# SIMD@OpenMP: A Programming Model Approach to Leverage SIMD Features

**Ph.D. Dissertation**

**Diego Luis Caballero de Gea**

**Department of Computer Architecture**
**Universitat Politècnica de Catalunya**

**November 2015**

Documento maquetado con TEXIS v.1.0+.

# SIMD@OpenMP: A Programming Model Approach to Leverage SIMD Features

Diego Luis Caballero de Gea

A dissertation submitted to the Department of Computer Architecture at Universitat Politècnica de Catalunya in partial fulfillment of the requirements for the degree of Doctor of Philosophy.

Thesis Supervisors:

Prof. Xavier Martorell Bofill, Universitat Politècnica de Catalunya
Dr. Alejandro Duran González, Intel Corporation

Dissertation Pre-defence Committee:

Prof. Jesús José Labarta Mancho, Universitat Politècnica de Catalunya
Prof. Daniel Jiménez González, Universitat Politècnica de Catalunya
Prof. Roger Espasa Sans, Universitat Politècnica de Catalunya

External Reviewers:

Dr. Ayal Zaks, Intel Corporation
Dr. Alexandre Eichenberger, IBM

Dissertation Defence Committee:

Dr. Albert Cohen, INRIA
Prof. Jesús José Labarta Mancho, Universitat Politècnica de Catalunya
Dr. Ayal Zaks, Intel Corporation

Prof. Roger Espasa Sans, Universitat Politècnica de Catalunya
Dr. Francesc Guim Bernat, Intel Corporation

Barcelona, November 2015

UNIVERSITAT POLITÈCNICA DE CATALUNYA
BARCELONATECH
Escola de Doctorat

| |
|---|
| **Curs acadèmic: 2015** |

## Acta de qualificació de tesi doctoral

Nom i cognoms
Diego Luis Caballero de Gea

Programa de doctorat
Arquitectura de Computadors

Unitat estructural responsable del programa
Departament d'Arquitectura de Computadors

## Resolució del Tribunal

Reunit el Tribunal designat a l'efecte, el doctorand / la doctoranda exposa el tema de la seva tesi doctoral titulada SIMD@OpenMP: A Programming Model Approach to Leverage SIMD Features.

Acabada la lectura i després de donar resposta a les qüestions formulades pels membres titulars del tribunal, aquest atorga la qualificació:

☐ NO APTE ☐ APROVAT ☐ NOTABLE ☐ EXCEL·LENT

| | |
|---|---|
| (Nom, cognoms i signatura)<br><br>President/a | (Nom, cognoms i signatura)<br><br>Secretari/ària |

| | | |
|---|---|---|
| (Nom, cognoms i signatura)<br><br>Vocal | (Nom, cognoms i signatura)<br><br>Vocal | (Nom, cognoms i signatura)<br><br>Vocal |

______________________, _______ d'/de __________________ de _______________

El resultat de l'escrutini dels vots emesos pels membres titulars del tribunal, efectuat per l'Escola de Doctorat, a instància de la Comissió de Doctorat de la UPC, atorga la MENCIÓ CUM LAUDE:

☐ SÍ ☐ NO

| | |
|---|---|
| (Nom, cognoms i signatura)<br><br>President de la Comissió Permanent de l'Escola de Doctorat | (Nom, cognoms i signatura)<br><br>Secretari de la Comissió Permanent de l'Escola de Doctorat |

Barcelona, _______ d'/de ____________________ de _________

# Abstract

SIMD instruction sets are a key feature in current general purpose and high performance architectures. In contrast to regular instructions that operate on scalar operands, SIMD instructions apply in parallel the same operation to a group of data, commonly known as vector. A single SIMD/vector instruction can, thus, replace a sequence of isomorphic scalar instructions. Consequently, the number of instructions can be significantly reduced leading to improved execution times.

However, SIMD instructions are not widely exploited by the vast majority of programmers. In many cases, taking advantage of these instructions relies on the compiler, domain-specific libraries and programming model support libraries used in the application. Nevertheless, despite the fact that compilers have a sophisticated vectorization technology, they struggle with the automatic vectorization of codes. Domain-specific libraries usually have support for limited SIMD architectures and programming model support libraries are not designed to exploit SIMD instructions. Advanced programmers are then compelled to exploit SIMD units by hand, using low-level hardware-specific intrinsics. This approach is cumbersome, error prone and not portable across SIMD architectures.

This thesis targets OpenMP to tackle the underuse of SIMD instructions from three main areas of the programming model: language constructions, compiler code optimizations and runtime algorithms. We choose the Intel Xeon Phi coprocessor (Knights Corner) and its 512-bit SIMD instruction set for our evaluation process. We make four contributions aimed at improving the exploitation of SIMD instructions in this scope.

Our first contribution describes a compiler vectorization infrastructure suitable for OpenMP. This vectorization infrastructure targets for-loops and whole functions. We define a set of attributes for expressions that determine how the code is vectorized. Our vectorization infrastructure also implements support for several vector features, such as vector function versioning, gather/scatter memory operations, vector math functions (SVML library), software prefetching and predicated vectorization. This infrastructure is proven to be effective in the vectorization of complex codes and it is the basis upon which we build the following two contributions.

The second contribution introduces a proposal to extend OpenMP 3.1 with SIMD parallelism. Essential parts of this work have become key features of the SIMD proposal included in OpenMP 4.0. We define the `simd` and `simd for` directives that allow programmers to describe SIMD parallelism and guide the compiler in the vectorization process of loops and whole functions. Furthermore, we propose a set of optional clauses that leads the compiler to generate a more efficient vector code. These SIMD extensions improve the programming efficiency when exploiting SIMD resources. At the same time, they allow programmers to exploit the advanced vectorization technology available in many compilers, overcoming the limitations found in automatic approaches. Our evaluation on the Intel Xeon Phi coprocessor shows that our SIMD proposal allows the compiler to efficiently vectorize codes poorly or not vectorized automatically with the Intel C/C++ compiler. For instance, we obtain up to a speed-up of 4.40 in the *Distq* benchmark and up to 12.90 in the Mandelbrot benchmark over their auto-vectorized versions.

In the third contribution, we propose a vector code optimization that enhances *overlapped* vector loads. These vector loads redundantly read scalar elements from memory that have been already loaded by other vector loads. Our vector code optimization improves the memory usage of these accesses building a vector register cache and exploiting register-to-register instructions. Our proposal also includes a new clause (*overlap*) in the context of the SIMD extensions for OpenMP of our first contribution. This new clause allows enabling, disabling and tuning this optimization on demand. Our evaluation on the Intel Xeon Phi coprocessor reports up to 29% improvement over the execution time of benchmarks highly optimized for this architecture.

The last contribution tackles the exploitation of SIMD instructions in runtime algorithms for OpenMP. We focus on the barrier and reduction primitives and propose a new combined barrier and reduction tree scheme specifically designed to make the most of SIMD instructions. Our barrier algorithm takes advantage of simultaneous multi-threading technology (SMT) and it utilizes SIMD memory instructions in the synchronization process. These instructions allow checking and releasing multiple barrier thread counters at a time. Our tree reduction algorithm performs vertical SIMD reduction throughout the different levels of the tree and a final horizontal SIMD reduction in the root node. Our evaluation on the Intel Xeon Phi coprocessor shows that our SIMD approach outperforms by up to 70% and 76% the best barrier and reduction algorithms in the Intel OpenMP Runtime Library, respectively. We obtained up to an 11% performance gain in the Cholesky kernel and up to 26% and 30% in the NAS CG and MG benchmarks, respectively.

The four contributions of this thesis are an important step in the direction of a more common and generalized use of SIMD instructions. Our work is having an outstanding impact on the whole OpenMP community, ranging from users of the programming models (through OpenMP 4.0) to compiler and runtime implementations. Our proposals in the context of OpenMP improves the programmability of the programming model, the overhead of runtime services and the execution time of applications by means of a better use of SIMD resources available in modern multi- and many-core architectures.

# Acknowledgments

This thesis has come to a successful end thanks to the help, support and disposal of many extraordinary people that I have had the opportunity to meet and work with.

Firstly, I would like to thank my advisors, Xavier Martorell and Alejandro Duran for going ahead with this project despite all the difficulties that have arisen over the last five years and for pushing me beyond my limits in order to get the best of me. They have always supported me, giving me freedom to develop my own ideas. Their valuable guidance has allowed me to get this far and to have an exciting future ahead. I am extremely grateful.

I would also like to thank my closer workmates as well as all the people working in the Computer Sciences department at Barcelona Supercomputing Center. I must highlight how much I have learned from them, since they are really outstanding and special people. I would especially like to mention Xavier Teruel, who was my mentor on my first work trip and conference. I also want to thank all the members of the Mercurium compiler group. Although the group is going through a difficult time, we have managed to push all our projects forward by means of collaborating with each other. My deepest gratitude goes to Roger Ferrer and Sara Royuela. I have been fortunate to work with the two most outstanding compiler engineers I have ever known and even better people. They truly feel passion for compilers and fortunately, they transmitted that passion to me. They both have selflessly devoted a lot of time to help me with this thesis. I thank them for their priceless advice and all the discussions we had. They both were co-authors in most of my articles and I do hope they also feel co-authors of this thesis. I am very proud and grateful to have worked with them.

I must also thank people from the Department of Computer Applications in Science and Engineering for their willingness to collaborate with our department. In particular, I thank Albert Farrès, Mauricio Hanzich, Josep de la Puente, Raul de la Cruz and Maria Jose Cela for sharing their expertise and wisdom with me.

My gratitude is also to Jesús Labarta, Daniel González, Roger Espasa, Ayal Zacks and Alexandre Eichenberger for reviewing this thesis. They kindly helped me improve its quality with their detailed analysis and brilliant comments, suggestions and discussion. I really appreciate it.

I shall also mention and express my gratitude to Anton Lokhmotov and his team from ARM Ltd. Cambridge. Anton was my supervisor during my Ph.D. internship.

I had the opportunity of working with an exceptional compiler team that allowed me to know how things work in industry.

Last but not least, I would like to thank all my family and friends. Especially, to my closest family, my source of inspiration and motivation: Tony, María Josefa, Felipe, Juan José, Felipe, Aurora, Isabel María, Juanjito and Adrián.

This Ph.D. candidacy has been funded by the FI-DGR grant from Agència de Gestió d'Ajuts Universitaris i de Recerca (AGAUR), Barcelona Supercomputing Center, the European projects EnCORE (FP7-248647), DEEP (FP7-ICT-287530) and DEEP-ER (FP7-ICT-610476), the HiPEAC-3 Network of Excellence (ICT FP7 NoE 287759), the Spanish Ministry of Education under contracts TIN2007-60625, TIN2012-34557 (Computación de Altas Prestaciones VI) and CSD2007-00050 (Supercomputación y e-Ciencia), and the Generalitat de Catalunya under contracts 2009-SGR-980 and 2014-SGR-1051 (MPEXPAR, Models de Programació i Entorns d'eXecució PARal.lels). Any opinions, findings and conclusions or recommendations expressed in this thesis are those of the author and do not necessarily reflect the views of the funding agencies.

Diego Caballero,

Barcelona, 2015

# Contents

# List of Figures

# List of Tables

# List of Listings

# Glossary

**barrier** Synchronization point for a group of threads where a thread waits until all the other threads have reached this point before proceeding with the execution.

**compiler directive** Language construction that provides the compiler with information about how to process a piece of code.

**intrinsic** Function that is handled specially by the compiler. In vectorization, SIMD intrinsics provide programmers with direct access to SIMD assembly instructions of a particular architecture.

**OpenMP** Parallel programming model for shared-memory machines based on compiler directives.

**reduction** Operation that consists of aggregating a set of values into a single one using a combiner operation.

**SIMD** It stands for Single-Instruction-Multiple-Data. It is a kind of data parallelism where a single instruction is simultaneously executed on a set of data at a time.

**vector** Synonym of *SIMD* in most of cases. It also refers to data parallelism exploited in traditional vector processors. We use this last meaning when vector processors are explicitly referenced.

**vectorization factor** In loop vectorization, number of scalar iterations of the loop computed within a single vector iteration. It is not necessarily related with the *vector register* of the architecture.

**vector lane** Independent sub-unit that represents each scalar element within a *vector register*.

**vector length** Number of data elements that fits into a physical *vector register*. This number can vary for the same architecture depending on the size of the data element used.

**vector mask** Special data type that is optionally used in vector operations to represent which *vector lanes* are enabled and disabled in the execution of a vector instruction.

**vector register** Hardware component used to host vector operands when executing vector instructions.

# Chapter 1

# Introduction

In the last decades, computer architecture research and development has experienced a fundamental shift in the way to increase performance of future generations of processors. Performance improvements mainly based on scaling clock frequency are no longer an option due to the arising of intractable physical barriers. These physical barriers, such as the well-known *power wall* (unaffordable overall temperature and power consumption mainly due to increasing too much the clock frequency of the processor), and energy efficiency [123, 31, 18], have turned aggressive hardware improvements focused on single-core performance into more energy-aware parallel solutions.

Particularly, in the High Performance Computing (HPC) segment, new highly parallel processors have been released with a huge aggregate performance, moderate power consumption but considerably low performance per single core. Examples of these processors are the new Power 8 CPU [102], the latest NVIDIA GPUs [41] and the Intel Xeon Phi processors family [38].

The first product of the Intel Xeon Phi family, a co-processor code-named *Knights Corner*, has up to 61 cores per chip and 4 hardware threads per core (up to 244 threads overall). Every individual core features 512-bit Single-Instruction-Multiple-Data (SIMD) units that are vital to achieve a peak performance of 1 TeraFLOPS in double precision with a Thermal Design Power (TDP) of 300W [76, 136, 65].

In this context of energy-aware high performance processors, SIMD instruction sets play a key role, as in the mentioned case of the Intel Xeon Phi coprocessor. Their appealing power consumption-to-performance ratio has led to a higher investment on these hardware components. Hardware architects are devoting more die area in wider vector units that can execute the same operation on a progressively larger amount of data at a time, without increasing the power consumption of that operation drastically.

The evolution of SIMD instruction sets from Intel and other vendors throughout the years attests this widening of the vector length of SIMD units, as shown in Figure 1.1. AMD 3DNOW! and Intel MMX were the first approaches that introduced 64-bit vector units in their general purpose processor. Later on, 128-bit SIMD in-

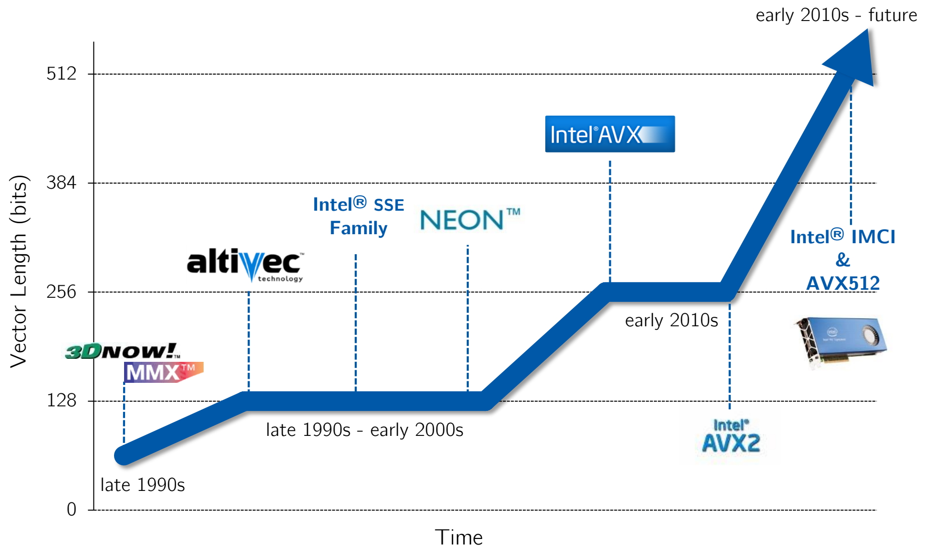


Figure 1.1: Vector length evolution of some SIMD instruction sets (time axis is just illustrative and it is not to scale)

structions arose from different hardware companies. The first one was AltiVec, also known as Velocity Engine and Vector/SIMD Multimedia eXtensions (VMX). It was proposed by Apple, IBM and Motorola in 1999 [33]. These extensions have been used in a wide variety of devices, ranging from embedded systems to HPC processors (Cell B.E.) and video game consoles [27].

Intel also released a series of 128-bit SIMD extensions called Streaming SIMD Extensions (SSE, SSE2, SSE3, SSSE3, SSE4.1 and SSE4.2) from 1999 to 2007 [72]. These extensions are used, even currently, in general purpose and HPC processors.

Meanwhile, ARM announced its advanced 64-bit and 128-bit SIMD extensions under the name Neon in 2004 [11]. These extensions are mainly included in low-power and mobile devices.

After some years, Intel stepped forward and released Advanced Vector Extensions versions 1 (AVX) and 2 (AVX2) with 256-bit vector units [37]. More recently, Intel moved to 512-bit vector extensions with the Initial Many Core Instructions (IMCI) [39] included in the Intel Xeon Phi coprocessor (Knights Corner). They also unveiled their continuation with this vector length in the upcoming AVX-512 SIMD instruction set [74] for general purpose and HPC processors.

Furthermore, SIMD extensions have gradually included more sophisticated, powerful and flexible instructions that bring vectorization to a broader range of applications and domains. In newer instruction sets, we can find, for instance, support for combined horizontal operations, improved support for unaligned memory accesses, gather and scatter memory operations, advanced support for predicated vectorization, improved shuffle and permutation instructions, non-temporal streaming stores, advanced software prefetching instructions for vector memory operations and instructions to detect write conflicts in scatter memory operations, among others [74].

In conclusion, this brief summary of some SIMD instruction sets shows that their relevance is increasing with time. These extensions are used in processors from different fields ranging from mobile and embedded devices to HPC systems. We observe the trend of increasing the vector length and features over generations, at least regarding Intel architectures. This fact also confirms that architects are relying more on these devices to achieve performance and energy goals.

We foresee that SIMD instructions will be determinant in future processors. Therefore, their optimal exploitation turns essential to take fully advantage of all the computational power of current and future architectures at a lower energy rate.

## 1.1 The Problem

The higher investment in SIMD resources per core makes extracting the full computational power of these vector units more important than ever before. However, on the software side, there still exists a considerable underuse of these vector resources which leads to not fully exploiting all the computational power of these new architectures. In the context of programming models, we find deficiencies from two different perspectives: exploitation of SIMD resources at programmer level and exploitation of SIMD resources in services and algorithms used in programming model support runtime libraries.

### 1.1.1 SIMD Exploitation at Programmer Level

As far as programmers are concerned, there exist three major ways of exploiting these vector units: compiler auto-vectorization, low-level intrinsics or assembly code and programming models/languages with explicit SIMD support [153].

Regarding auto-vectorizing compilers, fully automatic techniques do not always exploit SIMD parallelism successfully in real applications. The reasons that can prevent the compiler to auto-vectorize a code can be very varied: unknown trip-count of loops, pointer aliasing, data alignment constraints, complex data and control flow, mixed data lengths and complex data dependences (loop cross-iteration dependences), among others. Function calls are another highly limiting factor for auto-vectorization and for many code optimizations in general. They affect the flow of the execution hindering the analysis of the code.

A lot of work has been done by the community to overcome these limitations [54, 110, 157, 158, 108, 112, 80, 111, 92]. However, the truth is that many techniques have not stepped into production compilers [99]. The main reasons are the limitations found when applying aggressive code optimizations in general. Some of these limitations are the following:

**Lack of compilation technology:** limits in static analyses, cost models, idiom recognition and code transformations at production level. These limitations are especially acute for functions since inter-procedural approaches are complex and highly time-consuming. In addition, sometimes they have to be applied at link-time. Some examples for auto-vectorization are: inter-procedural pointer aliasing analysis, accurate data and control dependences (loop cross-iteration dependences) analysis, analysis and code transformations for data and control divergences, accurate cost models to decide which loop is better to be vectorized, code patterns detection and code transformations that allow mapping the code efficiently to available vector instructions and major code transformations to make complex codes more suitable for vectorization.

**Scheduling and aggressiveness of optimizations:** optimizations need correctness and safety validation, and performance estimation to be applied. They can lead to wrong code, degrade performance for some cases, interfere with other optimizations or even completely prevent others that are more profitable. In particular, for instance, vectorizing a loop could interfere with thread parallelism, loop interchange and other loop-level optimizations.

**Code expected behavior:** aggressive code transformations could cause an unexpected behavior for advanced programmers working on fine tuning their applications or interfere in their hand-coded optimization process. For example, vectorizing a loop could require aggressive changes in data structures or in the structure of a loop or loop nest. Optimizations applied by programmers on the original code could have then no effect, or being harmful after the transformations required by vectorization.

Furthermore, on the contrary to other aggressive code optimizations, vectorization, on the one hand, is a high-level optimization transforming entire loops whereas, on the other hand, it depends on the specific low-level SIMD capabilities of the target architecture.

Even if compilers were improved in all the previous aspects, they would have to follow conservative approaches to guarantee the correctness of the execution under any possible circumstance. In addition, compilers would still have to adopt conservative approaches that prevented optimizations for those cases where heuristics were not accurate enough to minimize performance degradation.

Consequently, compilers still could inhibit from vectorizing codes that programmers know that vectorization can be safe and efficient because of certain characteristics of the application or the input data set.

Regarding low-level intrinsics, on the bright side, they allow advanced programmers to have direct access to hardware, and thus, to all the SIMD features available in the underlying architecture. In order to use them, programmers must have in mind the vector length and those features supported by the target architecture. Unfortunately, most regular programmers do not have the required knowledge to take advantage of them. Furthermore, dealing with vector intrinsics is a very tedious

```
for(i = 0; i < width; i++){
    float x = x0 + i * dx;
    float y = y0 + j * dy;
    float z_re = x;
    float z_im = y;

    for (k = 0; k < maxIterations; ++k){
        if (z_re * z_re + z_im * z_im > 4.0f) break;

        float new_re = z_re*z_re - z_im*z_im;
        float new_im = 2.0f * z_re * z_im;
        z_re = x + new_re;
        z_im = y + new_im;
    }
    output[j * width + i] = k;
}
```

Listing 1.1: Mandelbrot scalar code snippet

and error prone task even for advanced programmers. Listing 1.1 shows a snippet of the Mandelbrot algorithm and Listing 1.2 shows only a part (epilogue loop is not shown) of the equivalent code vectorized with SIMD intrinsic for the Intel Xeon Phi coprocessor (Knights Corner). Both figures illustrate how cumbersome and challenging vectorizing by hand even a simple code can be, in terms of code rewriting and complexity of the resulting code. In addition, intrinsics are hardware-specific and they are not portable among different architectures.

In an attempt to release programmers from the burden of using SIMD intrinsics, new programming models with SIMD support have emerged, and well-established ones have been extended with SIMD features. Some examples are OpenCL [82], OpenACC [115], ISPC [122] and Intel Cilk [70]. Although there are important differences among them, they all provide a significantly higher level of abstraction and portability than SIMD intrinsics. However, while some of them are specific for particular architectures, others still expose too many low-level details and require hard redesign of algorithms and a significant amount of source code rewriting. This makes them neither useful nor efficient for the vast majority of programmers.

In our opinion, a high-level directive-oriented hardware independent programming model, such as OpenMP [117], seems to be the most appropriate way to exploit SIMD resources efficiently for most of the programmers. Compiler directives are orthogonal to the actual code which allows minimizing modifications of the original version of the application. Moreover, they offer programmers a high-level and programming efficient way to exploit several kinds of parallelism portable among different architectures.

```
for (i = 0; i <= width - 16; i = 16 + i){
    __m512i voffset = _mm512_set_epi32(15,14,13,12,11,10,9,8,7,6,5,4,3,2,1,0);
    __m512i vone = _mm512_set1_epi32(1);
    __m512i t2 = _mm512_add_epi32(voffset, _mm512_set1_epi32(i));
    __m512 t3 = _mm512_cvtfxpnt_round_adjustepi32_ps(t2,
                    _MM_ROUND_MODE_NEAREST, _MM_EXPADJ_NONE);
    __m512 t6 = _mm512_mul_ps(t3, _mm512_set1_ps(dx));
    __m512 x = _mm512_add_ps(_mm512_set1_ps(x0), t6);
    float y = y0 + j * dy;
    __m512 z_re = x;
    __m512 z_im = _mm512_set1_ps(y);
    __m512i k = _mm512_set1_epi32(0);
    __m512i t7 = _mm512_set1_epi32(maxIterations - 1);
    __mmask16 msk0 = _mm512_cmp_epi32_mask(k, t7, _MM_CMPINT_LE);

    for (; msk0 != 0; k = _mm512_mask_add_epi32(k, msk4, vone, k),
                   msk0 = _mm512_mask_cmp_epi32_mask(msk4, k, t7, _MM_CMPINT_LE)) {
        __m512 z_re2 = _mm512_mask_mul_ps(_mm512_undefined(), msk0, z_re, z_re);
        __m512 z_im2 = _mm512_mask_mul_ps(_mm512_undefined(), msk0, z_im, z_im);

        __m512 t11 = _mm512_mask_add_ps(_mm512_undefined(), msk0, z_re2, z_im2);
        __m512 t12 = _mm512_set1_ps(4.0f);
        __mmask16 msk1 = _mm512_mask_cmp_ps_mask(msk0, t11, t12, 14);
        __mmask16 msk3 = _mm512_kandnr(msk0, msk1);
        __mmask16 msk4 = _mm512_kor(_mm512_int2mask(0), msk3);

        __m512 new_re = _mm512_mask_sub_ps(_mm512_undefined(), msk4, z_re2, z_im2);
        __m512 z_mul = _mm512_mask_mul_ps(_mm512_undefined(), msk4, z_re, z_im);
        __m512 new_im = _mm512_mask_add_ps(_mm512_undefined(), msk4, z_mul, z_mul);
        z_re = _mm512_mask_add_ps(z_re, msk4, x, new_re);
        z_im = _mm512_mask_add_ps(z_im, msk4, _mm512_set1_ps(y), new_im);
    }
    _mm512_extpackstorelo_epi32(&output[j * width + i],
        k, _MM_DOWNCONV_EPI32_NONE, 0);
    _mm512_extpackstorehi_epi32(&output[j * width + i] + 64,
        k, _MM_DOWNCONV_EPI32_NONE, 0);
}
```

Listing 1.2: Mandelbrot vector code snippet using intrinsics for the Intel Xeon Phi coprocessor (Knights Corner)

### 1.1.2 Runtime Services

Parallel programming models require efficient runtime services that implement their semantic. These services make use of algorithms that must be as efficient as possible to reduce the overhead of creating and managing parallelism. This management can have a significant impact on the overall execution time of users' applications.

This issue gains importance with the advent of many-core architectures. Over these massively parallel machines, runtime systems have to deal with a larger number of threads than for multi-core processors. In this way, runtime algorithms must take fully advantage of the features of these new architectures in order to scale appropriately and not to become a performance bottleneck.

However, many traditional algorithms were designed years ago without having in mind new features available in current architectures, such as SIMD instructions or simultaneous multi-threading [62, 101, 109, 28, 55, 63, 52, 98]. This means that these algorithms could not be exploiting all the potential of modern multi and many-core architectures.

These runtime algorithms are kept hidden from regular programmers. They are implemented by developers of the programming model which have a deep knowledge of the hardware architecture and they are highly specialized in such a task. In this way, these runtime services are not easily accessible to regular or advanced programmers. Only developers of programming models are responsible for taking the best advantage of the underlying architecture.

## 1.2 Our Approach

We follow a programming model approach based on OpenMP to tackle the mentioned underuse of SIMD resources. This approach combines research and new proposals in the following three scopes of the OpenMP programming model:

**Language constructions:** Defining new language constructions in the programming model that allow programmers to efficiently instruct the compiler in the vectorization process.

**Compiler Code Optimizations:** Proposing vector code optimizations that can be directed by programmers using annotations of the programming model.

**Runtime algorithms:** Designing new runtime algorithms that exploit SIMD instructions and take better advantage of new hardware resources available in modern architectures.

As for language constructions, we propose extending OpenMP with new SIMD constructs that allow programmers to have further control over the vectorization process that the compiler applies to their applications [1]. Introducing these new SIMD extensions, we are moving the responsibility of vectorizing a code from the compiler to programmers, at the same that we are exploiting the advanced vectorization technology of the compiler.

Regarding code optimizations, we propose a vector code optimization that can be enabled, disabled and tuned using a new OpenMP directive in the context of the previous SIMD constructs.

On the runtime side of OpenMP, we propose new runtime algorithms that are specifically designed to take advantage of SIMD resources. These runtime algorithms naturally exploit SIMD instructions and other new hardware technologies such as simultaneous multi-threading (SMT).

[1] In the process of writing this thesis, SIMD extensions have been officially included in OpenMP 4.0, partially as the result of this work. We will discuss the impact of this thesis on the OpenMP 4.0 standard later on in the thesis.

## 1.3 Objectives of the Thesis

In the context of this Ph.D. thesis, we define the following objectives:

- Build a solid user-directed vectorization proposal in the context of OpenMP.
  - Propose an effective way to allow programmers to instruct the compiler in the vectorization process, providing it with more information that could improve the quality of the resulting vector code.
  - Bring more control into vector transformations for experienced users. Provide programmers with an agile way to perform a systematic benchmarking to find which optimizations/transformations yield better performance in their applications.
- Build a competitive research vectorization infrastructure using the Mercurium C/C++ source-to-source compiler.
  - Design and implement a prototype of our user-directed vectorization proposal in the Mercurium compiler in order to validate its applicability to actual applications and its usefulness in terms of improving the quality of the vector code generated by the compiler.
  - Leverage Mercurium as a source-to-source compiler with outstanding vectorization capabilities. Our goal is to be able to compete with the Intel C/C++ production compiler in terms of performance for some codes.
- Propose new vector code optimizations in the context of OpenMP.
  - Investigate on new aggressive vector code optimizations that are not easily applicable by the compiler automatically.
  - Extend OpenMP with new directives to bring the control of these optimizations to programmers.
  - Provide advanced programmers with an efficient approach to fine tune these optimizations.
- Propose runtime algorithms for OpenMP that take advantage of current SIMD capabilities of modern multiprocessors.
  - Investigate on runtime algorithms for OpenMP that are suitable to benefit from the use of SIMD instructions.
  - Choose some algorithms and propose new approaches where the main goal in their design is favoring the exploitation of SIMD instructions.

## 1.4 Contributions

In this thesis, we present several contributions in the field of the OpenMP programming model, compilers and runtime algorithms aimed at leveraging the exploitation of the SIMD instruction. The main contributions of this thesis are the following:

**Vectorization Infrastructure:** We present the vectorization approach developed in the Mercurium source-to-source compiler. This vectorizer is the largest software product resulting from this thesis. This contribution includes more than four years of design, implementation, integration, research and evaluation of compiler algorithms related with different aspect of vectorization.

**SIMD Extensions for OpenMP:** We introduce a set of new OpenMP directives and clauses that allow programmers to direct the compiler in the vectorization process. These directives and clauses provide the compiler with information valuable to weaken the constraints that prevent auto-vectorization in many cases. With this information, programmers cannot only tell the compiler that a loop is safely vectorizable but also describe further information about the characteristic of the code that will improve the vector code generated. Furthermore, our proposal brings vectorization to functions in a generic way, solving the important limitation with function calls present in auto-vectorizing compilers. In collaboration with Intel, this proposal established the basis of the official SIMD extensions now included in the 4.0 version of the OpenMP standard.

**Vector Code Optimization:** We propose a compiler vector code optimization that targets vector loads that redundantly load scalar elements from memory (*overlapped vector loads*). This optimization performs an aggressive code transformation. It builds and maintains a vector register cache to reuse data of those vector loads that overlap in memory. In this way, the optimization improves the use of the memory bandwidth. Moreover, in the context of the previous OpenMP SIMD extensions, we propose a new clause to allow advanced programmers to enable on demand and fine tune this optimization.

**SIMD Runtime Algorithms:** In the context of runtime services for OpenMP, we choose thread synchronization barrier and reduction primitives as relevant components to explore the applicability of SIMD instructions. As a result, we propose a novel synchronization barrier algorithm explicitly designed to exploit SIMD instruction. This new barrier approach makes use of distributed SIMD counters to carry out the synchronization of threads on a tree scheme. In conjunction with the SIMD barrier algorithm, we propose a novel reduction scheme that uses SIMD instructions to perform the computation of the OpenMP reduction primitive. This approach uses vertical and horizontal vector reductions operations according to different stages of the reduction scheme.

## 1.5 Context

This thesis has been conducted as part of the Doctor in Philosophy Program on Computer Architecture [30], in the Computer Architecture Department at Technical University of Catalonia (UPC).

The research activity has been developed in the context of the Parallel Programming Models Group [124] in the Computer Sciences Department at Barcelona Supercomputing Center (BSC). The main research line of the group comprises parallel programming models focused on high-performance computing and general-purpose parallelism. They aim to improve usability of low-level programming languages and to hide hardware details from programmers to increase their efficiency at working with complex computational systems and heterogeneous architectures.

Currently, the Parallel Programming Models Group is working on OmpSs [12], their own parallel programming model for heterogeneous architectures and parallel systems. They implement their solutions and prototypes on the Mercurium C/C++/FORTRAN source-to-source compiler and the Nanos++ OmpSs/OpenMP Runtime. Additionally, the research on OmpSs is used to propose valuable extensions to the OpenMP programming model.

Consequently, this dissertation has been carried out as much as possible in the context of the Programming Model Group interests and their tools and hardware/-software infrastructures.

This Ph.D. candidacy has been funded by the FI-DGR grant from Agència de Gestió d'Ajuts Universitaris i de Recerca (AGAUR), Barcelona Supercomputing Center, the European projects EnCORE (FP7-248647), DEEP (FP7-ICT-287530) and DEEP-ER (FP7-ICT-610476), the HiPEAC-3 Network of Excellence (ICT FP7 NoE 287759), the Spanish Ministry of Education under contracts TIN2007-60625, TIN2012-34557 (Computación de Altas Prestaciones VI) and CSD2007-00050 (Supercomputación y e-Ciencia), and the Generalitat de Catalunya under contracts 2009-SGR-980 and 2014-SGR-1051 (MPEXPAR, Models de Programació i Entorns d'eXecució PARal.lels).

## 1.6 Document Organization

The rest of this document is organized as follows: Chapter 2 presents the background of this thesis that includes the OpenMP programming model, the Mercurium source-to-source compiler, the Intel OpenMP Runtime Library and the Intel Xeon Phi coprocessor. Chapter 3 describes our vectorization infrastructure. Chapter 4 introduces our proposal on SIMD Extensions for OpenMP. Chapter 5 describes our Overlap vector code optimization. Chapter 6 presents our SIMD barrier and reduction algorithms. In Chapter 7, we address the related work and in Chapter 8 we conclude this thesis and present the future work. In addition, there is an appendix that contains the source code of some of the benchmarks used in our evaluations. Finally, the bibliography used in this thesis is at the end of the document.

# Chapter 2

# Background

In this chapter, we describe our previous related work conducted as part of a Master Thesis. In addition, we offer a brief description of several software and hardware components used in this PhD Thesis: the OpenMP programming model, the Mercurium source-to-source compiler, the Intel OpenMP Runtime Library and the Intel Xeon Phi coprocessor.

## 2.1 Master's Thesis Contributions

In the preceding Master's Thesis [21], we established the basis of this dissertation with a proof of concept of user-directed vectorization. For such a proof, we used an old version of the Mercurium compiler (Section 2.3) to develop a very limited prototype. It only had support for a preliminary standalone `simd` directive targeting Streaming SIMD Extensions (SSE).

Results showed a highly competitive performance, comparable to that achieved by manual vectorization using low-level languages and extensions, such as OpenCL and SSE intrinsics. Moreover, the gain was substantially higher than those from pure compiler auto-vectorization technology.

Due to the lack of compiler analysis in the Mercurium compiler at that time, the `simd` directive implementation required that programmers specified a list of pointers of those memory accesses that had to be vectorized. Furthermore, the Master's Thesis did not cover the interaction and combination of the `simd` directive with OpenMP fork-join parallelism. Thus, we started this Ph.D. thesis from this point, with certain experience in the field.

## 2.2 The OpenMP Programming Model

Some of the challenges that programmers have to face when working with a parallel architecture are related to the inherent non-deterministic behavior of parallelism. Thus, the main objective of programming models is to provide applications with a

```c
void saxpy(float *x, float *y, float *z, float a, int N)
{
    int j;

    #pragma omp parallel for private(j) shared(z, a, x, y, N)
    for (j=0; j<N; j++)
    {
        z[j] = a * x[j] + y[j];
    }
}
```

Listing 2.1: Saxpy kernel annotated with OpenMP constructs

deterministic, predictable, consistent behavior and performance while minimizing the programming effort.

OpenMP (Open Multi Processing) [117] is a multi-platform parallel programming model for shared-memory systems that meets the previous objectives in the context of thread-level parallelism. In addition to these objectives, the primary goal of OpenMP is making the development of C/C++ and FORTRAN parallel applications easier to programmers and portable across different architectures.

OpenMP is based on compiler high-level directives that ease the programmability preserving in most of cases the original structure of the code. It relies on compiler and runtime support to process and implement the functionality of these high-level directives. The programming model provides a fork-join execution model and a relaxed-consistency memory model that allow exploiting data-level and task-level parallelism in several ways with the main premise of maintaining the correctness of the sequential version of the code.

Releases of OpenMP previous to version 3.0 introduced support for the basis of the OpenMP fork-join model: parallel constructs, worksharing constructs and data sharing attributes. Parallel constructs and worksharing constructs allow programmers to create parallel regions and to distribute the computational work within a parallel region among threads. Data sharing attributes state whether each variable declared outside of the parallel region will be shared by all threads or each thread will have a private copy of that variable. Listing 2.1 shows an example of a code annotated with OpenMP constructs. As depicted, the for-loop is annotated with a combined `parallel for` directive. This directive creates a new parallel region around the for-loop where multiple threads will run in parallel. Then, the iterations of the for-loop are distributed among these threads to be executed in parallel. The `private` data sharing attribute states that each thread will host a private copy of the variable `j`. On the contrary, the `shared` attribute states that array pointers `z`, `x`, `y` and variables `a` and `N` will be shared by all threads within the parallel region.

Major version 3.0 introduced task parallelism in OpenMP with a set of task related constructs. These constructs allow describing arbitrary regions of code, not necessarily for-loops, within a parallel region, that will be assigned to a single thread for their execution. Further refinement on the task proposal was introduced in OpenMP 3.1 [116]. This version of the standard is the one that we use in our proposal described in Chapter 4.

Version 4.0 is the latest version of OpenMP at the time of writing this thesis [117]. This version introduced major features such as support for accelerators, thread affinity policies, cancellation constructs and improvements in the tasking model. SIMD extensions are also defined to exploit SIMD parallelism. These extensions are influenced by the contribution of this thesis described in Section 4.

In OpenMP, neither the compiler nor the runtime but the programmer is responsible for the correct description of the parallelism of the application. Accordingly, OpenMP-compliant implementations are not required to check for potential parallel issues, such as data dependences, deadlocks or race conditions. These facts ease its implementation even in infrastructures with low level resources. Furthermore, as long as the semantic of OpenMP is preserved, compilers and runtimes are allowed to use complementing automatic approaches for detecting and exploiting additional parallelism opportunities.

### 2.2.1 Execution Model

OpenMP is based on a fork-join execution model. In this model, an initial thread starts the serial execution of the program. In some regions of the code, the execution is *forked* into multiple parallel threads of execution, denoted as a *team* of threads. At the end of these regions, this parallel execution is *joined* again into the initial thread of execution.

This fork-join execution model is orchestrated by compiler directives. These directives are aimed at describing OpenMP parallelism without changing the original code of the application. Therefore, an OpenMP-compliant application should still be valid if it is executed in a serial way ignoring the OpenMP directives. In addition, the output and behavior of the application must be the same for a serial or a parallel execution, independently from the number of threads used. The only admitted exception is the precision error resulting from different order in the association of operations.

OpenMP 4.0 defines a host-centric execution model in the execution context of multiple devices or accelerators. This means that the execution of an OpenMP application starts and ends in the *host* device. From this host, some regions of the code and data can be *offloaded* to be executed in other devices.

### 2.2.2 Memory Model

OpenMP describes a relaxed-consistency shared-memory model. The key aspect in this memory model is the *temporary view* of the shared memory defined for each thread. These temporary views allow threads to cache variables of the code, avoiding reading/writing from/to shared memory each time that is required by the execution. In this way, at some point of the execution, shared memory can be in an inconsistent state if the temporary view has not been flushed to shared memory. For that reason, OpenMP allows this inconsistency at the same time that it defines where the temporary view must be flushed to shared memory.

### 2.2.3 Barriers and Reductions in OpenMP

A barrier in OpenMP is a collective synchronization primitive where threads must stop and wait for all the threads in the team to reach the barrier. Once all threads in the team have reached the barrier, threads are released to continue their execution. Barriers in OpenMP can be explicitly defined by the programmer using the *barrier* directive. In addition, OpenMP defines implicit synchronization barriers by default at the end of some other OpenMP constructs. For example, there is an implicit barrier at the end of the *parallel* construct and at the end of the *for* construct and the *single* construct in the absence of the `nowait` clause.

A reduction operation in OpenMP is also a collective primitive where all threads in the team collaborate to compute a shared value. In this reduction computation, all threads in the team have a local copy of the target reduction variable where they compute their partial reduction value. Then, threads collaborate to merge all the local variables using a combiner operator. The result of this combination is the result of the reduction operation. Programmers can explicitly describe a reduction operation in OpenMP using the *reduction* clause. With this clause, programmers can specify the reduction variables and the combiner operator to use in the combination of the thread local reduction values.

OpenMP defines a particular semantic that affects barrier and reduction primitives. This semantic has special interest in the context of this thesis. Synchronization barriers in OpenMP are not only a point to synchronize threads but also a point where many OpenMP features have to implement part of their functionality. For example, a *task scheduling point* is defined within barriers. This means that threads waiting in the barrier must check for OpenMP ready tasks and proceed with their execution. The barrier primitive is also a point where all the threads in the team must perform a memory flush. Furthermore, barriers are also a point to check for active cancellation points, as defined in the OpenMP 4.0 [117].

As for reduction operations, on the contrary to other programming models, in OpenMP the value of the reduction does not have to be broadcast to all the threads involved in the computation but written into a shared variable. This means that a reduction operation is not a synchronization primitive by itself because threads only have to provide their partial reduction value. At some point, these threads should be able to obtain the final reduction value from the shared reduction variable. Consequently, OpenMP defines that the final reduction value must be available for all threads after a synchronization barrier. This rule leads to a strong coupling of the computation of the reduction operations and the synchronization barrier.

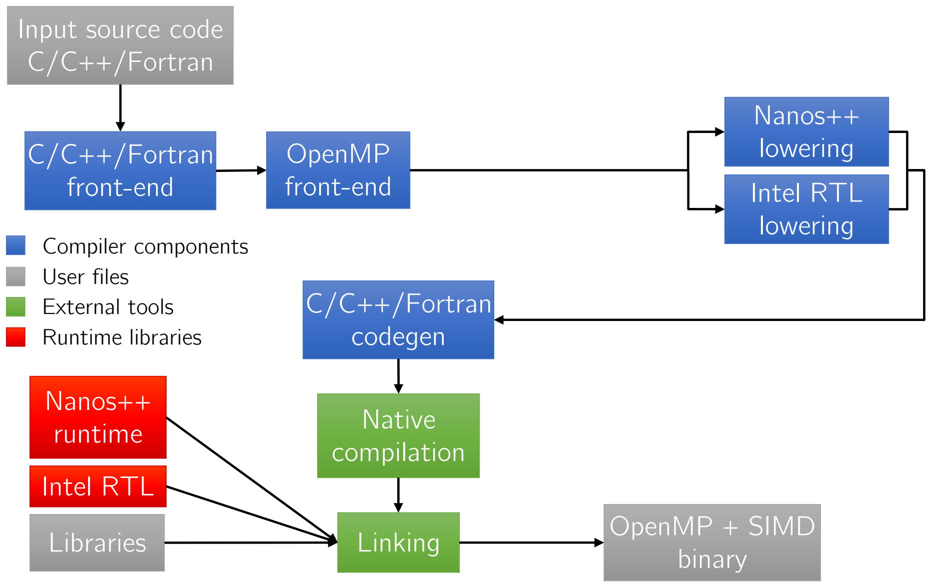


Figure 2.1: Mercurium compilation diagram

## 2.3 The Mercurium Source-to-source Compiler

Mercurium [124, 57] is a research source-to-source compiler with support for the C, C++ and FORTRAN programming languages. It is developed at Barcelona Supercomputing Center with the main goal of building an infrastructure for fast prototyping of new parallel programming models. Mercurium has already support for several programming models such as OpenMP 3.1 [116] (partial support), OmpSs and StarSs [12]. In addition, it is also able to manage CUDA and OpenCL codes.

Figure 2.1 shows a high-level scheme with the different phases of the Mercurium compiler. We can distinguish three main parts in the structure of the compiler. The first one is a front-end with full support for C, C++ and FORTRAN syntaxes that gathers all the symbolic and typing information into the intermediate representation of the compiler. The second one is a pipelined sequence of phases that performs all the source-to-source transformations and the specific features required by the target programming model and runtime. This sequence of phases is represented in Figure 2.1 with the OpenMP front-end phase and the lowering phases for the Nanos++ Runtime Library and the Intel OpenMP Runtime Library. Finally, the last part of the Mercurium compiler is the code generator. This part of the compiler is responsible for regenerating the reliable final source code from the intermediate representation of Mercurium. The output code is then ready to be compiled and linked with the chosen native compiler.

```
expression: NODECL_ADD([lhs]expression, [rhs]expression) type const-val
          | NODECL_LOWER_THAN([lhs]expression, [rhs]expression)
                              type const-val
          | NODECL_ARRAY_SUBSCRIPT([subscripted]expression,
                                   [subscripts]expression-seq)
                                   type const-val
          | NODECL_FUNCTION_CALL([called]expression, [args]argument-seq)
                                 type const-val
```

Listing 2.2: Example of nodes of the intermediate representation of Mercurium. The whole grammar can be found in the file `cxx-nodecl.def` [124]

### 2.3.1 Intermediate Representation

The intermediate representation (IR) of Mercurium is a tree-like IR based on an abstract syntax tree (AST) data structure. This AST represents the input code in a high-level and accurate way. Unlike low-level IRs, a high-level and accurate IR is indispensable to generate an output source code as similar as possible to the input code, which is one of the main goals of the Mercurium source-to-source compiler.

The AST is built in the Mercurium front-end with information of the explicit code and data of the compilation unit. However, information regarding declarations is minimal in this AST as the code generator of Mercurium is able to deduce it from the code representation. The type system and other symbolic information are represented separately from the AST. Mercurium uses independent data structures for these purposes that are accessible from fields of the AST nodes.

A single grammar describes the set of AST nodes that are used for the representation of the C, C++ and FORTRAN programming languages. These nodes are mainly shared by the three languages. However, there are also a few specific nodes that are aimed at representing particularities of a language that are not present in the others. An example of this grammar is shown in Listing 2.2, where four rules describe the nodes that represent *arithmetic additions*, *lower than* comparisons, *array subscripts* and *function calls*. Each AST node is comprised by kind of the node, up to 4 children nodes and a set of external attributes that are node-kind dependent. For example, these attributes can be the `type` of the node used of the type system, a symbol, a scope of the language or an associated constant value (`const-val`). This grammatical description is automatically translated to a non-hierarchical class system that is then used by the developers of the compiler.

### 2.3.2 Analysis Infrastructure

The Mercurium analysis infrastructure was developed in parallel with the work of this thesis. Although some analysis techniques were implemented with research purpose, they are able to provide the compiler with an outstanding and solid analysis resources.

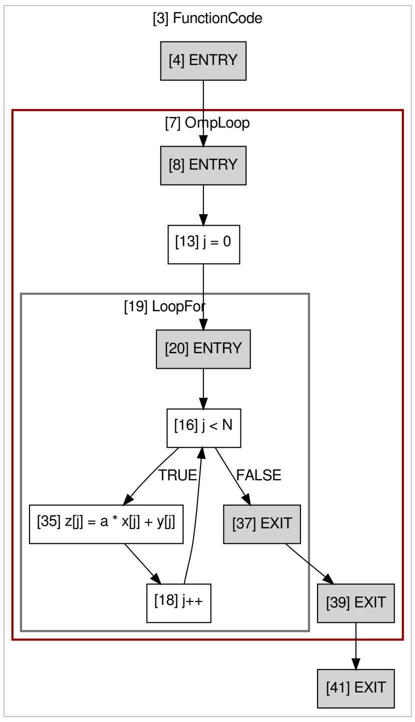


Figure 2.2: Example of a parallel control flow graph of the Saxpy computation

The main compiler analysis techniques implemented in Mercurium are use definitions, liveness, reaching definitions, induction variable analysis and range analysis [148, 125]. These techniques combine old and new traditional approaches applied from the particular and restricted viewpoint of a source-to-source compiler. The restrictions that affect the implementation of analysis techniques in the context of the Mercurium compiler are mainly the characteristics imposed on the intermediate representation (IR). The Mercurium compiler utilizes a high-level and accurate IR (see Section 2.3.1) in favor of keeping the original structure of the input source code. This means that the IR is generic and it is neither in SSA nor three-address form.

All the analysis techniques implemented in Mercurium use a parallel control flow graph (PCFG). This structure is a hyper-graph that extends the classic CFG representation to describe parallelism, such as information of the OpenMP parallel constructs. The nodes of this PCFG can be *simple* or *structured* and they contain pointers to their counterpart nodes in the AST. Simple nodes represent sequential execution of one or more statements. Structured nodes are PCFGs that represent control flow or parallel semantic. Further information about the PCFG infrastructure is detailed in our work about correctness of OpenMP tasks [131].

Figure 2.2 shows an example of a simple loop annotated with an OpenMP *for* directive, which is represented with the *OmpLoop* node. This node states that the iterations of the nested loop might be executed in parallel. This information will be taken into account in all the analysis algorithms implemented on this PCFG. The nodes *FunctionCode*, *OmpLoop*, *LoopFor* are structured nodes, i.e., PCFGs with their corresponding *Entry* and *Exit* nodes. The remaining nodes in the example are simple nodes.

In the context of this thesis, we intensively exploit use-definitions, liveness, reaching definitions and induction variable analyses.

## 2.4 The Intel OpenMP Runtime Library

The Intel OpenMP Runtime Library is a production library implemented by Intel that gives support for the OpenMP programming model. The first open-source version of this library was released in April, 2013 [75].

In this section, we give a brief overview of some components of this runtime library. We limit our description to those components related with the contributions of this thesis: some basic components of the runtime and those components related with the barrier and reduction mechanisms.

### 2.4.1 Threads and Teams of Threads

The Intel OpenMP Runtime Library defines structures to model the concepts of *thread* and *team* of threads in OpenMP, described in Section 2.2. The team structure contains basic information of the team, such as information about the parent team, threads in the team, global structures of barriers, and other variables needed to implement specific features of OpenMP. The thread structure has basic information of the thread, such as several IDs of the thread for different contexts, local structures of barriers, the team it belongs to and private internal control variables (ICVs) that control the behavior of a thread in a OpenMP program. The thread structure also contains fields to cache some information from the team structure. This is useful to reduce the memory overhead that happens when a large number of threads in the team have to access the team structure.

Threads and teams are created and placed in a *pool of threads* and a *pool of teams*, respectively, when they are not being used. When the runtime needs a team, it first checks whether there is one available in the pool of teams to avoid creating a new one. If there is a team available in the pool, the runtime removes the team from the pool and it initializes and assigns the threads needed to that team. If there is no team available in the pool, the runtime creates a new team. After its use, the runtime finalizes the team and it moves the threads of that team to the pool of threads. Finally, the team is placed in the pool of teams. In a similar way, when the runtime needs a new thread for a team, it first checks the pool of threads. If there is one available, the runtime initializes and reuses it. If there is no thread available, the runtime creates a new one from scratch. These pools of threads and teams improve the performance by reusing threads and teams instead of creating them from scratch and destroying them after a single use.

In addition to the pools, the runtime implements another mechanism to improve the reuse of teams that have been used recently. These teams are denoted as *hot teams*. When a team has just finished being used, it is in a *hot* state. Teams that are hot are not placed in the pool of teams and neither are their threads. Both team and threads remain in a latent state almost ready to be reused. When the runtime needs a new team, hot teams are reused before teams from the pool. Hot teams only require a light re-initialization as they already have threads assigned. This approach reduces even more the overhead of managing teams and threads.

### 2.4.2 Synchronization Barrier

As we described in Section 2.2.3, OpenMP introduces an important semantic load on the barrier primitive. When threads reach the barrier, there are a set of duties that they have to do besides the synchronization. For this reason, the barrier mechanism is a key component in the Intel OpenMP Runtime Library. This library uses the barrier mechanism for the following tasks:

- Performing a standard synchronization barrier among threads in a team.
- Forking and joining threads in a parallel construct.
- Putting threads on waiting when they are in the pool of threads.
- Computing reduction operations.
- Distributing ICVs among all the threads in the team.

According to these tasks, the barrier mechanism must work in several scenarios at the same time as it must provide all the functionality required by OpenMP. This fact leads to distinguishing between the following three kinds of barriers:

**Plain barrier:** The plain barrier is the common synchronization barrier used to implement implicit and explicit barriers in OpenMP. For example, the OpenMP *barrier* directive is translated to this kind of barrier.

**Fork-join barrier:** This kind of barrier is used in the creation and destruction of parallel regions, such as the one denoted by an OpenMP *parallel* directive. This kind of barrier requires two well-differentiated phases: a *release* phase to fork threads within the parallel region, and a *gather* phase to join all the threads at the end of the parallel region. If there is a reduction associated to the parallel region, the gather phase of the reduction barrier is used instead of the gather phase of this barrier.

**Reduction barrier:** The reduction barrier is a barrier that has reduction operations associated. This kind of barrier also requires split gather and release

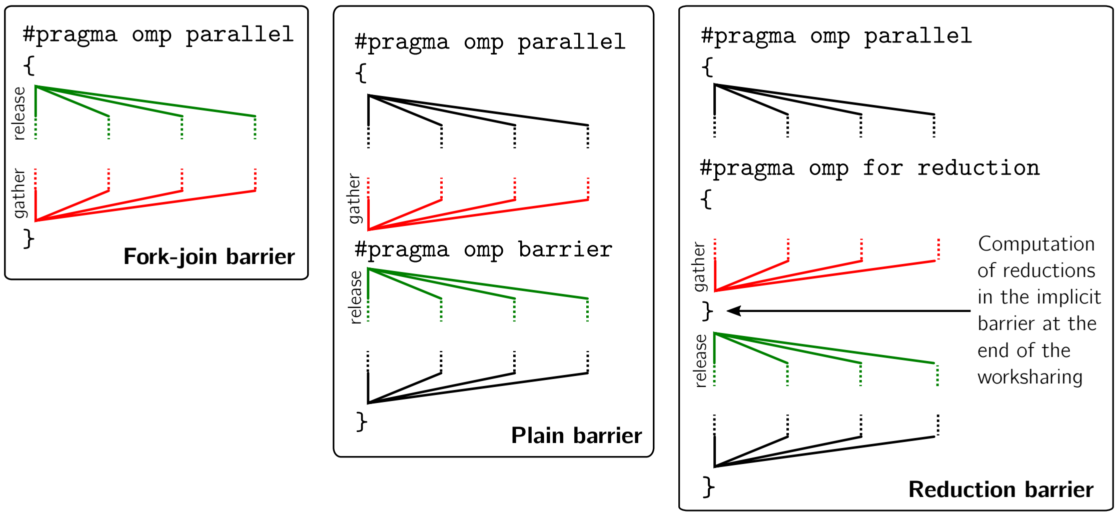


Figure 2.3: Gather (red) and release (green) phases of the fork-join, plain and reduction barriers

phases. The gather phase guarantees that all threads have reached the barrier and, therefore, all the partial reduction values are ready for the computation of the final reduction values. Then, reductions are computed and the corresponding reduction shared variables are updated before any thread leaves the barrier. Finally, the *release* phase allows threads to leave the barrier once the shared reduction variables contain the final reduction value.

Figure 2.3 shows the scheme of a fork-join, a plain and a reduction barrier with their respective gather and release phases. These gather and release phases are also used in a plain barrier. However, since a plain barrier does not require a split scheme, a single-phase barrier might be used instead.

Moreover, as a barrier describes an efficient communication pattern to distribute information among threads, the runtime exploits this fact to distribute information of the ICVs from the team master thread to all the other threads in the team. In this way, in the release phase of the fork-join barrier, these ICVs are propagated among threads of the barrier before they are released.

In addition to the kind of barrier, the Intel Runtime Library contains multiple barrier algorithms that can be used for every kind of barrier. At the time of writing this thesis, there are available the following four barrier algorithms:

**Linear:** The linear barrier algorithm is implemented as a degenerated tree, i.e., a tree structure where only one threads is the parent and the remaining threads are all in the second level of the tree as children.

**Tree:** the tree barrier algorithm is a traditional tree barrier with configurable arity. Threads are distributed statically across the tree. They have a single parent thread (except the root) and one or multiple child threads (except threads in the leafs of the tree).

**Hyper:** the hyper barrier algorithm is a tree barrier with an special tree structure. This structure describes a tree embedded in an hyper-cube. Jim Demmel describes this kind of tree structures in detail [49].

**Hierarchical:** the hierarchical barrier algorithm is a tree barrier that takes advantage of the memory hierarchy of the processor. In particular, this barrier has been optimized to perform a faster on-core synchronization step of those hardware threads running on the same core.

These four barrier algorithms are implemented following a split design with their respective gather and release phases. They perform the distribution of the ICVs in the release phase following their respective barrier structure.

### 2.4.3 Waiting Mechanism

The Intel OpenMP Runtime Library implements a wait/release system which is orthogonal to barriers and other components of the runtime. This mechanism consists of waiting on a provided flag until this flag has a predefined value. The runtime supports three kinds of flags: `flag32`, `flag64`, `flag_oncore`. The `flag32` and `flag64` are used to wait on flags of 32-bit and 64-bit integers, respectively. The `flag_oncore` is a `flag64` that implements special waiting and releasing mechanisms for on-core synchronization. This flag is used in the on-core synchronization phase of the hierarchical barrier.

During the waiting period, a thread spins on a loop until its flag has the predefined value that ends the wait. In the body of this loop, the waiting thread performs the duties allowed by the OpenMP standard. For example, the thread checks if there are ready tasks to be executed. Moreover, this loop contains the following two waiting mechanisms to reduce the overhead of pure busy waits:

**Yielding:** A thread yields its computational resources for a limited time to other potential threads. There are two kinds of yielding: clock cycles yielding and hardware context yielding.

**Suspension:** A thread is indefinitely suspended until it receives a signal that ends the suspension.

The clock cycles yielding is a light process that consists of executing an instruction that stops that thread from issuing instructions for a few number of cycles. In the Intel Xeon Phi coprocessor, this instruction is the *delay* instruction [40]. With this, the thread does not yield the hardware context but yields cycles to other potential threads running in other hardware contexts of the core (simultaneous multi-threading). The hardware context yielding is a more expensive process

```
switch(__kmpc_reduce(&loc1, gtid, 1, 4, &reduc_data, reduce_func, &lock))
{
    case 1: // Tree (barrier) reduction or critical reduction
        *shared_red_var += reduc_data.red_var;
        __kmpc_end_reduce(&loc1, gtid, &lock);
        break;

    case 2: // Atomic reduction
        __kmpc_atomic_float4_add(&loc10, gtid, shared_red_var,
            reduc_data.red_var);
        __kmpc_end_reduce(&loc1, gtid, &lock);
        break;

    default: // case 0: non-master non-atomic
}
```

Listing 2.3: Reduction code of a `float` variable generated for the Intel OpenMP Runtime Library

where the thread calls to the operating system to yield its hardware context. This process takes place immediately when a core is oversubscribed with more threads than hardware contexts available and when the thread has been spinning for a prefixed number of iterations in the loop. In addition, hardware context yielding only happens if the runtime library has been configured for throughput (`KMP_LIBRARY=throughput`) and not for latency (`KMP_LIBRARY=turnaround`).

In the suspension, the thread is put to sleep invoking an operating system call. This suspension happens after the thread is spinning in the loop a pre-established number of milliseconds. This time can be defined with the environment variable `KMP_BLOCKTIME`. If this variable is set to *infinite* the suspension of the thread is disabled. A sleeping thread will be awaken by another thread using the operating system unblock service.

### 2.4.4 Reductions

In Section 2.2.3, we explained that reductions in OpenMP are defined over variables that must be shared among all the threads involved in the reduction computation. In this context, the Intel OpenMP Runtime Library implements the following three kinds of reduction computation approaches:

**Tree:** The tree reduction approach performs the computation of reductions piggy-backed in the synchronization barrier, using the *reduction barrier* that we described in Section 2.2.3. The flavor of the tree barrier also determines the tree structure used in the computation of the barrier, i.e., *linear*, *tree*, *hyper* and *hierarchical*.

**Atomic:** In the atomic reduction approach each thread incorporates its partial re-

```
kmp_int32 __kmpc_reduce(
    ident_t *loc,          // Source location and flags
    kmp_int32 global_tid, // Global thread id
    kmp_int32 num_vars,    // # of reduction variables
    size_t reduce_size,    // Combined size in bytes of all the reduction vars
    void *reduce_data,     // Structure with the packed reduction vars
    void (*reduce_func)(void *lhs_data, // Function to reduce two
        void *rhs_data),                // 'reduce_data' structures
    kmp_critical_name *lck // Lock to compute reductions with critical
);                         // sections if enabled
```

Listing 2.4: Main function prototype of the reduction API

duction to the shared reduction variable using atomic instructions if the underlying architecture supports them. This approach requires a barrier after the atomic operations to guarantee that all threads have incorporated their partial reduction value to the shared variable after that point.

**Critical:** The critical reduction approach computes the reduction in the same way as the atomic approach but using a critical section instead of atomic instructions. Likewise, this approach also requires a synchronization barrier after the computation.

The atomic and critical approaches can work well for a small number of threads. However, the tree approach scales much better when the number of threads is large and the barrier algorithm is the *tree*, *hyper* or *hierarchical* barrier. In addition, computing reductions using the tree approach also keeps the same association order of the reduction operations across different executions. This results in the same approximation error when the reduction is on floating point variables. This error can vary between executions in the atomic and critical approaches because the association order of the operations depends on the order of arrival of the threads to the atomic instruction or the critical section, respectively.

Listing 2.3 shows an example of code generated to reduce a `float` variable with an addition operation for the Intel OpenMP Runtime Library. As depicted, this runtime uses the struct `reduce_data` to pack all the partial reduction variable of each thread. The `__kmpc_reduce` function determines the computation approach of the reductions and it starts the reduction process. Listing 2.4 describes the meaning of the parameters of this function.

When the tree reduction approach is used, all threads execute the function `__kmpc_reduce` shown in Listing 2.3. If that thread is the master of the team, it will execute the gather phase of a reduction barrier. All the other threads will execute a full reduction barrier (gather and release phases). In this barrier, threads compute reductions piggybacked with the barrier. For this approach, the `__kmpc_reduce` function returns 1 for the master thread and 0 for any the other threads. Therefore, when the master finishes the gather phase of the barrier, this thread has the final reduction value stored in its local reduction struct `reduc_data`. Then, the master thread will execute the case 1 of the switch statement shown in Listing 2.3. In this switch case, the master thread updates the shared reduction variable. The other threads are blocked in the release phase of the reduction barrier. In the `__kmpc_end_reduce`, the master thread executes the release phase of the reduction barrier that allows all the threads to leave the barrier.

When the atomic reduction approach is chosen, `__kmpc_reduce` does not execute any barrier phase for any thread. In this case, this function returns 2 for all the threads. This leads all the threads to execute the atomic operation in the case 2 of the switch in Listing 2.3. Then, a full barrier is executed by all the threads to guarantee that the final value of the reduction is available in the shared reduction variable after that point.

When the critical reduction is selected, all threads try to obtain access to the critical section in the `__kmpc_reduce` function. When the access to the critical section is granted for a particular thread, this thread leaves the function with a return value 1. Then, this thread executes the case 1 of the switch in Listing 2.3 to combine its partial reduction value with the shared reduction variable. Finally, in the `__kmpc_end_reduce` function, that thread releases the critical section and executes a barrier to wait for the completion of the reduction computation.

## 2.5 The Intel Xeon Phi Coprocessor

The first product of the Intel MIC architecture is the Intel Xeon Phi coprocessor code-named Knights Corner [38]. It features a large number of cores ranging from 50 to 61. Figure 2.4 shows an scheme of the main components of the architecture.

Every core in the chip is based on a simple in-order x86 architecture that runs approximately between 1.0 and 1.2 GHz. This is much lower frequency than the standard cores of the regular Intel Xeon processors. Each core has four hardware threads that are scheduled in a round-robin fashion and can issue up to two instructions per cycle from the same thread context. A relevant limitation in this aspect is that a core cannot issue instructions from the same thread context in back-to-back cycles [40]. This means that in order to take advantage of all the execution cycles of each single core we need at least two threads running on the same core.

The cache hierarchy is organized in two levels. The L1 cache is local to each core and has 32KB for instructions and 32KB for data. Therefore, this level of the cache is shared among all the hardware threads of the core. The L2 cache is distributed across the different cores in modules of 512KB for both data and instructions. The whole level is shared among all threads. However, accessing to the local module of the core is potentially faster than accessing to remote modules. In addition, despite the L2 cache is shared, data can be replicated in different modules of the L2 cache. This means that if a core requests a cache line from a remote L2 module, it will keep a copy of that cache line in its local L2 module.

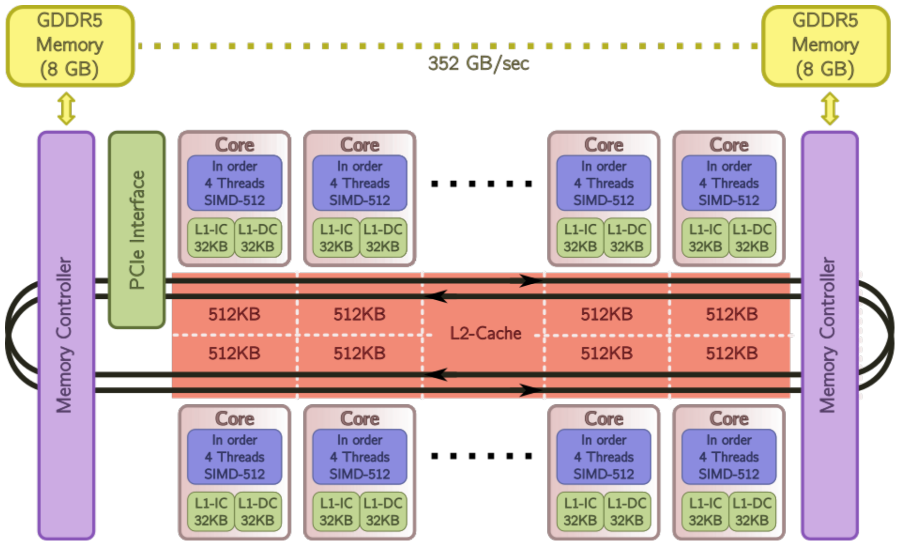


Figure 2.4: Intel Xeon Phi coprocessor diagram (Knights Corner). [1]

| | | | |
|---|---|---|---|
| Number of chips | 1 | VPU size | 512 bits |
| Cores / chip | 61 | Memory size | 16 GB |
| Hardware stepping | C0 | Memory bandwidth | 352 GB/s |
| Threads / core | 4 | ECC mode | Supported |
| Frequency | 1.238 GHz | Peak performance (DP) | 1.2 TFlops/s |
| L1 size / core | 32+32 KB | Power consumption | 300 W |
| L2 size / core | 512 KB | Software stack | Gold |

Table 2.1: Characteristics of the Intel Xeon Phi coprocessor 7120P

Both L1 and L2 caches are fully coherent. The coherence protocol is implemented by means of a distributed tag directory (DTD) which keeps the coherence information of each cache line. The tag information of each cache line is assigned to a DTD by means of a hash function. Dealing with the DTD can account for a large part of the overhead in a cache-hit memory access [127].

Cores and the L2 modules are connected through a double ring bus (one bus in each direction) as depicted in Figure 2.4. The ring also connects the memory controllers and the I/O interface that allows the coprocessor to communicate with the host through the PCIe bus. The memory controllers have up to 16 memory channels available to access the on-board GDDR memory.

Regarding the SIMD instruction set, the coprocessor comes with a specially designed Vector Processing Unit (VPU) that provides the architecture with 512-bit SIMD instructions. They are denominated Intel® Initial Many Core Instructions (Intel® IMCI). This SIMD instruction set supports gather/scatter memory instruc-

[1]Thanks to Mauricio Hanzich for this figure

tions, masked instructions for predicated execution, advanced shifts and permutation operations and fused multiply-and-add operations, among other features. As a result, the coprocessor can yield a sustained performance of 1.2 teraFLOPS in double precision on a 300W thermal design power (TDP) package.

The support of gather/scatter SIMD instructions is limited in Intel IMCI. In this instruction set, only gather/scatter operations that can be expressed with a single memory address as base for all vector lanes and a set of 32-bit integer offsets have direct support in hardware. In this thesis, we denote these gather/scatter operations as *simple* or *uniform-base*. Gather/scatter operations that cannot be expressed using a single memory address as base are denoted as *gather/scatter of pointers*.

Regarding Intel IMCI masked instructions, they have a particular feature. Most of them require an additional vector register argument that is used to set those vector lanes in the output disabled in the mask. We denote this argument *old value*. In this way, these masked instructions perform an implicit blend operations between the output of the instruction (for the vector lanes enabled) and the *old value* register (for the vector lanes disabled).

A particular feature of this SIMD instruction set exploited in this thesis is vector streaming stores. Vector streaming stores are useful for writing on non-temporal data, i.e., data that is not going to be read shortly. These vector stores perform the vector write without requesting the data of the involved cache line first (read for ownership). This allows saving memory bandwidth in case of a cache miss. In addition to regular vector streaming stores, the Intel Xeon Phi coprocessor also includes a special implementation denoted as non-globally ordered vector streaming stores. This special implementation may improve performance relaxing the memory consistency. This means that subsequent writes to a non-globally ordered streaming store can be observed before it.

The coprocessor supports a standard software stack with a Linux operating system and programming models such as OpenMP, OpenCL or MPI. In this sense, applications written in one of these paradigms are readily available to run on the Intel Xeon Phi coprocessor. Therefore, the optimization of these programming models for the Intel MIC Architecture is of great importance.

In this work, we use the Intel Xeon Phi coprocessor 7120P described in Table 2.1. In this thesis, we focus on the 61-core model 7120 that we used in our evaluation experiments with C0 silicon and ECC memory mode enabled.

# Chapter 3

# Source-to-source Vectorization in the Mercurium Compiler

## 3.1 Introduction

The largest software contribution of this thesis is the vectorization infrastructure that we have developed in the Mercurium C/C++ source-to-source compiler. This infrastructure is aimed at the vectorization of loops and whole functions and it has been progressively implemented throughout the years of development of this Ph.D. thesis. Therefore, it can be seen as the final product within the context of our research in compiler vectorization.

This chapter presents our user-directed vectorization infrastructure. We introduce an algorithm for for-loop and whole function vectorization based on a set of attributes that we will later use in our proposal on SIMD extensions for OpenMP, presented in Chapter 4 and Chapter 5. The following sections synthesize all the decisions made throughout all the process. They address from the design of the vector intermediate representation to the generation of SIMD intrinsics in a high-level back-end. We include a summary of the performance evaluation conducted in detail in Chapter 4, together with our user-directed vectorization proposal for OpenMP. Moreover, we present a use case that shows the potential usefulness of this source-to-source vectorization infrastructure as a programming tool.

It is important to note that this chapter does not describe a research contribution on new vectorization algorithms, but our four-year experience designing and implementing the vectorization infrastructure in the Mercurium compiler. The description about this vectorization infrastructure is not intended to offer low-level compiler-specific implementation details but just the opposite: a high-level overview. In this way, the chapter could be an introduction to vectorization concepts for those readers without a large expertise in the field.

## 3.2 Motivation

Nowadays, there are two compiler infrastructures that dominate the development of open-source compiler technology in industry and research. They are GCC (the GNU Compiler Collection) [68] and the LLVM Compiler Infrastructure [95, 94]. Both infrastructures cover all the compilation stages from the parsing of different programming languages to the assembly code generation for several target architectures. The maturity of both infrastructures is high. This fact allows many vendors to use them as production-level compilers and compilation infrastructures to develop other compiler products. Some examples are the Concept GCC [113] the Cilk Plus GCC [35], the Linaro GCC [60], the NVIDIA's compiler for CUDA [42], the Intel's SPMD Program compiler [122], the Intel's Clang-based C++ Compiler [17] the Intel's OpenCL compiler [36], and the Apple's compiler for Swift [9].

However, doing research on these compilation infrastructures can be hard and costly. The learning curve of these infrastructures is intrinsically high due to the complexity of all their components and their interactions. For example, a new contribution that required changes on several phases of the compiler could involve working with different intermediate representations and compilation technologies. This high learning and implementation effort has normally to be assumed by one or a few number of people which could turn it into a challenging or even unaffordable task.

In addition, evaluating new or upcoming architectures often depends on the will of vendors to release a back-end for that particular compilation infrastructure. An efficient implementation for a specific architecture requires a large amount of work and low-level knowledge of the architecture. This task is not affordable for many researchers that look forward to working with the latest cutting-edge architectures. For instance, this is the case of the first version of the Intel Xeon Phi coprocessor (Knights Corner). Currently, neither GCC nor LLVM have back-ends with full support for this architecture (GCC supports basic *system* compilation) which limits compiler research work on these devices. The only compiler infrastructures in production state with full support for Knights Corner are those compilers released by Intel, whose source code is proprietary.

From a more application-level viewpoint and related with vectorization and other compiler optimizations, it is very common that the most optimized version generated by the compiler can be further tuned by advanced programmers. When the compiler is not able to apply a particular optimization, advanced programmers could easily optimize the code by hand if they had the source code of a high-level optimized version generated by the compiler [135].

Moreover, even though some compilation infrastructures are able to generate source code as output, this output is a mere dump of its low-level intermediate representation. The resulting code is difficult to read and far from the original version of the code. Consequently, in some cases, advanced programmers are compelled to directly write the code in assembly or modify the assembly code generated by the compiler to achieve their goals.

In addition, the set of optimizations applied on a particular code usually vary among compilers. This fact creates an undesirable dependency for the programmer (not for the compiler vendor) between the best performance achieved in an application and the compiler used to compile that version of the code.

## 3.3 Objectives

All the facts presented in Section 3.2 motivate us to create a source-to-source compiler vectorization infrastructure targeting loops and whole functions and aimed at the fast prototyping of new compiler vectorization proposals. Furthermore, the source-to-source nature of this tool will allow programmers to have access to a high-level vectorized version of their applications. These vector versions are easier to modify, optimize and debug than those versions compiled with full compilers.

The main objective of this contribution is to build an infrastructure for the vectorization of loops and whole functions. This infrastructure must have the following characteristics:

**Manageable:** suitable for fast prototyping with low manpower resources.

**High-level development:** support for multiple SIMD architectures at high-level. Low-level details will be addressed by the back-end compiler.

**Editable code:** high-level and easily editable output vector code that allows advanced programmers to further tune the code.

In the context of this thesis, we target the Intel Xeon Phi coprocessor and its Intel Many-Core Initial SIMD instruction set (IMCI) to develop a first version of our vectorization infrastructure.

## 3.4 Vectorization Infrastructure

In this section, we provide a high-level description of our vectorization infrastructure developed in the Mercurium C/C++ source-to-source compiler. We introduce the different phases of this infrastructure and the extensions made to the type system and the intermediate representation (IR) of the compiler to add support for vector operations.

### 3.4.1 Vector data types and SIMD instructions

In general terms, vectorizing a loop consists of replacing multiple instances of the same scalar operation across loop iterations by an equivalent vector operation that performs the same scalar operations at once [5, 81, 155]. The resulting vector code must produce the same output as the original scalar counterpart, except for precision errors. In this way, operations between scalar operands often have equivalent operations using vector operands. This suggests that it is necessary to extend the type system of the compiler to introduce support for vector types.

Due to the close relationship between a scalar and a vector operation, a vector type is parameterized with a scalar type and the number of scalar elements of that type that there are within the vector type. Therefore, our vector type is of the form *vector of X of Y elements*. For instance, vector of `int` of 4 elements, vector of `float` of 16 elements and vector of `double` of 8 elements. Eventually, the compiler will generate code for a specific SIMD architecture with fixed length vector registers from these instructions with vector types. In this process, vector operations and their scalar number of elements will be adapted to fulfill the physical vector length and other constraints of the target architecture.

### 3.4.2 Representing Vector Operations

We considered two approaches in order to represent vector operations in the intermediate representation of Mercurium, described in Section 2.3.1. The first approach consists of representing vector operations mainly using the type system and adding the minimum number of new nodes to the IR of the compiler. The second approach entails defining new nodes in the IR to represent vector operations, in addition to use the type system as in the first approach.

In the first approach, the type system is extended with vector types, as described in Section 3.4.1. Furthermore, scalar nodes already in the original IR are allowed to operate on vector type operands. This means that the IR will be extended with new nodes only when there is no scalar node that can be overloaded using vector operands (for instance, an operation along the scalar elements of a vector that computes the maximum element). On the bright side, this alternative does not increase the number of nodes in the IR. In addition, it simplifies compiler phases that target nodes depending on their kind of operation but not on their scalar or vector type. On the downside, this approach overloads the scalar semantic of the original nodes with an additional vector semantic. Therefore, phases that require differentiating between scalar and vector types will need to check the type of the nodes to discern between both data kinds. This fact can make compiler programming a bit tedious because it leads to repetitive checks of the type of each parameter of each node all around the phase.

The second approach is the one used in our vectorization infrastructure. Benefits and drawbacks of this alternative are opposite to those from the first option. In this approach, the semantic of scalar and vector operations are separated in different nodes. This makes easier the programming of compiler phases that target a particular kind of nodes (scalar or vector). However, increasing the number of nodes in the IR could make less programming efficient more generic phases that do not need from this distinction. This number of nodes could be a problem in full compiler infrastructures, such as LLVM [94] and GCC [68], where the IR is expected to be extended to support many other compiler technologies and architectures. In this case, having specific nodes to represent vector operations would increase the total number of nodes even more. For this reason, these large infrastructures opted by approaches similar to the first one. However, the Mercurium source-to-source compiler does not have the same goal as a production infrastructure. We expect a modest progression in comparison to LLVM or GCC. For these reasons, we chose this second alternative in Mercurium to benefit from an easy distinction between scalar and vector operations.

### Modeling Predicated Vector Code

We modeled the support of complex predicated vector code in Mercurium with the concept of vector mask featured in SIMD instruction sets such as the IMCI or AVX-512 [74]. This approach introduces a vector mask that describes which vector lanes are enabled and disabled in the execution of a particular vector instruction [119, 16, 122, 80]. In the Mercurium compiler, we introduced a new mask data type that is exclusively used to represent this vector mask. This new data type is basically a special integer data type where each bit represents the state of each vector lane. If a bit is set to 1, it means that the lane is enabled in the execution. Otherwise, if the bit is set to 0, it means that the lane is disabled. The size of this mask data type is parameterized depending on the number of vector lanes that needs to be represented.

The number of operations allowed on the mask data type is very limited. This operations mainly comprise logical operations such as *or*, *and*, *nor* and *xor*, among others. For this reason, we defined a reduced number of nodes for specific operations on mask data types. This allows having an automatic control on the operations allowed on the mask data type and easing the implementation of compiler phases that only target operations with this kind of nodes.

### Vector Intermediate Representation

Listing 3.1 shows an example of the vector nodes that we use to extend the original IR of the Mercurium compiler, introduced in Section 2.3.1. We defined a new vector version of nodes for arithmetic operations (`NODECL_VECTOR_ADD`), comparisons (`NODECL_VECTOR_LOWER_THAN`), logical operations, bit-wise operations and other scalar operations that have a direct vector equivalence. We also introduced vector nodes for specific vector operations that do not have an equivalence in the original scalar IR. These new nodes comprise stride-one vector memory loads (`NODECL_VECTOR_LOAD`) and stores, vector memory gathers (`NODECL_VECTOR_GATHER`) and scatters, scalar-to-vector promotions (`NODECL_VECTOR_PROMOTION`) and vector-to-scalar reduction operations (`NODECL_VECTOR_REDUCTION_ADD`), among others.

In addition, as stated in the previous section *Modeling Predicated Vector Code*, we use vector masks to model predicated vectorization. Consequently, we also de-

```
expression:  NODECL_VECTOR_ADD([lhs]expression, [rhs]expression,
                               [mask]expression) type const-val
           | NODECL_VECTOR_LOWER_THAN([lhs]expression, [rhs]expression,
                                      [mask]expression-opt) type const-val
           | NODECL_VECTOR_PROMOTION([rhs]expression, [mask]expression)
                                     type const-val
           | NODECL_VECTOR_LOAD([rhs]expression, [mask]expression,
                                [flags]vector-flags) type const-val
           | NODECL_VECTOR_GATHER([base]expression, [strides]expression,
                                  [mask]expression-opt) type const-val
           | NODECL_VECTOR_REDUCTION_ADD([src]expression,
                                         [mask]expression) type const-val
           | NODECL_VECTOR_MASK_XOR([lhs] expression, [rhs]expression)
                                    type const-val
```

Listing 3.1: Example of the new vector nodes in the intermediate representation of Mercurium. The whole grammar can be found in the source file `cxx-nodecl.def` of the compiler [124]

fined a few nodes for those operations allowed on this special data type. For example, the node `NODECL_VECTOR_MASK_XOR` shown in Listing 3.1 describes an `xor` operation between two vector mask operands.

The use of vector masks requires adding an additional parameter in all vector nodes of our IR to describe which vector lanes should execute the vector operation. This parameter is the `mask` parameter shown in nodes at Listing 3.1. If we had decided not to add new nodes for specific vector operations, the additional `mask` parameter would have overloaded even more the semantic of the nodes.

### 3.4.3 Phases

Our vectorization infrastructure has two main phases: the Vectorizer and the Vector Lowering. The main objective of the Vectorizer is to transform the input scalar IR into a relatively generic vector IR. Then, at some later point in the compiler pipeline, the Vector Lowering phase generates architecture-specific SIMD intrinsics that will be the output vector code generated by our vectorization infrastructure.

Figure 3.1 shows a simplified diagram of the Mercurium C/C++ source-to-source compiler that includes the new two phases of the vectorization infrastructure. As we can see, the Vectorizer phase is placed after the OpenMP front-end phase. Some of the OpenMP information processed in this former phase can be necessary for the correct vectorization of the code, such as the OpenMP worksharing attributes. In addition, the Vectorizer phase utilizes a set of basic data flow analysis techniques available in Mercurium. These compiler analyses are mainly use-definition, liveness, reaching definition and induction variable analyses. All of these compiler analyses are implemented over a parallel control flow graph (PCFG), as we described in Section 2.3.

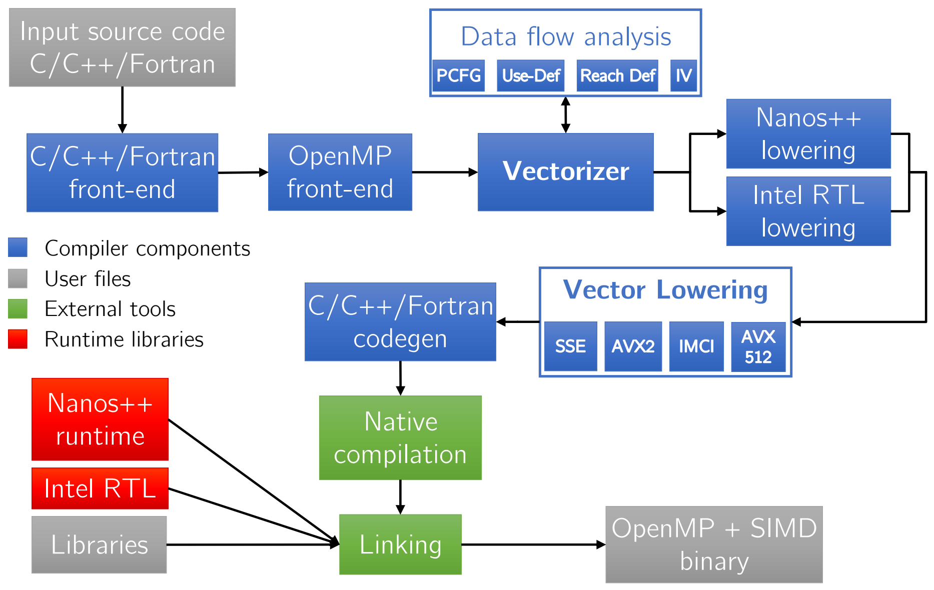


Figure 3.1: Mercurium compilation diagram including new phases of the vectorization infrastructure (original diagram is shown in Figure 2.1, Section 2.3)

### Vectorizer

The Vectorizer phase performs the code transformations necessary to generate an equivalent vector version of the input scalar code. Figure 3.2 shows the steps that comprise this phase.

Firstly, the input scalar IR is preprocessed to leave the code in the state expected by the following steps of the vectorization process. This code preprocessing includes the canonicalization of expressions [105], the simplification of compound assignment operations and the normalization of loops [81]. We do not perform further modifications of the input IR with the aim of preserving the original structure of the code as much as possible.

Secondly, in the Function Versioning Registration step, the compiler traverses the code to register all the vector versions of the functions that are going to be emitted/vectorized in the following steps of the Vectorizer phase. This registration requires vectorizing as many prototypes of each function as full vector versions of that function will be generated. Further details of the vectorizer algorithm are described in Section 3.5. This early registration is necessary to have all the information of the vector functions available during the vectorization process of the code. At that point, if the compiler finds a function call, it will be able to know whether there will exist the appropriate vector function version in the final code. Further details of the infrastructure that holds these vector function versions are described in Section 3.5.8.

The Code Vectorization is the third step of the Vectorizer phase. In this step, the

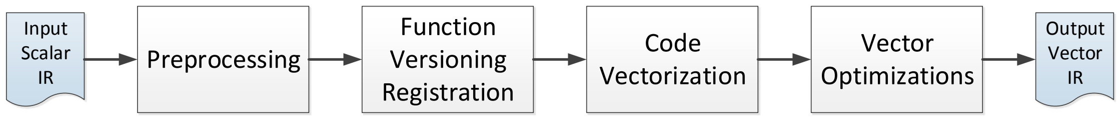


Figure 3.2: Internal scheme of the Vectorizer phase

scalar IR is transformed into an equivalent code expressed using vector operations represented by nodes from our vector IR (see Section 3.4.2). This includes generating vector versions of functions and loops chosen to be vectorized. The vectorization algorithm used is introduced in Section 3.5.

Finally, once the vector IR has been generated, the Vector Optimizations step applies compiler optimizations on the vector code that improve the vector IR from the previous step. These vector optimizations comprise strength reduction optimizations [105] on vector operations, optimizations on the base and offset components of gather/scatter memory instructions, software prefetching on vector memory operations (see Section 3.5.10), and other vector optimizations such as the one proposed in Chapter 5. This step is the only optional step in the Vectorizer phase.

**Vector Lowering**

The Vector Lowering phase translates the generic vector IR to architecture-specific intrinsics of the target SIMD instruction set. Figure 3.3 shows the steps that comprise this phase. As we can see, some of the steps of this phase are specific to the SIMD instruction set. Currently, we have support for several Intel instruction sets: Streaming SIMD Extensions 4.1 (SSE), Advanced Vector Extensions version 2 and 512 (AVX2 and AVX-512, respectively) and Initial Many-Core Instructions (IMCI). In this thesis, we focus on the IMCI instruction set as it is the SIMD instruction set supported by the Intel Xeon Phi coprocessor (Knights Corner).

The two main steps of the Vector Lowering phase are the following:

**Legalization:** The legalization step introduces hardware-specific restrictions into the generic vector IR obtained from the Vectorizer phase.

**Back-end:** The back-end step generates hardware-specific SIMD intrinsics that can run efficiently on the target architecture.

As depicted in Figure 3.3, there is also an additional step between the steps of legalization and back-end, denoted as Three-address Vector Code Transformation. This step decomposes nested/complex vector operations into three addresses form. Without this step, a small statement of the input code with a few scalar operations could turn into a multi-line statement with a set of SIMD intrinsics nested one inside another. This three-address transformation is aimed at making the output vector code more readable and more suitable for the potential manipulation of the programmer. It is important to note that this three-address transformation only

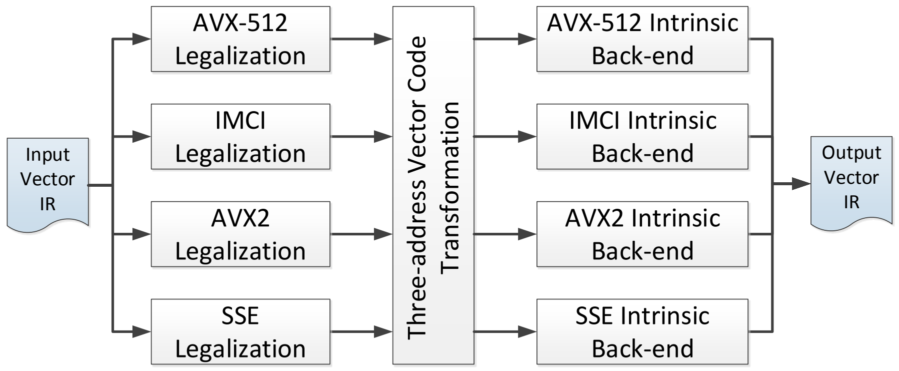


Figure 3.3: Internal scheme of the Vector Lowering phase

affects to vector operations. Scalar operations will not be decomposed in three addresses, preserving their original structure.

The idea of using a legalization step is inspired in the LLVM Target-Independent Code Generator [145]. In this step, the Mercurium compiler transforms the vector IR to make it more suitable and compatible with the target SIMD instruction set. For example, some of the specific transformations that happen for the IMCI instruction set are the following:

- Vector and mask instructions are adapted to the specific characteristic of the IMCI architecture, such as the supported vector length, and vector and mask data types.
- Masked unaligned stride-one vector load and store instructions are replaced by gather/scatter instructions, respectively, because the instruction set does not have direct support for them.
- Unnecessary vector conversion operations are removed.
- Information necessary to compute the *old value* parameter of IMCI masked vector instructions is added (lanes disabled in the mask parameter will be overwritten using this parameter. See Section 2.5).

The back-end step receives the code from the legalization step simplified enough to perform almost a direct translation between the vector IR and a single or a small set of SIMD intrinsics of the target architecture. It is also possible that multiple nodes of the vector IR are mapped to the same SIMD intrinsic but using different arguments.

These intrinsics do not have a special representation in the Mercurium compiler. Therefore, they are invoked as regular function calls. Depending on the vector IR node, its associated type, its number of parameters and the type of these parameters, the back-end of the specific architecture chooses the most appropriate intrinsic or set of intrinsics to perform those operations efficiently.

Particularly, the back-end for the IMCI instruction set also manages the generation of unmasked or masked SIMD instructions depending on whether the `mask` parameter of the vector node contains a mask value that disables some vector lanes. If a masked SIMD instruction is needed, the back-end is able to determine with the information generated in the legalization step whether the values in the disabled lanes must be preserved or not. If they have to be preserved, the back-end will generate instructions where these values are used to write on the disabled lanes (*old values*). Otherwise, the back-end will generate *undefined* values for the disabled lanes so that the native compiler can overwrite these lanes with any information.

## 3.5 Vectorization Algorithm

The vectorization algorithm used in our infrastructure follows the main idea of the traditional strip-mining/unroll-and-jam loop vectorization approach [81, 16]. This approach materializes the unrolling of the loop and the jamming of the unrolled instructions using separate compilation steps. However, our vectorization algorithm is more direct than the previous approach. The core of our algorithm performs an in-place vector code transformation for inner loops and outer loops without distinguishing between these two steps. This direct approach is also followed in several compilers, such as GCC [69], and research work. For example, Nuzman et al. [111, 108] used a similar approach for outer-loop vectorization.

Moreover, common compiler vectorization algorithms follow a multi-step vectorization approach where the scalar code is initially vectorized in a naive way. This vector code is then refined/optimized in later steps of the vectorization process. Instead, our vectorization algorithm generates vector IR with certain level of optimization from the beginning. This approach slightly increases the complexity of the core vectorization algorithm but it avoids additional steps to change or revert the initial vector decisions made for some components of the scalar IR.

Our vectorization algorithm is able to vectorize two kinds of code structures: loops and functions. These structures, respectively, are the vectorization scope during the vector code transformation. This transformation is guided by the attributes defined in Section 3.5.3. The definition of these attributes is more related with properties that happen in loops. Therefore, when vectorizing a function, the code is vectorized as if it was nested in a loop, i.e., assuming that the function is invoked from a loop that is already vectorized. In this way, the attributes used for the vectorization of the function are understood as in the context of such a caller loop. This kind of function vectorization is known as whole function vectorization [80].

In addition, the vectorization process is performed using a vectorization factor (VF). The VF establishes the number of scalar operations that will compose each equivalent vector operation in the vector IR. It can also be understood as the number of scalar iterations of the loop processed by each resulting vector iteration.

The core of the vectorization algorithm has two steps that traverse the target IR with the following goals:

```
for (i=0; i<1002; i++)
    a[i] = sinf(a[i]);
```

(a) Scalar loop to be vectorized

```
// Scalar prologue loop
for (i=0; !is_aligned(&a[i], 16); i++)
    a[i] = sinf(a[i]);

// Main vector loop
for (; i<999; i+=4)
    aligned_vstore4(&a[i],
        vsinf4(aligned_vload4(&a[i])));

// Scalar epilogue loop
for (; i<1002; i++)
    a[i] = sinf(a[i]);
```

(b) Loops resulting from the vectorization of (a)

Figure 3.4: Example of prologue, main vector and epilogue loops (b) resulting from the vectorization of a simple loop (a), assuming 16-byte vector units and VF = 4

**Vectorization of Local Variables:** vectorizing variables that are local to the vectorization scope, i.e., the target loop or function code.

**Vectorization of Code:** vectorizing statement and expression nodes of the abstract syntax tree (AST).

After the second step, the code is in a consistent vector IR. The following sections describe in more detail several aspects of the vectorization algorithm.

## 3.5.1 Vectorization of Loops

Vectorization of loops is the core of our algorithm. Currently, our algorithm is able to vectorize for-loops that are in a *canonical loop form*, as defined in OpenMP [117]. Roughly, a loop in this canonical form allows computing its iteration count before executing it, i.e., it is a countable loop. In addition, our algorithm is also able to vectorize simple while-loops that satisfy the restrictions imposed by the canonical loop form.

### Structure of a Vectorized Loop

The vectorization of a single scalar loop may result in one or more loops that, in conjunction, perform the computation equivalent to the scalar loop [16, 84]. These resulting loops can be of the following kinds:

**Prologue loop:** This kind of loop peels iterations from the beginning of the main vector loop. It is normally used to peel some scalar iterations to align vector memory accesses of the main vector loop [157, 16, 16, 93]. Its number of iterations is smaller than the VF.

**Main vector loop:** This is the main loop of the vector transformation. The resulting vector iterations try to exploit as much as possible all the computational resources available.

**Epilogue loop:** This loop computes the remaining iterations of the scalar loop not computed in the main vector loop. Its number of iterations is smaller than the VF.

The prologue and the epilogue loops might be scalar or vector, depending on the SIMD features of the target architecture. Furthermore, the compiler can generate one or multiple scalar and/or vector versions of each one of these loops with different characteristics, as an optimization for the same main vector loop.

Figure 3.4 shows an example of the loops resulting from the vectorization of a simple loop. Figure 3.4a shows the original scalar loop. As depicted in Figure 3.4b, the *first loop* generated is a prologue loop in this case. This loop is aimed at aligning `a[i]` accesses in the main vector loop (the second one). The prologue loop will execute scalar iterations whereas the address `&a[i]` is not aligned to the vector length boundary (16 bytes in this example). The main vector loop computes vector iterations using VF equal to 4. The remaining iterations of the original scalar loop that have not been computed in the main vector loop are then computed in the epilogue loop (third loop) in a scalar fashion.

Depending on the code of the scalar loop, the compiler can generate several combination of the previous loops as a result of the vectorization. For example, if the number of iterations of the scalar loop is known at compile time and it is smaller than VF, the resulting code could contain only an epilogue loop because the main vector loop will never be executed. If the number of iterations of the scalar loop is known at compiler time and it is multiple of the VF, the compiler could prevent the generation of epilogue loops as it will never be executed. Currently, the generation of prologue loops is not implemented in our algorithm.

### Loop Vectorization Steps

The vectorization of a scalar loop is conducted following three steps: the analysis of the loop, the vectorization of the loop header and the vectorization of the loop body.

In the analysis of the loop, the algorithm determines if the future main vector loop will need prologue or epilogue loops. In addition, the algorithm analyzes the data types of the operations in the loop body to determine a suitable VF to be used for the vectorization. Currently, our algorithm computes VF as the number of scalar elements of the largest data type found that fits into the vector register of the target architecture. In other words, the VF chosen is equal to the vector length (VL) for the largest data type found in the code.

With the information obtained in the loop analysis step, the algorithm continues with the vectorization of the header of the loop. At this point, the lower bound, the upper bound and the step of the induction variable of the new main vector loop

```
function VEC_LB
    if first_loop then
        new_lb = lb
    else
        new_lb = NULL
    end if
    return new_lb
end function
```

(a) Lower bound

```
function VEC_UB
    new_ub = ub−(VF−1)×st
    return new_ub
end function
```

(b) Upper bound

```
function VEC_ST
    new_st = VF×st
    return new_st
end function
```

(c) Step

Figure 3.5: Some of the rules to compute the new lower bound (a), upper bound (b) and step (c) of the induction variable of the main vector loop. `lb`, `ub`, `st` and `VF` are the lower bound, upper bound and step of the scalar loop, and the vectorization factor, respectively

are computed according to the rules described in Figure 3.5. The lower bound is processed as shown in Figure 3.5a. If the loop being vectorized is the *first loop* of the resulting set of vector loops, the original lower bound is maintained in the vector loop. Otherwise, the lower bound of that loop is removed. Its value will be the value of the induction variable at the end of the execution of the previous loop, as happens for the main vector loop in Figure 3.4b. The upper bound and the step of the main vector loop are computed as shown in Figure 3.5b and Figure 3.5c, respectively, as a function of the vectorization factor, the upper bound and the step of the original scalar loop.

Finally, the vectorization of the code within the body of the loop is vectorized using VF, as we describe in Section 3.5.4, Section 3.5.5, and Section 3.5.6.

### 3.5.2 Vectorization of Functions

A function code is vectorized as if the function was invoked from inside a vector loop [80, 79]. This means that the vectorizer will interpret the parameters of the function as if they evolved throughout the iterations of the loop.

The compiler is able to manage multiple vector versions of the same function using the function versioning infrastructure described in Section 3.5.8.

As described in Section 3.4.3, the prototype of the function, i.e., parameters and return type, is vectorized in the phase Function Versioning Registration. This vectorization is carried out according to the vectorization of local symbols described in Section 3.5.4.

Afterwards, the vectorization of the body of the function is performed in the phase Vectorization of Statements and Expressions, following the full main vectorization algorithm described in Section 3.5.5, and Section 3.5.6.

### 3.5.3 Vector Attributes

Our vectorization algorithm is based on a set of attributes that determine the kind of transformation that each particular scalar node needs in the vectorization process [80]. These attributes are the following:

**aligned:** This attribute applies to scalar pointer arithmetic expressions. It states that the address value is aligned to the boundary of the vector length (in bytes) of the architecture.

**suitable:** This attribute applies to scalar integer expressions. It states that the value of the expression at runtime will be multiple of the vector length (in elements) of the architecture.

**linear:** This attribute is applicable to scalar integer expressions. It states that the value of the expression will have a linear evolution with a particular step across the vector lanes of a vector register.

**uniform:** This attribute is applicable to scalar integer expressions. It states that the value of the expression will be the same for all the vector lanes of a vector register.

**adjacent:** This attribute is applicable to scalar memory operations. It states that the memory operation reads/writes scalar elements that are all adjacent in memory (stride-one) or at least they are adjacent in chunks of vector length elements.

It is important to note the differences between the *aligned* and the *suitable* attributes beyond their applicability to different data types (pointer and integer expressions, respectively). Whereas the value of a pointer is aligned to the vector length in bytes of the SIMD architecture, an integer variable can be suitable or not depending on the vector length but in number of elements. This means that an integer variable can be suitable for the `double` data type but not suitable for the `float` data type because the number of elements that fits into a vector register is different for both data types. For example, for a SIMD architecture with 512-bit vector registers, integer values multiple of 8 will be suitable for `double` but not for `float`.

Regarding the `adjacent` attribute for scalar memory accesses, it is equivalent to say that the access is *linear* with a linear step of 1. However, a linear memory access is the result of the specific set of linear and uniform expressions that comprise the access. Thus, we define the *adjacent* attribute for this particular purpose.

All these attributes can be determined automatically by the compiler using static analysis techniques, as those described in the Figure 2.3, or using annotations on the code, as we describe in Chapter 4.

```
function VECTORIZE_SCALAR_TYPE(type, VF)
    switch type do
        case integral or floating
            vec_type = vector of VF elements of type
        case boolean
            vec_type = mask of VF bits
    return vec_type
end function
function VECTORIZE_VARIABLE_TYPE(var, VF)
    if IS_NOT_UNIFORM(var) and IS_NOT_LINEAR(var) then
        var.type = VECTORIZE_SCALAR_TYPE(var.type, VF)
    end if
    return vec_type
end function
```

Figure 3.6: Rules for the vectorization of basic data types. Other data types have limited support or are not supported in our implementation

### 3.5.4 Vectorization of Local Variables

Variables declared within the scope of the target for-loop or function code may have to be transformed into vector form. In the simplest case, e.g., variables with a basic data type, this transformation consists of promoting the scalar type of the variable to its vector counterpart of VF elements. Figure 3.6 shows a simplified example of the logic used to vectorize a variable with a basic scalar type. As function VECTORIZE_SCALAR_TYPE depicts, integral and floating types are promoted to its equivalent integral and floating vector types with length VF. If the variable has boolean type, its type is promoted to vector mask type with length VF. Function VECTORIZE_VARIABLE_TYPE shows how the vectorization of the variable type is only necessary if the variable is neither *uniform* nor *linear*. The type of uniform variables is kept scalar because these variables will have the same value within a vector register of the target vector scope. The type of linear variables is also kept scalar because these variables need a special treatment. The vectorization of uniform and linear variables is address in Section 3.5.6.

Vectorizing local variables with non-basic types, such as array subscripts or classes, entails changing the original layout of data allocated in memory by means of the generation of a more complex vector data type. For example, as for local declarations of C/C++ structs/classes, our algorithm is able to vectorize simple ones that contain only fields of basic types. The procedure is to create a new class type with internal fields of vector type. The new class type is a duplicate of the original one where the types of the fields have been vectorized. This new class type is then used instead of the original class type for local declarations of the class.

### 3.5.5 Vectorization of Statements

The vectorization of statements mainly affects the generation of vector masks that turns simple vectorization into predicated vectorization (see Section 3.4.2). Vector masks are synthesized attributes generated by statements that may change the control flow of the execution of each vector lane within the same vector register. Some of these nodes are nested loops, if-then-else statements, continue and break statements, goto statements and return statements. Consequently, this vector mask attribute is used in the vectorization of expressions nested in this kind of statements (see Section 3.5.6). The generation of masks is inspired in the work of Shin Jaewook [140], later extended by Karrenberg and Hack [80, 79].

Currently, our algorithm only supports the generation of predicated execution triggered by for-loop, while-loop, if-then-else, break and return statements. The algorithm uses a stack to store the vector mask used for the vectorization of each basic block in the code [50]. This stack is initialized with a mask that has all the vector lanes enabled. Therefore, when one of the previous statements may produce a divergence within the vector lanes of a vector register, the top of the stack is updated with a new mask. This mask is the result of merging the current mask in the top of the stack and the new mask generated by the statement that causes the divergence. This new mask in the top of the stack is then used to vectorize the statements and expressions nested in the statement causing the divergence. If there are multiple independent branches of execution, such as in an if-then-else statement, the top of the stack is updated with the corresponding mask for each particular branch. Once all the branches have been executed, their associated masks are popped from the top of the stack, recovering the previous mask before the branching statement. It is possible that some masks affect the execution of code outside of their initial scope. For example, this is the case of break or return statements as they disable those vector lanes that reach them for the remaining execution of the loop body and the function body, respectively. In these cases, the mask of the previous scope is merged with the mask generated by these statements to propagate their effect.

#### Nested Loop Statement

In outer loop vectorization scenarios [111], variables that are part of the loop control of inner loops may not be uniform in the context of the iterations of the outer loop. If this happens, each vector lane could have different values for the induction variable of the inner loop and different stop conditions. This requires that the code of the control of the inner loop has to be specially vectorized, including the induction variable and the condition of that loop. Vectorizing the condition of the loop results in a vector mask that enables and disables those vector lanes that have not already reached their stop condition and those which have already done it, respectively. This mask will be used for the vectorization of the statements and expressions of the loop body of that loop.

Prologue and epilogue loops that are vectorized follow a similar approach.

### If-then-else Statement

If-then-else statements may need from predicated vectorization when the condition of the statement is not stated as uniform, i.e., vector lanes may have different result values in the evaluation of the condition. For these cases, the vectorization algorithm will generate two vector masks: one for the execution of the `if` branch and another one for the `else` branch if this branch exists. The mask of the if branch will have enabled those vector lanes that evaluate the if-then-else statement condition to true. The mask of the else branch will have the complementary value of the mask used for the execution of the if branch. If the mask that arrives to the if-then-else statement had vector lanes disabled, the lanes would be also disabled in both masks of the if-then-else branches.

### Break Statement

Vectorization of loops that contain break statements is only supported when the break statement does not introduce a control flow dependence across vector lanes but the it affects the control flow of each vector lane independently. The latter scenario normally happens in outer loop vectorization where the break statement is within the inner loop. Listing A.8 in Appendix A can illustrate both scenarios. Vectorizing the inner loop (line 5) would be an example of the first scenario not supported by our infrastructure. Vectorizing the outer loop (line 33) would be an example of the second scenario that can be vectorized with our tool.

Focusing on the supported case, the vectorization algorithm generates a vector mask that disables all the vector lanes that execute that break statement. This mask is used to execute the vector instructions that precede the break statement and to update successor masks.

### Return Statement

Return statements are exit points of a function code. Function codes may have multiple return statements. These exit points mean the end of the execution for those vector lanes that reach them.

The vectorization algorithm generates a mask for `return` statements that are guarded by non-uniform conditions. These masks are aimed at disabling vector lanes that reach these return statements. This mask is propagated throughout the subsequent code, even outside of the scope of the return statement, as those vector lanes will not execute any other vector instructions in that function.

For those functions that return a vector return value, the vectorization algorithm declares an auxiliary vector value to store the returning value of those vector lanes that are disabled by a return statement at any point of the function code. At the end of the function code, this auxiliary vector value is returned with the corresponding return value of each vector lane.

```
1: function VECTORIZE_ASSIGNMENT(assig)
2:   if IS_NOT_UNIFORM(assig.lhs) or IS_NOT_UNIFORM(assig.rhs) then
3:     if IS_LINEAR(assig.lhs) then
4:       return VECTORIZE_LINEAR_UPDATE(assig.lhs)
5:     else if IS_ARRAY(assig.lhs) then
6:       return VECTORIZE_MEMORY_WRITE(assig.lhs)
7:     else if IS_VARIABLE_REFERENCE(assig.lhs) then
8:       return VECTORIZE_VEC_REG_ASSIGNMENT(assig.lhs)
9:     end if
10:  else
11:    return assig                                        ▷ Kept Scalar
12:  end if
13: end function
```

Figure 3.7: Some of the rules for the vectorization of an *Assignment* node

### 3.5.6 Vectorization of Expressions

The vectorization of expressions address the vectorization of all kind of expressions that need to be vectorized, such as variable and memory accesses, arithmetic, logical, bit-wise and comparison operations, constant literals, data type conversions and function calls.

One of the most relevant expression nodes in this step is the vectorization of the *Assignment* node. This node represents an expression assignment, such as `a[i] = b[i] + c`. The left-hand side (`lhs`) and the right-hand side (`rhs`) of the assignment must be vectorized differently. The `lhs` denotes a memory store, i.e., it is an expression which is evaluated by its reference. The `rhs` performs an arithmetic computation, i.e., it is an expression which is evaluated by its value. Figure 3.7 summarizes some of the rules implemented for this node. As we can see, if both the `lhs` and the `rhs` of the *Assignment* node are *uniform*, the node is kept scalar because no one of its children needs vectorization. If one or both of its children are not *uniform*, specific vector transformations are applied depending on the characteristics of the `lhs` child. These transformations are the following:

**vectorize_linear_update (line 4):** The *Assignment* node is vectorized in the context of a *linear* variable update. The `lhs` child does not need vectorization in this context. Basic *linear* nodes in the `rhs` are increased with their linear offset. Other nodes in the `rhs` child are not vectorized. For example, the assignment expression `i = i + 1` is transformed into `i = i + 1 * 4` (assuming VF = 4).

**vectorize_memory_write (line 6):** The *Assignment* node is vectorized using the appropriate vector memory write operation: stride-one vector store or vector scatter. The `lhs` child is used as memory destination and the `rhs` child as the source value for this memory store. The `rhs` child and internal nodes of the `lhs` child, if required, are vectorized as a value expression following rules in Figure 3.8.

**vectorize_vec_reg_assignment (line 8):** The *Assignment* node is vectorized assuming a vector register-to-register assignment. Both children are vectorized as a value expression (see Figure 3.8), avoiding, in this way, explicit vector memory write operations on the `lhs` child.

Value expressions comprise those expressions that do not write into a memory location. Figure 3.8 summarizes some of the rules to vectorize this kind of nodes. Some of these rules describe a direct translation between a scalar node and a vector node. For example, a scalar addition operation is translated into a vector addition

```
function VECTORIZE_VALUE_EXPR(expr)
  switch expr.kind do
    case NODECL_ADD(expr1, expr2, type)
      vtype = VECTORIZE_SCALAR_TYPE(type)
      v1 = VECTORIZE_EXPRESSION(expr1, mask, VF)
      v2 = VECTORIZE_EXPRESSION(expr2, mask, VF)
      return NODECL_VECTOR_ADD(v1, v2, mask, vtype)
    case NODECL_ARRAY_SUBSCRIPT(subscripted, subscripts, type)
      if IS_UNIFORM(expr) then
        return NODECL_VECTOR_PROMOTION(var, vtype, VF)
      else if IS_LINEAR(expr) then
        return VECTORIZE_LINEAR_VAR(expr)
      else if IS_ADJACENT(expr) then
        return VECTORIZE_ADJACENT_MEMORY_LOAD(expr)
      else
        return VECTORIZE_MEMORY_GATHER(expr)
      end if
    case NODECL_SYMBOL(var, type)            ▷ Occurrence of variable 'var'
      vtype = VECTORIZE_SCALAR_TYPE(type)
      if IS_UNIFORM(var) then
        return NODECL_VECTOR_PROMOTION(var, vtype, VF)
      else if IS_LINEAR(expr) then
        return VECTORIZE_LINEAR_VAR(expr)
      else if IS_VECTOR_TYPE(var.type) then      ▷ Due to vec. of local vars
        expr.type = vtype
      else
        return expr                              ▷ Expression is kept scalar
      end if
end function
```

Figure 3.8: Some of the rules for the vectorization of value expressions

operation in lines 3 - 7. Other nodes require a more complex processing. For instance, the array subscript node is processed in lines 8 - 17. If this node is uniform, a vector promotion node will turn the scalar value of the node into a vector value with the scalar value replicated in all the vector lanes. If the node is adjacent, or is neither adjacent nor linear, the node is vectorized using the following transformations, respectively:

**vectorize_linear_var (line 12):** The variable is vectorized assuming it has a linear evolution across loop iterations. This means that the scalar variable will be promoted to vector where each vector lane will be increased with an offset based on the linear step of the variable.

**vectorize_adjacent_memory_load (line 14):** The memory access is vectorized as an adjacent vector memory load. At this point, the alignment properties of the node are evaluated (see Section 3.5.7) and the vector load is qualified as aligned or unaligned accordingly.

**vectorize_memory_gather (line 16):** This memory access is vectorized as a vector memory gather. This means that the vectorizer algorithm cannot guarantee that the memory access follows an adjacent memory access pattern. If the vectorization algorithm detects that the base pointer for each scalar access is the same, the vector gather is qualified as a simple memory gather. Otherwise, the vector gather is qualified as a gather of pointers.

Symbol occurrences are vectorized depending on their uniform and linear attributes, and the type of the variable after the Local Symbol vectorization step, as described Figure 3.8, lines 18 - 27. If the symbol occurrence is uniform, the symbol node is promoted to vector as happened with the uniform array subscript. If the symbol occurrence is linear, the symbol node is vectorized as a linear variable (`vectorize_linear_var`). If the symbol occurrence is neither uniform nor linear and the type of the variable was vectorized in the Local Symbol step, then the type of the occurrence is vectorized. Otherwise, the symbol does not require vectorization and it is kept scalar.

### 3.5.7 Contiguity and Alignment in Memory Accesses

A vector memory access that reads (writes) a set of consecutive scalar elements to (from) memory is usually denoted as contiguous, adjacent or stride-one vector memory access. Non-contiguous, non-adjacent or non-stride-one vector memory accesses are those vector accesses that read (write) scalar elements that are not consecutive in memory. Stride-one vector memory accesses are generally more efficient in the vast majority of SIMD architectures. In fact, some SIMD architectures do not even have support for the non-stride-one counterparts.

In addition, stride-one vector memory accesses can be qualified as aligned or unaligned accesses depending on whether the memory address that they read (write) from (to) is aligned to the boundary of the vector length of the architecture. In many architectures, unaligned vector memory accesses are significantly slower that their aligned counterparts due to hardware constraints. For this reason, the vectorization algorithm must try to maximize the use of aligned vector memory accesses. However, using an aligned vector memory access over an unaligned address usually results in a hardware exception that aborts the execution. Thus, the vectorization algorithm must ensure that the address of a particular vector memory access will always be aligned to the vector length boundary at runtime.

In the following sections, we briefly illustrate how memory accesses are analyzed to determine if they are adjacent and aligned or not. For the sake of simplicity, we limit the explanation of this analysis to array subscripts although our implementation is also able to deal with pointer accesses.

### Contiguity

Our vectorization algorithm computes the *adjacent* attribute defined in Section 3.5.6 to state that a vector access is stride-one or not.

In order to determine whether a scalar memory access performs loads/stores in consecutive positions in memory across loop iterations, the vectorization algorithm analyzes whether the evolution of the access is stride-one or not. To be considered adjacent, the array pointer must be uniform. Otherwise, the memory access is classified as non-adjacent as the array pointer could have different values across the loop iterations. If the array pointer is uniform, then the algorithm analyzes the expression in the subscript of the array. Figure 3.9 shows some of the rules used in this analysis based on the linear and uniform attributes:

**linear:** the evolution of linear expression is determined by their step. A linear expressions with a step one is the base of an adjacent array access (lines 2 and 3).

**uniform:** the value of uniform variables is always the same within a vector register. They cannot cause an adjacent access by themselves (lines 4 and 5).

**unqualified:** variables that are neither linear nor uniform (line 15) can evolve in an arbitrary way across loop iterations. If a variable of this kind is present in the subscript of an array access, the access is classified as non-adjacent.

The attributes of sub-expressions involved in the subscript determine the adjacent attribute of a compound expression. Lines 7 to 13 contain the rules for addition and subtraction expressions. As we can see, an array subscript with an addition or subtraction expression in the subscript will be adjacent if one of the sub-expression of the addition is adjacent and the other one is uniform.

```
function IS_ADJACENT(expr)
    if IS_LINEAR(expr) and LINEAR_STEP(expr) $==1$ then
        return TRUE
    else if IS_UNIFORM(expr) then
        return FALSE
    else
        switch expr.kind do
            case NODECL_ADD(expr1, expr2, type)
                return (IS_ADJACENT(expr1) and IS_UNIFORM(expr2)) or
                (IS_ADJACENT(expr2) and IS_UNIFORM(expr1))
            case NODECL_SUB(expr1, expr2, type)
                return (IS_ADJACENT(expr1) and IS_UNIFORM(expr2)) or
                (IS_ADJACENT(expr2) and IS_UNIFORM(expr1))
    end if
    return FALSE                    ▷ Default case. Arbitrary evolution.
end function
```

Figure 3.9: Some rules used in the computation of the *adjacent* attribute for the subscript expression of an array subscript

### Alignment

Our vectorization algorithm computes the *aligned* attribute defined in Section 3.5.6. Once the array access has been qualified as *adjacent*, the algorithm analyzes the access to determine if its address will be aligned to the vector boundary across all the iterations of the loop. The memory address accessed by an array subscript is computed as $subscripted\_alignment + subscript\_alignment \times sizeof(type)$. If the resulting alignment is multiple of the vector length boundary, the memory access is classified as aligned. An unknown value or alignment values not multiple of the vector length boundary are then classified as unaligned.

Figure 3.10 shows some simplified rules used in the computation of the alignment caused by the subscript expression of the array subscript. This alignment is computed in number of elements. Later, this number is multiplied by the size of the data type of the access to have an alignment in bytes. The alignment of a subscript expression depends on the attributes found on its sub-expressions:

**constant:** for compile-time constants the algorithm returns their constant value (lines 2 and 3).

**suitable:** suitable variables will have a runtime value multiple of the vector length in number of elements. The algorithm returns its suitable value (lines 4 and 5).

**linear:** linear variables require a *suitable* lower bound to produce an aligned access. In the same way, their step in the vector loop must also be *suitable* (lines

```
 1: function ALIGNMENT(expr)
 2:   if IS_CONSTANT(EXPR)( )then
 3:     return expr.const_value
 4:   else if IS_SUITABLE(expr) then
 5:     return GET_SUITABLE_VALUE(expr)
 6:   else if IS_LINEAR(expr) then
 7:     return ALIGNMENT(expr.lb) + ALIGNMENT(expr.step) × VF
 8:   else
 9:     switch expr.kind do
10:       case NODECL_ADD(expr1, expr2, type)
11:         return ALIGNMENT(expr1) + ALIGNMENT(expr2)
12:       case NODECL_MUL(expr1, expr2, type)
13:         return ALIGNMENT(expr1) × ALIGNMENT(expr2)
14:   end if
15:   return UNKNOWN    ▷ Default case. The value of expr is unknown.
16: end function
```

Figure 3.10: Some simplified rules used in the computation of the alignment (in elements) caused by subscript expression of an array subscript

6 and 7).

Computing the alignment of compound expressions requires a special algebra that defines the results for the operations on operands with the UNKNOWN value. For example, addition and multiplication operations depicted in lines 9 - 13 must be able to combine alignment values where the UNKNOWN value could be present. In this way, when the UNKNOWN value is operated with an addition operation, their result is also UNKNOWN. The same happens with the multiplication operation except when the other operand is *suitable*. A multiplication of an UNKNOWN value by a *suitable* value results in another *suitable* value.

### 3.5.8 Function Versioning

A function can be invoked from different regions of the code using arguments with disparate properties. For this reason, the compiler may need to generate multiple vector versions with specific characteristics for the same scalar function.

We designed an infrastructure to manage the versions of each particular scalar function. This infrastructure provides an interface to register and remove versions for a particular function and to find the best version given a set of properties of the scalar function and the conditions at the invocation point.

In this function versioning infrastructure, each vector version of a function contains the following fields:

**Version code:** Function code of the particular version.

**Prototype footprint:** Mask that represents the attributes (*aligned*, *suitable*, *linear* and *uniform*, see Section 3.5.3) of the parameters of that version.

**Priority:** Priority of that version. We assign each version a priority depending on its nature:

- **Naive function:** Lowest priority. Scalar version that unpacks the vector parameters of the function, executes the code of the body in a scalar fashion and packs the scalar results again in a vector data type.
- **Built-in function:** Middle priority. Compiler internal version.
- **SIMD function:** Highest priority. Version that has been vectorized.

**ISA:** Target SIMD instruction set of the version.

**Vector length:** Target vector length of the version.

**Masked:** Flag that states if this vector version is predicated with an initial mask.

When this Function Versioning infrastructure is asked for the most appropriate version of a given scalar function, it chooses the most suitable version based on the parameters provided in the request.

The version returned must target at least the requested ISA and vector length. If the infrastructure does not have a prototype footprint that perfectly matches with the requested one, it will return a version more generic and compatible with the properties of the requested prototype, but probably less efficient. If a masked version is returned when an unmasked version is needed, the compiler will invoke this function with a mask with all the vector lanes enabled. If there are multiple versions that fit the requirements of the query, the priority field is also considered to choose the most appropriate one.

For example, let us assume that a loop contains a call to the math function *sinf*. The compiler will register a naive vector version of this function for the target ISA and vector length. Then, let us also assume that a vector math library is enabled and a SIMD version of *sinf* is registered with the same characteristics. When the Function Versioning infrastructure is used to get a version of *sinf*, it will return the function with the highest priority, i.e., the SIMD version from the math library.

### 3.5.9 Vector Math Library Support

Scalar codes often contain calls to well-known math functions. These functions have also their vector hardware or software counterpart in many SIMD architectures. They are implemented in auxiliary libraries or in the set of intrinsics that provide direct access to the SIMD instructions.

Our vectorization algorithm has support for the Intel Short-Vector Math Library (SVML). This library implements a large set of math functions for several Intel SIMD instruction sets. We register the SVML functions of the target architecture in the Function Versioning infrastructure described in Section 3.5.8. These new vector functions are associated to their scalar counterpart defined in the standard C library. Therefore, the vectorization algorithm will have available these vector math functions in the Function Versioning infrastructure when it is necessary to vectorize calls to these supported scalar math functions.

### 3.5.10 Software Prefetcher

Software prefetching is a memory optimization that consists of requesting cache lines before these cache lines are really necessary for the execution. In this way, the latency resulting from the memory coherence and transfers could potentially be hidden. Those prefetched cache lines could be ready to be used in the local cache memory when the core really needs them.

One of the targets of software prefetching techniques are loops. In a given loop iteration, prefetching techniques aim to request cache lines used in future iterations so that these cache lines have time to arrive to the local cache of the core [25, 13, 97]. This number of iterations in advance is known as *prefetching distance*.

Physical vector registers are reaching the length of a whole cache line in some architectures which, at the same time, is the basic memory transfer unit in software prefetching. In addition, new gather and scatter memory operations may require accessing very disperse elements, involving a relevant number of cache line accesses. These vector memory accesses may benefit from special software prefetching instructions that efficiently bring in advance the cache lines involved in the whole vector memory access. For these reasons, software prefetching is increasingly becoming a technique related with vectorization in some SIMD instruction sets. For example, there are dedicated prefetching instructions for gather and scatter memory operations in the Intel Xeon Phi coprocessor [85].

We designed and implemented a simple software prefetcher for loops that is able to generate prefetching instructions for contiguous vector memory accesses targeting the Intel Xeon Phi coprocessor. The decisions that this software prefetcher has to make are the following:

- Determining which vector memory accesses should be prefetched.
- Choosing a prefetching distance.
- Inserting prefetching instructions in the original code.

In our current algorithm, we only consider contiguous vector memory accesses within a single iteration. Accesses are analyzed assuming that pointers involved do no alias. Unaligned accesses are analyzed as if they had been decomposed in two independent, yet consecutive, vector accesses, one for each cache line involved.

Vector accesses of the same iteration to the same base pointer are analyzed in group to determine the relationship among them. Two different vector memory accesses in the group can access in the same loop iteration to the same memory positions, to consecutive positions or to partially overlapped positions. This may cause that cache lines from one loop iteration are reused in the next loop iteration. Prefetching this particular cache lines with a distance larger than one is useless as the cache line will be needed before the expected prefetching distance. This may introduce an unnecessary overhead.

Consequently, our algorithm only generates prefetching instructions for those cache lines used within a loop iteration that will not be reused again in the next loop iterations. If a cache line is chosen for prefetching and it is read and stored in the same loop iteration, the algorithm prefetches that cache line as exclusive. This means that the cache line will be requested for writing (the cache line will only be present in the private cache of that core, it will be clean and its value will be consistent with main memory).

Our algorithm uses implementation-defined prefetching distances. These distances can be modified by the programmer by means of compiler flags or a programming level interface, defined in Section 4.

Once the prefetching instructions have been generated, the algorithm chooses where to place them within the body of the loop. We implemented two placement strategies:

**On top:** All the prefetching instructions are placed at the beginning of the loop body. This placement strategy is useful to prefetch data of the same loop iteration.

**In place:** Prefetching instructions are placed just before their respective memory accesses. This placement strategy is the common approach to prefetch data across iterations.

## 3.6 Performance Evaluation

The Mercurium vectorization infrastructure does not have auto-vectorization capabilities. This means that the vectorization process requires guiding from the programmer. For this reason, the performance evaluation of our infrastructure is carried out in Chapter 4 and Chapter 5 of this thesis, where a user-directed vectorization approach for OpenMP in introduced.

Figure 3.11 briefly summarizes the performance of some benchmarks evaluated in Chapter 4. This figure shows the speed-up over the scalar baseline achieved with the Mercurium source-to-source vectorization infrastructure (*mcc*) and the Intel C/C++ compiler 15.0.2 (*icc*). The same version of the Intel compiler is used as back-end compiler of Mercurium. The baseline, *mcc* and *icc* versions were compiled with the same level of optimizations and all of them were executed with the same number of threads. As depicted, the Mercurium vectorization infrastructure yields a speed-up comparable to the Intel C/C++ compiler, even outperforming it in some

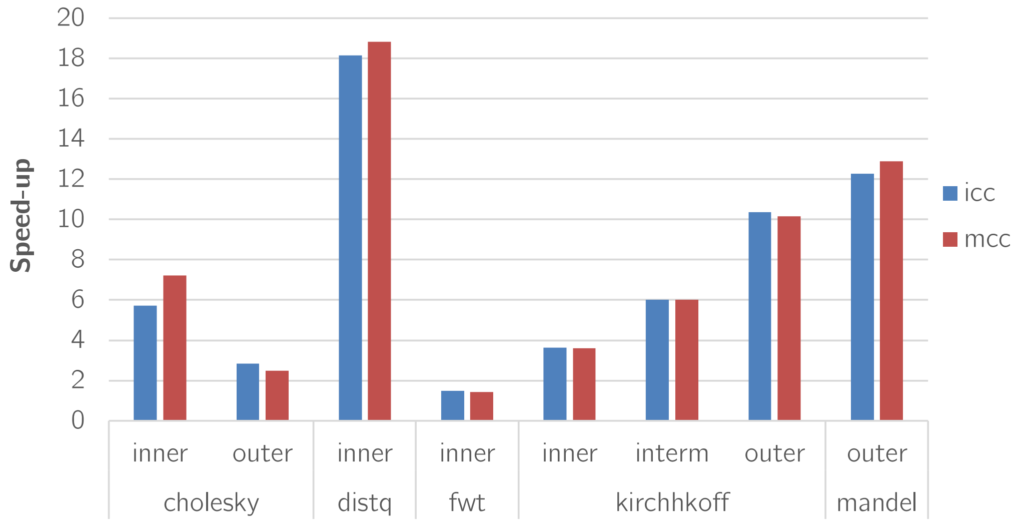


Figure 3.11: Performance summary of our Mercurium vectorization infrastructure (mcc). Baseline scalar version. Intel Xeon Phi coprocessor (1 core, 4 threads/core)

benchmarks. This demonstrates that our vectorization infrastructure is able to yield a performance competitive with an outstanding compiler in production state, at least for the benchmarks evaluated. In Chapter 4, we provide further details about these performance results.

## 3.7 Use Case: A 3D Elastic Wave Propagator

In this section, we present a use case where our source-to-source vectorization infrastructure is used to optimize a scientific application for the Intel Xeon Phi coprocessor [23]. Independently from the performance achieved, with this use case we want to show how useful this vectorization tool can be for programmers.

The application implements a Finite Difference numerical method that solves elastic wave propagation equations considering the anisotropy of the field. The propagation of elastic waves is the current trend in seismic imagining. It provides an improved physical model in comparison to other approaches, at the cost of raising the computational resources needed.

Figure 3.12 shows the evolution of the performance of the elastic propagator application after applying several optimizations incrementally. Our initial implementation has the 3D spatial grid loop split into 16 parts. The outermost loop of the grid has been parallelized using OpenMP, leaving to each thread a number of consecutive memory planes to update. This version, named OpenMP, is our evaluation baseline. It is compiled with the Intel C/C++ compiler 15.0.2 with auto-vectorization capabilities enabled. We run our experiments with 244 threads on the Intel Xeon Phi coprocessor (61 cores and 4 threads/core) as this was the best thread configura-

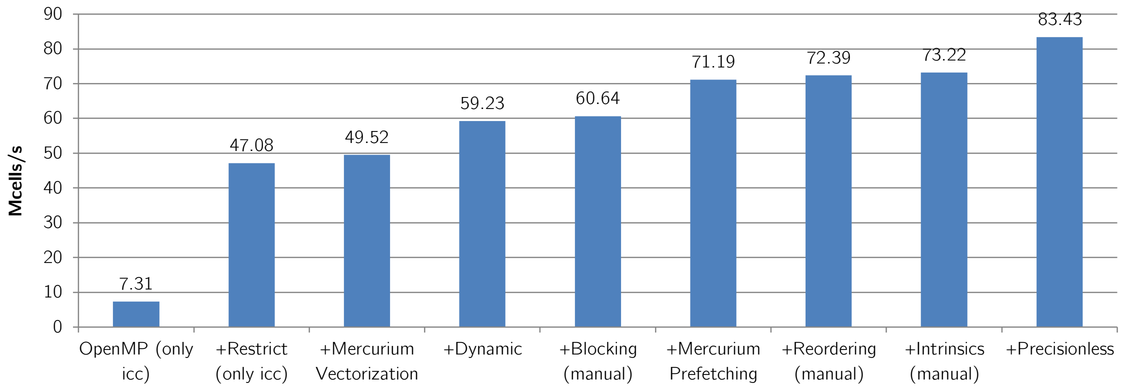


Figure 3.12: Absolute performance evolution (Mcells/s) of a Full Anisotropic Elastic Wave Propagator after applying incrementally a set of optimizations [23]. Run on the Intel Xeon Phi coprocessor using 61 cores and 4 threads/core (244 threads in total)

tion. We briefly describe the optimizations that have been applied to the baseline version to improve its performance on the Intel Xeon Phi coprocessor:

**Restrict:** We add the `__restrict` keyword to function parameters to inform the compiler that the pointers in our code do not alias with each other. This enables the compiler to remove false loop-carried dependences and auto-vectorize our code.

**Mercurium Vectorization:** We use the Mercurium source-to-source vectorization infrastructure to vectorize the code using our OpenMP user-directed vectorization proposal, described in Chapter 4. The Intel C/C++ Compiler 15.0.2 is used as native/backend compiler.

**Dynamic:** We evaluated the three most relevant loop scheduling strategies for threads: static, dynamic and guided. The best performance in our case resulted from using dynamic scheduling.

**Loop Blocking:** We implemented a loop blocking strategy. We performed a systematic benchmarking of block sizes to find out the best blocking configuration that best fits the characteristics of the memory hierarchy and the number of threads used.

**Mercurium Prefetching:** We enabled the software prefetcher of our vectorization infrastructure and carried out an exhaustive exploration to find out the best prefetching distance in conjunction with the best block size.

**Reordering:** We reordered the code of the innermost loops by hand so that produced variables were consumed as soon as possible. This helps the native compiler with its liveness analysis and improves the generated code.

**Intrinsics:** We apply some optimizations by hand over the vector code with intrinsics generated by Mercurium. For example, we improved vector memory accesses.

**Precisionless:** We enabled the use of some fast but less precise floating point instructions using the –fimf–domain–exclusion=31 compiler flag. This flag guarantees that our application will not produce values from some special domains. The native compiler can then generate more efficient code.

Leaving aside the performance achieved by each optimization, we used our source-to-source vectorization infrastructure to vectorize the application in version *Mercurium Vectorization*. Afterwards, we also used the software prefetcher to generate prefetching instructions in version *Mercurium Prefetching*. Finally, in version *Intrinsics*, the source-to-source nature of our vectorization infrastructure allowed us to apply further manual optimizations over the vector code with intrinsics generated by Mercurium. Even though we did not get much performance improvement with these optimizations, having the possibility of modifying the vector code saved us from vectorizing the code by hand to be able to apply these optimizations.

## 3.8 Chapter Summary and Discussion

In this chapter, we have described the source-to-source vectorization infrastructure designed and implemented in the Mercurium C/C++ source-to-source compiler. This infrastructure is the largest piece of software developed during this PhD and it has been used in the two following contributions of the thesis (presented in Chapter 4 and Chapter 5).

We extended the type system of the compiler to be able to represent vector data types. Furthermore, we extended the scalar intermediate representation (IR) with new vector nodes that allow representing vector operations without overloading the semantic of scalar nodes.

The vectorization infrastructure has two compiler phases: Vectorizer and Vector Lowering. In the Vectorizer phase, the compiler transforms the intermediate representation of the target scalar code into a vector IR. Afterwards, the Vector Lowering phase lowered this vector IR to intrinsics of the target SIMD instruction set.

Our vectorization infrastructure targets loops and function codes, where function codes are vectorized as if they were invoked from a vector loop. For the vectorization process, we define the following five attributes: *adjacent*, *aligned*, *suitable*, *linear* and *uniform*. These attributes describe the evolution of the value of expressions in the code, and other characteristics such as alignment information and its memory access pattern. Based on this information, the vectorization algorithm determines which vector transformation must be applied to each expression and statement in the code.

Our infrastructure is able to vectorize a wide variety of codes. For example, it includes support for predicated vectorization with masks, alignment-aware stride-one vector memory accesses, vector gather and scatter memory accesses, vector math libraries, multi-versioning of vector functions and software prefetching.

The evaluation of our vectorization infrastructure in a set of benchmarks shows outstanding performance results. These results are competitive with those achieved with the Intel C/C++ Compiler 15.0.2 (the last production version at the time of writing this thesis) with the same level of optimizations.

In addition, we present a use case that shows the usefulness of our source-to-source vectorization tool in the process of optimizing a scientific application. Using the Mercurium vectorizer, programmers cannot only easily vectorize the application but they can also apply further manual optimizations on the source vector code generated with our infrastructure. This releases them from vectorizing the code by hand in order to have a source vector code on which to apply manual optimizations.

Further discussion and related work on source-to-source vectorization infrastructures and vectorization techniques is Chapter 7, Section 7.1.2 and Section 7.1.3.

### 3.8.1 Impact

The impact of this contribution is being remarkable as a programming and research tool. The Computer Applications in Sciences and Engineering (CASE) department at Barcelona Supercomputing Center is currently starting to use our vectorizer to optimize a production application in collaboration with Repsol.

Moreover, our source-to-source vectorization infrastructure has been considered as candidate to be used in two European projects. The first one is the RoMoL project [43]. In this project, the Mercurium source-to-source vectorization infrastructure could be used to carry out further work on vectorization targeting architectures with long vector instruction sets. The second project is Montblanc 3 [29]. In this project, our vectorization infrastructure could be used to generate vector code for ARM architectures, targeting the Neon SIMD instruction set.

# Chapter 4

# SIMD Extensions for OpenMP

## 4.1 Introduction

In this chapter, we introduce our proposal to extend OpenMP 3.1 with SIMD constructs. These constructs allow programmers to describe SIMD parallelism and guide the compiler in the vectorization transformation. We implemented a prototype of this proposal in the Mercurium C/C++ source-to-source compiler to conduct the empirical evaluation targeting the advanced SIMD instruction set of the Intel Xeon Phi coprocessor. We compare our results with scalar and auto-vectorization approaches using the Intel C/C++ compiler. We also provide performance results of the official OpenMP 4.0 SIMD extensions that were influenced by this work.

### 4.1.1 Motivation

Automatic vectorization has been an active topic in research for several decades. Essentially, it seems a simple and well limited problem that even programmers can tackle successfully with advanced programming skills. Then, why is automatic vectorization so hard for compilers?

First of all, we must have in mind that the first goal for a compiler, even for an optimizing compiler, is generating a binary code whose execution does what is expected. This means that the binary code must do precisely what the programmer described, complying with the standard specification of the programming language. In other words, the correctness of the execution takes priority by default over any other matter. Generating a faster code that might not produce the expected output or behave according to the programmer specification is not an option. This is true even if the wrong output or unexpected behavior could happen in very unlikely cases. Consequently, the compiler must prevent vectorization unless it can guarantee that this code transformation can be applied with all certainty of producing an equivalent output (some precision error could be accepted) and behaving as if the code had not been optimized (scalar code).

The requirements considered to determine if vectorization is safe to be applied

```
float tmp = 0.0f
for(i=halo; i<N-halo; i++){
    tmp += foo(b[i-halo]);
    if (tmp > threshold) break;
    a[i] = tmp;
}
```

Listing 4.1: Motivating code with potential problems for auto-vectorizing compilers

on a scalar code are mainly those related with the exploitation of parallelism in general terms. Targeting the C/C++ programming language, we classify some of the most outstanding issues that can prevent a compiler to automatically vectorize a code. For this classification, we use the following six groups: programming language flexibility, common concurrency issues, lack of compilation technology, hardware limitations, software limitations and lack of information at compile time.

### Programming Language Flexibility

Some programming language features can be very powerful and increase the programming efficiency, flexibility and expressiveness of the language. However, at the same time, such semantic power can hinder the static analysis of the code and the subsequent code optimization. Pointers, complex data structures and direct and indirect function calls are examples of these features.

**Pointer aliasing**, i.e., the possibility that several pointers point to the same memory location, can prevent vectorization if the resulting code does not satisfy the potential data dependences among the pointers that can alias. For example, `a` and `b` in Listing 4.1 could alias. If they do alias, there will be a data dependence that potentially could generate a wrong output if the loop is vectorized. This dependence will be satisfied only if the dependence distance (`halo`) is greater or equal to the vectorization factor.

**Data structures** composed by other non-basic static data types, multiple levels of indirections or dynamic memory fields can jeopardize vectorization. This is particularly problematic if the vectorization algorithm needs to emit a vector instance of such a structure.

**Function calls** are very limiting components for code optimizations and compiler analysis in general. They imply changing the execution flow to a different part of the application. The callee function could be unknown at compile time or the source code might not be available when compiling the caller code. Therefore, when vectorizing the code that performs the function call, the compiler might not know whether there is a suitable vector version of the callee function with the expected interface and calling-convention. When this happens, the compiler may decide to vectorize the surrounding code, maintaining the function invocation scalar. This approach introduces additional overhead as it requires to scalarizing the vector function call and to turn the returning value again into a vector. In Listing 4.1, the call to the function `foo` is an example of a code where a function call with no function code available hinders the vectorization of the loop.

### Common Concurrency Issues

Vectorization is constrained by some generic parallelism limitations as it exploits single-instruction-multiple-data parallelism. Examples of these limitations are flow and control dependences, and pointer aliasing.

**Flow dependences**, also known as *true* or *read-after-write* (RAW) data dependences, define the situation where two entities of the program refer to the same data and have a producer-consumer relationship. This kind of dependences establishes an order in the execution of instructions that prevents the execution of the involved instructions in parallel. In Listing 4.1, if we assume that `a` and `b` pointers alias with each other, some values of the `b[i-halo]` memory read may depend on values of the `a[i]` memory write from previous iterations. In other words, there may be a true data dependence with a distance determined by the runtime value of the variable `halo`.

**Control dependences** happen when the execution of a code region depends on the result of a previous condition. Some control dependences can be specially harmful for vectorization as **speculative execution of code** may be required. For example, in Listing 4.1 the `break` statement defines a control dependence on every iteration of the loop with their previous iteration. In other words, the iteration `i` of the loop *should* be executed if and only if the `break` statement has not been reached in iterations lower or equal to `i-1`. If the code is vectorized without any previous code transformation, all vector lanes will be enabled at the beginning of their corresponding iteration. If one vector lane reaches the `break` statement, that vector lane and all the following ones will be disabled. The code already executed of that iteration by the disabled threads will have been speculatively executed and their side effects will have to be counteracted or discarded. Thus, the compiler will not be able to vectorize the motivating example unless it can guarantee that the function `foo` can be speculatively executed and it has no side effects or its side effects can be counteracted.

### Lack of Compilation Technology

Vectorization of complex codes may require sophisticated analysis techniques and code transformations. Particularly, function calls usually need from inter-procedural approaches sometimes only applicable at link-time. Although there exist relevant research proposals, their applicability to production compilers is limited for being very aggressive or expensive in terms of execution time.

Other examples of codes that require important compiler technology to be vectorized are codes with complex control flow or **idioms** that need a special treatment, such as reduction operations.

Moreover, vectorizing codes with **divergences in the control flow** usually requires aggressive code transformations. When the code allows it, these transformations are aimed at producing an equivalent code without control flow divergences. When the control flow is complex and the previous approach is not feasible, the compiler must generate predicated vector code where vector lanes are enabled and disabled according to each divergent region. In the case of the code in Listing 4.1, predicated vector code might be necessary since the `break` statement increases the complexity of the control flow.

Furthermore, the compiler has to use **cost models** to determine if the vectorization of a piece of code will yield better or worse performance than the scalar version. Again, the complexity of the code, the complexity of the vector instruction set and the constraint on the compilation time lead the compiler to use suboptimal and conservative approaches.

### Hardware Limitations

Even when the compiler has the technology to vectorize a piece of code, the code might not finally be vectorized because the target SIMD instruction set does not support the needed instructions. In some cases, missing vector instructions are simulated in software by means of other vector instructions or scalar instructions (partial scalarization of the code). However, the performance of this approach is limited and it might even yield lower performance than the scalar version. For example, predicated vector codes require instruction sets with support for masks that enable and disable each vector lane execution accordingly. As stated before, the success or failure addressing these issues also depends on compiler cost models that make the decision of vectorizing or not vectorizing a code.

In addition, some SIMD architectures impose certain constraints that make vectorization even harder. For instance, many current SIMD architectures have strong alignment constraints when dealing with vector memory operations. In many cases, accessing memory addresses that are not aligned to the vector length boundary entails more latency than those accesses that are aligned. Some other architectures do not support vector memory operations on unaligned memory references.

### Lack of Information at Compile Time

The compiler may prevent vectorization due to a **lack of runtime information at compile time**. For instance, the correctness of the vectorized code might not be guaranteed for the whole range of some variables. However, the problematic values might not be realistic or even possible according to external factors of the application not represented in the code. Therefore, the code could have been safely vectorized since this problematic cases will not take place at runtime.

In the same way, compiler cost models have to make important assumptions when dealing with variables with an **unknown value at compile time**. These unknown values can affect important semantic components of the code such as loop trip counts, the alignment condition of memory accesses, and the execution guards of basic blocks, among other components. These assumptions can be highly determinant in the decision of vectorizing or not vectorizing a code and, unfortunately, they may be totally wrong when the real execution of the code takes place. Based on these assumptions, cost models could conclude that the code is not worth to be vectorized when it is really worth it according to the runtime execution circumstances.

### Software Limitations

Some codes or algorithms are simply not suitable for vectorization. They were designed without having vectorization in mind and it is very difficult for vectorization to be applied to them, or, if to, to yield better performance than the scalar counterpart. There are many reasons that make a **code unsuitable for vectorization**, such as codes that contain:

- Intensive unstructured memory access patterns.
- Complex data structure such as graphs, trees, lists, or any other structure with a high number of indirect accesses.
- `Goto` statements or any other feature that results in an unstructured control flow.
- High level of control flow divergences, such as codes that need large `switch` statements.
- Input/output primitives that do not have an equivalent vector version.
- Synchronization or concurrency control primitives, such as atomic operations critical sections, mutexes or any other artifact that affects or limits parallelism.

Codes with any of these characteristics should be redesigned or rewritten aiming at removing or minimizing all of these problematic components.

### Practical Example

Figure 4.1 shows the performance of several vectorization approaches for the Blackscholes and Mandelbrot benchmarks executed on the Intel Xeon Phi coprocessor. These vectorization approaches comprise two compiler auto-vectorization versions: a first one with regular function calls (not inlined) and a second one with function calls inlined. In addition, we evaluate the equivalent inlined and not inlined versions applying vectorization by hand using the specific SIMD intrinsics of the Intel Xeon Phi coprocessors.

It is interesting to note how the Intel compiler is not able to vectorize the code of any benchmark when function calls are not inlined. This happens even when

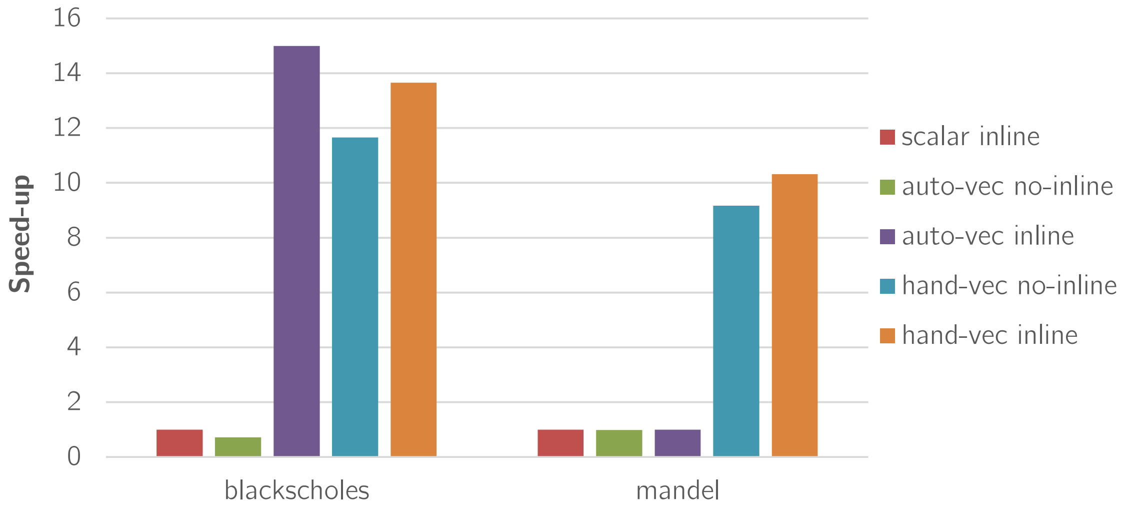


Figure 4.1: Speed-up of auto-vectorization and hand-vectorization versions of Blackscholes and Mandelbrot benchmarks with inlined and not inlined functions. Baseline: scalar version with inlined functions. Intel C/C++ Compiler 13.1.3. Running on an Intel Xeon Phi coprocessor using 1 core and 4 threads/core.

the code of these functions is available at compile time in the same file and the compiler can analyze it. When function calls are inlined, the compiler vectorizes the Blackscholes benchmark yielding a speed-up of 15 over the scalar version with inlined functions. Nevertheless, the Mandelbrot benchmark is not automatically vectorized by the compiler even when function calls are inlined. The reasons are that the innermost loop of this benchmark is not vectorizable and the code has a complex control flow that leads the compiler to keep the scalar version of the code.

The considerable speed-up of the hand-vectorized versions without inlined functions reveals the compiler limitations when vectorizing codes with function calls that cannot be inlined. However, when functions are inlined in Blackscholes, the hand-made version yields less performance than the auto-vectorized version. The reason is that the Intel C/C++ compiler does not apply some code optimizations on SIMD intrinsics. It relies on the code written by the programmer and it attempts to preserve the original code as much as possible. This means that the programmer should provide a vector code with intrinsics as optimized as possible for the target architecture. This reveals that optimally vectorizing a code by hand is more complex than just vectorizing the code with SIMD intrinsics. This fact motivates even more the use of the compiler for vectorization. The compiler has a great knowledge of the target architecture and outstanding vectorization capabilities that can even outperform the skills of advanced programmers, as in this case.

### 4.1.2 Objectives

The main goal of this proposal is to boost the exploitation of SIMD instructions by enabling the vectorization of codes that are not auto-vectorized by the compiler. The key concept to achieve this goal is moving to the programmer part of the responsibility of the vectorization process that now is assumed by the compiler. Thus, the programmer will guide the compiler vectorization process, indicating to the compiler which code regions are safe and should be vectorized.

To achieve this goal in a generic, efficient and easy way for programmers, we propose a set of SIMD extensions to the 3.1 version of the OpenMP programming model. We chose OpenMP because it is a high-level architecture-independent programming model that allows programmers to efficiently describe several kinds of parallelism. This description is performed with compiler directives that do not require to change the original serial code of the application. The new SIMD extensions that we propose for OpenMP bring SIMD parallelism and vectorization to this programming model. With these extensions, now programmers will be able to describe SIMD parallelism and even how this interacts with other kinds of parallelism available in OpenMP. Moreover, we propose a set of optional clauses that can be used to provide the compiler with further tuning information that may lead to a more optimized vector code.

## 4.2 Standalone SIMD Directive

The simplest construct that we propose to describe SIMD parallelism is the standalone SIMD directive. This construct is used to delimit a code region that is safe to be vectorized according to the programmer. These regions of code can be for-loops and functions (declarations and definitions). Figure 4.2 shows the syntax of the SIMD directive for C/C++.

This directive is useful to instruct the compiler to vectorize the annotated code, relaxing some of the strong restrictions that prevent vectorization in fully automatic approaches (see Section 4.1.1). This means that the compiler does not have to determine whether the vectorization of that code is safe or whether it is profitable using cost models. It will assume that vectorization can be safely applied without considering hypothetical data dependences, pointer aliasing, function call limitations and cost models.

It is important to note that this directive must be understood as a hint to the compiler. In spite of the fact that the programmer is now responsible for providing the compiler the right information, each compiler is free to decide not to vectorize the annotated code. For instance, this could happen if the compiler cannot ensure that the resulting vector code will be correct or the target architecture does not support a SIMD version.

When a function is annotated with the SIMD directive, the compiler will try to generate a vector version of that function, converting all parameters and the return

`#pragma omp simd` *[clause [clause] ...] new-line*
*for-loop — function-decl — function-def*

Figure 4.2: C/C++ syntax of the standalone `simd` construct

```
#pragma omp simd
float max(float a, float b);

#pragma omp simd
for(i=0; i<N; i++)
{
    b[i] = max(a[i], b[i]);
}
```

Listing 4.2: Simple example using the standalone SIMD directive

value to vector types by default. When the SIMD directive is applied to a for-loop, the compiler will try to vectorize the loop. If a SIMD-annotated function is called inside a SIMD-annotated loop, the compiler will emit a call to the vector version of that function.

Listing 4.2 shows an example of the SIMD directive for a loop and a function definition. Without the `simd` annotation, the compiler would not be able to vectorize the loop automatically if the `max` function is not inlined, as was demonstrated in Section 4.1.1. If vectorization was carried out, it would entail scalarizing the `max` call, i.e., extracting scalar elements of those vector variables that will be used as parameters, invoking the scalar function `max` several times with these scalar parameters and building a vector with the scalar results of the function calls. This approach would be highly inefficient.

However, by using the SIMD construct on the loop, we are asking the compiler to vectorize it without taking into account the previous performance consideration. Therefore, the compiler will generate a vector version of the loop using some vectorization strategy, such as the one that we described in Chapter 3. The resulting equivalent vector version of the code will normally contain at least one loop with vector instructions. As we described in Section 3.5, it is also very common that the vector version of the code also contains an epilogue loop to compute the remaining iterations not computed by the main vector loop. Furthermore, a prologue loop is sometimes added before the main vector loop to compute some scalar iterations with optimization purposes. Additionally, if we annotate the function definition, we are ensuring to the compiler that there will be a compatible vector function of `max` ready to use instead of the scalar version. Using the vector version of the function `max` will result in a much more efficient vector version of the loop.

`#pragma omp simd for` *[clause [clause] ...] new-line*
  *for-loop*

Figure 4.3: C/C++ syntax of the SIMD-for construct

```c
float *a, *b, *c;
...
#pragma omp simd for
for(i=0; i<1600; i++)
{
    a[i] = b[i] + c[i];
}
```

Listing 4.3: Simple example using the SIMD-for directive

## 4.3 SIMD Directive and Fork-Join Parallelism

SIMD parallelism and fork-join thread parallelism are different approaches to exploit the same level of parallelism: data parallelism. As a thread of execution can execute vector instructions, it seems natural that both approaches can be exploited at the same time to maximize the use of all the parallel resources of the chip.

We define the combined construct *SIMD-for* with this purpose. This construct tells the compiler that a loop can be vectorized and then, its vector iterations must be distributed among threads. In this way, each thread will execute a subset of SIMD iterations, instead of the potentially scalar iterations executed using the traditional standalone *for* directive defined in OpenMP.

Figure 4.3 shows the C/C++ syntax of the SIMD-for directive. As is stated, this directive can only be used on for-loops. Functions cannot be annotated with this construct. However, vector versions of functions annotated with the standalone SIMD directive can also be called within loops annotated with the SIMD-for directive.

Listing 4.3 illustrates the use of this directive on a loop with 1600 scalar iterations. For convenience, let us assume that the target architecture has a vector length of 16 single-precision floating point elements and we are using 4 threads. Based on this information, the compiler will firstly vectorize the code, turning the scalar loop into a vector loop with 100 iterations. Then, it will distribute these vector iterations based on the static schedule (default schedule used very often in OpenMP implementations) in the same way as the standalone *for* directive would do. Therefore, iterations from 0 to 24 will be assigned to the first thread, iterations from 25 to 49 to the second thread, iterations from 50 to 74 to the third thread and iterations from 75 to 99 to the fourth thread.

### 4.3.1 Loop Iteration Scheduling

The SIMD-for construct has a new semantic meaning that affects the schedule of iterations. This change is in the way of interpreting the chunk size used in such scheduling. In a SIMD-for directive, the chunk size of every schedule refers to the number of SIMD iterations after applying vectorization, both if the default chunk is used or if it is a user-defined chunk. For instance, a `schedule(static, 2)` will distribute contiguous chunks of two scalar iterations for a *for* directive, but two vector iterations in a SIMD-for directive.

This definition precludes from carrying out a fine-grained scalar scheduling of iterations. However, expressing a scalar chunk size in terms of a particular vector length is not portable across architectures with different vector lengths. Furthermore, a chunk size not multiple of the vector length would not make use of the whole vector register of the architecture. In addition, using a chunk size not multiple of the vector length would potentially lead to a less efficient vector code in architectures with alignment constrains. The reason is that the generated code would have to deal with unaligned memory accesses or to introduce a prologue loop per thread to aligned memory references in the main vector loop, as we described in Section 3.5.

### 4.3.2 Distribution of Prologue/Epilogue Loop Iterations

As we described in Section 3.5, the vectorization of a loop can result in an equivalent vector code with multiple loops. These loops can be the main vector loop, an epilogue loop after the main vector loop and a prologue loop before the main vector loop.

The prologue and the epilogue loops can compute a maximum of $vector\_length - 1$ scalar iterations each one. These iterations must be assigned to the threads involved in the computation of the SIMD-for loop. This distribution is implementation defined. If the prologue or the epilogue loops are vectorized, their (partial) single vector iteration could be assigned to a single thread. If they are kept scalar, their iterations might be distributed among multiple threads. Different strategies about how to distribute these prologue and epilogue loop iterations are discussed in Section 4.4.8.

## 4.4 Optional Clauses

In Section 4.2 and Section 4.3, we introduced SIMD constructs that enable and make the vectorization of loops and functions easier to the compiler. However, besides informing the compiler that a loop can be safely vectorized, these SIMD annotations do not provide further information of the target code. Thus, the compiler can emit generic and conservative vector code for the annotated region. Nevertheless, maybe this code is not as optimal as it could be if the compiler had more actual information from the code or even from its runtime execution.

```
aligned(expressions_list[:alignment])
```

Figure 4.4: Syntax of the *aligned* clause

```
#pragma omp simd aligned(x,y,z:64)
for (i=0; i<N; i++)
{
    z[i] = a * x[i] + y[i];
}
```

Listing 4.4: Example of the *aligned* clause in the Saxpy kernel

In this section, we introduce a group of clauses aimed at extending the functionality and enriching the expressiveness of the SIMD constructs. These optional clauses can be utilized to provide the compiler with more information that can be useful to generate better vector code. With these clauses, programmers will have an agile way of performing a systematic benchmarking of their applications. In this way, they will have available the possibility of easily testing different vector versions of their code to find which one better fulfills their requirements.

### 4.4.1 The *aligned* clause

The compiler assumes by default that all pointers to memory are not aligned to any specific boundary. This assumption leads the compiler to generate a vector code with unaligned vector memory accesses or to use techniques that align these accesses at runtime. For instance, the compiler could generate multiple versions of the code, one with aligned accesses and one with unaligned accesses, to execute the most appropriate introducing alignment checks at runtime. Another common option when vectorizing a for-loop is to introduce a prologue loop before the main vector loop. This prologue loop will peel at runtime those scalar iterations that make the target accesses unaligned, as we described in Section 3.5. Both approaches can introduce a considerable overhead. In addition, these approaches do not scale well when there are multiple pointers and variables involved in the memory accesses of the loop that have to be checked at runtime. This leads the compiler to generate a sub-optimal code in terms of alignment.

We propose the *aligned* clause to tackle this problem. The *aligned* clause provides information about the alignment of the base memory address of arrays and pointers. The syntax of this clause is shown in Figure 4.4. The `expression_list` enumerates a set of aligned expressions. Optionally, the `alignment` parameter can be used to indicate the alignment of those expressions in bytes. This alignment must be constant at compile time. If `alignment` is not specified, the list of expressions is assumed to be aligned to the vector length boundary of the target architecture.

```
 1 float (*u)[sizey];
 2 float (*utmp)[sizey];
 3
 4 for (i=1; i < sizex-1; i++)
 5 {
 6 #pragma omp simd suitable(sizey:16) aligned(u,utmp:64)
 7     for (j=1; j < sizey-1; j++)
 8     {
 9         float tmp = CONST *
10             (u[i  ][j-1] + u[i  ][j] + u[i  ][j+1] +
11              u[i-1][j-1] + u[i-1][j] + u[i-1][j+1] +
12              u[i+1][j-1] + u[i+1][j] + u[i+1][j+1]);
13         ...
14         utmp[i][j] = tmp;
15      }
16 }
```

Listing 4.5: Example of the *suitable* and the *aligned* clauses

In Listing 4.4, the compiler would generate unaligned vector loads for accesses `x` and `y`, and an unaligned vector store for access `z` if the *aligned* clause was not used. However, the *aligned* clause informs the compiler that the base addresses of arrays `x`, `y` and `z` will be aligned to 64-bytes at runtime. Therefore, the alignment of each access depends now on the alignment of the address yielded by the respective offset of the access from the aligned base address. In the example, the offset of all the accesses is `i`, which is the induction variable of the loop with a lower bound of 0 and a step of 1. With this information, the compiler can infer that the addresses of all accesses in the loop will be aligned. Consequently, it will generate two aligned vector loads for accesses `x` and `y` and an aligned vector store for access `z`. In this way, the compiler could generate a code with no runtime checks related with the alignment of pointers `x`, `y` and `z`. Furthermore, it will be more efficient in architectures with alignment constraints.

### 4.4.2 The *suitable* clause

The *aligned* clause might not be enough to generate aligned vector memory accesses when dealing with complex array accesses or pointer arithmetic expressions that contain other variables different from the loop induction variable. The alignment of these accesses also depends on the runtime value of those variables that add an extra offset to the address of the access.

For instance, the linearized version of the memory store `utmp[i][j]` in line 14 of Listing 4.5 is `utmp[i*sizey + j]`, where `sizey` is the size of the least significant dimension of `utmp`. The alignment of this access depends on whether the memory address $utmp + i * sizey + j$ is aligned to the vector length boundary. Assuming that `utmp` is said to be aligned, the alignment of the access will then depend on whether value of `sizey` at runtime is multiple of the vector length. As this value is

```
suitable(expressions_list[:multiple])
```

Figure 4.5: Syntax of the *suitable* clause

unknown at compile time, the compiler will generate an unaligned vector store for this memory access or it will generate several versions of the code orchestrated by runtime checks, as we described in Section 4.4.1.

This problem can be addressed with the *suitable* clause. This clause informs the compiler that a list of variables or expressions will have a runtime value multiple of a given value provided as argument.

Figure 4.5 depicts the syntax of this clause where `expression_list` is the list of suitable variables and `multiple` is the constant value these variables are multiple of. It is important to note that the vector length of the architecture does not need to be explicitly specified in the clause. This favors portability across architectures with different vector lengths.

As stated before, the *suitable* clause is useful in codes with multidimensional arrays to state that the first element of every row is aligned to a particular boundary. In such cases, setting one or some of the least significant dimensions as `suitable` allows the compiler to generate aligned memory accesses in a more efficient way. Moreover, knowing of suitable expressions is also convenient to compute a more optimal epilogue loop if suitable expressions are involved in the computation of the loop iterations.

The example of Listing 4.5 shows the combined usage of the clauses *suitable* and *aligned*. The *aligned* clause states that `u` and `utmp` pointers are aligned to the vector length boundary. The variable `sizey` is the size of the least significant dimension of both arrays. The clause suitable states that this variable will have a runtime value multiple of 16. This information will be used by the compiler in order to generate aligned memory accesses for vector loads on `u[i][j]`, `u[i-1][j]` and `u[i+1][j]`, and the vector store on `u[i][j]`, assuming a vector length of 16 single precision floating point scalar elements. Furthermore, the compiler will be able to emit an epilogue loop with a constant number of iterations because `sizey` is set as suitable and it is used in the upper bound of the vectorized loop. This epilogue loop will compute exactly 14 iterations if we assume that the main vector loop processes 16 scalar elements at each vector iteration.

### 4.4.3 The *uniform* and *linear* clauses

The evolution of some loop induction variables can be very hard to analyze for the compiler. Their iteration steps can depend on function calls or complex control flows. If the compiler is not able to characterize the evolution of these induction variables appropriately, the generated vector code might contain unnecessary gather/scatter memory operations or other expensive vector instructions.

```
uniform(expressions_list)

linear(expressions_list[:linear_step])
```

Figure 4.6: Syntax of the *uniform* and *linear* clauses

```
#pragma omp simd uniform(x, y, z) linear(j)
void foo (float *x, float *y, float *z, float a, int j)
{
    z[j] = a * x[j] + y[j];
}

void saxpy(float *x, float *y, float *z, float a, int N)
{
    #pragma omp simd
    for(int j=0; j<N; j++)
    {
        foo(x, y, z, a, j);
    }
}
```

Listing 4.6: Example of the *linear* and the *uniform* clauses

Moreover, SIMD-enabled functions can be called from a SIMD loop body and receive as arguments induction variables or variables that keep a uniform value in all the loop iterations. However, this information is not available for the compiler when it is vectorizing the function code. Therefore, the compiler generates a vector code for that function that is less efficient but it is ready to work with more generic parameters.

We propose the *uniform* and *linear* clauses to allow the compiler to generate a more efficient code in these cases. These clauses give information to the compiler about the evolution of a variable within the vector lanes of a vector register. With this information, the compiler will be able to potentially emit more specialized and faster vector code.

The *uniform* clause states that the value of a variable will have no evolution throughout the vector lanes of a vector register, i.e., the value of such a variable will always be the same within chunks of consecutive vector length iterations but it could change across these chunks. The *linear* clause states that the value of a variable will have a linear evolution within chunks of consecutive vector length iterations, i.e., the value of such a variable for a vector lane can be expressed as the value of that variable in the previous vector lane plus a linear step.

Figure 4.6 shows the syntax of the `uniform` and `linear` clauses. The keyword `expression_list` refers to the list of *uniform* and *linear* variables, respectively. By default, the step of the linear evolution is assumed to be 1. Optionally, it is

```
vectorlength(expression)

vectorlengthfor(data-type)
```

Figure 4.7: Syntax of the *vectorlength* and the *vectorlengthfor* clauses

possible to specify a different linear step using the `linear_step` parameter. This parameter must be constant at compile time.

The *uniform* clause is useful when passing pointers or variables that do not change within the caller SIMD loop (or another SIMD function) to a SIMD function. The *linear* clause allows describing loop induction variables with a complicated evolution that might not be detected by the compiler. In addition, the *linear* clause can be used in function declarations/definitions to state that a parameter will always have a linear evolution. The information provided with these clauses allows the compiler to emit a more efficient code, especially when the annotated variables are used in array subscripts.

Listing 4.6 illustrates the use of the *uniform* and *linear* clauses. If no *uniform* nor *linear* clauses were used on `x`, `y`, `z` and `j`, the compiler should assume that these variables will have different and arbitrary values in each function call. Therefore, vector memory accesses `z[j]`, `x[j]`, `y[j]` would be vectorized as scatter and gathers with a different pointer base per vector lane. When `x`, `y` and `z` are qualified as *uniform*, the compiler can now vectorize the three accesses as scatter and gathers with the same pointer base per lane, respectively. These vector operations are significantly simpler than the previous ones. With the `j` variable annotated as *linear* with a (default) step 1, the compiler can then generate stride-one vector loads and a vector store for the three vector memory accesses.

### 4.4.4 The *vectorlength* and the *vectorlengthfor* clauses

When the compiler has to vectorize a loop, it has to choose the vectorization factor, i.e., the number of scalar iterations of the loop that will be computed within each vector iteration. This factor is normally a multiple of the vector length of the target SIMD architecture. However, when the loop to be vectorized contains mixed data lengths, the number of elements that fits into a vector length depends on the length of chosen data type. For instance, assuming a vector register of 64 bytes, a loop that executes operations on `float` (4 bytes) and `unsigned char` (1 byte) data elements could be vectorized using a vectorization factor of 16 (16 `float` elements fully fill a vector register) or a factor of 64 (64 `unsigned char` elements fully fill a vector register). In this way, the compiler will have to choose the most appropriate vectorization factor based on heuristics that do not always lead to the best performance.

We propose the *vectorlength* and the *vectorlengthfor* clauses to allow program-

```
void vl_example (float *a, float *b, int m, int N)
{
    #pragma omp simd vectorlength(8)
    for (int i=0; i<N; i++)
    {
        a[i] = a[i-m] * b[i];
    }
}
```

Listing 4.7: Example of the *vectorlength* clause

```
void image_filter(unsigned char * input_img, unsigned char * output_img,
    float * filter, const int img_height, const int img_width)
{
    for (int i = 0; i < img_height; i++)
    {
        #pragma omp simd vectorlengthfor(float)
        for (i = 0; i < img_width; i++)
        {
            float pixel_c = input_img[j][i] * filter[i];

            pixel_c = ( pixel_c > 255.0f) ? 255.0f : pixel_c;
            pixel_c = ( pixel_c < 0.0f) ? 0.0f : pixel_c;

            output_red[j][i] = pixel_c;
        }
    }
}
```

Listing 4.8: Example of the *vectorlengthfor* clause

mers to have control on this decision. Both clauses provide the compiler with information related with the vectorization factor that must be used in the vectorization process, but they have different purposes. Their syntax is shown in Figure 4.7. The `expression` provided must be a constant expression known at compile time and the argument `data-type` refers to a basic data type.

The *vectorlength* clause establishes the maximum vectorization factor that the compiler can use for the vectorization of the target code. Providing this maximum vectorization factor, the compiler may be able to vectorize codes with data dependences that, otherwise, could prevent vectorization.

Listing 4.7 shows an example of the *vectorlength* clause. As depicted, there is a potential true data dependence on the array `a` with distance `m`. Since `m` is a parameter of the function, its value is unknown at compile time. However, the programmer knows that the runtime value of `m` will be eight or greater which turn the code into safely vectorizable for a maximum vectorization factor of eight, as the *vectorlength* clause indicates.

The `vectorlengthfor` clause instructs the compiler to vectorize the code us-

```
reduction(operator:variable_list)
```

Figure 4.8: Syntax of the *reduction* clause

```
void simd_red(float* in, float* out,
              int N, int M)
{
  #pragma omp parallel for
  for(int i=0; i<N; i++)
  {
    float red = 0.0f; //Thread-local
    #pragma omp simd reduction(+:red)
    for(int j=0; j<M; j++)
    {
      red += in[i*M + j];
    }
    out[i] = red;
  }
}
```

(a) Standalone SIMD directive

```
float simd_for_red(float* b, int N)
{
  float red = 0.0f; //Thread-shared

  #pragma omp parallel
  {
    #pragma omp simd for reduction(+:red)
    for(int i=0; i<N; i++)
    {
      red += b[i];
    }
  }

  return red;
}
```

(b) SIMD-for directive

Figure 4.9: Examples of the *reduction* clause

ing a vectorization factor that is suitable for the specific data type specified in the clause. This means that in a scenario with mixed data lengths, the compiler will select a vectorization factor according to the number of elements of the specified data type that fits into a physical vector register.

In the example shown in Listing 4.8, the innermost loop contains operations on `float` and `unsigned char` data types. Given 64-byte vector registers, if we use `vectorlengthfor(float)`, the compiler will use a vectorization factor of 16. Otherwise, if we specify `vectorlengthfor(unsigned char)`, the compiler will use a vectorization factor of 64. In the first case, operations on `unsigned char` data types will misspend 48 bytes of the vector register. However, in the last case, the larger vectorization factor requires to use 4 vector register per `float` operation. This could significantly increase the register pressure when the number of vector operations with the larger data type is high.

### 4.4.5 The *reduction* clause

The *reduction* clause is already defined for parallel and worksharing constructs in OpenMP 3.1. It enables the computation of a specific reduction operation on a variable or set of variables across multiple thread within a parallel or worksharing construct. We extend the meaning of the *reduction* to the context of our SIMD constructs.

```
#pragma omp simd mask
float max(float a, float b)
{
    return a < b ? b : a;
}

#pragma omp simd
for(i=0; i<N; i++)
{
    if (b[i] < 0.0f)
        b[i] = max(a[i], b[i]);
}
```

Listing 4.9: Example of the *mask* clause

The syntax of the clause is depicted in Figure 4.8. The *reduction* clause in a SIMD construct creates a private copy per vector lane of each reduction variable specified in the `variable_list` field. At the end of the annotated code, these partial values per lane are reduced using a combining `operator`.

Figure 4.9 illustrates the use of the *reduction* clause with two examples. If the SIMD construct is a standalone SIMD directive, as in Figure 4.9a, the result of the previous reduction operation across vector lanes is the final result of the reduction operation. In this case, reduction variables must be local to the thread, as depicted in Figure 4.9a, line 7. Therefore, the reduction operation only takes place at vector register level and not at thread level. If the SIMD construct is a SIMD-for directive, as in Figure 4.9b, after the reduction across vector lanes, a reduction across threads is performed, as occurs with parallel and worksharing reductions in OpenMP. In this case, reduction variables must be shared by all threads involved in the reduction process, as shown in Figure 4.9b, line 3.

### 4.4.6 The *mask* and *nomask* clauses

When the compiler vectorizes a SIMD-annotated function, by default it has to emit at least two vector versions of that function. The first version is a regular vector version of the scalar function where all vector lanes are enabled and they all execute the function call. The second version is a predicated vector version of the scalar function that assumes that there could be vector lanes disabled that do not execute the function call. Depending on the SIMD architecture, the compiler could implement this second version adding an extra parameter to the function that describes which lanes are enabled and disabled at the moment of the function invocation.

During the vectorization of a SIMD code region with SIMD-enabled function calls, the compiler will emit calls to the appropriate SIMD function. If the function call will be executed by all vector lanes, the compiler will use the regular non-predicated SIMD version. Otherwise, if the function call is within a conditional branch or predicated region of code that might not be executed by all the vector lanes, the compiler will use the predicated version of the function.

Nevertheless, it is possible that both SIMD function versions are generated but one of them is not called at any point of the application. This useless version could unnecessarily increase the compilation time and the size of the binary file.

To address this issue, we propose the *mask* and *nomask* clauses. They are aimed at preventing the compiler to generate one particular version of a SIMD-annotated function. If neither of these clauses is used in conjunction with the SIMD directive on the target function, the compiler will generate both the regular and the predicated vector version of the function. If the *nomask* clause is specified, the compiler will only emit the regular non-predicated vector version. Otherwise, if the *mask* clause is used, the compiler assumes that the function will only be invoked from predicated or divergent regions of code and only the predicated vector function will be generated.

Listing 4.9 shows an example where the function `max` has been annotated with the *mask* clause because it is only used within a conditional branch in the SIMD for-loop. Therefore, the compiler will only generate the predicated vector version of the function, avoiding the generation of the regular vector version that will never be called. This can potentially reduce the compilation time and the size of the binary file.

### 4.4.7 The *unroll* clause

Loop unrolling [3, 81] is a well-known loop optimization that consists of generating a new version of the loop with a loop body that computes multiple iterations (unroll factor) of the original loop. This loop optimization is intended to increase the instruction-level parallelism (ILP) as the unrolled loop contains more instructions in the loop body that can be reordered. In addition, the unrolled loop computes a lower number of iterations which reduces the overhead of the loop control instructions.

We define the clause *unroll* in the context of our SIMD extensions. This clause is aimed at giving the loop unrolling optimization a SIMD semantic in the context of the SIMD directives. In this way, when the *unroll* clause is used with a SIMD construct, the compiler will unroll the loop *after* the vectorization process. This means that the unrolled iterations will be vector iterations and not scalar iterations. If the SIMD construct is a SIMD-for, the loop unrolling will take place *before* the scheduling of iterations among threads. This will increase the amount of work assign to each thread with respect to not using the *unroll* clause. Specific chunks of iterations refers then to iterations after the unrolling transformation, in the same way as happens with chunks of iterations in the SIMD-for construct (see Section 4.3.1).

Figure 4.10 describes the syntax of the *unroll* clause. The optional parameter `constant-integer` is the number of vector iterations to unroll after the vectorization of the code (unroll factor). If this parameter is not specified, the unroll factor will be chosen by the compiler (implementation defined).

```
unroll [(constant-integer)]
```

Figure 4.10: Syntax of the *unroll* clause

```
float *a, *b, *c;

#pragma omp simd for unroll(4)
for(i=0; i<1600; i++)
{
    a[i] = b[i] + c[i];
}
```

Listing 4.10: Example using the *unroll* clause

Listing 4.10 extends the example from Section 4.3 with an *unroll* and an unroll factor of 4. Assuming a vectorization factor of 16, the vector version of the loop will have 100 iterations after the vectorization process. Then, this vector loop will be unrolled 4 times, resulting in a final loop with 25 vector iterations that will then be scheduled among threads.

### 4.4.8 The *prologue* and *epilogue* clauses

We propose the *prologue* and *epilogue* optional clauses for SIMD constructs applied on for-loops. These clauses allow programmers to have more control on the execution of the prologue and the epilogue loops generated in the vectorization process. This control comprises the vectorization strategy applied on these loops and how their iterations are scheduled among threads.

Figure 4.11 shows the syntax of both clauses. We define two kinds of execution policies: the *SIMD* policy (`simd-policy`) and the *thread* policy (`thread-policy`). The SIMD policy can take the following values:

- `simd`: the prologue/epilogue loop should be vectorized.
- `scalar`: the prologue/epilogue loop should be kept scalar.

In addition, one of the following thread policies can be optionally specified after the SIMD policy:

- `single`: the prologue/epilogue loop will be executed by the first thread that reaches the loop.
- `integrated`: the iterations of the prologue loop, the main vector loop and the epilogue loop will be seen as the iterations of a single loop. The iterations of the prologue loop will be placed before the iterations of the main vector loop. The iterations of the epilogue loop will be placed after the iterations of

```
prologue | epilogue (simd-policy [, thread-policy])
```

Figure 4.11: Syntax of the *prologue* and *epilogue* clauses

```c
float *a, *b, *c;

#pragma omp simd for prologue(scalar, single) epilogue(simd, integrated)
for(i=0; i<1600; i++)
{
    a[i] = b[i] + c[i];
}
```

Listing 4.11: Example using the *prologue* and the *epilogue* clauses

the main vector loop. The corresponding thread scheduling strategy will be applied on this integrated set of iterations.

- `worksharing`: the prologue loop, the main loop and the epilogue loop will be seen as independent loops. The prologue/epilogue loops will be executed in parallel with the same thread worksharing as the one used for the main loop.

When a standalone SIMD directive contains a *prologue* and/or *epilogue* clause, the only valid thread policy is `single` as only one thread is involved in the execution of that region of code. In this way, the `integrated` policy and the `worksharing` policy can only be used in a SIMD-for construct. It is also important to note that the `single` policy in a SIMD-for construct with static scheduling could not preserve the same scheduling of iterations among loops with the same number of iterations. This is critical when the `nowait` clause is used to prevent synchronization between loops with dependences.

Listing 4.11 shows an example of use of the *prologue* and *epilogue* clauses. According to the SIMD and thread policies used, the prologue loop will not be vectorized and will be executed by the first thread that reaches the loop. The epilogue loop will be vectorized and its iterations will be executed by the thread that computes the last iterations of the main loop.

### 4.4.9 The *nontemporal* clause

Non-temporal streaming stores refer to those memory access patterns that only issue memory writes and do not read that memory position neither before nor after such write shortly [85]. Therefore, caching non-temporal streaming stores could impair performance since they pollute the cache hierarchy with data that does not exploit temporal locality.

To palliate this issue, some architectures, such as our target Intel Many Integrated Core (MIC) architecture, introduce special instructions to perform non-

```
nontemporal(variable_list[:ntem_attributes_list])
```

Figure 4.12: Syntax of the *nontemporal* clause

```
#pragma omp simd nontemporal(z:relaxed,evict)
for(i=0; i<N; i++)
{
    z[j] = a * x[j] + y[j];
}
```

Listing 4.12: Example of the *nontemporal* clause

temporal stores on aligned memory accesses. Unlike regular stores, stream store instructions write data directly into the last cache level or even main memory. In this way, they avoid polluting the smallest cache levels with non-temporal data. In addition, they do not perform a previous read of the cache lines involved in stream stores as the old data will be overwritten.

The *nontemporal* clause, firstly introduced by Intel [85], allows telling the compiler which vector write accesses will be non-temporal. We add this proposal to the context of the OpenMP SIMD constructs and extend it to be able to specify a more fine-grained information about the non-temporal stores.

The syntax of our clause is described in Figure 4.12. As depicted, it is possible to use optional attributes (`attributes_list`) to complement the list of variables stated as non-temporal (`variable_list`). So far, we define the following two attributes that can be used together or individually:

- `relaxed`: states that non-temporal data can be stored using a relaxed memory consistency model (weakly-ordered stores).
- `evict`: instructs the compiler to evict non-temporal data from any cache level after performing the non-temporal streaming store.

These two attributes are motivated by the Intel MIC architecture, where there are specific instructions with this functionality, as we described in Section 2.5. With this clause, we offer programmers an easy alternative to use these features on each specific store of their code.

Listing 4.12 shows an example where write accesses to array `z` are explicitly defined as relaxed non-temporal stores with cache eviction. Consequently, targeting the Intel MIC architecture, the compiler will perform a relaxed non-temporal vector store followed by an eviction of the involved cache line for each `z[j]` vector store of the loop.

```
prefetch(variable_list[:l1-dist[,l2-dist...][,pref-policy]])

    noprefetch(variable_list[:l1-dist[,l2-dist...)]]
```

Figure 4.13: Syntax of the *prefetch* and *noprefetch* clauses

### 4.4.10 The *prefetch* and *noprefetch* clauses

Software prefetching is an optimization technique that consists of introducing special instructions to request data from memory before this data is actually needed by the computational units of the processor. This optimization can be applied automatically by means of compiler techniques and it is increasingly becoming an optimization closely related with vectorization, as we discussed in Section 3.5.10.

We introduce two clauses in the context of our SIMD proposal for OpenMP. These clauses are named *prefetch* and *noprefetch* and they are inspired in the previous work of Intel [85]. They allow directing the compiler on the software prefetching code generation for SIMD-annotated for-loops when the underlying architecture supports it. As we stated in Section 3.5.10, software prefetching is increasingly related to SIMD instructions. For that reason, we propose these clauses in the context of our SIMD proposal. However, this problem could be addressed independently from SIMD and vectorization.

Figure 4.13 shows the syntax of these clauses. The *prefetch* clause enables software prefetching on a set of pointer variables (`variable_list`) whose accesses want to be prefetched. The *noprefetch* clause disables the software prefetching for the set of pointer variables specified.

The `prefetch` clause allows optionally describing a more accurate prefetching strategy. Programmers can specify prefetching distances (loop iterations ahead) for each particular level of the memory hierarchy following an order from closer to farther memory levels of the core processor (`l1-dist`, `l2-dist`, and so on). These distances are in units of vector loop iterations after the whole SIMD transformation, including any transformation produced by the `unroll` clause. In addition, programmers can specify a software prefetching policy (`pref-policy`) from the following three policies that we define:

- `auto`: The compiler will chose the most appropriate software prefetching strategy for each particular target vector access.
- `in-place`: The compiler will place the software prefetching instructions as close as possible to the target vector access.
- `on-top`: The compiler will place all the software prefetching instructions on the top of the loop body.

The default prefetching policy is `in-place`. The `on-top` policy can be useful in loops with a large body where issuing prefetching instructions of memory positions

```
#pragma omp simd nontemporal(z) prefetch(x,y:4,16,in_place) noprefetch(z)
for(i=0; i<N; i++)
{
    z[j] = a * x[j] + y[j];
}
```

Listing 4.13: Example of the *prefetch* and *noprefetch* clauses

that are going to be used in the same loop iteration (distance 0) could be beneficial.

Listing 4.13 shows an example of the clauses *prefetch* and *noprefetch*. In this code, the compiler will generate software prefetching instructions for accesses on arrays x and y. The compiler will not generate prefetching instructions for accesses on array z. We use the *noprefetch* clause on z appropriately because vector stores on this array are also qualified as non-temporal.

## 4.5 User Guidelines

In this section, we introduce some hints to guide programmers on the application of the SIMD extensions in our proposal. Since OpenMP provides a model for parallel programming, potential programmers are expected to have a minimum background in parallelism and performance analysis. In addition, some notions in vectorization are also required in order to take advantage of the SIMD extensions of our proposal. For the sake of simplicity, we do not consider other kinds of parallelism besides the data parallelism exploited by the OpenMP SIMD clause.

The first step when we want to make use of a SIMD construct, and perhaps the most relevant in terms of performance, is to choose the most appropriate loop to vectorize. The innermost loop is the traditional option for many auto-vectorizing compilers [81]. However, outer loop vectorization has been proven to yield more performance in some codes when outer loops offer a large amount of data parallelism and data locality than innermost loops [111]. In this way, programmers may consider the following hints to decide which potentially can be the most appropriate loop to be annotated with a SIMD directive:

- Loops that traverse the contiguous memory dimension of basic array data structures in a stride-one fashion are generally better options. These memory access pattern should produce stride-one vector memory accesses instead of vector gather/scatter instructions.

- Loops with a large trip-count will maximize the number of vector iterations of the loop and will minimize the impact of normally less efficient epilogue loop codes. Vectorizing a loop that does not have enough iterations to fill a whole vector registers could yield lower performance than the scalar version of the code in some architectures.

- If both the innermost and the outer loops satisfy the previous hints, vectorizing the outer loop is usually the most suitable option. The reason is that vectorizing the outer loop will bring vectorization to a larger number of instructions, thus increasing the amount of SIMD parallelism exploited.

In some cases, the only way to find the vectorization strategy that offers the best performance is benchmarking the most suitable alternatives. This is one of the greatest benefits of our SIMD proposal. It is only a matter of moving the SIMD construct from one loop to another, recompiling and measuring the performance of a totally different vector version of the code.

## 4.6 Legacy and Relationship of SIMD Extensions

As a result of this work and in collaboration with Intel [83], OpenMP was officially extended with SIMD constructs in its 4.0 version released in July of 2013. Table 4.1 shows a comparison of our SIMD proposal, the Intel proposal for C/C++ [147] and the final proposal implemented in the OpenMP standard [117].

It is important to note that all the directives and clauses included in our SIMD proposal are not a novel contribution of this thesis. Checkmarks in Table 4.1 indicate that a directive or clause is available in a proposal but it is not novel from that proposal. When the name of the directive or clause is explicitly indicated in a proposal, it means that it is novel from that proposal. If the name of the directive or clause is shown in multiple proposals, it means that the work was developed dependently in both proposals, reaching the same or very similar semantic. Crossmarks mean that the directive or clause is not included in that proposal.

To the best of our knowledge, the idea of user-directed vectorization of loops was first introduced by Krzikalla et al. in Scout [87, 86]. Intel and we were working independently on some basic aspects of this field before the SIMD extensions for OpenMP 4.0 were published.

We presented an initial prototype of the standalone SIMD directive for loops and functions as Master Thesis [21]. Intel presented a more extended proposal with clauses such as `linear`, `uniform`, `mask` and `nomask` [147] that we incorporate to our proposal. Intel and we presented some results and shared ideas with the OpenMP committee and this resulted in a joint proposal [83]. Some of the clauses in our proposal (`suitable`, `unroll`, `prologue`, `epilogue`) are in the process of being presented to the OpenMP committee to study if they are interesting from the point of view of the standard.

Currently, OpenMP 4.0 incorporates the standalone SIMD directive for for-loops and it includes the directive *declare simd* for the same purpose on function declarations/definitions. Moreover, another construct combines the semantic of the SIMD directive with the semantic of the worksharing *for* construct in OpenMP. However, this combined construct is significantly different from our SIMD-for construct. In the case of OpenMP, a for-SIMD construct is included. This *for simd* directive (note

| Our Proposal | Intel C/C++ proposal | OpenMP 4.0 |
|---|---|---|
| `#pragma omp simd` for for-loops | `#pragma simd` for for-loops | ✓ |
| `#pragma omp simd` for functions | `__attribute__(vector)` for functions | `#pragma omp declare simd` for functions |
| `#pragma omp simd for` | ✗ | `#pragma omp for simd` |
| `aligned` and `reduction` | `aligned` and `reduction` | ✓ |
| ✓ | `linear` and `uniform` | ✓ |
| ✓ | `mask` and `nomask` | renamed to `inbranch` and `notinbranch` |
| `vectorlength` and `vectorlengthfor` | `vectorlength` and `vectorlengthfor` | ≈ `safelen` (not exactly the same semantic) |
| `suitable`, `unroll`, `prologue` and `epilogue` | ✗ | ✗ |
| ✓but with extended functionality | `nontemporal`, `prefetch` and `noprefetch` | ✗ |
| ✗ | data sharing attributes: `private`, `firstprivate` and `lastprivate` | ✓ |
| ✗ | ✗ | interaction with other OpenMP clauses: `collapse` |

Table 4.1: Comparison of our proposal, the Intel C/C++ proposal [147] and the SIMD extensions included in OpenMP 4.0 [117]. Checkmark means that the directive/-clause is included in that proposal but it is inspired by other proposal. Crossmark means that the directive/clause is not included in that proposal. Same directive/ clause in multiple proposals without checkmark means independent development

that it is not *simd for* as in our proposal) instructs the compiler to firstly schedule the scalar iterations among threads and then to vectorize those scalar iterations. The alternative SIMD-for construct that we described in Section 4.3 does exactly the opposite: firstly the scalar iterations are vectorized and then the resulting vector iterations are scheduled among threads.

Regarding optional clauses, *aligned* and *reduction* have been included as part of the OpenMP 4.0 standard with similar meaning. The *vectorlengh* clause has been renamed to *safelen* and *mask* and *nomask* clauses to *inbranch* and *notinbranch*, respectively. Clauses *uniform* and *linear* have been incorporated with slightly more restrictive meaning. The uniform and linear properties are defined in terms of the iterations of the SIMD loop in OpenMP and not in terms of the vector lanes within the vector register as in Intel's and our proposal. This means that uniform and linear properties must be fulfilled within and across vector registers in OpenMP whereas they must be only fulfilled within vector registers in Intel's and our proposal.

In addition, OpenMP 4.0 also defines some extra SIMD clauses. They are more related with data sharing attributes (*private*, *firstprivate* and *lastprivate*) and other clauses already in the standard, such as the *collapse* clause.

For comparison, we include the OpenMP SIMD extensions in our evaluation and discuss further this topic in Section 4.8.2.

## 4.7 Evaluation

We implemented a prototype of our proposal in the Mercurium source-to-source Compiler, introduced in Section 2.3. This prototype makes use of the vectorization infrastructure widely described in Chapter 3.

### 4.7.1 Goals and Interpretation of Results

The main goal of this evaluation is to know how SIMD directives and clauses from our proposal impact the performance versus the scalar version of the code and the compiler automatic vectorization approach.

As we stated in Section 4.6, our SIMD proposal laid down the basis of the SIMD extensions released in OpenMP 4.0. The OpenMP 4.0 SIMD proposal is already supported by the last stable version of the Intel C Compiler (15.0.2) at the time of writing this thesis. However, we found that this support is still in an early stage in this version of the compiler. Some clauses still do not work appropriately and their evaluation could lead to wrong conclusions. Consequently, we only include performance results of the OpenMP 4.0 SIMD directives (with no clauses) implemented in the Intel C/C++ Compiler. In no case, we want to fully compare the performance from both approaches and compilers. Results obtained with OpenMP 4.0 SIMD extensions are aimed at giving credibility to the performance achieved with the Mercurium compiler and our SIMD approach. In addition, we want to illustrate which similarities and differences exist between both approaches and their impact on performance. Performance results must be understood in the context of a small research source-to-source compiler (the Mercurium compiler) and a full production compiler (the Intel C/C++ compiler).

### 4.7.2 Benchmarks

We decided to use a small set of benchmarks for our evaluation instead of including a large set that offered redundant results. Dealing with a small number of benchmarks allow us to analyze and describe in depth all the vectorization details that take place for each code and how they affect the execution time. We evaluate the following 6 benchmarks:

**Kirchhoff:** The Kirchhoff benchmark implements a seismic migration method that uses the Kirchhoff equation to reconstruct a 2D earth sub-surface image using a set of simulated traces as input.

**Distsq:** The Distsq benchmark is a simple but widely used kernel that computes the square distance of two points and compares the result with another input distance, choosing the minimum distance as output.

**Cholesky:** The Cholesky benchmark computes the Cholesky decomposition of an input square matrix. This matrix is decomposed into the product of two matrices: a lower triangular matrix and its conjugate transpose.

**Nbody:** The Nbody benchmark simulates the interaction of a set of particles or *bodies* under the influence of physical forces. This algorithm is used in astrophysics to simulate the motion of celestial objects and how they interact with each other.

**Fastwalsh:** The Fastwalsh benchmark calculates the Fast Walsh–Hadamard transform (FWHT). The Walsh-Hadamard transform (WHT) is a generalization of the Fourier transform using a square matrix of $2^N$ elements called a Hadamard matrix. This transform has application in the field of data encryption and compression algorithms. The FWHT is an efficient algorithm to compute the WHT with a $O(NlogN)$ complexity (compared to the naive $O(N^2)$ algorithm) that follows a divide and conquer algorithmic schema.

**Mandelbrot:** The Mandelbrot algorithm computes the Mandelbrot set of complex numbers. This set comprises the sequence of numbers resulting from applying iteratively a mathematical operation on an initial complex value. Only those numbers that do not diverge to infinity are considered as part of the Mandelbrot set. The Mandelbrot benchmark iterates on a discretized region of the complex space. It computes the number of iterations that it takes for each discretized point to diverge to infinite (threshold).

Table 4.2 summarizes the input configuration, the data types and the memory footprint of each particular benchmark. We utilize input sizes that widely exceed the size of the L2 cache (512 KB) cache of a single core. The execution time using these input sizes takes several minutes using our thread configuration defined in the Section 4.7.3.

| Benchmark | Input Size | # Tsteps / Runs | Data Types | Memory Footprint |
|---|---|---|---|---|
| Kirchhoff | $512^2$ | 10 | `int` & `float` | $\sim$ 2 MB |
| | $1{,}024^2$ | 3 | | $\sim$ 8 MB |
| Distsq | 6,400,000 | 100 | `float` | $\sim$ 98 MB |
| Cholesky | $1{,}600^2$ | 5 | `float` | $\sim$ 10 MB |
| Nbody | 8,192 | 10 | `float` | $\sim$ 354 KB |
| Fastwalsh | $2^28$ | 1 | `float` | $\sim$ 2 GB |
| Mandelbrot | $1{,}600^2$ 500 | 10 | `int` & `float` | $\sim$ 9 MB |

Table 4.2: Benchmarks setup

### 4.7.3 Environment and Methodology

The Intel Xeon Phi coprocessor (Knights Corner) with its 512-bit vector units is the target architecture in our experiments. Further details about this many-core architecture are discussed in Section 2.3. For our evaluation, we use the set of benchmarks described in Section 4.7.2. For each benchmark, we measure the execution time of the following versions:

**scalar icc:** This is the scalar version of the benchmark. The code does not contain any SIMD construct. It is compiled with the Intel C/C++ Compiler. The Intel C/C++ Compiler auto-vectorization is disabled.

**auto-vec icc:** This is the auto-vectorized version of the benchmark. The code does not contain any SIMD construct. It is compiled with the Intel C/C++ Compiler. The Intel C/C++ Compiler auto-vectorization is enabled.

**simd icc:** This is the version that makes use of the OpenMP 4.0 SIMD constructs. The code contains standalone SIMD directives and/or for-SIMD directives, as appropriate. It is compiled with the Intel C/C++ Compiler.

**simd mcc:** This is the version that uses our SIMD proposal. The code contains standalone SIMD directives and/or SIMD-for directives, as appropriate. It is compiled with the Mercurium C/C++ source-to-source compiler using the Intel C/C++ Compiler as a native/backend compiler.

**simd mcc + clauses:** This version is similar to the previous *simd mcc* version but it contains optional SIMD clauses from our proposal.

We use the same source code in all these versions, except for the SIMD annotations. Furthermore, all these versions, including the scalar one, contain the same OpenMP *parallel* and *for* constructs that exploit fork-join parallelism. The version of the Intel C/C++ Compiler is the 15.0.2 in all our experiments.

We run all the experiments with 4 threads, using one single core of the architecture. A small number of threads allows us to evaluate the quality of the vector code of each version without any interference from memory issues, very common in this massively parallel architecture.

By default, the only flags that we use for all versions at compile time are `-O3 -fopenmp`. We do not alter the inlining of functions at compile time unless the contrary is specifically stated for a particular version of the benchmark. However, we preserve the original single file or multi-file structure of the benchmark that may have an impact on this optimization.

Our evaluation mainly focuses on the execution time of each version of the benchmarks. We report the speed-up over the scalar version of the code based on the mean of 10 executions. For some benchmarks, we also evaluate how some of the SIMD clauses impact on the binary code size. The size of this binary code is measured with the Unix tool `size`.

### 4.7.4 Results

#### Kirchhoff

The main kernel code of the Kirchhoff benchmark is included in the Appendix A, Listing A.1. The inputs of this benchmark are an array with the earth `traces`, the `width` and `height` of the traces, and a set of components involved in the computation (`velocity`, and three delta components). The output is an array with the result of the `model`.

The computation in this benchmark consists of three nested for-loops that we denote as *outermost*, *intermediate* and *innermost* loop, respectively. For each output value of the `model`, the algorithm gathers several elements from the `traces` and reduces the selected elements with an addition operation. The following equation is a simplified formula that describes the evolution of the memory reads on the array `traces`:

$$(a^2 - 2ab + b^2 + c^2) * width + a \tag{4.1}$$

In this formula, $a$, $b$ and $c$ are the induction variables of the innermost, the intermediate and the outermost loop, respectively.

In all our experiments, we used a *for* directive to parallelize the outermost loop. This approach offers the best performances in comparison to the parallelization of the intermediate or the innermost loop.

The 3-loop nest does not have any data dependence that could prevent vectorization in any of the three loops. In this way, we explore the following three different approaches of vectorization using SIMD constructs:

- vectorizing the innermost loop with a standalone SIMD directive and a `reduction` clause on the `tsum` variable (versions `simd icc inner` and `simd mcc inner`).

- vectorizing the intermediate loop with a standalone SIMD directive (versions `simd icc interm` and `simd mcc interm`).

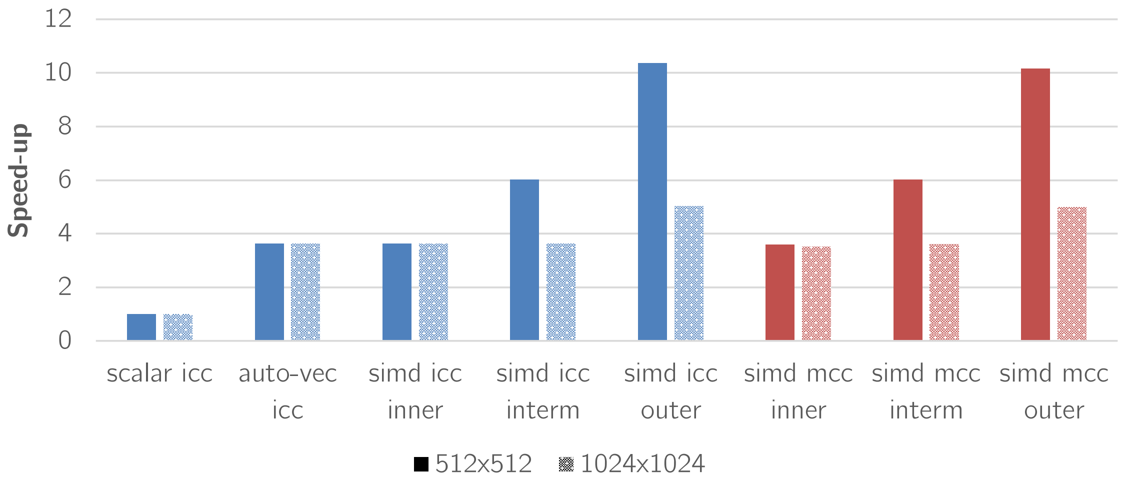


Figure 4.14: Speed-up of the Kirchhoff benchmark (`float` data types)

- vectorizing the outermost loop with a combined for-SIMD/SIMD-for directive (versions `simd icc outer` and `simd mcc outer`).

We do not use any extra SIMD clause apart from the `reduction` clause in the innermost loop version that it is needed to describe the reduction operation. Figure 4.14 shows the speed-up factor of all the evaluated versions obtained over the scalar version of the code with two different problem sizes: traces of 512x512 and 1024x1024 elements. As we can see, the Intel compiler yields 3.60 gain in the auto-vectorized version where the compiler chooses the innermost loop to be vectorized. This version yields the same performance as `simd icc inner`, where we use the SIMD clause on the same loop. This means that the Intel compiler is able to auto-vectorize the innermost loop efficiently without additional help. The Mercurium compiler offers a similar performance when using the SIMD directives on the innermost loop (`simd mcc inner`). This means that Mercurium vectorization capabilities are competitive on this benchmark in comparison with the Intel compiler.

The 3.60 speed-up that results from vectorizing the innermost loop is relatively low for the 16-element vector length of the architecture. However, vectorizing this loop requires expensive vector instructions such as a gather operation on the array `traces`, a call to the vector version of the math function `sqrtf` and a horizontal vector addition operation at the end of the loop to reduce the vector version of the variable `tsum` to a single scalar value. According to Equation 4.1, vectorizing the innermost loop, i.e., increasing the innermost induction variable $a$ while maintaining $b$ and $c$ constant per vector lane, leads to the vector gather instruction with the most dispersed scalar elements. This fact increases the latency of the gather instruction. Furthermore, the write on the array `model` and other operations are not vectorized as they are outside of the innermost loop.

Vectorizing the intermediate loop also generates a vector gather instruction on the array `traces`. This gather reads scalar elements that are a bit less dispersed than in the innermost loop version, since we are increasing $b$ and keeping constant $a$ and $c$ per vector lane in Equation 4.1. This could reduce the number of cache lines involved in the gather operation. In addition, the horizontal reduction operation is no longer necessary. In this version, multiple independent reductions are now computed in a vector way for every value of the intermediate loop induction variable `ix`. Furthermore, the write on the array `model` is now vectorized with a stride-one vector store operation.

The vectorization of this intermediate loop (`simd mcc interm`) offers striking performance results in Figure 4.14. This version yields 6.0 times more performance than the scalar version for the small input. This speed-up outperforms the 3.60 gain of the innermost loop version. However, performance results are the same for both versions when the larger input size is used. The reason behind this performance difference among both input sizes is that a larger input size increases the dispersion in the gather instruction and, therefore, the latency of the operation. Thus, even vectorizing a larger piece of code, the latency of the gather instruction dominates the execution time.

The outermost loop vectorization yields the best performance with more than 10 and 5 speed-up for the small input and the large input, respectively. This gain is due to that this vectorization approach leads to the gather instruction with the closest elements according to Equation 4.1. In this case, we increase $c$, keeping $a$ and $b$ constant per vector lane. Again, the performance difference between both input sizes is due to that the larger input data size impacts more negatively in the dispersion of the elements read by the gather instruction. Furthermore, the `model` memory write is vectorized using a vector scatter instruction that performs a column-wise access pattern. Although this instruction is more expensive than the stride-one vector store used in the intermediate loop version, it does not penalize that much the performance. Accessing to memory column-wise only entails cache misses for the iteration that accesses the first element of the cache line, and cache hits for the following iterations that access the same cache line.

This detailed analysis of the different version strategies shows some factors that the compiler should take into account to choose the most suitable loop to be vectorized in this simple code. Even if the compiler carried out this analysis, the best performance and, consequently, the best vectorization strategy could depend on the input data set, as our results suggest. In addition, other OpenMP constructs, such as the *for* directive used in this benchmark, will prevent the compiler from changing the semantic of the execution specified by the programmer. This would lead to discarding the outermost loop vectorization which is the strategy that offers better performance. All these reasons justify the need of our proposal on SIMD extensions for OpenMP.

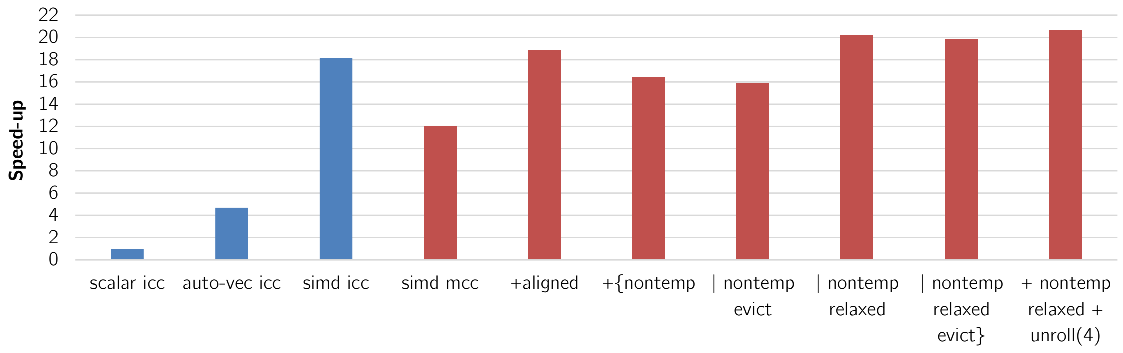


Figure 4.15: Speed-up of the Distsq benchmark (`float` data types)

## Distsq

The source code of the Distsq benchmark is shown in the Appendix A, Listing A.2. This benchmark receives three arrays of floating point elements (`a`, `b` and `c`) and returns another array (`d`) with the result of the computation. There are two functions, `distsq` and `min`, that return the square distance and the minimum value of two floating points variables, respectively. Using these functions, the algorithm computes the square of the distance between each element of arrays `a` and `b`, it compares that value with the corresponding element of the array `c` and it writes the minimum one into the array `d`.

We use this simple benchmark as a first evaluation of the SIMD directive applied to functions. We parallelize the main loop of the benchmark with a *for* directive in all its versions. In the SIMD-annotated versions, we make use of the combined for-SIMD/SIMD-for compiled with the Intel compiler (`simd icc`) and the Mercurium compiler (`simd mcc`), respectively. Moreover, we accumulatively add the clauses *aligned*, *nontemporal* and *unroll*, in this order, to measure their impact on the resulting vector code. We test several configurations of the *nontemporal* clause, using combinations of their *relaxed* and *evict* parameters.

Figure 4.15 shows the speed-up of the different vectorized versions over the scalar version of the code. The auto-vectorized version improves the scalar performance by a modest factor of 5, even though `distsq` and `min` functions are inlined by the compiler. The improvement is moderate because the compiler cannot guarantee that the pointers to arrays do not alias. This fact leads the compiler to emit several loops with scalar and vector versions of the loop. The appropriate version is chosen at runtime with extra code that checks whether pointers do not alias and the vector version can safely be executed.

Versions with the SIMD construct allow the compiler to get rid of these dependence checks at runtime and to generate a more efficient code. These versions yield 18.15 and 12.00 speed-up when they are compiled with the Intel compiler and the Mercurium compiler, respectively. This difference in performance is due to that the Intel compiler generates a prologue loop that is able to align vector memory accesses within the main vector loop. On the contrary, the Mercurium compiler does not have this technology and generates unaligned vector memory accesses for all memory accesses.

The *aligned* clause allows Mercurium to generate aligned vector memory accesses that increase the speed-up to 18.85, which is comparable to the previous Intel compiler performance. For this simple case, the Intel compiler alignment algorithm is enough to reach an outstanding performance only using the SIMD construct without any extra clauses. This goes in favor of programmability. However, this approach is not enough in more complex codes, for instance, with multi-dimensional arrays. Multi-dimensional arrays entail the analysis at runtime of several elements of the access, such as the base pointer and the size of the dimensions which may be different among different arrays. These facts turns the number of checks unaffordable at runtime. Therefore, the compiler has to opt by sub-optimal and lighter approaches that can be outperformed with the combined use of the *aligned* and the *suitable* clauses, as we will see in the forthcoming benchmark evaluation.

We use the *nontemporal* clause to instruct the compiler to generate nontemporal stream stores on the array `d`. This clause allows us to find out that the only option that boosts performance over a speed-up of 20 is the use of stream stores with relaxed memory consistency and no eviction. Just changing the parameters of the clause, we can discard the use of regular nontemporal stores and the cache eviction approach because they worsen memory usage. The speed-up obtained with this clause is moderate because we are running our experiments on a single core. This configuration does not saturate the whole memory hierarchy as much as when using the 61 cores available in the Intel Xeon Phi architecture. Stream stores will potentially have a greater impact on performance when the application uses more cores.

Regarding the *unroll* clause, we tested several unroll factors from 2 to 32. Despite of the fact that loop unrolling is increasing the granularity of the chunk of the scheduling and reducing the loop control instructions, we only achieve better performance, and by a marginal amount, with an unroll factor of 4. This speed-up is marginal because this architecture executes instructions in order. Therefore, a greater basic block does not increase instruction-level parallelism at runtime. In addition, the throughput issue of the jump instruction (*nop* instruction immediately after the jump) is hidden by the four hardware thread contexts running concurrently on the same core.

### Cholesky

Listing A.3 at Appendix A depicts the main kernel of the Cholesky benchmark. This version implements a simple in-place Cholesky decomposition without using math libraries. It receives the input square matrix `a` and the size of the matrix (`N`), and returns the two output triangular matrices in the same `a` data structure. The code has a main loop (`#1`) with data dependences across iterations. This loop

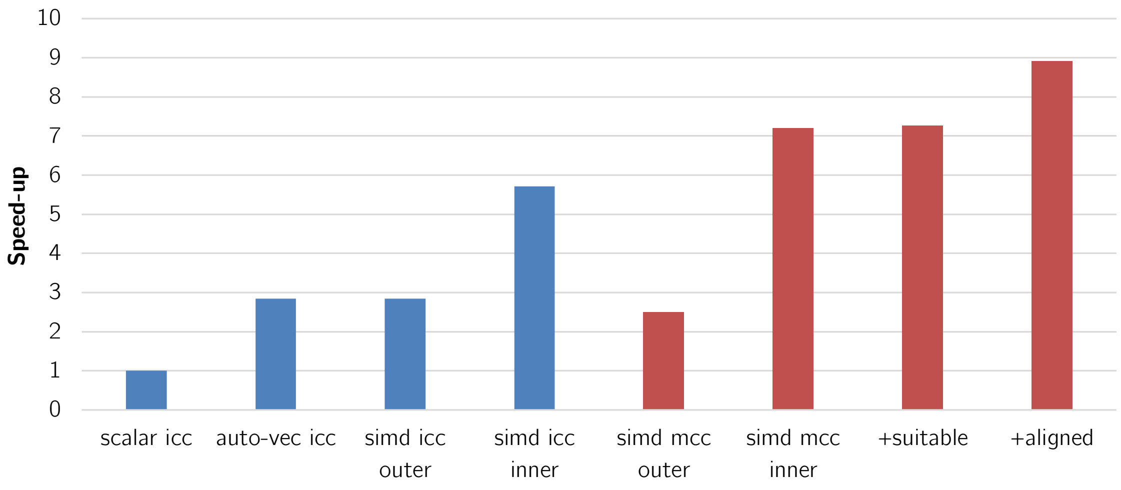


Figure 4.16: Speed-up of the Cholesky benchmark (`float` data types)

nests two loops that do not have data dependences across their own iterations but there are data dependences among some of the loops. We parallelize the first nested loop (#2) with a *for* directive that performs a thread synchronization barrier at the end of the execution. After the loop #2, we use an OpenMP `master` directive to initialize the shared variable `a_jj` with the value of `a[j][j]` (line 11). Then, we parallelize loop #4 with another *for* directive and a `reduction` clause on the shared variable `a_jj`. Once the reduction operation has finished, another *master* directive is used on the statement that computes the square root of the reduction result (line 18). Finally, we parallelize the last nested loop (#5) with a *for* directive. We apply this parallelization approach to all versions of the benchmark, included the scalar version.

Using the previous parallelization strategy, in this benchmark we evaluate several versions using SIMD constructs. Loops #4 and #5 are vectorized using a combined for-SIMD/SIMD-for directive in all the SIMD versions that we evaluate. For loops #2 and #3, we explore two different vectorization approaches:

- vectorizing the outer loop #2 with a combined for-SIMD/SIMD-for directive.
- vectorizing the inner loop with a standalone SIMD directive and a `reduction` clause.

Then, we use the clauses *suitable* and *aligned* onto the previous version that yields the best performance.

Figure 4.16 shows the speed-up results of all the evaluated versions over the scalar version of the code. As we can see, the compiler is able to improve by almost a factor of 3.0 the execution time with auto-vectorization (`auto-vec icc`). However, this speed-up factor is considerably low for a vector length of 16 single precision floating point elements. The compiler chooses to auto-vectorize all the innermost loops, i.e., loops `#3`, `#4` and `#5`. However, multiple versions of each vectorized loop are emitted to consider several alignment possibilities for the single two-dimensional matrix. As we detailed in the analysis results of the benchmark Distsq, multi-dimensional arrays require more checks at runtime to determine if their memory accesses are aligned, and sub-optimal versions has to be adopted. These facts are limiting performance in this version.

When we use SIMD constructs, the outer loop version (vectorization of the loops `#2`, `#4` and `#5`) yields similar performance with both compilers. This performance is also the same as the performance achieved by the auto-vectorized version. The inner loop version (vectorization of the loops `#3`, `#4` and `#5`) yields 5.72 and 7.21 speed-up gain with the Intel compiler and the Mercurium compiler, respectively. When vectorizing the outer loop `#2`, accesses `a[i1][j]` and `a[i1][k]` result in column-wise access patterns, i.e., a vector gather/scatter instruction. In addition, `a[j][k]` requires a scalar-to-vector promotion. On the contrary, the inner loop `#3` vectorization turns `a[i1][k]` and `a[j][k]` into adjacent vector memory accesses and `a[i1][j]` into a reduction operation, which is more efficient in terms of the latency of vector instructions.

In the inner version, our approach significantly outperforms the speed-up of the Intel compiler. The main reason is that both approaches implement the combined SIMD and *for* directive with an important semantic difference. As we detailed in Section 4.6, the implementation in the Intel compiler (for-SIMD directive defined in OpenMP 4.0), distributes scalar iterations among threads and then vectorizes that chunk of iterations. In our approach (SIMD-for directive), scalar iterations are first vectorized and then the resulting vector iterations are distributed among threads. This discrepancy turns noticeable when dealing with triangular loops, as happens with loops `#4` and `#5` of this benchmark.

When the OpenMP 4.0 approach is applied to these triangular loops, some iterations are executed in a less efficient way because there are not enough iterations per thread to fill a whole vector register. Running with 4 threads needs at least 64 iterations (vector length of 16 x 4 threads) to fully fill a vector register per thread. This problem will be much worse for a larger number of threads because more iterations are needed to fill a whole vector register per thread. In addition, this scheduling problem is exacerbated by false sharing problems when the number of iterations of the loop is small.

When our approach is applied, a large number of full vector iterations is executed because we vectorize the code before the scheduling of iterations among threads. This also means that a lower number of threads is involved in the computation when the number of iterations is small. Running with 4 threads needs at least 16 iterations (vector length) to fully fill the vector register of the first threads. 32 iterations will be executed by 2 threads using full vector registers, and so on. In addition, using a large number of threads will not cause false sharing as threads execute full vector iterations from the beginning of the triangular loop execution (the size of the vector register is the same as the size of the cache line).

We choose the inner loop version to study the impact of the *suitable* and *aligned* clauses because it yields better performance than the outer loop version. We use the *suitable* clause to tell the compiler that the size of the matrix (`N`) will be multiple of the vector length at runtime. As we can see in Figure 4.16, this extra information (version `+suitable`) does not have impact on performance because all vectorized loops are triangular and always need and epilogue loop. However, only if we combine the *suitable* clause with the *aligned* clause (version `+aligned`) the compiler is able to optimally generate aligned vector memory accesses without any runtime check. This increases the speed-up to a factor of 8.92 over the scalar version. This performance is acceptable if we have in mind that the parallel strategy requires several synchronization points per iteration.

**Nbody**

The Nbody benchmark uses an AOSOA (Array Of Structures Of Arrays) approach to represent the information of the particles and forces. The main algorithm contains multiple functions that compute the forces involved in the computation and update the position of each particle. We include the main code of this benchmark in Appendix A. Listing A.4 and Listing A.5 show the functions that compute the position update and the forces of particles, respectively. Functions `update_particle_soa` and `calculate_force_soa` implement the basic functionality of the algorithm for a single particle. These functions are invoked from functions `update_particle_block` and `calculate_force_block`, respectively, that iterate on every single particle of the AOSOA block. Finally, two nested loops iterate on all AOSOA blocks of the simulation where the interaction between every pair of particles is computed calling to the latter block functions (this code is not shown).

For all the evaluated versions, we parallelize the outer loop of the two nested loops that iterates on the AOSOA blocks with a *for* directive. Regarding the use of SIMD constructs, we use the standalone SIMD directive on the loops that iterates on every particle within each AOSOA block. Particularly, we vectorize the single loop in function `update_particle_soa`, and the outer loop in function `calculate_force_soa`. As these loops contain calls to the functions that compute the basic functionality of the algorithm per particle (`update_particle_soa` and `calculate_force_soa`), we also annotate these functions with the SIMD directive. Afterwards, we investigate how the clauses *linear*, *uniform* and *aligned* affect the vectorization and, therefore, the performance of the benchmark. Finally, we also study how the use of the clause *nomask* can impact the size of the code reducing the number of version of SIMD functions generated.

Figure 4.17 shows the performance results of the evaluation. As we can see, the Intel compiler decides not to auto-vectorize the code because function calls are not inlined. The reason is that the functions that compute the forces and position per particle are in files separated from the functions that compute these properties for the whole AOSOA block. When we use the SIMD directive on loops and

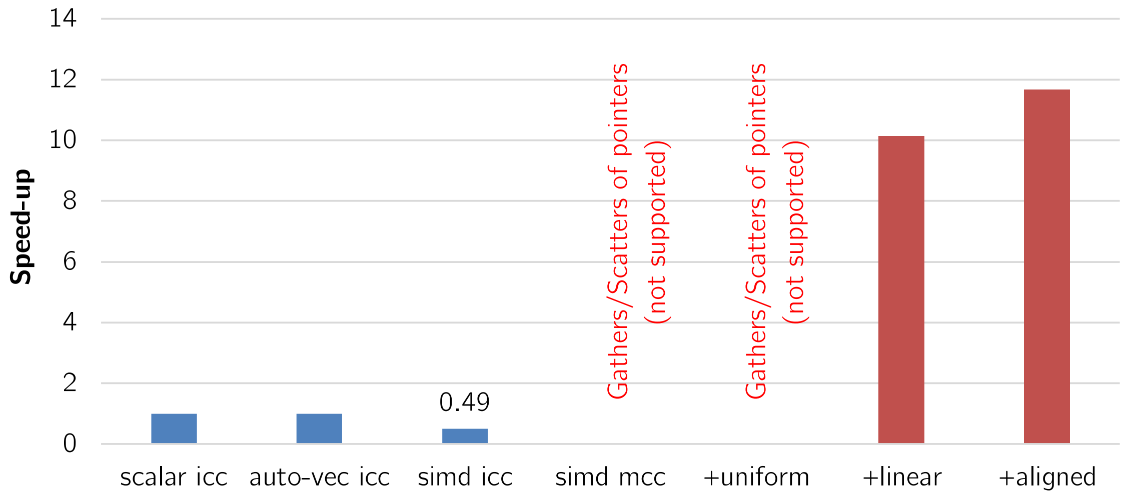


Figure 4.17: Speed-up of the Nbody benchmark (`float` data types)

functions without any other clause, the performance with the Intel compiler (`simd icc`) drops by 51%. This performance degradation happens because the generic vectorization of the functions requires the use of vector gather and scatter of pointers. Gather/scatter of pointers are generic gather/scatter memory operations whose scalar memory accesses do not share the same base pointer. These vector memory operations do not have direct support in hardware on the Intel Xeon Phi coprocessor and they have to be implemented using a very expensive software approach. Currently, Mercurium does not have support to deal with these gather/scatter of pointers. For this reason, it cannot vectorize the code when only the SIMD clause is used (`simd mcc`).

Nevertheless, as we can see in the SIMD-annotated loops that invokes the SIMD-enabled functions, many of the pointers of the code will have the same pointer value across loop iterations. For this reason, we use the *uniform* clause to qualify them as follows:

- function `update_particle_soa`: `px`, `py`, `pz`, `pvx`, `pvy`, `pvz`, `pm`, `fx`, `fy` and `fz`.
- function `calculate_force_soa`: `p1x`, `p1y`, `p1z`, `p1m`, `p2x`, `p2y`, `p2z`, and `p2m`.

Variable `l` from function `calculate_force_soa` is also qualified as *uniform* for the same reason. The use of the *uniform* clause turns previous gather of pointers `p2x[l]`, `p2y[l]` and `p2z[l]` into scalar-to-vector promotions, which is a much faster operation. However, memory accesses `*fx`, `*fy` and `*fz` from `calculate_force_soa` still need scatter of pointers operations. Consequently, the Mercurium compiler cannot vectorize this version of the code yet (`+uniform`).

As a next step, we add the *linear* clause to the previous version of the code to

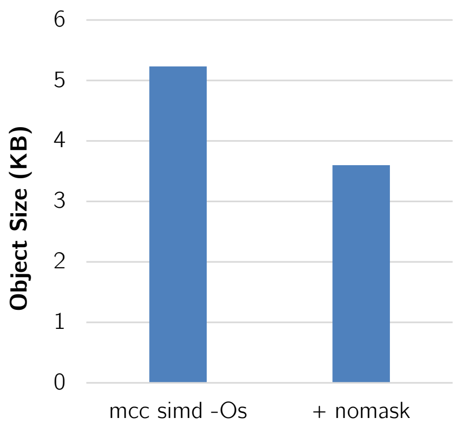


Figure 4.18: Code size variation of the main functions of the Nbody benchmark

qualify those pointers and variables that have a linear evolution when they are passed as arguments of the SIMD-annotated functions. We qualify pointers `*fx`, `*fy` and `*fz` and the variable `k` from `calculate_force_soa`, and variable `i` from `update_particle_soa`. As a result, memory accesses `*fx`, `*fy` and `*fz`, and those accesses indexed with the variable `k` and are now vectorized as stride-one vector loads and vector stores in function `calculate_force_soa`. In the same way, memory accesses indexed with the variable `i` in function `update_particle_soa` turns also into adjacent vector loads/stores. This version yields 10.14 speed-up gain over the scalar version. This improvement is striking if we take into account that the use of the SIMD directive without any clause entails a 51% slowdown.

When we add the *aligned* clause to tell the compiler that pointers are aligned to the vector length boundary, we reach a factor of 11.67 overall improvement (version `+aligned`).

So far, SIMD annotations on functions lead the compiler to generate two vector versions of each function: one without an initial mask and one with an initial mask. However, the original scalar code never invokes these functions from branches, so the masked version of these functions will never be used in the vector version of the code. To prevent the generation of this useless function versions we annotate them with the *nomask* clause. Figure 4.18 shows the code size in bytes of the main part of the benchmark (`main` function is not included) compiled with the `-Os` flag that optimizes the code size. As depicted, when the *nomask* clause is used, the code size decreases from 5.30 KB to 3.60 KB, which is a reduction of a 45% over the version without this clause.

### Fastwalsh

The code of the Fastwalsh benchmark is shown in Listing A.6 included in the Appendix A. The algorithm receives the Hadamard input matrix in `h_Input` and the total number of elements of this matrix in power of two in `log2N`. It returns the

```
if (stride >= 16){ // stride and base are 'suitable' in this branch
    #pragma omp for
    for(base = 0; base < N; base += 2 * stride){
        ...
    }
}
else{
    #pragma omp for
    for(base = 0; base < N; base += 2 * stride){
        ...
    }
}
```

Listing 4.14: Multiversion code snippet of the Fastwalsh benchmark without SIMD constructs

matrix resulting from the Walsh-Hadamard transform in h_Output.

The code has a first loop that copies the whole input matrix h_Input to the output matrix h_Output. Then, a 3-loop nest computes the transformation only using information already in the output matrix. In this computation, the induction variable of the outermost loop (stride) evolves with a decreasing butterfly stride. This induction variable is then used to compute the step ($2 * \texttt{stride}$) of the induction variable of the intermediate loop (base). Finally, the innermost loop updates elements of the output matrix within the range [base, base+2*stride). This update is performed in pairs of elements per iteration with a stride stride.

In our experiments, we parallelize the first independent loop and the intermediate loop of the loop nest with a *for* directive in all the benchmark versions. The outermost loop of the loop nest cannot be parallelized because there are true dependences across iterations.

This benchmark presents a peculiar memory access pattern because memory accesses on the 2D matrix h_Output depend on the value of three induction variables. Although the induction variable of the innermost loop is unit-stride, the algorithm accesses data with no spatial locality across iterations of the two outermost loops, because their induction variables are not unit-stride. For that reason, we use SIMD constructs on the independent loop and the innermost loop of the loop nest (simd icc and simd mcc versions compiled with the Intel compiler and the Mercurium compiler, respectively). The independent loop is annotated with a for-SIMD/SIMD-for directive and the innermost loop with a standalone SIMD directive. This vectorization strategy produces unit-stride vector memory accesses on all h_Output accesses. Vectorizing the intermediate loop is also possible, but these memory accesses turn into vector gather/scatter operations. This leads us to discard this vectorization strategy as it worsens even more the memory locality issues already present in the algorithm.

Over the selected SIMD-annotated version, we measure the performance im-

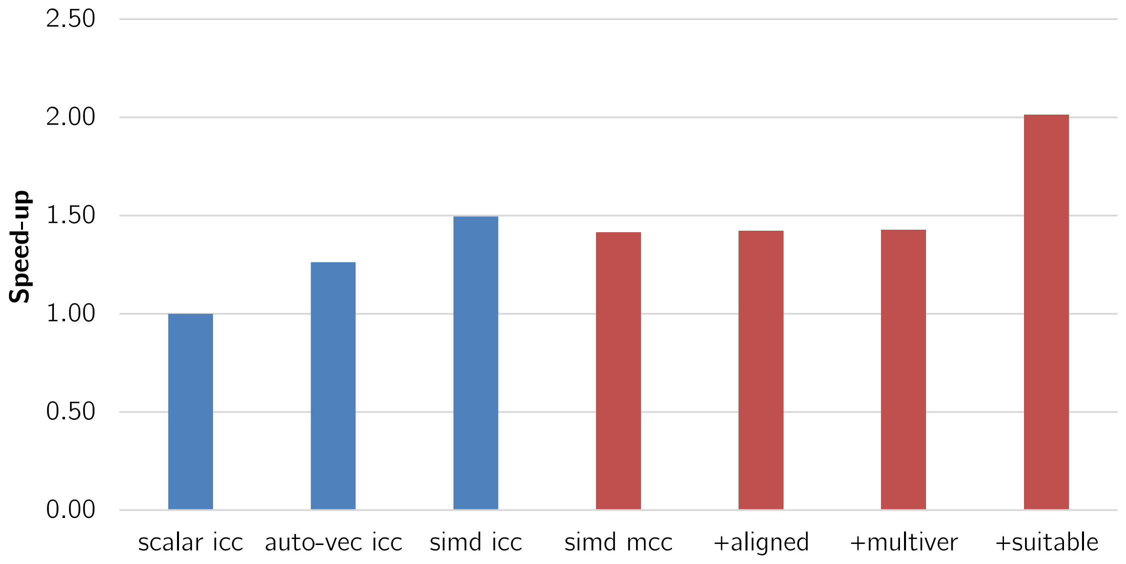


Figure 4.19: Speed-up of the FastWalsh benchmark (`float` data types)

pact of adding the *aligned* clause (+ *aligned* version). Then, we modify the original source code to be able to use the *suitable* clause on variables `stride` and `base` with positive effects on the alignment of vector accesses. Both variables will be multiple of the vector length only if `stride` $\geq 16$. In this way, we add this logic before the intermediate loop and we replicate the intermediate and the innermost loop into both branches of the condition, as we show in Listing 4.14.

Before adding the *suitable* clause we measure the impact of this transformation (`+ multiver` version). Then, we add the *suitable* clause with the value 16 on the `stride` and `base` variables (`+ suitable` version) to the SIMD clause of the first branch.

Figure 4.19 shows the performance results of all the versions. As we can see, the Intel auto-vectorizer is not able to exploit all the vectorization potential of the code. It yields only 24% of improvement over the scalar version due to the runtime checks that are needed to discard potential aliasing of memory accesses. The use of SIMD constructs provides 50% and 42% of gain with the Intel compiler and the Mercurium compiler, respectively. This performance difference, again, is due to the dynamic alignment strategy that the Intel compiler implements, as occurred with the Distsq benchmark. The single use of the *aligned* clause only turns memory access of the independent loop into aligned. Memory accesses within the loop nest remain still unaligned because the compiler does not have information about the runtime values of `stride` and `base`. As we can see, the *aligned* clause by itself does not have a significant impact on the overall execution time. Introducing the code multi-versioning does not penalize the execution time in comparison with previous versions. However, this multi-versioning jointly with the *suitable* clause increase the performance improvement to a factor of 2.01. This performance improvement happens because vector memory accesses within the loop nest are now aligned when `stride` and `base` have suitable values. In addition, the *suitable* clause on the `stride` variable allows the compiler not to generate an epilogue loop after the main vector loop.

Even though we obtain certain performance improvement with the use of the SIMD constructs and clauses, the overall improvement, an speed-up of 2.01, is not outstanding. However, the lack of spatial locality of memory accesses across the intermediate and the outermost loop gives rise to a memory bandwidth issue that overcomes the performance provided by vectorization.

**Mandelbrot**

The Mandelbrot benchmark implements a simple algorithm which is challenging for vectorization. The code is shown in Listing A.8 at Appendix A. The `mandelbrot` function is the main function of the algorithm. It receives two initial coordinates of the complex space, the maximum number of iterations used in the Mandelbrot algorithm (`maxIterations`), and the `width` and `height` used in the discretization of the complex space between the two initial coordinates. For each discretized point, the `mandel` function computes the number of iterations that the Mandelbrot computation takes to diverge over a threshold. If the Mandelbrot computation for a given coordinate exceeds this threshold, a `break` statement stops the computation and the function returns that number of iterations. If the algorithm has not diverged in the maximum number of iterations established, the `mandel` function stops the computation and it returns that maximum number. The resulting number of iterations for each complex coordinate are stored and returned in the `output` array.

This simple code poses a challenge to the vectorizer because of the irregular control flow and the control flow dependence introduced by the `break` statement. For the vectorization of this code, the compiler must guarantee that the control flow dependence can be *relaxed* without introducing side effects or rising exceptions that would not happen in the scalar version of the code. The relaxation of this control flow dependence would allow the speculative execution of the code previous to the `break` statement across multiple vector lanes of the same vector register. This execution would take place even though one of those vector lanes finally came to execute the `break` statement and this fact disabled the execution from that vector lane to all the following vector lanes involved in the execution of the loop body.

In our experiments, we parallelize the innermost loop within the `mandelbrot` function with a *for* directive. On this parallel scheme, we use the for-SIMD/SIMD-for directive on the innermost loop of the `mandelbrot` function and a SIMD directive on the `mandel` function (`simd icc` and `simd mcc` compiled with the Intel compiler and the Mercurium compiler, respectively). Afterwards, we incrementally add to the SIMD clause on the `mandel` function the clauses *uniform*, *suitable* and *aligned* to evaluate their impact. We use the *uniform* clause on the function pa-

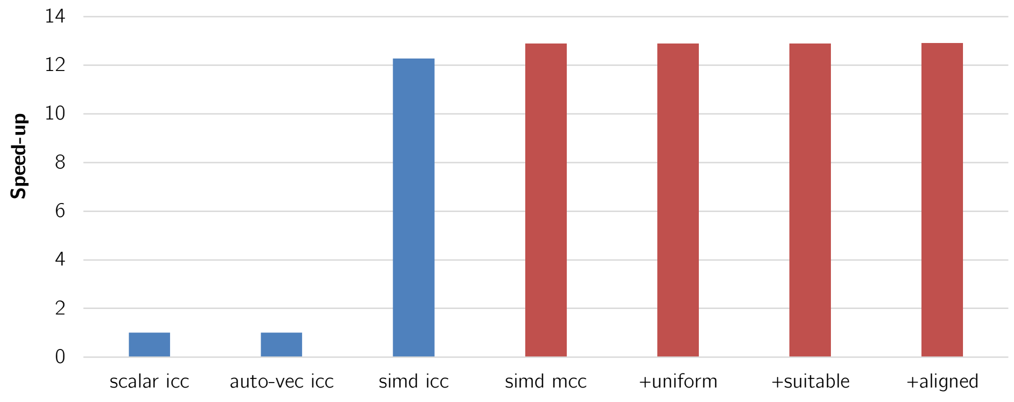


Figure 4.20: Speed-up of the Mandelbrot benchmark (`int` data types)

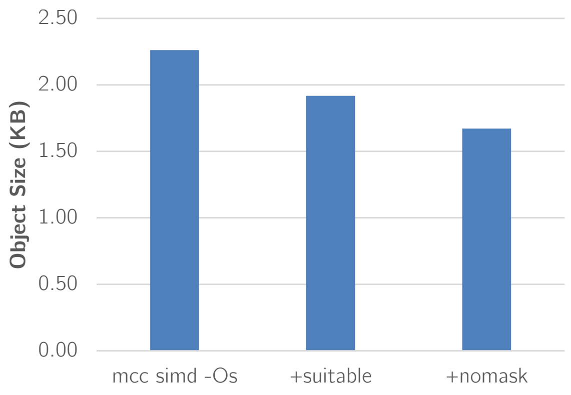


Figure 4.21: Code size variation of the main functions of the Mandelbrot benchmark

rameters `c_im` and `count`, as all iterations of the SIMD caller loop will call the `mandel` function with the same value in these parameters at runtime. We use the *suitable* clause on the SIMD loop to tell the compiler that the runtime value of variable `width` will be multiple of the vector length. Finally, we introduce the *aligned* clause on the SIMD loop to set the pointer `output` as aligned.

Figure 4.20 shows the performance of all the versions. As we discussed in the motivation of this chapter at Section 4.1.1, the Intel compiler is not able to auto-vectorize this code due to the complexity of the control flow and the need of executing part of the code in an speculative way. However, when the SIMD directive is used, both compilers are able to vectorize the code, yielding 12.29 (Intel compiler) and 12.90 (Mercurium compiler) performance speed-up over the scalar version. The Mercurium compiler outperforms the Intel compiler because of the already mentioned differences in the semantic of the combined SIMD construct: the for-SIMD directive instead of SIMD-for directive of our proposal.

It is interesting to note how optional clauses do not provide any extra performance even though they are applicable to this benchmark. The use of these clauses is affecting the generated code. Parameters `c_im` and `count` are kept scalar because they are set as *uniform*. The *suitable* clause lead the compiler to avoid the generation of the epilogue loop of the SIMD loop and jointly with the *aligned* clause, the vector store on the array `output` is turned aligned. Nevertheless, any of these optimizations have enough impact on the overall execution time. This fact support that in some cases just annotating the loop with a SIMD construct is enough to achieve the best performance with our proposal. This facts keeps low the complexity of use.

For this benchmark, we also evaluate how the *suitable* and the *nomask* clauses impact on the code size. Figure 4.21 shows that the *suitable* clause reduces the code size in a 18% with respect to the same version without such a clause using the compiler flag `-Os`. In addition, this improvement reaches 35% when we also annotate the `mandel` function with the *nomask* clause.

## 4.8 Summary and Discussion

In this chapter, we propose a set of directives and clauses that extends OpenMP 3.1 to deal with SIMD parallelism. These SIMD extensions allow programmers to annotate for-loops and functions that guide the compiler in the vectorization process.

Our proposal includes two main directives: a standalone SIMD directive and a combined SIMD-for directive. The standalone SIMD directive can be used on for-loops and functions to inform the compiler that the annotated code is safely vectorizable. The SIMD-for directive combines the standalone SIMD directive with the *for* directive that describes thread-level parallelism in OpenMP. This latter directive can only be used on for-loops.

We also define several optional clauses that give programmers more control on the vectorization optimization and provide the compiler with further information. These clauses are mainly aimed at improving the efficiency of the vector code generated by the compiler. The information provided by these clauses comprises alignment information, runtime behavior of variables, description of idioms and tuning options.

With this proposal, we not only offer programmers the possibility of describing where SIMD parallelism can be exploited but we also provide programmers with a productive way of taking advantage of the outstanding vectorization capabilities that many compilers have.

We implemented a prototype of our proposal in the Mercurium C/C++ source-to-source compiler. Our evaluation on a set of six benchmarks shows that SIMD directives are crucial to efficiently extract the best performance from vectorization. The use of the SIMD directives commonly outperforms the speed-up achieved by auto-vectorization with the latest public version of the Intel C/C++ compiler. This happens even for simple codes easily auto-vectorizable, such as the Distsq or the Cholesky benchmarks. The main reason is that SIMD directives lead the compiler to remove runtime checks and multi-versioning necessary for correctness in the completely automatic approach.

More complex codes not auto-vectorized by the compiler yield an outstanding speed-up when they are annotated with a SIMD directive, such as the Mandelbrot benchmark. This demonstrates that auto-vectorization techniques still have strong limitations that prevent the vectorization of codes with high vector potential.

In some cases, a closely-to-the-best performance is achieved only using the SIMD directive without any extra clauses. For example, this happens in the Mandelbrot and the Distsq benchmarks. This fact is relevant for programmers without experience in vectorization as they will be able to achieve a great performance just using SIMD directives without extra clauses. However, more complex codes require a finer tuning using optional clauses. For example, in the Nbody benchmark, the usage of the clauses *uniform*, *linear* and *aligned* turns 51% performance penalty from using a standalone SIMD directive into 11.67 speed-up over the scalar version of the code. This demonstrates how powerful these clauses can be in some codes if they are used appropriately.

Besides performance results, the whole set of directives and clauses allows us to evaluate quickly many vectorization approaches just by means of adding or removing a clause or moving the SIMD annotation to a different loop. Following this trial-and-error approach, we were able to find the best vectorization strategy for each benchmark. On the contrary, generating all these different vector versions of the benchmarks using SIMD intrinsics would have been unaffordable.

All these reasons widely justify the benefit of including our SIMD proposal in the OpenMP programming model.

### 4.8.1 Where is the limit?

During the research and development of this part of the thesis, we made ourselves the following questions whose answers are not still clear:

- Where is the limit of making programmers responsible for the correctness of vectorization?
- Which constraints should SIMD constructs relax and which not?
- Should the compiler vectorize a SIMD-annotated code even though it knows that the generated code is not correct?
- Should the compiler apply bidirectional checking to guarantee that the constraints stated by programmers are fulfilled at runtime?

These questions have been part of the history of optimizing compilers since their first existence. Traditional optimization approaches conservatively answer these questions in favor of the correctness of the code. However, SIMD directives are aimed at overcoming this conservatism and releasing programmers from the burden of optimizing the code by hand. From our point of view, any decision towards not considering programmers guidance or information will make the SIMD proposal useless. The reason is that the main goal of this proposal will not have been achieved: releasing programmers from vectorizing the code by hand to extract the best performance from vectorization.

As we said, we do not have a firm position on this. However, as programmers, to some extent, we would expect that a sensitive compiler opt by more aggressive and risky approaches when we had deliberately and consciously assumed the responsibility on the correctness and feasibility of a particular optimization.

### 4.8.2 OpenMP 4.0

As we introduced in Section 4.6, OpenMP 4.0 incorporated SIMD extensions as a result of this work and in collaboration with Intel [83]. The most significant difference with our proposal is the definition of a for-SIMD directive instead of the SIMD-for directive. To the best of our knowledge, the reasons that led to opt by this for-SIMD construct in OpenMP 4.0 were:

- The scheduling of iterations do not change from OpenMP scalar constructs to SIMD constructs.
- Implementing the SIMD-for construct supposed an important challenge, according to some vendors.

Both arguments are totally acceptable but we want to give our opinion on the first one. We agree on that in our proposal, we had to redefine the scheduling of instructions for SIMD construct, as we described in Section 4.3.1. However, we do not see any inconvenient or inconsistency in doing this, but quite the contrary. From our point of view, we do not see the advantage of being able to specify a chunk of iterations not multiple of the vector length when we are asking the compiler to vectorize the code. If this chunk is not multiple of the vector length, the remaining iterations will be executed in a less efficient way, even using scalar instructions depending on the architecture. In fact, keeping the schedule of scalar iterations for SIMD constructs will entail the following issues:

- Programmers must specify a chunk of iterations multiple of the vector length for an optimal execution.
- Default chunk values lead to not very efficient scenarios.

Regarding the first issue, specifying the chunk of iterations as a function of the vector length is not portable across architectures of different vector lengths. In addition, the level of abstraction is affected as it requires programmers to be aware of such a low level detail of the architecture.

The second issue can be illustrated with some example. A dynamic schedule with the default chunk (one iteration) will not execute any full vector iterations. A static schedule will have a chunk multiple of the vector length only if the total number of iterations is multiple of the vector length. If this condition is not satisfied, the remaining iterations could be executed in a scalar fashion. This is particularly critical when the number of iterations of the loop is low and the number of threads is considerably high. This happened in the evaluation of the Cholesky benchmark even using four threads. From our point of view, a more suitable approach is that a lower number of threads executes full SIMD iterations making use of full vector registers, as in our proposal. Furthermore, this remaining iterations not multiple of the vector length introduce an offset across chunks of iterations that lead to problems with the alignment of memory operations. Consequently, either unaligned memory access will have to be used or more iterations will have to be executed in a less efficient way by a prologue loop with alignment purpose.

In conclusion, we think that the approach adopted in OpenMP 4.0 might not be the most appropriate for programmers. The optimal execution of the application highly depends on the intervention of programmers and default behaviors can lead to poor performance difficult to understand for inexperienced programmers.

# Chapter 5

# Optimizing Overlapped Vector Memory Accesses

## 5.1 Introduction

This chapter introduces our proposal on a compiler code optimization that exploits register-to-register vector instructions to improve *overlapped* vector memory loads. This kind of vector loads read scalar elements from memory that have already been read by other vector load. Overlapped vector loads arise naturally after the vectorization of algorithms based on neighboring computation, e.g., stencil computation.

In addition to this optimization and in the context of the OpenMP SIMD extensions proposed in Chapter 4, we introduce a new clause for enabling, disabling and tuning this optimization on demand. In this way, this clause allows programmers to have further control on the compiler optimization process.

We implement a prototype of this vector code optimization in our vectorization infrastructure described in Chapter 3, targeting the Intel Xeon Phi coprocessor. Then, we evaluate the performance of the optimization on a set of stencil codes highly optimized to run on this massively parallel architecture.

### 5.1.1 Motivation

As we discussed in Chapter 1, SIMD instructions have become more relevant in the instruction set architecture (ISA) of the latest multi- and many-core processors. They allow to achieve high computational performance ratios at moderate energy consumption. Examining past and present SIMD instruction sets, we observe the trend of widening vector registers and introducing more powerful and flexible instructions generation after generation. These powerful and flexible new instructions try to overcome limitations from previous instruction sets and issues that naturally arise in the process of vectorizing a scalar code.

### Advanced SIMD Instructions

However, some of these powerful SIMD instructions are so sophisticated that there exists no direct translation from scalar instructions. Their usage may require an aggressive code transformation of the code. Therefore, their exploitation can be hard, not only for programmers but also for compilers. This fact can lead to the underuse of these instructions. These are only three examples of this kind of instructions available in current SIMD instruction sets:

Inter-register shift instructions: These instructions emerged to emulate vector memory operations on unaligned memory addresses in architectures without specific hardware support for them [54]. Furthermore, they are also included in architectures with specific hardware support to palliate the high latency of these unaligned memory operations in comparison to the aligned counterparts [15]. They can also be used to implement circular single-register permutations. The *valignd* instruction is one of these instructions available in several Intel SIMD instruction sets.

Instructions on multidimensional matrices: There are specific vector instructions that implement operations interpreting the vector register with a layout of a 2D matrix. For instance, the NEON instruction set contains instructions that perform a transposition on a 2D matrix or the computation of its determinant [11].

Generic permutations: These instructions are used to compose a vector register by means of reorganizing scalar elements from a single or multiple vector registers. For example, the AVX-512 instruction set [74] features a wide variety of these instructions with different permutation patterns and latencies.

### Compiler Limitations

As stated in Chapter 4, production compilers still have strong limitations when dealing with vectorization in general terms. If these limitations happen even in the vectorization of a simple scalar code where there exists a direct correspondence between scalar instructions and vector instructions, the exploitation of advanced vector instructions that require special code transformation is even harder. Compilers might not exploit these advanced vector instructions mainly for the following three reasons:

- Lack of compiler technology such as analyses, cost models, idiom recognition and code transformations. This technology is necessary to detect the applicability and to generate the appropriate code to be able to use a particular advanced vector instruction.
- Aggressiveness of optimizations and code transformations. Code transformations can jeopardize the correctness of the code if they are applied indiscriminately, degrade performance or prevent other more relevant optimizations.

```
#pragma omp simd aligned(a,b:64)
for(i=0; i <= N-points-1; i++)
{
    float tmp = 0.0f;

    for(j=0; j <= points-1; j++)
        tmp += a[i+j];

    b[i] = tmp / points;
}
```

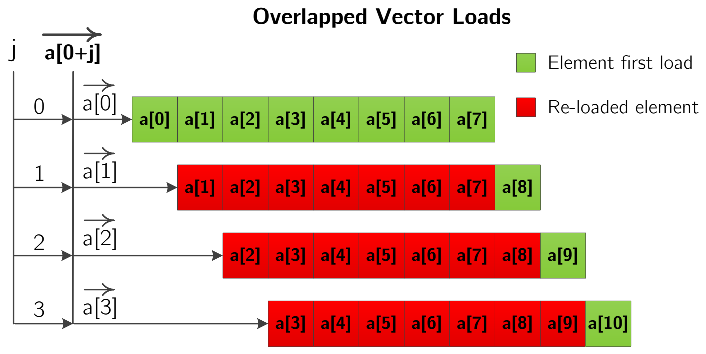
(a) Scalar code with a standalone SIMD directive on the outer loop

```
for(i=0; i <= N-points-VF; i+=VF)
{
    floatVF tmp = vpromotionVF(0.0f);

    for(j=0; j <= points-1; j++)
        tmp += unaligned_vloadVF(&a[i+j]);

    tmp = tmp / vpromotionVF(points);
    aligned_vstoreVF(&b[i], tmp);
}
```

(b) Vector pseudo-code after the outer loop vectorization. VF is the vectorization factor


Overlapped Vector Loads
j
a[0+j]
🟩 Element first load
🟥 Re-loaded element
0
a[0]
a[0] a[1] a[2] a[3] a[4] a[5] a[6] a[7]
1
a[1]
a[1] a[2] a[3] a[4] a[5] a[6] a[7] a[8]
2
a[2]
a[2] a[3] a[4] a[5] a[6] a[7] a[8] a[9]
3
a[3]
a[3] a[4] a[5] a[6] a[7] a[8] a[9] a[10]

(c) Original overlapped vector loads of the array a after the outer loop vectorization

Figure 5.1: Moving Average motivating example

- Interference with manual optimizations. Code transformations can interfere with manual optimizations of programmers that are fine tuning the code.

As a result, advanced programmers pursuing the best performance for their applications are compelled to apply these optimizations manually [135]. This also implies vectorizing the code by hand using low-level hardware-specific intrinsics. This task is very cumbersome, error prone and inefficient in terms of programming because some of these optimizations might be implemented and discarded if they penalize performance.

### Practical Example using SIMD Extensions for OpenMP

The SIMD extensions that we proposed in the context of OpenMP in Chapter 4 are aimed at reducing compiler auto-vectorization issues and easing SIMD exploitation. Nevertheless, these extensions still leave room for performance improvements. For instance, Figure 5.1a shows a snippet of the scalar code of our motivating example: the Moving Average benchmark. As depicted, we use a standalone SIMD directive to inform the compiler that the outer loop is safely vectorizable. Figure 5.1b shows

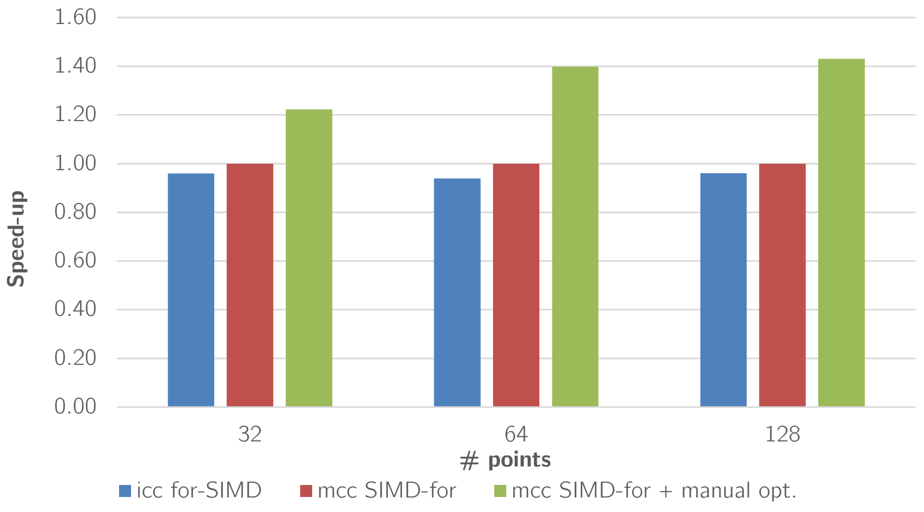


Figure 5.2: Speed-up of the Moving Average motivating example on a Intel Xeon Phi coprocessor 7120. Running with 183 threads (61 cores, 3 threads/core. The best experimental thread configuration). `N`=536M, `points`=128. Baseline `mcc SIMD-for`

the output vector pseudo-code that the compiler generates following the SIMD annotations. However, as we can see in Figure 5.1c, vector loads on the array `a` result in a not very efficient memory access pattern. These vector loads are *overlapped*: they redundantly read some scalar elements that have already been loaded by previous vector loads.

The *overlap* optimization that we will introduce later in this chapter applies an aggressive code transformation to improve this overlapped vector memory loads. Unfortunately, at this point, the only option for programmers in order to apply this optimization is by hand. This process first entails to vectorize the code by hand using intrinsics or assembly and then to apply the optimization over the resulting vector code.

We followed these steps, but thanks to our source-to-source vectorization infrastructure introduced in Chapter 3, we did not have to vectorize the code by hand. We used the output code with intrinsics generated by the Mercurium compiler. Then, we applied the overlap optimization by hand. Figure 5.2 shows the speed-up of three versions running on an Intel Xeon Phi coprocessor. Version `icc for-SIMD` is compiled with the Intel C/C++ compiler 15.0.1 using a for-SIMD directive on the outer loop. Version `mcc SIMD-for` is compiled with the Mercurium compiler using a SIMD-for directive on the outer loop. In version `mcc SIMD-for + manual opt.`, we optimized the overlapped vector loads using the vector code generated by the previous version. As we can see, the Intel compiler version yields slightly less performance than the version compiled with the Mercurium compiler. This is due to the differences between the for-SIMD and the SIMD-for constructs described in Chapter 4. The manual optimization of the code greatly outperformed both SIMD-annotated versions. This demonstrates that SIMD extensions for OpenMP might lead the compiler to generate a code still far from the most optimal hand-vectorized version.

### 5.1.2 Objectives

Our goal in this contribution is to propose a vector code optimization aimed at improving overlapped vector loads. This optimization will improve this inefficient memory access pattern by means of an aggressive code transformation that allows using advanced vector register-to-register instructions.

In order to control this aggressive code transformation, we propose a new clause (*overlap*) in the context of our OpenMP SIMD extensions proposed in Chapter 4. This new clause will allow programmers to have further control on the code optimization process and the vector code generated by the compiler. The clause will allow enabling and, optionally, fine tuning this vector code optimization.

## 5.2 Overview of the Overlap Optimization

The key concept of our proposal is to classify vector memory loads that partially overlap in memory, as depicted in Figure 5.1c, into groups, denominated *overlap groups*. The compiler will create and maintain a software register cache for each overlap group [112, 108, 141]. This cache will be used to load elements from memory only once. Then, the original vector memory loads will be replaced by vector register-to-register operations that build the original vector operands from the register cache.

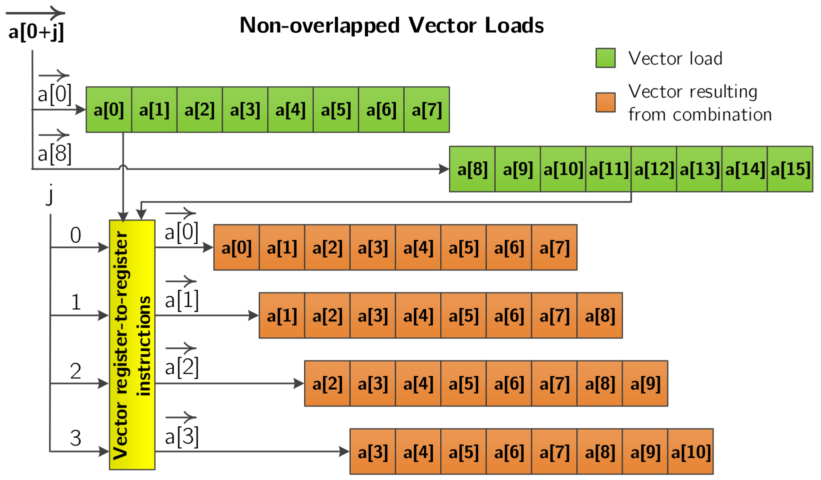
Each one of these register caches uses vector registers and it only hosts data from the same overlap group. Its management is simple since the compiler knows exactly which data is in which position. For this reason, this register cache do not require sophisticated mechanisms as a hardware cache normally does.

Figure 5.3 shows the resulting scheme from applying the overlap optimization on the example shown in Figure 5.1. This scheme contains the vector register cache (in green) and how vector register-to-register instructions (in yellow) rebuild the original vector operands (in orange).

The overlap optimization can target different inner and outer loop vectorization scenarios, where:

- the total number of vector loads in an overlap group can be known or unknown at compile time.
- vector loads in an overlap group can overlap only in the same iteration, across iterations or both.

Furthermore, we extend our SIMD proposal for OpenMP introduced in Chapter 4 with a new clause called *overlap*. This clause allows programmers to enable and guide the compiler on which arrays to apply this vector code optimization.



$\overrightarrow{a[0+j]}$
Non-overlapped Vector Loads
Vector load
Vector resulting from combination
$\overrightarrow{a[0]}$
$\overrightarrow{a[8]}$
a[0] a[1] a[2] a[3] a[4] a[5] a[6] a[7]
a[8] a[9] a[10] a[11] a[12] a[13] a[14] a[15]
j
0
1
2
3
Vector register-to-register instructions
$\overrightarrow{a[1]}$
$\overrightarrow{a[2]}$
$\overrightarrow{a[3]}$
a[1] a[2] a[3] a[4] a[5] a[6] a[7] a[8]
a[2] a[3] a[4] a[5] a[6] a[7] a[8] a[9]
a[3] a[4] a[5] a[6] a[7] a[8] a[9] a[10]

Figure 5.3: The overlap optimization applied to vector loads of the array a in Figure 5.1

Optionally, the clause can contain extra arguments to fine tune the optimization. These arguments affect the construction of the overlap groups and, therefore, the code transformation algorithm.

The potential benefits of the overlap optimization are:

- Improved use of memory bandwidth: overlapped memory elements are loaded only once.
- Reduced number of unaligned vector loads in SIMD architectures with alignment constraints.
- Reduced number of vector loads and less pressure in the memory units of the architecture.

Moreover, the *overlap* clause gives users different levels of control over a code optimization that the compiler would not be able to apply automatically. In addition, this clause favors the agile benchmarking of different vector versions when looking for the best performance, thus avoiding hand-coded vectorization.

## 5.3 Formal Framework and Definitions

In this section, we describe a framework to formally introduce the concepts of *overlap group*, *overlapped vector memory load* and other definitions of our proposal. We apply our formalization to the Moving Average code shown in Figure 5.1 and to the Jacobi code shown in Figure 5.4a to offer a practical point of view.

```
for (i=1; i<=sizex-2; i++){
  #pragma omp simd aligned(u+1,utmp+1:64) suitable(sizey:16) reduction(+:sum)
  for (j=0; j<=sizey-3; j++){
    float tmp = 0.25f * (u[i*sizey+j] + u[i*sizey+j+2] +   /*left + right +*/
                u[(i-1)*sizey+j+1] + u[(i+1)*sizey+j+1]);  /*top + bottom  */
    float diff = tmp - u[i*sizey+j+1]; /*center*/
    utmp[i*sizey + j+1] = tmp;
    sum += diff * diff;
  }
}
```

(a) Scalar code with a standalone SIMD directive on the inner loop. The lower bound of the inner loop has been normalized to 0

```
for (i=1; i<=sizex-2; i++){
  float_VF vsum = vpromotion_VF(0.0f);
  for (j=0; j<=sizey-VF-2; j+=VF){
    float_VF tmp = vpromotion_VF(0.25f) *
      (unaligned_vload_VF(&u[i*sizey+j]) + unaligned_vload_VF(&u[i*sizey+j+2]) +
       aligned_vload_VF(&u[(i-1)*sizey+j+1]) +
       aligned_vload_VF(&u[(i+1)*sizey+j+1]));
    float_VF diff = tmp - aligned_vload_VF(&u[i*sizey+j+1]);
    aligned_vstore_VF(&utmp[i*sizey+j+1], tmp);
    vsum += diff * diff;
  }
  sum += vhorizontal_reduction_VF(vsum);
}
```

(b) Vector pseudo-code after the inner loop vectorization. `VF` is the vectorization factor

Figure 5.4: Code of a 2D Jacobi solver for heat diffusion

### 5.3.1 Preliminaries

We use $a[i]$ to denote a scalar access to the $i^{\text{th}}$ element of the array $a$, and $a[l:u]$ to designate the set of scalar accesses to $a[j]$, $\forall j \; l \leq j \leq u$.

Let $K$ be a scalar loop defined as:

$$K : IV_K = \langle LB_K, UB_K, ST_K, S_K \rangle \tag{5.1}$$

where $IV_K$, $LB_K$, $UB_K$, $ST_K$ and $S_K$ are the induction variable, the inclusive lower and upper bounds, the step and the scalar statements of $K$, respectively. Let us assume that the loop has been normalized, i.e., $LB_K = 0$ and $ST_K = 1$, and array accesses have been linearized [81].

We define $V$ as the vectorized version of $K$ following a strip-mining/unroll-and-jam vectorization approach as our approach introduced in Chapter 3, such that:

$$V : IV_V = \langle LB_V, UB_V, ST_V, S_V \rangle \tag{5.2}$$

where $LB_V = 0$, $ST_V = VF_V$ (vectorization factor of loop $V$, i.e., the number of scalar iterations of $K$ computed within each vector iteration of $V$), $UB_V = UB_K - VF_V + 1$ and $S_V$ is the set of statements obtained from the vectorization of $S_K$.

**Definition 1.** (Stride-one Vector Memory Access). $\overrightarrow{a[e]}$ denotes the *stride-one vector memory access*, henceforth *svma*, of a scalar access $a[e]$ that satisfies that $e$ is of the form $e = IV_K + c$ where $c$ is *invariant* in $K$. This kind of vector accesses performs the set of scalar memory accesses $a[e : e + VF_V - 1]$ in a vector way.

A *stride-one vector memory load*, denoted *vl*, is an *svma* that reads data from memory. From now on, we limit our description to *vls*, although it can be easily extended to *stride-one vector memory stores*. However, our proposal has limited applicability to stores in real applications.

In the Moving Average example, the outer loop in Figure 5.1a is $K : i = \langle 0, N - points - 1, 1, \{\text{lines } 3 - 10\}\rangle$, the outer loop in Figure 5.1b is $V : i = \langle 0, N - points - VF, VF, \{\text{lines } 2 - 10\}\rangle$ and `a[i+j]` is a *vl*. In the Jacobi 2D example, the inner loop in Figure 5.4a is $K : i = \langle 0, sizey - 3, 1, \{\text{lines } 3 - 9\}\rangle$, the inner loop in Figure 5.4b is $V : i = \langle 0, sizey - VF - 2, VF, \{\text{lines } 3 - 11\}\rangle$ and accesses on arrays `u` and `utmp` are *vls*.

## 5.3.2 Overlap Relations

The following definitions are used to establish an equivalence relation among *vls* of each particular array in $V$.

**Definition 2.** We say that two *vls* $\overrightarrow{a[e_1]}$ and $\overrightarrow{a[e_2]}$ *overlap*, denoted $\overrightarrow{a[e_1]} \sqcap \overrightarrow{a[e_2]}$, if and only if $\{a[i] \mid i \in e_1 \ldots e_1 + VF_V - 1\} \cap \{a[j] \mid j \in e_2 \ldots e_2 + VF_V - 1\} \neq \emptyset$.

**Definition 3.** We say that two *vls* $\overrightarrow{a[e_1]}$ and $\overrightarrow{a[e_2]}$ are *transitively-overlapped*, denoted $\overrightarrow{a[e_1]} \sqcap^* \overrightarrow{a[e_2]}$, if and only if $\overrightarrow{a[e_1]} \sqcap \overrightarrow{a[e_2]}$ or there exists $\overrightarrow{a[e_3]}$ such that $\overrightarrow{a[e_1]} \sqcap \overrightarrow{a[e_3]}$ and $\overrightarrow{a[e_3]} \sqcap^* \overrightarrow{a[e_2]}$.

It is important to note that some SIMD architectures with alignment constraints need two aligned *vl* instructions to perform one unaligned *vl*. In this way, the effective aligned *vls* must be considered in those architectures when evaluating whether two *vls* overlap.

## 5.3.3 Overlap Groups

We describe several sets of *vls* that are necessary to define the concept of an *overlap group* and its construction.

Given loop $V$, we define the set of loops $\mathcal{L}_V = \{V\} \cup \{\text{all loops in } S_V\}$. Then, for a loop $L \in \mathcal{L}_V$, we define:

$$A_{L,a} = \{ \text{ all } vls \text{ on the array } a \text{ in the body of } L \ \} \tag{5.3}$$

Moreover, the set of *vls* on the array $a$ directly nested in the loop $L$ is defined as follows:

$$D_{L,a} = A_{L,a} - \left\{ A_{L',a} \,|\, L' \in \mathcal{L}_L \wedge L' \neq L \right\} \tag{5.4}$$

In the Moving Average example, let $Lo$ and $Li$ denote the outer loop and the inner loop from Figure 5.1b, respectively. Then, $\mathcal{L}_{Lo} = \{Lo, Li\}$ and $\mathcal{L}_{Li} = \{Li\}$. Focusing on the array a, $A_{Lo,a}$, $A_{Li,a}$, $D_{Lo,a}$ and $D_{Li,a}$ sets are defined as:

$$A_{Lo,a} = A_{Li,a} = \{\ \overrightarrow{\texttt{a[i+j]}}\ \}$$
$$D_{Lo,a} = A_{Lo,a} - A_{Li,a} = \emptyset \qquad D_{Li,a} = A_{Li,a}$$

In the Jacobi 2D example, the inner loop is the only loop affected by vectorization. Then, let $Li$ denotes the inner loop from Figure 5.4b. Then, $\mathcal{L}_{Li} = \{Li\}$. Hence, targeting only *vls* from array u, $A_{Li,u}$ and $D_{Li,u}$ are defined as:

$$\begin{aligned} A_{Li,u} = \{ & \overrightarrow{\texttt{u[i*sizey+j]}}, \overrightarrow{\texttt{u[i*sizey+j+1]}}, \\ & \overrightarrow{\texttt{u[i*sizey+j+2]}}, \overrightarrow{\texttt{u[(i-1)*sizey+j+1]}}, \\ & \overrightarrow{\texttt{u[(i+1)*sizey+j+1]}}\ \} \\ D_{Li,u} = {} & A_{Li,u} \end{aligned}$$

**Definition 4.** (Overlap Group). An *overlap group* $\mathcal{O}_{L,a}$ is an equivalence class of the equivalence relation $\sqcap^*$ on $D_{L,a}$.

The i<sup>th</sup> overlap group is denoted by $\mathcal{O}_{L,a,i}$. We call $P_{L,a}$ to the quotient set of the equivalence relation of $D_{L,a}$ by $\sqcap^*$.

Depending on how *vls* overlap in the group, an overlap group can be of one or both of the following non-disjoint kinds:

**intra:** $|\mathcal{O}_{L,a,i}| > 1$, i.e., there exist at least two $vls$ in $\mathcal{O}_{L,a,i}$ that *overlap* in the same iteration.

**inter:** There exist $w$ and $w'$ in $\mathcal{O}_{L,a,i}$ that *overlap* across iterations, i.e., $w \sqcap (w'/IV_L \mapsto IV_L + ST_L)$. Note that $w$ and $w'$ may be the same *vl*.

If the overlap group does not satisfy any of these properties, its kind is *unknown*.

**Definition 5.** (Overlapped Vector Memory Load). A *vl* in $\mathcal{O}_{L,a,i}$ is an *overlapped vector memory load*, denoted *ovl*, if and only if $\mathcal{O}_{L,a,i}$ is of kind *inter* and/or *intra*.

The length of each overlap group is computed as:

$$\begin{aligned} & length(\mathcal{O}_{L,a,i}) = M - m + 1 \\ \text{where} \quad & \begin{array}{ll} m & = \min\ \{\ l \mid \forall r \in \mathcal{O}_{L,a,i}\ r = a[l:u]\} \\ M & = \max\{u \mid \forall r \in \mathcal{O}_{L,a,i}\ r = a[l:u]\} \end{array} \end{aligned} \tag{5.5}$$

Again, in the Moving Average example, we apply the *transitively-overlapped* relation to the only non-empty set, $D_{Li,a}$. It results in one *inter* overlap group $\mathcal{O}_{Li,a,0} = \{\ \overrightarrow{\texttt{a[i+j]}}\ \}$ with $|\mathcal{O}_{Li,a,0}| = 1$ and $length(\mathcal{O}_{Li,a,0}) = VF$. Likewise in the

Jacobi 2D example, we apply the ***transitively-overlapped*** relation on the set $D_{Li,u}$ which generates three overlap groups:

$$\begin{aligned}\mathcal{O}_{Li,u,0} &= \{\overrightarrow{\texttt{u[i*sizey + j]}}, \overrightarrow{\texttt{u[i*sizey + j+1]}},\\ &\quad\ \overrightarrow{\texttt{u[i*sizey + j+2]}}\ \}\\ \mathcal{O}_{Li,u,1} &= \{\overrightarrow{\texttt{u[(i-1)*sizey + j+1]}}\ \}\\ \mathcal{O}_{Li,u,2} &= \{\overrightarrow{\texttt{u[(i+1)*sizey + j+1]}}\ \}\end{aligned}$$

where the group $\mathcal{O}_{Li,u,0}$ has intra and inter kind, and groups $\mathcal{O}_{Li,u,1}$ and $\mathcal{O}_{Li,u,2}$ have unknown kind because their cardinality is 1 (no intra kind) and their single *vl* does not *overlap* across iterations (no inter kind). The cardinalities and lengths of the resulting overlap groups are:

$$|\mathcal{O}_{Li,u,0}| = 3 \quad |\mathcal{O}_{Li,u,1}| = 1 \quad |\mathcal{O}_{Li,u,2}| = 1$$
$$length(\mathcal{O}_{Li,u,0}) = VF + 2$$
$$length(\mathcal{O}_{Li,u,1}) = length(\mathcal{O}_{Li,u,2}) = VF$$

Since $\mathcal{O}_{Li,u,1}$ and $\mathcal{O}_{Li,u,2}$ have unknown kind, both overlap groups do not contain *ovls*.

## 5.4 The Overlap Optimization

The overlap optimization is applied to stride-one vector memory loads (*vls*) occurring within the already vectorized loop ($V$). Figure 5.5 depicts the implementation phases and their flow. At this point, we assume that basic compiler optimizations, such as common subexpression elimination or local value numbering [2, 81], have already been applied on the input vector code. The high-level steps to apply the overlap optimization are:

- Detecting and building overlap groups in the vector code (Group Building Phase, Section 5.4.2).
- Applying loop unrolling to adequate overlap groups properties to programmer/compiler constraints and vector units of the target architecture. (Unrolling Analysis Phase, Section 5.4.3).
- Pruning overlap groups according to programmer/compiler constraints (Group Pruning Phase, Section 5.4.4).
- Transforming the code to implement the register cache infrastructure for each overlap group (Cache Building Phase, Section 5.4.5).

Throughout this section we use the Moving Average example from Figure 5.1 to illustrate some implementation aspects of the overlap optimization. Listing 5.1 shows the optimized pseudo-code generated for this example, assuming both a vectorization factor (*VF*) and vector length (*VL*) equal to 16.

### 5.4.1 Overlap Group Constraints

We define three constraints on the overlap groups that will affect their construction and the optimized output code:

CARDCONSTR (cardinality): establishes the minimum number of *ovls* that an overlap group must contain. This constraint is useful to discard small groups that will not yield enough performance or that do not have enough data reuse to overcome the overhead of maintaining the vector register cache.

VREGSCONSTR (vector registers): limits the maximum number of vector registers per overlap group. The minimum number allowed in practice is two as a vector register cache with a single vector register can only host data for a single *vl* that would not require this optimization. This constraint is valuable in circumstances where register pressure may lead to register spilling and, therefore, to a performance degradation.

GROUPSCONSTR (number of groups): limits the maximum number of groups per array. This constraint is useful to limit the number of groups in scenarios with a large number of them. In this situation, the overlap optimization might lead to a performance penalty due to register spilling if it was applied to all groups.

These constraints can be set either by the compiler or the programmer. A compiler could establish the default values for this constraints by means of cost models and microbenchmarking for each particular architecture. In Section 5.5, we describe how programmers will be able to provide the compiler with the value of some of these constraints.

### 5.4.2 Group Building Phase

In the Group Building Phase, the compiler builds the overlap groups and it computes their basic properties (length and cardinality), as explained in Section 5.3.3. To do so, it constructs the quotient sets $P_{L,a}\ \forall L \in \mathcal{L}_V \forall a\ vl \in L$.

In this phase the compiler also computes the number of vector registers necessary to maintain a cache for that group, denoted $VecRegs(\mathcal{O}_{L,a,i})$. The minimum number of vector registers is computed as:

$$VecRegs(\mathcal{O}_{L,a,i}) = \lceil length(\mathcal{O}_{L,a,i})/VL\rceil \tag{5.6}$$

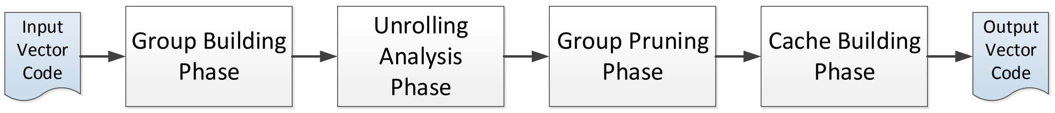


Figure 5.5: Phases of the overlap optimization

where $VL$ is the vector length of the target SIMD architecture. However, this minimum number could be tied to unaligned *vls* in architectures with alignment constraints if the first *ovl* (lowest memory address) is unaligned. In that case, using aligned *vls* requires the addition of a negative offset from this first memory reference to the first backward aligned address. This could entail one extra register and one extra *vl* over the minimal unaligned approach.

As stated in Section 5.4.1, the minimum number of registers used to maintain an overlap group is two. However, at this point of the optimization, $VecRegs$ could be one. When this happens, the Unrolling Analysis Phase (see Section 5.4.3) will attempt to increase this value. If it fails to do so, the group will finally be pruned in the Group Pruning Phase (see Section 5.4.4).

If $VecRegs$ is greater than one, the compiler computes the efficiency of the overlap group, defined as:

$$\eta(\mathcal{O}_{L,a,i}) = \frac{|\mathcal{O}_{L,a,i}|}{(\,VecRegs(\mathcal{O}_{L,a,i}) - 1) * VL} \tag{5.7}$$

Efficiency equal to one means that all data loaded into the register cache is optimally reused in the loop body. Values lower than one mean that some data loaded into the register cache is not reused as much as it could be, or even that some elements are not used at all in the loop body. Values greater than one happen when some vector load reads exactly the same data as another vector load from the next iteration (for example, if we have a[i] and a[i+$VL$] in the loop body). The efficiency will be undefined if $VecRegs$ is one.

We recall from Section 5.3.3 that the Moving Average example has one *inter* overlap group $\mathcal{O}_{Li,a,0} = \{\ \overrightarrow{\texttt{a[i+j]}}\ \}$ with $|\mathcal{O}_{Li,a,0}| = 1$ and $length(\mathcal{O}_{Li,a,0}) = VF = 16$. The number of registers is $VecRegs(\mathcal{O}_{Li,a,0}) = 1$, and, therefore, $\eta(\mathcal{O}_{Li,a,0})$ is undefined.

### 5.4.3 Unrolling Analysis Phase

In the Unrolling Analysis Phase, the compiler analyzes whether loops with overlap groups associated need to be unrolled. Loop unrolling is aimed at:

- making overlap groups satisfy CARDCONSTR (Equation 5.8) without breaking VREGSCONSTR (Equation 5.9).
- better fitting the length of each overlap group into the register cache considering the vector length ($VL$) of the target architecture (Equation 5.10).
- minimizing cache readjusting instructions per loop iteration, mapping original vector loads to the same register cache position across iterations (Equation 5.10).
- improving the register cache efficiency ($\eta$), maximizing the reuse of data loaded into the cache.

Frequently, loop unrolling is indispensable in outer loop vectorization scenarios where *inter* overlap groups are within the innermost loop, as occurs in the Moving Average example. These groups tend to have minimal lengths ($\approx$ *VL*) and cardinalities ($\approx 1$) that could not satisfy CARDCONSTR nor VREGSCONSTR. Innermost loop vectorization scenarios can also benefit from loop unrolling and the reduction of cache management instructions across iterations.

The unroll factor $\mathit{UF}$ of a loop is computed as the minimum common multiple of the unroll factor that better fits each overlap group in the loop, denoted $\mathit{UF}_{\mathcal{O}}$. Each $\mathit{UF}_{\mathcal{O}}$ is chosen based on the new cardinality ($|\mathcal{O}_{L,a,i}|_{UF}$), the new number of vector registers ($\mathit{VecRegs}_{UF}(\mathcal{O}_{L,a,i})$), the new length ($\mathit{length}_{UF}(\mathcal{O}_{L,a,i})$) and the new efficiency ($\eta_{UF}(\mathcal{O}_{L,a,i})$) that the group would have after unrolling the loop with a particular unroll factor. Starting from $\mathit{UF}_{\mathcal{O}} = 2$, the compiler will increment $\mathit{UF}_{\mathcal{O}}$ iteratively to find those values that satisfy the following equations:

$$|\mathcal{O}_{L,a,i}|_{UF} \geq \textsc{CardConst} \tag{5.8}$$

$$1 < \mathit{VecRegs}_{UF}(\mathcal{O}_{L,a,i}) \leq \textsc{VRegsConst} \tag{5.9}$$

The number of iterations is bound by the upper limit of Equation 5.9. If no $\mathit{UF}_{\mathcal{O}}$ is found that satisfies Equations 5.8 and 5.9, the group will be discarded in the Group Pruning Phase. If more than one value is found that satisfies them, the compiler will chose the one that minimizes *dist* as defined in Equation 5.10 and maximizes $\eta_{UF}(\mathcal{O}_{L,a,i})$, in this order of priority.

$$\begin{gathered} \mathit{dist} = (k * \mathit{VL} - \mathit{ST}_L,\ k > 1) - \mathit{length}_{UF}(\mathcal{O}_{L,a,i}) \wedge \\ \mathit{dist} \geq 0 \end{gathered} \tag{5.10}$$

If loop unrolling successfully takes place on the innermost loop in an outer loop vectorization scenario, several consecutive epilogue sections compute the remaining iterations of the unrolled loop:

**overlap epilogue:** computes up to $(\lceil \mathit{UF}/\mathit{UF}_{EPI} \rceil - 1) * \mathit{UF}_{EPI}$ iterations of the original loop in blocks of $\mathit{UF}_{EPI}$ iterations, where $\mathit{UF} > \mathit{UF}_{EPI} > 1$. This epilogue code still exploits the overlap vector register cache and, therefore, it must fulfill the constraints of each overlap group. $\mathit{UF}_{EPI}$ can be the minimum number that satisfies the constraints or be a default value chosen from microbenchmarking experiments, as in our implementation (Section 5.7).

**regular epilogues:** compute the remaining iterations not covered by the overlap epilogue and the main loop. These epilogue codes do not use the overlap vector register cache but some of them could still be vectorized. At least the last epilogue code will be a scalar epilogue.

If we apply the Unrolling Analysis Phase to the Moving Average example assuming $\mathit{VF} = \mathit{VL} = 16$, CARDCONSTR $= 4$ and VREGSCONSTR $= 2$, the inner loop

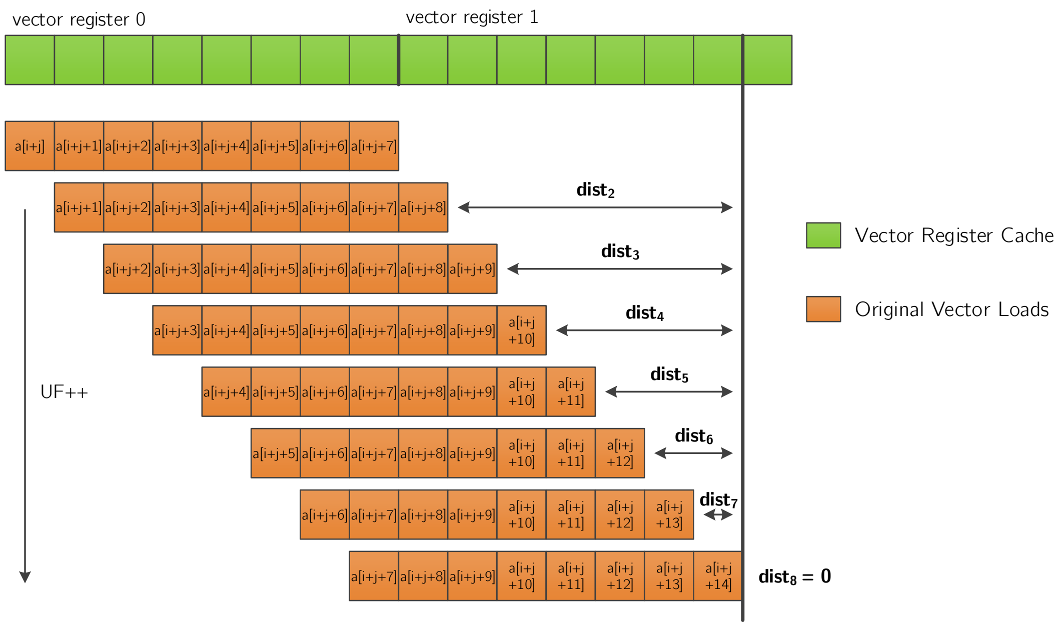


Figure 5.6: Graphical representation of *dist* for increasing *UF* in the Moving Average example

in Figure 5.1b will be unrolled with $UF = 16$ as shown in Listing 5.1, lines from 9 to 21. This value is the minimum number that satisfies Equations 5.8 and 5.9 as $|\mathcal{O}_{Li,a,0}|_{16} = 16$ and $VecRegs_{16}(\mathcal{O}_{Li,a,0}) = 2$, respectively. In addition, this unroll factor minimizes Equation 5.10 and it maximizes efficiency (Equation 5.7), as it is shown in Figure 5.6 and Figure 5.7, respectively. The overlap epilogue generated after the unrolled loop is depicted in Listing 5.1, lines from 24 to 41. It is generated with $UF_{EPI} = 4$, which is the minimum number that fulfills CARDCONSTR. It computes up to 12 iterations of the original loop in blocks of 4 iterations. In this case, only one regular epilogue is generated, as depicted in Listing 5.1 from lines 44 to 45.

### 5.4.4 Group Pruning Phase

In the Group Pruning Phase, the compiler processes and prunes overlap groups from the previous phase according to CARDCONSTR, VREGSCONSTR and GROUPSCONSTR, defined in Section 5.4.2. Figure 5.8 describes the pruning algorithm. This algorithm is independently applied to each quotient set $P_{L,a}$. It returns a new set $P'_{L,a}$ with the final overlap groups that have been chosen to receive the overlap optimization.

The input overlap groups with data dependences that could make the overlap optimization unsafe are discarded, as shown in line 4 from Figure 5.8. Groups that satisfy dependences are processed according to constraints on vector registers and

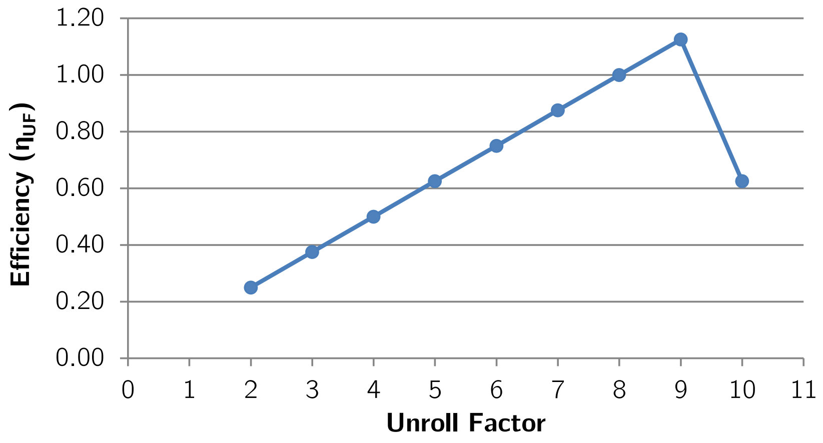


Figure 5.7: Evolution of efficiency ($\eta_{UF}$) for increasing *UF* in the Moving Average example

cardinality, as shown in Figure 5.8, line 5 and lines from 6 to 8, respectively.

In APPLYVREGSCONSTR, the compiler decides if the group satisfies VREGSCONSTR. It also determines what will be the final number of vector registers necessary to build the vector register cache for each overlap group. This number depends on the use of aligned or unaligned *vls*, as we described in Section 5.4.2. The compiler will decide the most appropriate approach depending on the register pressure and VREGSCONSTR. It can also decide to discard some *vls* to reduce the number of registers involved in the cache and satisfy VREGSCONSTR. Discarding *vls* might also be useful when one or a few *vls* increase this number of registers and reduce the efficiency of the cache.

Regarding the group cardinality, the compiler chooses the overlap group only if its cardinality after the processing of VREGSCONSTR is greater than or equal to the limit imposed by CARDCONSTR, as shown in Figure 5.8 from lines 6 to 8.

Once all groups have been processed and pruned according to CARDCONSTR and VREGSCONSTR, the compiler processes the whole set $P'_{L,a}$ to fulfill GROUPSCONSTR (APPLY-GROUPSCONSTR function in Figure 5.8, line 11). At this point, the compiler will discard some groups if they exceed the limit on the total number of groups imposed by this constraint. Groups with higher efficiency (Equation 5.7) and lower *VecRegs* (Equation 5.6), in that order, will be prioritized not to be discarded.

If all groups affected by the loop unrolling from the Unrolling Analysis Phase have been pruned at this point, the loop unrolling is undone.

In the Moving Average example, we assumed $VF = VL = 16$, CARDCONSTR $= 4$ and VREGSCONSTR $= 2$. The resulting code from the Unrolling Analysis Phase is shown in Listing 5.1. The only group $\mathcal{O}_{Li,a,0}$ has *ovls* without unsafe data dependences. $VecRegs(\mathcal{O}_{Li,a,0})$ is equal to two, so VREGSCONSTR is satisfied. In addition, the first *vl* (the *vl* with the lowest memory address) in the group is aligned after the loop unrolling, as shown in line 13 of Listing 5.1. CARDCONSTR $= 4$ is satisfied in

```
1: function PRUNINGGROUPS($P_{L,a}$)
2:     $P'_{L,a} \leftarrow \emptyset$
3:     for all $\mathcal{O}_{L,a,i} \in P_{L,a}$ do
4:         if NODATADEPENDENCES($\mathcal{O}_{L,a,i}$) then
5:             $\mathcal{O}'_{L,a,i} \leftarrow$ APPLYVREGSCONSTR($\mathcal{O}_{L,a,i}$)
6:             if $|\mathcal{O}'_{L,a,i}| \geq$ CARDCONSTR($\mathcal{O}'_{L,a,i}$) then
7:                 $P'_{L,a} \leftarrow P'_{L,a} \cup \left\{\mathcal{O}'_{L,a,i}\right\}$
8:             end if
9:         end if
10:    end for
11:    $P'_{L,a} \leftarrow$ APPLYGROUPSCONSTR($P'_{L,a}$)
12:    return $P'_{L,a}$
13: end function
```

Figure 5.8: The overlap group pruning algorithm

```
1: procedure BUILDGROUPCACHE($\mathcal{O}_{L,a,i}$, $L$)
2:     DECLAREGROUPSYMBOLS($\mathcal{O}_{L,a,i}$, $L$)
3:     ADDGROUPINITSTMTS($\mathcal{O}_{L,a,i}$, $L$)
4:     ADDGROUPPREITERUPDATESTMTS($\mathcal{O}_{L,a,i}$, $L$)
5:     ADDGROUPPOSTITERUPDATESTMTS($\mathcal{O}_{L,a,i}$, $L$)
6:     for all $\overrightarrow{a[e_j]} \in \mathcal{O}_{L,a,i}$ do
7:         $new\_vr2r\_inst \leftarrow$ GETVR2RINSTR($\overrightarrow{a[e_j]}$)
8:         REPLACE($\overrightarrow{a[e_j]}$, $new\_vr2r\_inst$)
9:     end for
10: end procedure
```

Figure 5.9: The cache building algorithm

the main loop, with $|\mathcal{O}_{Li,a,0}| = 16$, as depicted in Listing 5.1, lines from 9 to 21. The overlap epilogue also satisfies this constraint since there are 4 *ovls* ($UF_{EPI} = 4$) in each block code, as shown in lines from 26 to 29, from 32 to 34 and from 37 to 39 in Listing 5.1. GROUPSCONSTR is not set and it has no effect.

### 5.4.5 Cache Building Phase

In the Cache Building Phase, the code of each vector register cache is generated for each *ovls*. It is important to note that these register caches are simple and they host read-only data. The compiler always knows exactly which data is available in the cache, in which position and which data must be replaced because it will no longer be used. For these reasons, their management does not require a sophisticated infrastructure but only the vector registers to host the data.

For each $\mathcal{O}_{L,a,i} \in P'_{L,a}$ resulting from the Group Pruning Phase, the compiler

```
for(i=0; i <= N-points-16; i+=16){
  float16 cache_a_0, cache_a_1;
  float16 tmp = vpromotion(0.0f);

  // Cache init, j=0
  cache_a_0 = aligned_vload16(&a[i+0]);

  // Unrolled loop (UF=16)
  for(j=0; j <= points-16; j+=16){
    // Cache pre-iteration update
    cache_a_1 = aligned_vload16(&a[i+j+16]);
    // Rebuild original operands from cache
    tmp += cache_a_0;//aligned_vload(&a[i+j])
    tmp += vr2r_op16(//unaligned_vload(&a[i+j+1])
             cache_a_1, cache_a_0, 1);
    ...
    tmp += vr2r_op16(//unaligned_vload(&a[i+j+15])
             cache_a_1, cache_a_0, 15);
    // Cache post-iteration update
    cache_a_0 = cache_a_1;
  }

  // Overlap Epilogue (UF_EPI=4)
  if (j <= points-4){ // Block 1
    cache_a_1 = aligned_vload16(&a[i+j+16]);
    tmp += cache_a_0;
    tmp += vr2r_op16(cache_a_1,cache_a_0,1);
    tmp += vr2r_op16(cache_a_1,cache_a_0,2);
    tmp += vr2r_op16(cache_a_1,cache_a_0,3);
    j += 4;
    if (j <= points-4){ // Block 2
      tmp += vr2r_op16(cache_a_1,cache_a_0,4);
      ...
      tmp += vr2r_op16(cache_a_1,cache_a_0,7);
      j += 4;
      if (j <= points-4){ // Block 3
        tmp += vr2r_op16(cache_a_1,cache_a_0,8);
        ...
        tmp += vr2r_op16(cache_a_1,cache_a_0,11);
        j += 4;
  } } }

  // Regular Epilogue
  for(; j <= points-1; j++)
    tmp += unaligned_vload16(&a[i+j]);

  tmp = tmp / vpromotion16(points);
  aligned_vstore16(&b[i], tmp);
}
```

Listing 5.1: Pseudo-code resulting from the overlap optimization on the Moving Average example from Figure 5.1. Assuming $VF$=$VL$=16. `vr2r_op`$_{16}$ refers to vector register-to-register instructions

```
#pragma omp simd aligned(a,b:64) overlap(a:4)
for(i=0; i < N-points; i++){ ... }
```

Figure 5.10: Example of the `overlap` clause for Figure 5.1

applies the algorithm shown in Figure 5.9. Function DECLAREGROUPSYMBOLS declares vector symbols that will be used to host the cache data, as shown in Listing 5.1, line 2.

Function ADDGROUPINITSTMTS inserts the *vl* instructions that initialize the register cache. They will be aligned or unaligned depending on the decision made in the Group Pruning Phase. If $\mathcal{O}_{L,a,i}$ is only of kind *intra*, these statements will be added to the bodies of $L$ and the overlap epilogue. If $\mathcal{O}_{L,a,i}$ is *inter* or both *intra* and *inter*, these statements will precede $L$. The latter is the case in the Moving Average example, as depicted in line 6 of Listing 5.1.

Function ADDGROUPPREITERUPDATESTMTS inserts *vl* instructions in the body of $L$ and the overlap epilogue, as shown in Listing 5.1, lines 11 and 25). They load the new data that will be used in that iteration into the last registers of the cache. They are necessary for both *inter* and *intra* overlap group kinds.

It is important to note that the *vl* instructions generated to initialize and maintain the vector register cache could be partially speculative, but safe. This means that, even though not all the elements loaded may be needed, the fact that at least one element is guarantee that these instructions load valid cache lines. Otherwise, fully speculative loads could raise hazardous exceptions.

ADDGROUPPOSTITERUPDATESTMTS inserts statements in the body of $L$ that shift or readjust data already loaded in the cache, as depicted in Listing 5.1, line 20. They rearrange data that will be reused in the next loop iteration to its expected position. Data that will not be reused is overwritten. These statements are required only for *inter* overlap groups.

Once the register cache has been implemented, the compiler replaces original *vls* in $L$ and in the overlap epilogue code by vector register-to-register instructions, as depicted in Figure 5.9, lines from 6 to 9. They rebuild the original vector operands from data in the register cache. Lines from 13 to 18 in Listing 5.1 shows an example of this rebuild operation.

## 5.5 The Overlap Optimization in OpenMP

In the context of our proposal on SIMD constructs for OpenMP introduced in Chapter 4, we present a new clause named `overlap`. This clause instructs the compiler where and how to apply the overlap optimization (see Section 5.4). It has the following syntax:

```
overlap(var_list[:min_group_loads[,max_groups]])
```

where:

- `var_list` is a list of array and/or pointer variables that the overlap optimization will target.
- `min_group_loads` is the minimum number of *ovls* per loop iteration necessary to constitute an overlap group for each target variable (CARDCONSTR).
- `max_groups` is the maximum number of overlap groups that can be generated for each target variable (GROUPS-CONSTR).

The `overlap` clause allows programmers to perform a systematic benchmarking with different overlap groups and constraints that affect the code generated by the overlap optimization. The parameters `min_group_loads` and `max_groups` are optional, independent from each other and their value must be constant and known at compile time. A value of 0 indicates an undefined parameter. If they are undefined or not specified, their values will be implementation defined. We decided not to expose the control of vector registers to programmers as they are a low-level hardware component.

The compiler could use heuristics to determine default values for these parameters or fix them by default (see our implementation decisions in Section 5.7), but it is hard to automatically obtain optimal values for them. In addition, the use of the `overlap` clause is necessary to guide the compiler on where and over which arrays to apply the optimization to get better performance. Determining this automatically at compile time is even harder due to the number of factors that must be taken into account: selected overlap groups, loop unroll factor, overlap transformation performance/overhead ratio, register and memory unit pressure, and instruction scheduling, among others. These facts support the use of the `overlap` clause and make hints from programmers indispensable to optimally apply the overlap optimization.

The `overlap` clause must be used in conjunction with SIMD constructs defined in OpenMP, as depicted in Figure 5.10. Multiple `overlap` clauses can be used to specify different sets of parameters for different variables.

## 5.6 User Guidelines

In Section 4.5, we described several hints that may help with the application of the new SIMD directives proposed in Chapter 4. Once the vectorization strategy has been chosen using these directives, the next step is to enable the overlap optimization with the `overlap` clause if *ovls* are present in the vectorized code, as described in Section 5.3.2.

The general advice for programmers pursuing the best performance is to apply the overlap optimization on those arrays with a larger number of *ovls* that are as *close* as possible to each other. According to the vectorization scenario, the meaning of *close* is defined in the following way:

**Innermost loop vectorization:** *ovls* are close to each other when their indexes only differ in a small constant offset, such as $\overrightarrow{a[i]}$ and $\overrightarrow{a[i+1]}$, where $i$ is the induction variable of the loop. The smaller this offset is, the closer the *ovls* are.

**Outer loop vectorization:** indexes of *vls* that could overlap across the inner loop iterations typically contain both loops induction variables, such as $\overrightarrow{a[i+j]}$. The inner loop induction variable with a constant step can be seen as a constant offset and then we can proceed as in the innermost loop scenario. Likewise, arrays could also overlap within the same iteration ($\overrightarrow{a[i+j]}$ and $\overrightarrow{a[i+j+1]}$).

In the outer loop vectorization scenario of the Moving Average example in Figure 5.1, the array `a` contains both loops induction variables, so their *vls* will be *ovls*, as described previously.

Again, the only way to know which arrays with *ovls* are the best options to apply the overlap optimization in each particular code is to carry out a trial and error approach. The parameter `min_group_loads` may also help in this process. For example, in an application with several groups of 2 and 4 *ovls*, programmers may run this application twice: one with the parameter unset and another one fixing this parameter to 4. In this way, programmers would know if overlap groups with 2 *ovls* increase or decrease performance. In those outer loop vectorization scenarios where discarding groups is not necessary, programmers could use their application knowledge to set `min_group_loads`. For example, in Moving Average, we know that the number of `points` could be small. To exploit the overlap optimization also in the small cases, programmers could find by benchmarking the minimum number that yields the best performance and overcome the overhead of the code transformation.

The parameter `max_groups` can be used in the same way for the same purpose. Thus, programmers could benchmark their application increasing this parameter to find out which value offers the best performance. As vector registers are usually a scarce resource, a large number of groups is generally not recommended.

Providing all the possible information about runtime values of variables in the code can help the compiler to choose better decisions in the overlap algorithm. For example, programmers could make use of the `suitable` or `aligned` clauses introduced in Chapter 4 to provide information regarding loop bounds and alignment of pointers. This information could help the compiler to optimize the unroll factor and group sizes.

Programmers can also take these guidelines into account to apply the overlap optimization manually. However, this manual application requires the previous vectorization of the code. This initial vectorization step could be applied also by hand or using a source-to-source vectorization tool, as the one described in Chapter 3.

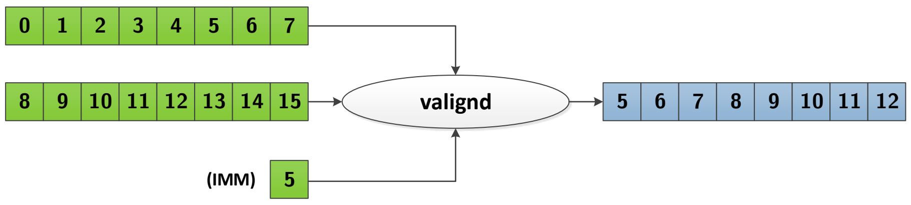


Figure 5.11: Example of the functionality of the *valignd* instruction

## 5.7 Implementation

We implemented a prototype of our approach in our Mercurium C/C++ source-to-source compiler vectorization infrastructure that we proposed in Chapter 3 targeting the Intel Xeon Phi coprocessor (see Section 2.5 for more details on this architecture). As we also described in Chapter 4, this vectorization infrastructure has support for our proposal on SIMD extensions for OpenMP.

Our implementation uses the *valignd* vector register-to-register instruction available in the Intel Xeon Phi coprocessor. Figure 5.11 illustrates how this instruction works. This instruction receives two vector registers and a 32-bit immediate number as input (in green). The instruction combines scalar elements from these two input vector registers by means of shifting them a number of elements specified by the immediate number. The resulting value of the operation is returned in a new vector register (in blue).

Table 5.1 summarizes most of the implementation decisions regarding default policies and default values. We set to 4 *ovmls* the default value of the cardinality constraint (CARDCONSTR). This number, far from being optimal, offered some performance improvement and moderate slowdowns in microbenchmarking experiments for the Intel Xeon Phi architecture.

The constraint on the maximum number of vector registers (VREGSCONSTR) is set to 32 by default. This number favors the choice of aligned *vls* instead of unaligned *vls* whenever this threshold of vector registers is not exceeded. In this way, the algorithm is not limited by this resource in most cases. This was our intention since in a source-to-source compiler we do not have control on this low-level component of the architecture.

The constraint on the number of groups (GROUPSCONSTR) is set to unlimited by default. This value can easily be changed using the *overlap* clause.

The policy to maintain data on vector register caches depends on how this policy affects the number of vector registers and VREGSCONSTR. We always try to use aligned *vls* to maintain the register cache. However, if using aligned *vls* results in a higher number of vector registers than the limit imposed by VREGSCONSTR, we then use unaligned *vls* to try to fulfill VREGSCONSTR.

The policy to unroll a loop in our implementations depends on the vectorization scenario where we are applying the overlap optimization. In inner loop vector-

| Constraint | Default Value | Register Cache Load Policy |
|---|---|---|
| CARDCONSTR | 4 | if(VREGSCONSTR is OK) |
| VREGSCONSTR | 32 | aligned *vls* |
| GROUPSCONSTR | unlimited | else unaligned *vls* |

| Unrolling Analysis Phase | | |
|---|---|---|
| Unrolling Policy | Outer loop vectorization: unroll inner loop | |
| | Inner loop vectorization: no unroll | |
| Parameters | $UF_O$ = Smallest | $UF_{EPI}$ = CARDCONSTR |

Table 5.1: Default policies and values adopted in our implementation

ization scenarios, we prevent the loop to be unrolled. We only apply unrolling to the innermost loop of outer loop vectorization scenarios. We choose the minimum $UF_O$ that satisfies Equations 5.8 – 5.10 and it maximizes efficiency. The parameter $UF_{EPI}$ used to generate the overlap epilogue is set to the value of CARDCONSTR, i.e., to 4 by default.

## 5.8 Evaluation

We conducted experiments targeting the Intel Xeon Phi coprocessor using our implementation described in Section 5.7. In these experiments we followed two steps. First, we fine optimized five well-known kernels used in real-world applications from diverse fields for the Intel Xeon Phi coprocessor (see Section 5.8.3). Afterwards, we evaluated the performance of the overlap optimization over these highly-optimized versions of the kernels (see Section 5.8.4).

### 5.8.1 Benchmarks

Our benchmark set contains five kernels from real-world applications using realistic workloads. They cover a wide spectrum of stencil algorithms and vectorization scenarios. Table 5.2 summarizes their main characteristics.

**Moving Average** is a low-pass finite impulse response (FIR) filter that is widely used in statistics, economics and signal processing. One of its most popular uses is the smoothing of short-term fluctuations when analyzing long data series. Our benchmark implements the Simple Moving Average Filter that we have use throughout this chapter to exemplify several aspects of the overlap optimization. A snippet of the code is shown in Figure 5.1 at Section 5.1.1. The $\texttt{i}^{th}$ output value of `b` is computed as the average of `points` elements forward from the $\texttt{i}^{th}$ element of the input array `a`. We ran experiments for different number of `points`: 4, 8, 16, 32, 64 and 128.

**Convolution** implements a stencil convolution filter on an image. These kinds

| Benchmark | Version | Stencil Kind (Vect. Strategy) | Input Size, Tsteps, Mem. Footprint | Evaluation Key Factor |
|---|---|---|---|---|
| Moving Average | 1D | 1D forward (1st outer loop) | 536M, 50, 4.0GB | Unknown order at compile time |
| Convolution | 2D | convolution (2nd outer loop) | $24000^2$, 30, 4.3GB | Unknown order at compile time. `min_group_loads` |
| Stencil Probe | 1D | 28th-order star (1D: innermost loop 2D&3D: 1st outer loop) | 525M, 200, 3.9GB | Very high-order star stencil. 1D, 2D and 3D performance comparison |
| | 2D | | $22964^2$, 50, 3.9GB | |
| | 3D | | $820^3$, 25, 4.1GB | |
| Acoustic Propagator | 3D | 8th-order star (innermost loop) | 928x448x840, 100, 5.5GB | Multi-grid stencil |
| Cubic Stencil | 2nd-order | cubic (innermost loop) | $832^3$, 100, 4.3GB | Cubic stencil evaluation with 27 and 125 points |
| | 4th-order | | $834^3$, 20, 4.3GB | |

Table 5.2: Benchmarks Summary

of algorithms are widely used in image processing to modify or analyze certain image properties (image blurring, sharpening or edge detection, among others). We used the code of the motivating example from Naishlos et al. [149]. We ran experiments for different filter sizes (3x3, 5x5, 7x7, 10x10 and 16x16).

**Stencil Probe** [78] is a benchmark based on the Chombo's Heat Equation. It is used to explore the behavior of optimizations in star stencil codes. We chose the 28th-order 3D version (85 stencil points) proposed by de la Cruz et al. [48]. We extended it to also have 1D (29 stencil points) and 2D (57 stencil points) versions.

**Acoustic Propagator** computes the acoustic wave propagation in an isotropic non-elastic medium. It solves a Partial Differential Equation (PDE) that involves velocity and two spatial components from previous time steps [10]. Our private algorithm is based on a 8th-order 3D star stencil similar to that published by Intel [73].

**Cubic Stencil** implements a cubic stencil algorithm where points involved in the neighboring computation form a dense cube around the central point to be updated. We used the 2nd-order 3D cubic stencil code from Datta et al. [47]. We also extended it to have a 4th-order version (125 points).

### 5.8.2 Environment and Methodology

We compiled versions of every benchmark with the Intel C/C++ compiler version 15.0.1 (*icc* versions) and the Mercurium C/C++ source-to-source compiler (*mcc* versions), both using the Intel OpenMP Runtime Library included in the Intel compiler release. Mercurium versions are vectorized with the Mercurium compiler vectorization infrastructure that we introduced in Chapter 3, using the Intel C/C++ compiler as native compiler to generate the binary code.

All versions in our experiments, including the baseline, are annotated with OpenMP worksharing and the SIMD extensions that we proposed in Chapter 4.

Memory allocations are aligned to the page boundary of the architecture (4096 bytes) and pointers are qualified with the `restrict` attribute to maximize compiler optimizations. We also used the following compilation flags: `-fopenmp -O3 -mmic -restrict -ansi-alias`. Both compilers offer similar vectorization performance over baseline codes.

As evaluation metrics, we use the absolute performance in millions of elements or pixels updated per second. We report the speed-up of the highly-optimized versions for the Intel Xeon Phi coprocessor over the SIMD-annotated baseline (see Section 5.8.3), and the speed-up of the overlap optimization over the highly-optimized versions compiled with the Mercurium compiler (See Section 5.8.4). We aim to differentiate the improvement of the data reuse of the overlap optimization from the one of other co-lateral transformations in the overlap optimization, such as loop unrolling. Our performance figures are the arithmetic mean of 50 executions.

We ran our benchmarks natively in an Intel Xeon Phi coprocessor C0PRQ-7120 (61 cores @ 1.23 GHz, 16 GB of GDDR Memory) introduced in Section 2.5 at Chapter 2. We use 122, 183 and 244 threads on a balanced thread affinity policy (2, 3 and 4 threads per core, respectively). We added the compile-time flag `-qopt-threads-per-core=#threads_per_core` to optimize the scheduling of instructions depending on the number of threads per core at runtime.

### 5.8.3 Generic Optimizations

We applied the following optimizations to our set of benchmarks in order to improve their performance and scalability in the Intel Xeon Phi coprocessor:

**Array Indexes Simplification (AIS):** We applied common sub-expression elimination and loop-invariant code motion [3, 2] by hand to optimize the indexes of array accesses.

**Loop Unrolling (U):** To minimize the impact on the speed-up of the loop unrolling [3, 81] applied by the overlap optimization, we unrolled the same loop in versions without the overlap optimization.

**Memory Alignment (MA):** We used the *aligned* clause proposed in our SIMD extensions and the `__assume_aligned` and `__assume` compiler hints to provide the compiler with information to maximize aligned vector memory accesses and prevent prologue and epilogue loops in the vectorization process [76].

**Software Prefetching (SP):** We implemented manual software prefetching for L1 and L2 cache levels using the `_mm_ prefetch` intrinsic. We performed an extensive exploration to find out the best prefetching distances. Our prefetching strategies include data prefetching among loop iterations and initial prefetching before loops [85]. This approach noticeably outperforms software prefetching automatically generated by the Intel C/C++ compiler.

**Loop Blocking (LB):** We applied loop blocking [154, 81] to loops that traverse

| Benchmark | Version | Optimizations | | Input Param. | Threads | Baseline | Speed-up | icc/mcc (time) |
|---|---|---|---|---|---|---|---|---|
| Moving Average | 1D | AIS, MA, U, SP | | 4 | 244 | 484.74 | 1.09 | 1.02 |
| | | | | 8 | | 449.10 | 1.18 | 1.02 |
| | | | | 16 | | 337.06 | 1.51 | 1.06 |
| | | | | 32 | | 122.02 | 1.66 | 1.07 |
| | | | | 64 | | 212.68 | 1.72 | 1.06 |
| | | | | 128 | | 65.86 | 1.76 | 1.05 |
| Convol. | 2D | AIS, MA, U, SP | | $3^2$ | 183 | 276.43 | 1.10 | - |
| | | | | $5^2$ | | 135.18 | 1.53 | - |
| | | | | $7^2$ | | 78.33 | 1.60 | - |
| | | | | $10^2$ | | 41.92 | 1.67 | - |
| | | | | $16^2$ | | 12.06 | 2.58 | - |
| Stencil Probe | 1D | AIS, MA, SP, | - | - | 122 | 9683.38 | 1.10 | 0.99 |
| | | | | | 183 | 10400.45 | 1.06 | 1.00 |
| | 2D | | LB, LI | - | 183 | 1840.77 | 3.58 | 1.27 |
| | | | | | 244 | 1825.26 | 3.48 | 1.12 |
| | 3D | | LB, LI, LD | - | 183 | 66.61 | 27.66 | 2.27 |
| | | | | | 244 | 49.65 | 38.42 | 2.15 |
| Acoustic Propag. | 3D | AIS, MA, SP, LB | | - | 122 | 4410.02 | 1.21 | 1.00 |
| | | | | | 183 | 3790.25 | 1.49 | 1.02 |
| Cubic Stencil | 2nd order | AIS, MA, SP, LB, LD | | - | 122 | 8020.97 | 1.23 | 0.96 |
| | | | | | 183 | 7675.79 | 1.35 | 1.00 |
| | 4th order | | | - | 122 | 1604.78 | 1.71 | 1.02 |
| | | | | | 183 | 1129.22 | 2.43 | 0.98 |

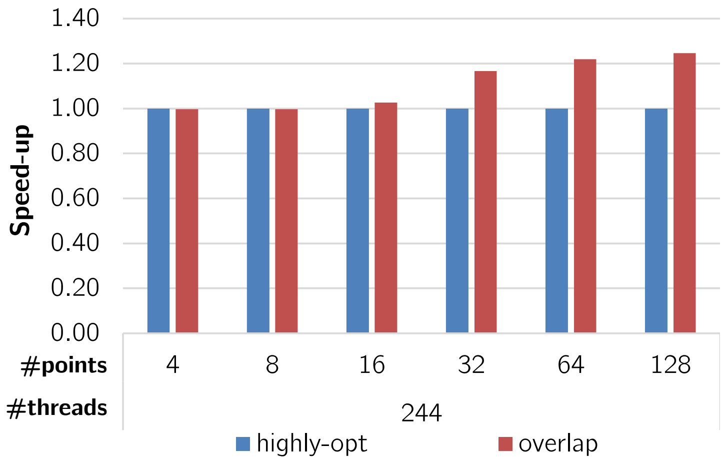
Table 5.3: Generic optimizations applied to each benchmark. Absolute performance (in M{elements/pixels}/s) of the best thread configurations for the baseline. Speed-up of the optimizations over the baseline. The input parameters for Moving Average is the number of `points` and for Convolution is the filter size

large multidimensional data structures. We widely explored different block sizes and their scheduling among threads.

**Loop Interchange (LI):** We applied loop interchange [6, 81] to explore several traversal strategies on grid structures with significant reuse in other dimensions than in the memory contiguous one. If the innermost loop is annotated with the SIMD directive and it is interchanged, initial innermost loop vectorization turns into outer loop vectorization [111].

**Loop Distribution (LD):** Splitting a single loop body computation into two loops [118, 81] reduced register pressure and memory contention issues in some benchmarks.

Best prefetching distance and best block size explorations were jointly carried out since they affect each other. Table 5.3 shows which generic optimizations were applied to each particular benchmark, the absolute performance of the best thread



| #points | Melems/s | $\mathcal{O}$s, Kind, (%*ovls*/iteration) |
|---|---|---|
| 4 | 528.44 | 1 group of 16 *ovls*, inter, (100%) |
| 8 | 527.07 | |
| 16 | 522.03 | |
| 32 | 410.79 | |
| 64 | 255.97 | |
| 128 | 144.61 | |

Figure 5.12: Moving Average absolute performance, speed-up and kind of groups (244 threads)

configurations before optimizing the code (baseline) and the speed-up obtained after it. Both versions were vectorized with the Mercurium compiler. As depicted, generic optimizations yield substantial performance over the baseline, even though the latter is also annotated with SIMD extensions that lead to the successful vectorization of the code. These optimizations mainly address memory bandwidth issues. However, loop distribution (LD) also targets register spilling and it is critical in some benchmarks, such as in the 3D Stencil Probe (improvement of 27% and 38%).

Whereas Intel and Mercurium compilers yield similar performance on baseline codes, icc/mcc ratio in Table 5.3 states how Mercurium vectorization capabilities are on the optimized versions with respect to the Intel compiler version. Values close to or greater than one state that Mercurium vectorization is comparable or better than Intel's. This fact increases the confidence in the performance results obtained with Mercurium according to state-of-the-art vectorization techniques. In addition, it also validates our vectorization infrastructure proposed in Chapter 3. Performance differences are mainly due to problems in the Intel compiler when vectorizing outer loops. That is the reason why the icc/mcc ratio is not available in Convolution.

### 5.8.4 Overlap Optimization Performance

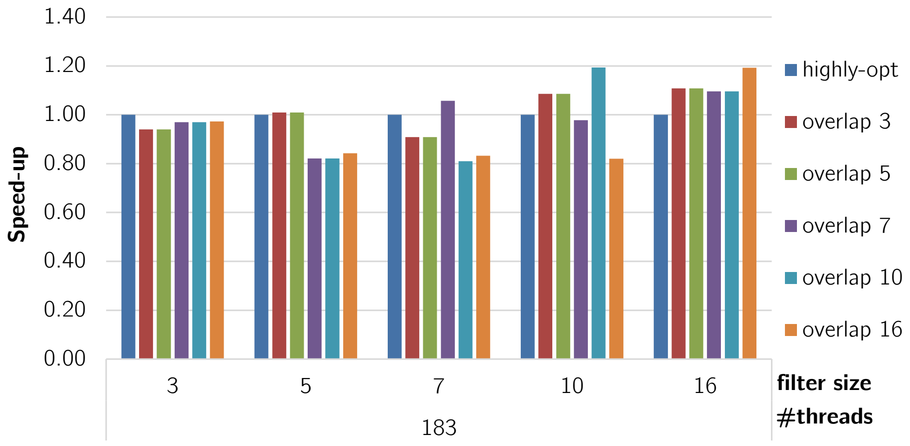
In this section, we present the performance results of the overlap optimization in Figure 5.12, Figure 5.13 and Figure 5.14. These figures contain a chart on the top with the speed-up over the highly-optimized version of the benchmarks. On the bottom, they have a table that contains the absolute performance results of the benchmark version using the overlap optimization. The last column of each table depicts the number and kind of overlap groups and the percentage of vector loads affected by the overlap optimization (*ovls*) out of all vector loads per iteration. Both the highly-optimized version and the version with the overlap optimization are compiled with the Mercurium compiler.

The performance of the **Moving Average Filter** is shown in Figure 5.12. Due to the outer loop vectorization scenario, the overlap optimization unrolls the innermost loop 16 times. It builds a 2-register cache for the single overlap group of the array `a`. An overlap epilogue is also generated with $UF_{EPI} = 4$. The optimized version used as baseline includes the best unroll factor found for every input set. This counteracts the performance gain from the loop unrolling applied by the overlap optimization.

The speed-up results of this benchmark show a gradual performance gain when `points` increases (greater than or equal to 16) and so does the number of *ovls*. This means that this benchmark needs an overlap group with a large cardinality to overcome the overhead of the register cache.

In the **Convolution Filter** benchmark, vectorization takes place from the second outer loop (image column traversal). We evaluate how the parameter `min_group_loads` (CARDCONSTR) impacts performance in the overlap optimization. We use the values of the filter size, which are unknown at compile time. As in our implementation we set the default value of the $UF_{EPI}$ parameter to the value of CARDCONSTR, changing the values of the parameter `min_group_loads` also affects the generation of the overlap epilogue (see Section 5.7).

The performance results of this benchmark are summarized in Figure 5.13. The best configuration for each filter size is highlighted except in filter 3x3, where the overlap optimization only causes a performance slowdown. Speed-up reveals how sensitive the transformation is to the input size and the `min_group_loads` parameter. For the smallest filter sizes, the overlap optimization can obtain a minimal gain (if any) or up to 19% of penalty (size=7, `min_group_loads`=10) depending on the value of `min_group_loads`. For larger filter sizes, the most suitable `min_group_loads` value yields up to 19% of speed-up (size=`min_group_loads`=10), whereas other values lead to up to 18% of slowdown (size=10, `min_group_loads`=16) for the same filter size. The slowdown is higher when the filter size is lower than `min_group_loads` and the overlap code is generated but not executed because CARDCONSTR is not satisfied. Since filter size is unknown at compile time, the `min_group_loads` parameter is truly important to tune the overlap optimization for this benchmark. If programmers know the usual input characteristics, they can set



| Filter Size | Mpixels/s varying #min_group_loads | | | | | $\mathcal{O}$s, Kind, (%*ovls*/iteration) |
|---|---|---|---|---|---|---|
| | 3 | 5 | 7 | 10 | 16 | |
| 3 | 286.82 | 286.75 | 295.69 | 295.90 | 296.86 | 1 group of 16 *ovls*, inter, (100%) |
| 5 | 208.77 | **208.80** | 169.77 | 169.87 | 174.30 | |
| 7 | 113.61 | 113.59 | **132.17** | 101.30 | 104.04 | |
| 10 | 76.07 | 76.07 | 68.49 | **83.65** | 57.50 | |
| 16 | 34.51 | 34.50 | 34.15 | 34.15 | **37.16** | |

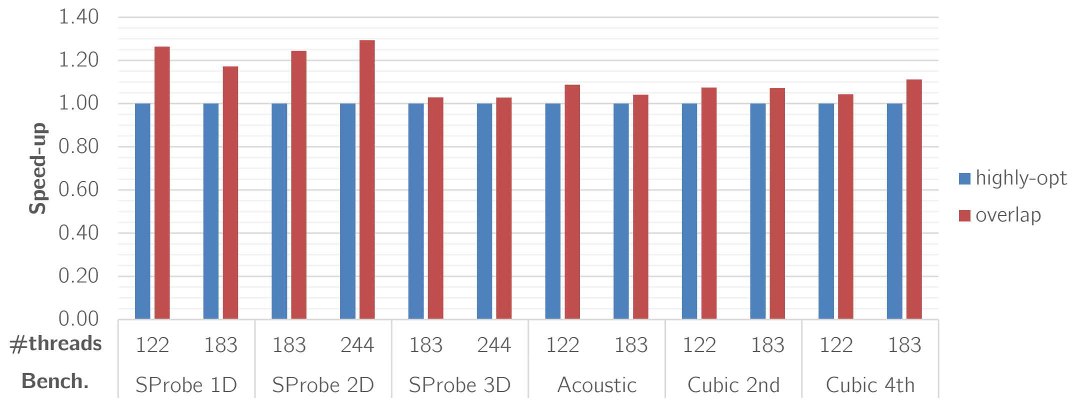
Figure 5.13: Absolute performance in Mpixels/s and speed-up of the overlap optimization on the Convolution Filter (183 theads, 3 threads/core)

this parameter to get the best performance.

Figure 5.14 gathers performance results from StencilProbe, Acoustic Propagator and Cubic Stencil benchmarks. In the three versions of the **StencilProbe** benchmark, the overlap optimization finds only one overlap group with 29 *ovls*. In the 1D version, all *vls* are *ovls*. The overlap optimization yields a 27% speed-up over the highly optimized version with 122 threads. This speed-up greatly outperforms any other version with higher thread configuration. In the 2D version, the overlap optimization obtains a gain of 29% and 24% with 183 and 244 threads, respectively. Even when only 51% of *vls* are optimized, the optimization reduces pressure on the memory system. In the 3D version, the overlap optimization only yields 3% improvement because it only affects 34% of *vls* per iteration. Furthermore, memory bandwidth and computation per updated elements are dramatically high due to the high-order of the stencil.

In the **Acoustic Propagator** benchmark, however, the overlap optimization only provides a modest improvement of 4-9%. This low improvement is due to the high memory requirements of this application caused by the several grids involved in the computation. Furthermore, the overlap optimization only affects 33% of *vls* in the code.

In the **Cubic Stencil** benchmark, unlike in previous star stencil codes, there



| Bench. | Ver. | Threads | Melements/s | $\mathcal{O}$s, Kind, (%*ovls*/iteration) | |
|---|---|---|---|---|---|
| Stencil Probe | 1D | 122 | 13522.11 | 1 group of 29 *ovls*, | both, (100%) |
| | | 183 | 12943.67 | | |
| | 2D | 183 | 8195.33 | | intra, (51%) |
| | | 244 | 8213.63 | | |
| | 3D | 183 | 1895.67 | | both, (34%) |
| | | 244 | 1960.73 | | |
| Acoustic Propagator | 3D | 122 | 5783.41 | 1 group of 9 *ovls*, both, (33%) | |
| | | 183 | 5892.36 | | |
| Cubic Stencil | 2nd order | 122 | 10570.29 | 9 groups of 3, | both, (100%) |
| | | 183 | 11135.66 | | |
| | 4th order | 122 | 2877.68 | 25 groups of 5, | |
| | | 183 | 3082.98 | | |

Figure 5.14: Acoustic Propagator, Cubic Stencil and StencilProbe

are multiple overlap groups that affect 100% of *vls*. Even in the 2nd-order version, with only three elements per group, performance results show that the overlap optimization yields an extra 7% of performance over the optimized version. In the 4th-order version, larger overlap groups yield 12% gain. This demonstrates that the overlap optimization can improve performance under scenarios with high memory and computational requirements if the percentage of *ovls* per loop iteration is high.

## 5.9 Chapter Summary and Discussion

In this chapter, we propose a compiler optimization for vector code in the context of the SIMD extensions that we presented in Chapter 4 for OpenMP. This optimization targets overlapped vector loads, i.e., vector loads that redundantly read scalar elements from memory. This memory access pattern occurs naturally after the vectorization process of stencil codes. Instead, our optimization loads these elements once creating a vector register cache and it rebuilds the original vector operand of the code using advanced register-to-register vector instructions. This code is potentially more efficient as it improves the memory bandwidth. In addition, our approach make use of advanced vector instructions that compilers do not widely exploit automatically.

In order to guide the compiler in the application of this optimization, we propose a new clause called *overlap*. This clause is intended to be used in conjunction with SIMD constructs. It is aimed at allowing programmers to easily enable and tune our optimization.

We implement a prototype of our proposal in the Mercurium vectorization infrastructure targeting the Intel Xeon Phi coprocessor. We evaluate our proposal on a representative set of different stencil kernels and workloads used in real-world applications. Performance results show improvements of up to 29% over previously highly-optimized versions that successfully exploited SIMD extensions. This gain is significant even in codes with high memory requirements if the number of vector loads affected by the optimization is large. In some cases, the overlap optimization helps to achieve the best benchmark performance with a considerably smaller number of threads than versions not exploiting this optimization. Moreover, results show how the performance of this optimization is highly sensitive to many factors, such as number of overlapped vector loads per group and the characteristics of the surrounding code. Therefore, user guidance and systematic benchmarking is crucial to obtain maximum performance and avoid slowdowns with this optimization.

### 5.9.1 Automatic Application of the Overlap Optimization

One of the questions that arose in several occasions when we introduced the overlap optimization was whether the *overlap* clause is really necessary.

Intel compiler engineers pointed out that maybe a production compiler could apply this optimization automatically. Particularly, applying the overlap optimization as a source-to-source transformation could inherently suffer from being far ahead of the register allocation phase, where actual register availability is determined. For example, a full compiler could automatically create overlap groups as large as possible in a middle-end phase. Later, in the register allocation phase, the compiler could detect whether there are too much register pressure and partially or totally undo the optimization to prevent register spilling.

From our point of view, the idea introduced by the Intel compiler team is a great idea. Following that approach, the compiler could apply the optimization successfully in some cases. However, we firmly believe that some other cases will lead to performance penalty or, at least, not to the best achievable performance. We motivate our position with the following thoughts:

- In a scenario with multiple overlap groups, if the compiler had to revert partially the overlap optimization in the register allocation phase, it could be difficult for it to determine which final configuration of groups would yield the best performance.

- If the overlap clause entailed loop unrolling, the register allocation phase would be too late in the compilation process to be able to successfully revert this unrolling of the loop.
- It could be difficult for the compiler to determine which set of optimizations will yield better performance if the overlap optimization cannot be jointly applied with other optimizations.
- If some optimizations are discarded in favor of the overlap optimization and the compiler reverts the optimization in the register allocation phase, it would be too late to apply the discarded optimizations.

These are just a few thoughts that offer a glimpse into the complexity that entails this optimization and its relation with other process involved in the optimization of code. We think that further work in this direction would be interesting aiming at determining how far the compiler could go in the automatic application of the overlap optimization.

### 5.9.2 Incorporation into the OpenMP Standard

This proposal was presented to the OpenMP committee in the face-to-face meeting held in Barcelona in May, 2015. The feedback was positive and the addition of our *overlap* clause will be studied for the 5.0 version of the standard.

We think that this proposal has room in OpenMP as the latest incorporation to the standard, such as the SIMD extensions, are towards providing programmers with further control on the optimization process of their codes. It is known that compilers can apply further optimizations on top of OpenMP-annotated codes. However, these optimizations are limited by the OpenMP semantic. Optimizations cannot violate the OpenMP semantic even though the compiler knows it is worth their application. In order to palliate this problem, we think that further extensions are necessary to describe how code optimizations can be applied over the semantic of some OpenMP constructs. These extensions will give more control to those programmers that want to take advantage of them instead of opt for other more manual and cumbersome approaches.

# Chapter 6

# SIMD Synchronization Barrier and Reductions in OpenMP

## 6.1 Introduction

In this section, we describe our proposal for a new synchronization barrier and reduction scheme for OpenMP. Our approach has been specifically designed to take advantage of new hardware resources, such as simultaneous multi-threading (SMT) and SIMD instructions available in modern multi and many-core processors.

We implemented a prototype of our combined SIMD barrier and SIMD reduction algorithms in the production-state Intel OpenMP Runtime Library. This implementation demonstrates that our barrier and reduction schemes allow an efficient implementation in the real context of the OpenMP programming model.

We performed the evaluation of our prototype with benchmarks that make intensive use of OpenMP barrier and reduction primitives in order to measure the overhead of each one of these parallel components. In addition, we also included in our evaluation a set of more application-like benchmarks that exploit fine-grained parallelism.

### 6.1.1 Motivation

Synchronization barriers and reduction operations have been widely studied topics since the beginning of the supercomputer era. They are the most common operations in a set of collective synchronization and communication primitives available in many parallel programming models, such as MPI [103] and OpenMP [117]. These primitives became especially relevant for large-scale supercomputers. In these large systems, the high latency of their complex network and the large number of threads involved in the computation turn communication primitives into a potential bottleneck. The increasing core count within a single chip processor and their associated complex interconnection network are raising the relevance of the efficient implementation of these communication primitives also for multi and many-core architectures.

### Barriers and Reductions in OpenMP

Some aspects of barrier and reduction primitives in the OpenMP standard significantly restrict their implementation alternatives in OpenMP runtimes.

In a thread parallel region, reduction operations defined on a variable are only allowed if the variable is shared by all threads involved in the execution of that parallel region. This means that at some point of the computation such a variable must be updated with the final reduced value. In addition, this approach requires an extra synchronization step to establish a point where the final reduction value will be available to all threads.

This definition of the reduction primitive makes some reduction algorithms inefficient. For instance, using a traditional all-to-all dissemination algorithm, very common in MPI, is inefficient as the update of the reduction shared variable makes the all-to-all communication useless.

As for synchronization barriers, they have a special meaning in OpenMP. Apart from the synchronization point, a barrier primitive has more semantic connotations that increase its implementation complexity. According to the OpenMP 4.0 standard, these are some of them:

- The synchronization barrier does not apply to all threads of the application but to threads within the same `team` of threads. Several teams of threads can be defined in the same application and their synchronization at the barrier must be independent from each other.
- The global result from reduction primitives is only guaranteed to be available to all threads after a synchronization barrier.
- The barrier primitive is a task scheduling point. This means that all OpenMP tasks created by that team of threads can be scheduled and executed by those threads waiting at the synchronization barrier.
- A synchronization barrier implies a memory flush. This means that threads at the synchronization barrier must make their local memory view consistent with main memory.
- The barrier primitive is an implicit cancellation point. A cancellation point is a region where threads must check if they have to cancel the execution of that parallel region.

Although naive and inefficient implementations could allow a simple barrier scheme, these semantic requirements lead to more elaborated schemes that introduce important restrictions in the implementation of this primitive in OpenMP runtimes.

The Intel OpenMP Runtime Library is an interesting example of a production level OpenMP runtime library. In this OpenMP implementation, the barrier primitive is split into a *gather* phase and a *release* phase, as we described in Section 2.4. This scheme allows to separate the moment when it is known that all threads have reached the barrier from the releasing of threads. In this way, once all threads have arrived in the barrier, reduction operations are computed and the reduction shared variable is updated before the release phase. This piggybacked implementation of reduction operations does not require extra synchronization steps beyond the barrier itself.

As said above, the additional semantic that OpenMP defines for the barrier primitive increases its complexity. Thus, general approaches used in other programming models may not be applicable or may be inefficient in the context of OpenMP. In addition, both operations are strongly coupled and they need to be addressed jointly.

### Barriers and Reductions for Many-core Architectures

Barriers and parallel reductions have gone quite unnoticed for general-purpose chip multi-processors (CMPs) with a small number of cores and threads/core. As the core integration has improved, new multi- and many-core processors have largely increased their number of cores and threads. Collective synchronization and communication primitives have a higher performance impact on these processors with a large count of threads. Computational work is distributed among a large number of workers. As a consequence, the ratio of computation versus communication easily worsens, as occurs when exploiting fine-grained parallelism[133].

Figure 6.1 shows the clock cycles and the execution time of a single barrier in a quad-socket Intel Xeon processor and an Intel Xeon Phi coprocessor. We vary the number of cores but we always use the maximum number of threads per core available in the architecture (2 threads/core in the Intel Xeon and 4 threads/core in the Intel Xeon Phi).

Regarding the number of cycles, in general terms, the Intel Xeon Phi coprocessor always needs more cycles than the Intel Xeon to synchronize the same number of cores. However, the scalability as a function of the number of cores is better in the many-core architecture as they are specifically designed to scale well with large number of cores [127]. From the single-chip point of view, the 8 cores (16 threads) of the Intel Xeon processor single-socket take 2,400 clock cycles in average to synchronize whereas the 61 cores (244 threads) of the Intel Xeon Phi coprocessor take 15,900 cycles. Even in the quad-socket scenario (32 cores / 64 threads), a single barrier consumes only 9,700 clock ticks in average. Despite the fact that synchronizing 4 multi-core processors requires a more expensive off-chip communication, the performance is still notably better than in the single-chip Intel Xeon Phi coprocessor.

Such significant difference in the number of cycles is mainly because the many-core architecture requires a more complex communication network to successfully

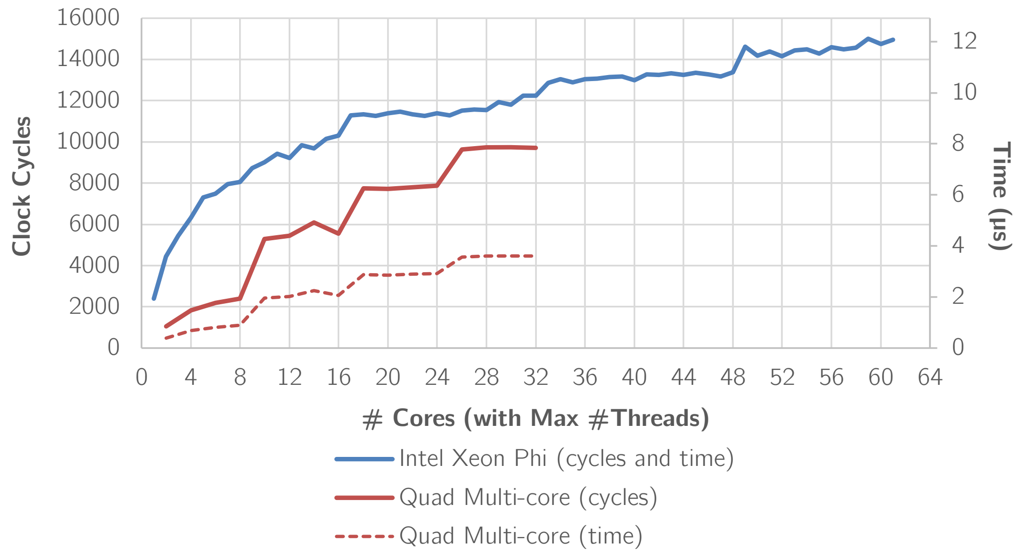


Figure 6.1: Clock cycles (left axis) and execution time (right axis) of EPCC barrier with the default Intel OpenMP barrier (Intel Composer XE 2013 Update 4). Quad-socket Intel Xeon E5-4650 at 2.7 GHz (2 threads per core / 32 cores / 64 threads) and Intel Xeon Phi 7120P at 1.238 GHz (4 threads per core / 61 cores / 244 threads). Running with the maximum number of threads per core. Mean of 10 executions.

deal with a larger number of cores on-chip. In addition, the many-core architecture features more threads per core (4-way SMT). This involves a larger number of threads per core in the synchronization process. Even though synchronizing threads running on the same core should be very fast, barrier algorithms require to be specifically designed or configured to take advantage of that fact.

Regarding time results, many-core architectures may suffer from harder frequency constraints than multi-core architectures [66]. The Intel Xeon Phi coprocessor used in our experiments runs at a considerably lower frequency (1.238 GHz) than the Intel Xeon processor (2.7 GHz). Thus, as far as a single barrier execution time is concerned, the Intel Xeon processor further outperforms the Intel Xeon Phi coprocessor.

All these facts reveal that software barriers and reduction algorithms need to be revisited from the many-core architectures viewpoint. In these architectures, the interconnection network is essential but it is also important to be aware of other core features, such as the number of hardware threads per core and SIMD instruction sets.

### SIMD instructions for Barriers and Reduction Algorithms

Outstanding SIMD resources are also available in modern architectures and their use on barriers and reduction algorithms has not yet been explored.

Barrier algorithms may benefit from vector memory instructions that can read or write several scalar elements at a time. These instructions can be used either to check the arrival in the barrier or to signal the release of multiple threads at once.

Reduction algorithms can use vector memory and arithmetic instructions to implement the partial reduction value of multiple threads at once. In addition, new SIMD instruction sets also include more advanced vector instructions that can be used efficiently to reduce the scalar elements in a vector register into a single scalar element.

The aforementioned facts lead us to think that SIMD instructions can impact the performance of barrier and reduction algorithms. They could reduce the number of executed instructions and even the traffic through the communication network. However, the application of SIMD instructions to barrier and reduction algorithms requires a specific design of those algorithms. As we already know from Chapter 3 and Chapter 4, the efficient exploitation of SIMD instructions implies the fulfillment of certain constraints related with the stride of data in memory and its alignment.

### 6.1.2 Objectives

The main objective of this contribution is to propose new synchronization barrier and reduction schemes specifically designed to exploit SIMD instructions. Furthermore, our design will also take into account that multiple hardware threads could be running within the same core with the simultaneous multi-threading technology. Our proposal must fit into the OpenMP programming model and fulfill the requirements of the barrier and reduction primitives defined in the standard.

We target the Intel Xeon Phi coprocessor as many-core architecture with a large number of cores, a 4-way simultaneous multi-threading and a powerful 512-bit SIMD instruction set. The evaluation of this proposal must be compared with current production-state barrier and reduction schemes available and tuned for the chosen architecture.

## 6.2 SIMD Combining Tree Barrier Algorithm

We propose a tree barrier algorithm for current multi and many-core architectures that makes use of SIMD instructions. Our proposal also benefits from having multiple hardware threads per core. We denominate our barrier algorithm as *reconfigurable multi-degree combining tree barrier with lock-free distributed SIMD counters*. The reasons for this qualification are the following:

**Combining tree:** The barrier scheme is based on a traditional combining tree data structure.

**Reconfigurable:** The combining tree data structure of the barrier can be reconfigured depending on the number of threads and how we want to distribute them across this structure.

**Multi-degree:** The combining tree data structure of the barrier can have a different branching factor (number of children) per level.

**Distributed SIMD counters:** Each node of the combining tree contains individual counters per thread (distributed counters) partially orchestrated by SIMD memory operations.

**Lock-free:** All the counters of the combining tree are lock-free and locks are not necessary at any point of the synchronization.

The tree-like internal data structure of the barrier allows a split design with independent *gather* and *release* phases, as we motivated in Section 6.1.1. Consequently, this barrier algorithm is suitable for implementing the barrier primitive in any OpenMP runtime library, including that of Intel.

### 6.2.1 Barrier Scheme

Our barrier algorithm deploys a combining tree data structure which is walked from leaves to root in the *gather* phase and from root to leaves in the *release* phase. In this tree structure, a pre-established number of threads is assigned to each node in the same level of the tree. The group size of each level is statically defined in the initialization phase of the barrier. This group size makes the number of threads per group the same for every node within that level whereas it may be different for each particular level of the tree. If the number of threads reaching a level is not divisible by the group size, there will be an additional group (the last one) containing the remainder of threads. The total number of nodes/groups per level depends on the group size of that level and the total number of threads that execute that level. Figure 6.2 illustrates the scheme of the barrier for the synchronization of 21 threads with group sizes of 6, 2 and 2 for levels 0, 1 and 2 of the tree, respectively.

In each particular tree node only one thread is designated to play the master role (M) of the group. The remaining threads assume the slave role (S). These roles will affect their duties in the gather and release phases of the barrier, described in Section 6.2.2 and Section 6.2.3, respectively.

Those threads assigned to the same tree node constitute an independent group of synchronization. Inside this group, each thread has an exclusive 1-byte counter available for taking part into the synchronization process. All the group counters are allocated contiguously in memory satisfying the alignment constraints of the underlying SIMD instruction set. These group counters will be handled with SIMD memory operations by the master thread. To prevent false-sharing, only one group of counters is placed per cache line. The remaining memory in the line is padded. In Figure 6.2 1-byte counters are depicted in red and green, and padding in white.

This particular tree-based design with distributed counters allows exploiting SIMD resources and inter-thread cache locality in cores with simultaneous multi-

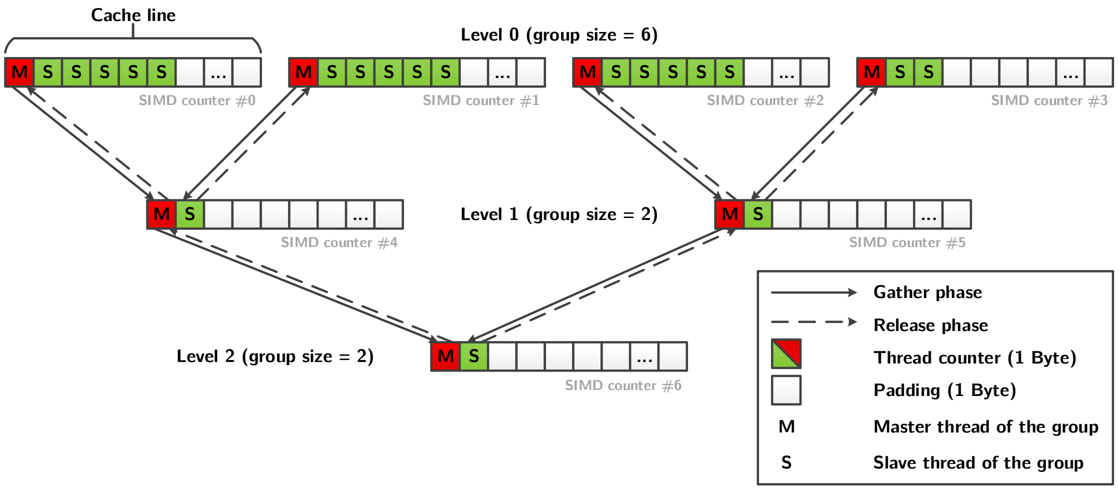


Figure 6.2: SIMD Combining tree barrier scheme for 21 threads. Group sizes of level 0, 1 and 2 are 6, 2 and 2, respectively. Seven tree nodes in total with their respective distributed SIMD counters.

threading. Inter-thread locality may be useful to carry out a first intra-core synchronization step.

In addition, the tree structure also offers the possibility of reshaping the flavor of the barrier from a multi-level combining tree structure to a lock-free totally centralized barrier. Hence, it will be possible to take advantage of two utterly different barrier algorithms with only one implementation. The most appropriate one can be chosen depending on the number of threads and the characteristics of the system.

### 6.2.2 Gather Phase

Listing 6.1 shows the pseudo-code of the gather phase of the SIMD barrier. In this phase, the intra-group synchronization is performed through the distributed counters introduced in Section 6.2.1. These counters are represented in Listing 6.1 with the `tree_flags` variable.

Each slave thread signals its arrival in the barrier changing the value of its exclusive 1-byte counter (line 29). It is important to note that cache line false sharing may occur in case that several slave threads from different cores in the same group perform the signaling arrival at the same time. Nevertheless, simultaneous multi-threading scenarios can benefit from this fact because groups with only slave threads from the same core will not suffer from this false sharing penalty.

Regarding the master thread, it waits for all its slave threads to reach the group by checking the slave counters using SIMD instructions (lines 16 and 17). Just one vector load allows reading at the same time as many slave counters as bytes the vector length has. Therefore, the use of SIMD instructions prevent the master thread from iterating in a scalar fashion on each single counter. While the master

```
void simd_barrier_gather(int tid)
{
    int level = 0;

    // Threads copy their reduction values on the appropriate SIMD buffer
    push_reduction_values(...);

    // Current thread is in a valid level and it is master at that level
    while (level < num_levels && is_group_master(level, tid))
    {
        int my_group_idx = get_my_group_idx(level, tid);
        vtype group_arrival_state =
            get_group_arrival_state(level, my_group_idx);

        // Master waits for group slaves using SIMD memory loads
        while(vload(&tree_flags[level][my_group_idx]) != group_arrival_state)
            OpenMP_barrier_duties();

        // Master computes group reductions
        master_group_reductions(...);

        level++;
    }

    // Current thread is in a valid level and it is slave at that level
    if (level < num_levels) // i.e., !is_group_master(level, tid)
    {
        // Slaves mark arrival in group and leave the gather phase
        tree_flags[level][my_level_idx] = THREAD_ARRIVAL_STATE;
    }
}
```

Listing 6.1: Generic scheme of the SIMD barrier gather phase (pseudo-code)

thread is waiting for its slaves, it may perform some of the OpenMP barrier duties (`OpenMP_barrier_duties`) that we commented in Section 6.1.1, such as task scheduling or checking active cancellation points.

Once all threads of a particular group have reached the barrier, the master thread continues to the next level of the tree (continuous arrow in Figure 6.2). In the meantime, slave threads leave the gather phase of the barrier. In the next level, several master threads from different groups converge at the same group, and new master and slave roles are reassigned as in the previous step. This process is repeated until the last level (tree root) is reached. At that point, it is guaranteed that all threads have arrived at the barrier. It is important to note that each thread can be slave in only one level at most. In every level before that level, each thread will have a master role (except for slave threads at level 0). No thread will take part in any level after the level where they play a slave role.

The computation of reduction operations can happen during the gather phase as

```
 1 void simd_barrier_release(int tid)
 2 {
 3     int level = get_deeper_level_of(tid);
 4 
 5     // The current thread is a group slave thread
 6     if (!is_group_master(level, tid))
 7     {
 8         int my_level_idx = get_my_level_idx(level, tid);
 9 
10         // Group slaves wait for their master to release them
11         while((*tree_flags[level][my_level_idx]) == THREAD_ARRIVAL_STATE)
12             OpenMP_barrier_duties();
13 
14         level--;
15     }
16 
17     // The current thread is a group master thread
18     if (is_group_master(level, tid))
19     {
20         do{
21             int my_group_idx = get_my_group_idx(level, tid);
22             vtype group_release_state =
23                 get_init_group_state(level, my_group_idx);
24 
25             // Master releases to group slaves using a SIMD memory store
26             // that resets all the counters of the group
27             vstore(&tree_flags[level][my_group_idx], group_release_state);
28 
29             level--;
30         } while(level >= 0);
31     }
32 }
```

Listing 6.2: Generic scheme of the SIMD barrier release phase (pseudo-code)

shown in Listing 6.1 (functions `master_group_reductions` and `push_reduction_values` in lines 20 and 6). This aspect will be discussed in Section 6.3.

## 6.2.3 Release Phase

The pseudo-code in Listing 6.2 illustrates the different steps of the release phase of the barrier. In this phase, the tree structure is traversed from root to leaves until all threads have been released. Each thread starts playing its last role in the last level visited in the gather phase (line 3). This means that at the beginning of the release phase there is only one master threat in the root node. The remaining threads are slave threads spread across the different levels of the tree.

These slaves threads are all waiting on their respective counters for their respective master threads to release them (lines 11 and 12). In the waiting time, slave threads may perform some of the OpenMP barrier duties (`OpenMP_barrier_` `duties`).

The master thread that starts from the root node performs a vector store upon all the slave counters of that level (line 27). This vector store releases to all those slave threads at the same, setting all the group counters to their initial state. Function `get_init_group_state` (line 23) returns a vector value that contains the initial state of the counters of a group, given the level of the three (`level`) and the index of the master thread of that group (`my_group_idx`). Afterwards, both master and slaves move back to their respective previous groups of the previous level and retake their former master role (dashed arrow in Figure 6.2). It is important to note that once a thread plays a master role at some level, it will continue to serve as master until level 0.

As in the gather phase, these steps are applied to each level until the first level is revisited again and each master thread releases to all its slaves. It is at that point when the released slave threads and their masters are allowed to leave the barrier and the synchronization is completed.

The biggest benefit of performing only one vector store to release to all the slaves threads of the group is that master threads avoid the intensive time-consuming ping-pong effect. This effect would occur if master threads wrote each slave counter in a scalar way, as slaves, in between, were requesting the same cache line to read their counters. Thus, per each scalar store, the master thread could have to reclaim the exclusive ownership of the cache line if some slave thread had already granted with a copy that line for reading.

## 6.3 SIMD Reduction Algorithm

As in our barrier approach, we propose a SIMD reduction algorithm targeting the exploitation of SIMD instructions available in modern multi- and many-core processors. Our current proposal is limited to work only on basic data types which are the most common reduction operations in OpenMP. However, reductions on more complex data types could also be computed following the same approach.

### 6.3.1 Reduction Scheme

Our reduction algorithm follows a scheme that is integrated with the OpenMP synchronization barrier, as we motivated in 6.1.1. For the sake of simplicity, we frame our reduction description in the context of our SIMD barrier proposal described in Section 6.2.1. Nevertheless, our approach could easily be extended to different barrier algorithms.

Our reduction scheme is applicable thanks to the split gather/scatter barrier scheme. The computation of reduction operations occurs in the gather phase. In this phase, the arrival of one thread to the barrier implies that its partial reduction value is ready to be used in the reduction computation. Therefore, the final reduction computation is completed after the gather phase when the reduction global variable is updated with the resulting value. Afterwards, the release phase can safely start as all threads will have the reduction result available through the global variable.

The whole pseudo-code with both the gather phase of the SIMD barrier and the SIMD reduction algorithms has been shown in Listing 6.1 from Section 6.2.2. We take advantage of the groups of threads of the tree-like structure of our barrier proposal to perform the computation of partial reductions within the group. These partial reductions happen only when all threads in the group have reached the barrier.

In order to use efficient SIMD memory instructions in the computation of reductions, we use temporal buffers per group of threads to rearrange contiguously in memory the partial reduction of each thread. In this way, all threads copy their partial reduction value to their corresponding buffer at the beginning of the gather phase (function `push_reduction_values` at line 6). These buffers will allow using stride-one vector loads to read multiple partial reduction values at a time.

In addition, we also keep the master/slave thread roles per group in our reduction approach. The following steps summarize the detailed process of computing the reduction operations piggybacked on the SIMD barrier scheme:

1. All threads reach the gather phase of the barrier and copy their partial reduction value to their assigned position of the group reduction buffer (function `push_reduction_values`, line 6).
2. Slave threads signal the arrival in their group of threads and go to step 7 (line 29).
3. Master threads realize that all the slave threads in their group have reached the barrier and that their reduction data is ready to be used (line 16).
4. Master threads compute the local reduction of their group of threads (function `master_group_reductions`, line 20).
5. Master threads continue to the next level of the tree and they repeat steps from 2 to 5 for all the tree levels.
6. The master thread of the last group (root) computes the final reduction value and updates the global reduction variable (function `master_group_reductions`, line 20).
7. The thread leaves the gather phase of the barrier.
8. The release phase of the barrier starts.

Figure 6.3 shows the scheme of the execution of an integer reduction for 21 threads. Unlike the SIMD counters of the barrier, there is only one buffer per group in the level zero of the tree. This means that these group buffers are reused in the subsequent levels of the tree. As depicted, *buffer #0* is also reused in level 1 and level 2, and *buffer #2* is reused only in level 1.

```
void master_group_reductions(...)
{
    for each single reduction
    {
        if (level == 0) // Leaf tree nodes
        {
            // Multi-register Leaf Reduction
            for i = 1 to num_vregister_per_group
                Vop_reduce(my_red_group_buffer, my_red_group_buffer + VL*i);
        }
        // Vertical Tree Reduction (non-leaf tree nodes)
        else
        {
            for each slave in my group
               Vop_reduce(my_red_group_buffer, slave_red_group_buffer);
        }

        // Horizontal Root Reduction (root node)
        if (level == (num_levels-1))
            Hop_reduce(global_reduction_data), my_red_group_buffer);
    }
}
```

Listing 6.3: SIMD reduction steps of the master thread (pseudo-code)

### 6.3.2 SIMD Reduction Steps

As we described in Section 6.3.1, master threads incrementally compute the reduction operation in the gather phase of the barrier. These master threads perform a partial reduction of their group of threads for each level of the tree. This process is represented with the function `master_group_reductions`, at line 20 in Listing 6.1 from Section 6.2.2. Listing 6.3 shows the pseudo-code of reduction steps of the function `master_group_reductions`.

In order to perform this incremental reduction computation, we define the following two SIMD reduction operations:

**Vertical SIMD reduction operation (Vop):** It reduces two input vector registers into a single output vector register using a combiner vector operation.

**Horizontal SIMD reduction operation (Hop):** It reduces all the scalar elements within the same input vector register into a single output scalar element.

Moreover, we use these SIMD reduction operations in the following three different computation steps:

**Multi-register leaf reduction:** This step only happens at level 0 (leaf tree nodes) if the group reduction buffer has a length larger than a single vector register length (`VL`). In such a case, the master thread reduces the group reduction

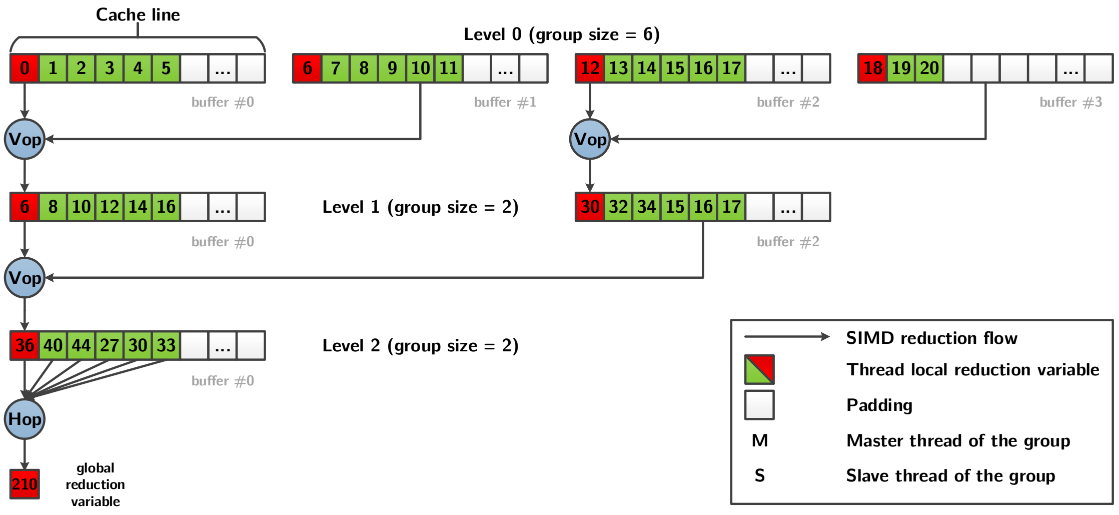


Figure 6.3: SIMD reduction scheme for 21 threads. Addition reduction defined on an integer data type. Tree group sizes of level 0, 1 and 2 are 6, 2 and 2, respectively. 4 SIMD reduction buffers.

buffer into a buffer with a maximum length of a single vector register length. Lines 8 and 9 from Listing 6.3 show the pseudo-code of this step. This reduction step is performed using vertical SIMD reduction operations.

**Vertical tree reduction:** For all levels greater than 0, the group master thread reduces all the slaves buffers into its buffer. Lines 14 and 15 from Listing 6.3 show the pseudo-code of this step. This reduction is also performed with vertical SIMD reduction operations.

**Horizontal root reduction:** At the root node, after the corresponding vertical tree reduction of that level, the group master thread reduces its single register buffer into the final scalar reduction value. This value is copied to the global reduction variable. Lines 19 and 20 from Listing 6.3 shows the pseudo-code of this step. This step is performed with a horizontal SIMD reduction operation.

Figure 6.3 shows the scheme of the SIMD reduction computation using the vertical (Vop) and horizontal (Hop) SIMD reduction operations in the vertical tree reduction and the horizontal root reduction steps. The multi-register leaf reduction step is not shown. We assume that the initial buffer at level zero is smaller than the vector register length.

## 6.4 Implementation on the Intel OpenMP RTL

In this section, we discuss the design and implementation decisions that we made to develop a prototype of our SIMD barrier and SIMD reduction algorithms in the production state Intel OpenMP Runtime Library. We analyzed the main characteristics of this library in Section 2.4. We used the public version of the runtime aligned with Intel Parallel Studio XE 2015 Composer Edition Update 2 (rev 43657) which was the latest public version at the time of starting the implementation. In this prototype, we target the Intel Xeon Phi coprocessor (Knights Corner) and its 512-bit SIMD instructions.

For the sake of simplicity, all the details provided in this section have been simplified assuming that threads can be evenly distributed across groups, i.e., all the groups in the same level have the same number of threads. Our implementation, however, is able to deal with any number of threads.

### 6.4.1 SIMD Barrier

#### Barrier Data Structures and Initialization

In the Intel OpenMP Runtime Library there are two data structures to keep track of the barrier state: a team-local data structure and a thread-local data structure. The team-local data structure, denoted `kmp_barrier_team_union`, only contains a single flag to represent the global state of the barrier at team level. This global structure is part of the state information of each team. The thread-local data structure, denoted `kmp_bstate`, contains information from the barrier state of a single thread. This information comprises the barrier counters/flags for that thread, the parent id, a pointer to the structure representing the team, local control variables, and other relevant information to perform the synchronization barrier. This local structure is part of the state information of each particular thread.

We had to modify considerably the way the barrier state is allocated in order to use stride-one vector memory instructions on the counters of the SIMD barrier. We allocate the group counters for all the groups within a particular tree level in a contiguous memory buffer. This buffer includes the corresponding padding to keep each group of counters aligned to the cache line size boundary. This approach results in as many buffers as levels the tree has.

Listing 6.4 shows a snippet of the new team-local barrier state structure with some of the most relevant fields. We renamed the original team-local data structure to `kmp_bstate_team_t` and we extended it to store a pointer to each buffer of counters mentioned above (`tree_flags`). In addition, we added other read-only information to this data structure that will be shared by all the threads of the team during the synchronization process.

The initialization of this global information of the barrier takes place during the initialization of the team. The team master thread computes and sets all the fields before creating any other thread. This information is reinitialized if the character-

```
typedef struct KMP_ALIGN_CACHE kmp_bstate_team {
  __m512i init_flags_state[MAX_LEVELS];       // Init state of the counters
  volatile kmp_uint8 *tree_flags[MAX_LEVELS]; // Group counters of each level
  kmp_int32 level_threads[MAX_LEVELS];        // # threads of each level
  kmp_int32 group_size[MAX_LEVELS];           // Group size of each level
  kmp_int32 num_levels;                       // # levels in the tree
  kmp_int32 num_participants;                 // # threads in the barrier

  KMP_ALIGN_CACHE kmp_uint8 b_arrived;        // Original team synch. flag
} kmp_bstate_team_t;
```

Listing 6.4: Snippet of the team-local barrier state structure of the SIMD barrier

```
typedef struct KMP_ALIGN_CACHE kmp_bstate {
  kmp_bstate_team_t *global_bar;          // Team-local barrier structure
  kmp_int32 level_thread_id[MAX_LEVELS]; // Id for each level
  kmp_int32 level_idx[MAX_LEVELS];        // Offset in the buffer of counters
  kmp_int32 last_level;                   // Deepest tree level of the thread
  bool level_group_master[MAX_LEVELS];    // Group master role flag per level
} kmp_bstate_t;
```

Listing 6.5: Snippet of the thread-local barrier state structure of the SIMD barrier

istics of the team change. In our current implementation, group sizes are read from environment variables and `MAX_LEVELS` is a compile time constant fixed to 5.

Regarding the thread-local data structure, Listing 6.5 depicts the main important fields that we add to this data structure. Each thread stores information about its local id per level of the tree (`level_thread_id`), its offset in each buffer of counters per level (`level_idx`), the deepest level reached in the gather phase (`last_level`) and if it plays a master role in the group of each particular level `level_group_master`). In addition, this data structure also contains a pointer to the team global state of the barrier (`global_bar`).

The initialization of the thread-local state data can take place in different points of the barrier, depending on the barrier kinds defined in the Intel OpenMP Runtime Library (Section 2.4). In a plain or reduction barrier, every thread initializes its structure at the beginning of the gather phase. In a fork-join barrier, this initialization is partially performed at the beginning of the release phase when the team of threads is still being built. This initialization is completed at the end of such a phase, once the thread has been released as a consequence of being assigned to a team.

The details included in this section show how the implementation complexity increases in an OpenMP runtime implementation that is in a production state and it has fully support for OpenMP.

### Waiting Mechanism

The Intel OpenMP Runtime Library provides the template class kmp_flag to implement a waiting mechanism on different types of flags or counters. The main waiting mechanism of the library is implemented at high-level in a function called wait. This function utilizes different auxiliary functions that have to be explicitly defined in the specializations of this template class. The waiting mechanism includes the OpenMP duties that must be performed during the barrier primitive. It also features different waiting phases such as stopping the issue of instructions of the thread, using the _mm_delay_32 intrinsic to reduce the intensity of the busy-wait, thread sleeping and thread yielding (thread context switch). We do not change at all this original waiting mechanism.

We add specializations of this class for two new flag types: SIMD counter with type __m512i, denoted kmp_flag_simd, and 1-byte counters with type unsigned char, denoted kmp_flag_8. Listing 6.6 shows a snippet of these two specializations. The checker variable in both specializations is initialized with the parameter c of the class construction. This is the value that the flag p must have to finish the wait. The notdone_check function is used by the waiting mechanism to check if the waiting condition is still not satisfied. As depicted, for the SIMD specialization, this function utilizes a single vector load _mm512_load_epi32 and a vector comparison __mm512_cmpneq_epi32_mask. The vector load (line 9) reads from memory the value of the SIMD counter p (returned by the get function) that contains all the individual counters of the treads in a group. Then, those individual counters are compared with the reference value checker using the vector comparison (line 6). This comparison returns a special 16-bit vector mask (integer) where each bit represents the comparison between every pair of 4-byte integers of the two source registers.

### Gather and Release Phases

We followed the gather and release generic schemes proposed in Section 6.2.2 and Section 6.2.3. We made use of the information contained in the team-local and thread-local data structures described in Section 6.4.1 to implement the basic thread functionality of the barrier. For instance, using the information in these structures a thread can determine if it is a master or a slave thread in a level and it can access the group counters of each level. We also utilized the two new waiting classes introduced in Section 6.4.1 in order to implement this waiting and release steps in the gather and release phases of the barrier.

Listing 6.7 shows the main code of the actual implementation of the gather phase. It is based on the generic scheme described in Section 6.2.2. In the gather base, all the thread counters of the barrier initially have the value 0xFF whereas the padding bytes are set to 0x00. All the slave threads signal their arrival writing the value 0x00 on their counters (line 23). In this way, master threads wait on the counters of their groups using the SIMD waiting class kmp_flag_simd (lines from

```
 1 class kmp_flag_simd : public kmp_flag<__m512i> {
 2   __m512i checker;
 3 
 4   kmp_flag_simd(volatile __m512i *p, __m512i c, ...) {}
 5   bool notdone_check_val(__m512i old_loc) {
 6     return _mm512_cmpneq_epi32_mask(old_loc, checker);
 7   }
 8   bool done_check() { return !notdone_check(); }  /* 'get' returns 'p' */
 9   bool notdone_check() { return notdone_check_val(_mm512_load_epi32(get()));}
10 };
11 
12 class kmp_flag_8: public kmp_flag<kmp_uint8> {
13   kmp_uint8 checker;
14 
15   kmp_flag_8(volatile kmp_uint8 *p, kmp_uint8 c, ...) {}
16 
17   bool done_check_val(kmp_uint8 old_loc) { return old_loc == checker;}
18   bool done_check() { return done_check_val(*get() /* returns 'p' */); }
19   bool notdone_check() { return !done_check();}
20 };
```

Listing 6.6: Snippet of the two new waiting flag classes for the SIMD barrier

12 to 14). We use a vector of all zeros as state to satisfy in the waiting condition, as shown in line 10. Only when all the group counters are zero, the SIMD wait will be over.

As for the release phase, Listing 6.8 shows the most relevant code of actual implementation. based on the generic scheme of Section 6.2.3. In the release phase, all the thread counters of the barrier and the padding bytes are set to `0x00`. All the slave threads are waiting on their local counters using the waiting class `kmp_flag_8` (lines from 11 to 12). They use the 1-byte value `0xFF` as state to finish the wait, as shown in line 8. To release their slave threads in a SIMD way, master threads perform a vector store with the initial state of the counters (lines 23 to 25). The vector array `init_flags_state` contains the initial state of the groups of each level, where each slave thread counter takes the value `0xFF` and each master thread counter and padding byte take the value `0x00`. In order to optimize this SIMD store, we use the intrinsic `_mm512_storenrngo_ps` (non-globally ordered stream vector stores) which performs a 64-byte store with a no-read hint. This hint avoids reading the original content of the entire 64-byte cache line as we are completely overwriting it. Moreover, it relaxes the memory consistency model in the way that these stores are not globally-ordered. As a consequence of this, the memory fence-like operation in line 8 is necessary to make this store available to slave threads as soon as possible. The use of non-globally ordered stream stores yielded better benchmarking results than traditional vector stores and globally ordered stream stores.

```
void simd_barrier_gather(int tid)
{
  ...
  // Current thread is in a valid level and it is master at that level
  while (level < team->bar.num_levels && thr->bar.level_group_master[level])
  {
    ...
    // Master waits for group slaves using SIMD memory loads
    kmp_int32 my_group_idx = thr->bar.level_idx[level];
    __m512i group_arrival_state = _mm512_setzero_epi32();

    kmp_flag_simd flag((__m512i*)&team->bar.tree_flags[level][my_group_idx],
                       group_arrival_state, ...);
    flag.wait(...);
    ...
  }

  // Current thread is in a valid level and it is slave at that level
  if (level < team->bar.num_levels) // i.e.,
  {                                 // !thr->bar.level_group_master[level]
    kmp_int32 my_level_idx = thr->bar.level_idx[level];
    // Slaves mark arrival in group and leave the gather phase
    tree_flags[level][my_level_idx] = 0x00; //THREAD_ARRIVAL_STATE
  }
}
```

Listing 6.7: Snippet of the gather phase implementation in the Intel OpenMP Runtime Library following the generic scheme from Listing 6.1

### 6.4.2 SIMD Reductions

In the implementation targeting the Intel Xeon Phi coprocessor, we use the generic scheme detailed in Section 6.3.1 without significant changes. The only hardware specific decision that we make is the use of masked instructions to perform the vector reduction operations. By means of this mask data type (`__mmask16`), we are able to apply reduction operations only on the vector lanes that contain partial reduction values.

The following sections give further implementation details on some aspects of the generic SIMD reduction scheme, such as the implementation of the temporal buffers for reductions, newly defined data structures and some changes introduced in the Intel OpenMP Runtime Library API.

#### Temporal Buffers for Reductions

As we introduced in Section 6.3.1, threads have to copy their partial reduction values to a temporal buffer. This copy is necessary to have several partial reduction values contiguous in memory and to be able to use SIMD memory instructions efficiently.

```
void simd_barrier_release(int tid)
{
  kmp_int32 level = thr->bar.last_level;

  if (!thr->bar.level_group_master[level]) // I'm a group slave
  {
    kmp_int32 my_level_idx = thr->bar.level_idx[level];
    kmp_uint8 release_state = 0xFF;

    // Group slaves wait for their master to release them
    kmp_flag_8 flag(&tree_flags[level][my_level_idx], release_state, ...)
    flag.wait(...);
    ...
  }

  if (thr->bar.level_group_master[level]) // I'm the group master thread
  {
    do{
        kmp_int32 my_level_idx = thr->bar.level_idx[level];

        // Master releases group slaves using a SIMD memory store
        // that resets all the counters of the group
        __m512i group_release_state = team->bar.init_flags_state[level];
        _mm512_storenrngo_ps((void *)&tree_flags[level][my_level_idx],
                             group_release_state);
        level--;
    } while(level >= 0);                    // Memory fence
    __asm__ volatile ("lock; addl $0, 0(%%rsp)" ::: "memory");
  }
}
```

Listing 6.8: Snippet of the release phase implementation in the Intel OpenMP Runtime Library following the generic scheme from Listing 6.2

We allocate a single reduction buffer per team of threads. We define the new runtime function __kmpc_push_estimated_reduction_info so that the compiler can provide an estimation of the size of this buffer. Its prototype is shown in Listing 6.9. With this function, the compiler can provide information in advance of the maximum number (estimated_red_vars) and sizes (estimated_red_sizes) of the reduction operations required to compute in a parallel region. When this information cannot be provided, such as in the case of reductions in orphan OpenMP constructs, this initial size is set by a default value or using an environment variable. We set this size to have room for two independent double data type reductions. If at some point the algorithm detects that a larger buffer is needed, the size of the buffer is increased in the initialization process of the barrier.

This large buffer is then split into chunks that are assigned to each individual reduction operation that has to be computed. Each chunk will contain the reduction groups of the SIMD reduction scheme described in Section 6.3.1.

```
void __kmpc_push_estimated_reduction_info(ident_t *loc,
    kmp_int32 global_tid,
    kmp_uint32 estimated_red_vars, // Number of estimated reductions
    size_t * estimated_red_sizes   // Pointer to an array with the sizes
);
```

Listing 6.9: Prototype of the function for the estimation of reductions in a parallel region

```
typedef struct kmp_simd_team_red_info
{
  kmp_int32 vregs_l0;          // # vector registers per group at level 0
  kmp_int32 elements_per_vreg; // # reduction elements per vector register
  __mmask16 reduction_mask;    // Bit mask with elements_per_vreg enabled
} kmp_simd_team_red_info_t;

typedef struct KMP_ALIGN_CACHE kmp_bstate_team {
    KMP_ALIGN_CACHE kmp_uint8 *reduction_buffer;
    kmp_simd_team_red_info_t *reduction_info_pool;
    size_t reduction_buffer_size;
} kmp_bstate_team_t;
```

Listing 6.10: On the top, snippet of the data structure that contains information of SIMD reductions at team level. On the bottom, new fields added to the team-local structure of the barrier to support SIMD reductions

### Data Structures for Reductions

In order to implement our SIMD reduction approach introduced in Section 6.3, we have to define two new data structures and add a new field to the team-local and thread-local data structures of the barrier defined in Section 6.4.1.

The first new data structure is called `kmp_simd_team_red_info_t` and it is shown on the top of Listing 6.10. This structure contains static information about a SIMD reduction of a particular data type size according to some configuration parameters of the SIMD barrier, such as the group size per level and the number of levels of the tree. Such information comprises the number vector registers per group at level 0 (`vregs_l0`), the number of reduction elements per vector register (`elements_per_vreg`) and a bit mask that represents the vector lanes that contain valid reduction elements to be reduced.

There will be an instance of the `kmp_simd_team_red_info` structure for each data type size supported by the SIMD reduction algorithm. So far, supported reductions are only on basic data types. Thus, we will have four instances of this structure for 1 byte, 2 bytes, 4 bytes and 8 bytes data type sizes. These instances are stored in a pool within the team-local data structure (`reduction_info_pool`), as depicted on the bottom of Listing 6.10. The team-local structure has been extended to contain a single buffer to host all the reduction temporal buffers (`reduction_`

```
typedef struct kmp_simd_thread_red_info
{                                                     // Pointer to the team
  kmp_simd_team_red_info_t *team_reduction_info; // reduction info
  volatile kmp_uint8 *reduction_buffer; // Pointer to a red. buffer section
  kmp_int32 red_group_idx;              // Reduction group index
  kmp_int32 red_group_offset;           // Reduction offset within the group

}  kmp_simd_thread_red_info_t;

typedef struct KMP_ALIGN_CACHE kmp_bstate {
    kmp_simd_thread_red_info_t *reduction_info;
    kmp_uint8 num_allocated_reductions;
} kmp_bstate_t;
```

Listing 6.11: On the top, snippet of the data structure that contains information of SIMD reductions at thread level. On the bottom, new fields added to the thread-local structure of the barrier to support SIMD reductions

buffer) and the size of this single buffer (reduction_buffer_size). The previous sub-subsection *Temporal Buffers for Reductions* contains more information about how these temporal buffers are implemented.

The reduction information local to each thread is stored in the kmp_simd_thread_red_info_t structure shown on the top of Listing 6.11. This structure stores information of each independent reduction operation. team_reduction_info is a pointer to the appropriate team reduction information depending on the size of the reduction data type. reduction_buffer points to the assigned chunk of the single temporal buffer allocated in the team-local structure. The field red_group_idx identifies the reduction group within the assigned reduction buffer and red_group_offset contains the offset within that particular group of the position assigned to that thread to store its partial reduction value.

The thread-local barrier structure is extended as depicted in Listing 6.11 to store as many kmp_simd_thread_red_info_t as number of reductions the thread has to compute. The field num_allocated_reductions contains the number of reductions currently allocated for that thread.

### Application Program Interface

The SIMD reduction scheme and the implementation described in this section require to change the reduction functions defined in the Intel OpenMP Runtime Library API. The main reason for this change is that the SIMD reduction scheme computes reductions independently from each other and it requires two reduction functions to perform the SIMD vertical reductions and the SIMD horizontal reduction.

The original main function of the reduction interface is described in Listing 2.4 from Section 2.4.4. The new reduction function is shown in Listing 6.12 and an

```
kmp_int32 kmpc_reduce (
    ident_t *loc,
    kmp_int32 global_tid,
    kmp_int32 num_vars,
    size_t *reduce_size, // Independent sizes of each reduction variable
    void **reduce_data,  // Pointers to each independent reduction variable
    // Vertical and horizontal functions for each reduction variable
    void (**reduce_func)(void *lhs_data, void *rhs_data, void *mask),
    kmp_critical_name *lck );
);
```

Listing 6.12: Prototype of the new main reduction function of the API

example for two `float` reduction variables is shown in Listing 6.13. This new interface contains the same number of parameters as the original one but some of them have different meaning. In the original API, the field `reduce_size` contains the aggregated size of all the reductions combined in a C structure. In the new API, this field is a pointer to an array that contains all the independent sizes of each reduction variable, as shown in Listing 6.13, line 21. In the original API, the field `reduce_data` is a pointer to a `struct` that contains the aforementioned combined reduction variables. However, in the new API, this field is a pointer to an array of pointers to each independent reduction variable, as shown in Listing 6.13, line 22. Finally, in the original API, the field `reduce_func` is a pointer to the function that reduces two structures with the combined reduction variables. In the new API, this field is a pointer to an array of pointers to functions. For each reduction, the array will contain two pointers: a pointer to the vertical SIMD reduction function and a pointer to the horizontal SIMD reduction function, as shown in Listing 6.13, lines 13 to 16. This array will have as many pairs of pointers as reductions to compute. It is important to note that the reduction functions now contain an extra parameter with the vector mask used in the SIMD operations that compute the reduction, as shown in Listing 6.13, lines 1 and 8.

## 6.5 Evaluation

We conducted several kinds of experiments to measure the performance and scalability of our SIMD barrier and reduction implementations in the Intel OpenMP Runtime Library. In a first step, we performed an extensive exploration to find the best configuration of each algorithm for each number of threads used in the evaluation. Afterwards, using this best configuration, we evaluated our SIMD approaches on several benchmarks that make intensive use of the barrier and the reduction primitives. These benchmarks range from synthetic kernels to more application-like codes. Some of them have been used in other published studies on barriers and reductions algorithms. We compare our performance results with all the barrier and reduction algorithms implemented in the Intel OpenMP Runtime Library.

```
void _red_V(float *red_out, float *red_in, __mmask16 mask)
{
  __m512 tmp0 = _mm512_mask_load_ps(_mm512_undefined(), mask, red_out);
  __m512 tmp1 = _mm512_mask_load_ps(_mm512_undefined(), mask, red_in);
  __m512 tmp2 = _mm512_mask_add_ps(_mm512_undefined(), mask, tmp0, tmp1);
  _mm512_mask_extstore_ps(red_out, mask, tmp2, _MM_DOWNCONV_PS_NONE, 0);
}
void _red_H(float *red_out, float *red_in, __mmask16 mask)
{
  __m512 tmp3 = _mm512_mask_load_ps(_mm512_undefined(), mask, red_in);
  (*red_out) = _mm512_mask_reduce_add_ps(mask, tmp3);
}
void (*reduce_func_arr[4L])(void*, void*) = {(void(*)(void*, void*))_red_V,
                                             (void(*)(void*, void*))_red_H,
                                             (void(*)(void*, void*))_red_V,
                                             (void(*)(void*, void*))_red_H};

struct  red_pack_t { float red1; float red2; };
redu_pack_t red_pack;

unsigned long int red_sizes[2L] = {4LU, 4LU};
void *red_data[2L] = {&red_pack.red1, &red_pack.red2};

...
switch (__kmpc_reduce(&_loc_0, gtid, 2, /* # reductions */, red_sizes,
                      red_data, reduce_func_arr, &lock))
{
    ...
    case 1 :
    {
        (*red1) += red_pack.red1;
        (*red2) += red_pack.red2;
        __kmpc_end_reduce(&_loc_0, gtid, &lock);
        break;
    }
    ...
}
```

Listing 6.13: SIMD reduction code of two `float` variables (`red1` and `red2`) and addition combiner operator generated for the Intel OpenMP Runtime Library targeting the Intel Xeon Phi coprocessor. Code not shown is similar to Listing 2.3

### 6.5.1 Benchmarks

We used two different sets of benchmarks: a first set that contains benchmarks with barrier primitives (no reduction primitives); a second set that contains benchmarks with barrier and reduction primitives. In this way, we can evaluate our barrier algorithm separately from the reduction algorithm using the first set of benchmarks. The reduction algorithm cannot be evaluated in an independent way since the reduction primitive in OpenMP requires a synchronization barrier.

The first set of benchmarks that we used to evaluate the performance of the barrier primitive contains the following benchmarks:

**EPCC Barrier:** The EPCC OpenMP Micro-benchmark Suite is aimed at measuring the impact of different OpenMP parallel services [20]. Within this benchmark suite, the EPCC barrier benchmark intensively uses the OpenMP barrier directive within a loop. We use this benchmark to calibrate our SIMD barrier and to measure the performance of a single synchronization barrier.

**MCBench:** The MCBench is a micro-benchmark used to measure the overhead of maintaining memory consistency in OpenMP applications [156]. In the kernel of the benchmark, threads exchange information using a shared memory buffer. Barrier primitives are used to guarantee that the data is visible to all threads after writing operations. In our experiments, we used the input parameters `--nomc --nreps 100000 --cbytes 32768 --abytes 7995392`.

**Livermore Loop:** Livermore Loops is a set of kernels that has long been used to evaluate the optimization capabilities of compilers [100]. We chose the 6th loop (kernel 6) which contains an OpenMP *for* directive with an implicit barrier on a nested loop. We modified the code in a similar way to how is done by Sampson et al. [133] in order to exploit fine-grained parallelism. We also added an outermost loop to run the kernel multiple times which requires the output array to be reset to its initial values. We reinitialize these values with a loop annotated with an OpenMP *for* directive.

**MG:** The MG benchmark belongs to the NAS Parallel Benchmarks suite [14] and it represents a full application-like algorithm that computes the approximate solution to a scalar Poisson problem using multi-grid data structures. This benchmark contains multiple nested loops annotated with an OpenMP *for* directive with an implicit barrier. The original benchmark has been modified by adding a `collapse(2)` clause to all the OpenMP *for* directives in order to reduce the grain of the parallelism and to make it suitable for running on the Intel Xeon Phi coprocessor. We use the Class Y input size proposed by Rodchenko et al. [129] to exploit fine-grained parallelism (`problem_size=16; nit=800`).

The second set of benchmarks contains codes that make use of the OpenMP reduction primitive. These benchmarks are the following:

**Reduction Loop:** This benchmark is the equivalent version of EPCC barrier for reductions. It computes reduction operations intensively in an OpenMP *for* directive. This code reduces to a single value all the elements of a shared array. This array has as many elements as threads involved in the computation. The code of this benchmark is shown in Listing A.7, Appendix A.

| Benchmark | Input Size, # Tsteps | Memory (Data Type) |
|---|---|---|
| EPCC Barrier | 300.000 | - |
| MCBench | 7.995.392, 100.000 | $\sim$ 7.6 MB (`char`) |
| Livermore Loop | 24.400, 10 | $\sim$ 2.21 GB |
| MG | Class Y | Up to $\sim$ - MB (`double`) |
| Reduction Loop | -, 300.000 | Up to $\sim$ 1KB (`float`) |
| EPCC Reduction | -, 300.000 | (`float`) |
| Cholesky | $3.904^2$, 10 | $\sim$ 58 MB (`float`) |
| CG | Class Y | Up to $\sim$ - MB (`double`) |

Table 6.1: Main characteristics of the benchmark sets

**EPCC Reduction:** The EPCC Reduction benchmark intensively computes a reduction operation in an OpenMP *parallel* directive. This means that this benchmark measures the overhead of forking and joining thread parallelism (*parallel*) as well as the overhead of the reduction operation. In contrast to the Reduction Loop benchmark, reduction operations in this benchmark do not involve reading from any shared data buffer.

**Cholesky:** The Cholesky benchmark computes the Cholesky decomposition of an input square matrix. This matrix is decomposed into the product of two matrices: a lower triangular matrix and its conjugate transpose. The serial code is shown in Figure A.3. The parallelism scheme that we described in Chapter 4 utilizes several OpenMP *for* directives with reduction primitives. Their triangular loops make it interesting for this evaluation.

**CG:** The CG benchmark is another benchmark of the NAS Parallel Benchmarks suite [14]. It solves an unstructured sparse linear system by means of conjugate gradient method. The code contains multiple loops annotated with OpenMP `for` directives, some of which also contains reduction primitives with up to 2 reduction operations per loop. As in the MG benchmark, we use the Class Y proposed by Rodchenko et al. [129] to exploit fine-grained parallelism (`na=240; nonzer=2; niter=300; shiftY=5.0`). However, as we run our experiments with up to 244 threads, we redefined `na` to 244.

Table 6.1 summarizes the input sizes that we used in our experiments, the main data type used in the computation and the memory footprint.

## 6.5.2 Environment and Methodology

We used the open source Intel OpenMP Runtime Library [75] version 20141212 (rev 43657). This version is aligned with the runtime library version included in the Intel Parallel Studio XE 2015 Composer Edition Update 2. Over this version, we implemented our SIMD barrier and SIMD reduction algorithms. In addition, we evaluated the *linear*, *tree*, *hyper* and *hierarchical* barrier algorithms included in this version of the runtime. Further details about the Intel OpenMP Runtime Library are described in Section 2.4. We compiled all the versions of the library targeting OpenMP 3.0 and with the Instrumentation and Tracing Technology (ITT) support disabled.

We run our experiments natively in an Intel Xeon Phi coprocessor, model 7120P. Further details of the architecture are described in Section 2.5. We used two different thread affinity policies that affect how threads are scheduled among cores and their respective hardware threads: the *compact* policy and the *balanced* policy [34]. In the *compact* policy, threads are assigned first to hardware threads of the same core before using a new core. In the *balanced* policy, threads are assigned first to different cores before using hardware threads of the same core. However, when multiple threads have to be assigned to the same core, the id of these threads will be contiguous (not round-robin). We used the `KMP_AFFINITY` environment variable in order to choose the corresponding thread affinity policy.

We conducted our experiments in two phases. In a first calibration phase, we run a large set of experiments aimed at finding the best group size configuration for our SIMD barrier and SIMD reduction algorithms. We used the EPCC Barrier and the Reduction Loop benchmarks for this purpose, respectively. In a second phase, we run all the benchmarks with the best configuration found in the first phase of the evaluation.

In the calibration phase, we performed a large exploration of possible configurations computing the mean of 5 executions of the EPCC Barrier and the Reduction Loop for the SIMD barrier and the SIMD reduction algorithms, respectively. Afterwards, we repeat the executions of the 20 best configurations per number of threads using the mean of 50 executions on each benchmark. These configurations are discussed in Section 6.5.3.

All the experiments were run using the following environment variables: `KMP_LIBRARY=turnaround`, `KMP_BLOCKTIME=infinite` and `KMP_TASKING=0`.

### 6.5.3 Calibration Phase

In this section, we present the results of the exhaustive exploration of all the possible configuration of group sizes per number of threads for the SIMD barrier and the SIMD reduction algorithms.

#### SIMD Barrier Calibration

Table 6.2 and Table 6.3 contain the three best group size configurations per number of threads for the compact and balanced thread affinity policy, respectively. In Table 6.2, almost all the best configurations entail the synchronization of 4 cores in the first level of the SIMD tree (16 threads due to compact policy). The best group size configuration for the second level of the tree differs a bit more depending on

| SIMD Tree Barrier | | | | | | |
|---|---|---|---|---|---|---|
| Num | Cores | Compact Binding | | | | |
| Thrds | (Thrds/Core) | Level Group Sizes | Avg | Min | Max | Sd |
| 8 | 2 (4) | 8 | 2595.64 | 2119.77 | 2977.26 | 248.77 |
| | | 2, 4 | 3041.86 | 2644.20 | 3534.69 | 257.61 |
| | | 4, 2 | 3205.41 | 2853.84 | 3620.09 | 216.70 |
| 16 | 4 (4) | 16 | 3310.63 | 2487.46 | 3952.44 | 430.97 |
| | | 12, 2 | 3430.75 | 2925.99 | 3887.42 | 254.88 |
| | | 4, 4 | 3715.00 | 3074.89 | 4445.48 | 490.38 |
| 32 | 8 (4) | 16, 2 | 4488.43 | 3794.28 | 5668.55 | 526.69 |
| | | 12, 3 | 4512.04 | 3694.64 | 5387.67 | 503.74 |
| | | 12, 2, 2 | 4578.95 | 3853.39 | 5429.29 | 406.66 |
| 61 | 16 (4*) | 16, 3, 2 | 5062.58 | 4535.80 | 5965.93 | 463.50 |
| | | 12, 4, 2 | 5155.43 | 4011.74 | 6304.06 | 697.13 |
| | | 16, 4 | 5195.90 | 3982.22 | 6485.37 | 860.84 |
| 122 | 31 (4*) | 16, 5, 2 | 6299.54 | 5527.92 | 7105.22 | 333.81 |
| | | 16, 6, 2 | 6343.62 | 5882.59 | 6895.64 | 342.05 |
| | | 12, 4, 3 | 6426.90 | 5739.06 | 7547.42 | 611.44 |
| 183 | 46 (4*) | 16, 3, 4 | 7216.68 | 6385.25 | 7926.21 | 483.40 |
| | | 12, 3, 4, 2 | 7326.39 | 6745.27 | 7710.04 | 232.39 |
| | | 16, 4, 3 | 7332.26 | 6768.12 | 8085.13 | 396.63 |
| 244 | 61 (4) | 16, 4, 4 | 7843.97 | 7427.63 | 8491.86 | 299.49 |
| | | 16, 3, 4, 2 | 7998.37 | 7697.41 | 8892.03 | 187.14 |
| | | 12, 4, 4, 2 | 8026.36 | 7165.25 | 8604.74 | 383.74 |

Table 6.2: Exploration of the SIMD barrier using EPCC barrier with *compact* thread policy. Clock cycles of the best three group size configurations of the SIMD barrier on the Intel Xeon Phi 7120 coprocessor using 1, 2, 3 and 4 threads/core. *Last core has a fewer number of threads

the number of threads. However, in general terms, the best configuration involves usually a group size of 4 cores, as well. Regarding the following levels, the best group size ranges from 2 to 6 depending on the number of threads. It is interesting to remark that for 8 and 16 threads, the best configuration of the barrier is the one that results in a totally centralized barrier, i.e., a single group size of 8 and 16 threads, respectively. This centralized approach makes sense when the number of threads and cores are not enough to congest the memory hierarchy.

In Table 6.3, the best group sizes for the first level of the SIMD tree are smaller when using a balanced affinity policy in comparison with the compact policy. This results in a deeper tree with smaller groups. The main reason is that a first intra-core synchronization step is not possible when only one thread is bound to each core (scenarios with 61 threads and lower). Thus, a small first level group size implies the synchronization of a larger number of cores than in the compact policy. As long as the number of threads increases and cores host several threads, the first level group allows exploiting a first intra-core synchronization step. However, indepen-

| SIMD Tree Barrier | | | | | | |
|---|---|---|---|---|---|---|
| Num | Cores | Balanced Binding | | | | |
| Thrds | (Thrds/Core) | Level Group Sizes | Avg | Min | Max | Sd |
| 8 | 8 (1) | 3, 3, | 3480.90 | 2691.12 | 4448.08 | 527.84 |
| | | 2, 4, | 3649.81 | 2785.65 | 4949.65 | 585.12 |
| | | 3, 2, 2 | 3650.59 | 2920.49 | 4457.16 | 422.44 |
| 16 | 16 (1) | 3, 4, 2, | 4207.74 | 3626.18 | 5095.52 | 448.38 |
| | | 4, 4, | 4411.93 | 3145.22 | 5824.64 | 822.74 |
| | | 4, 3, 2, | 4443.04 | 3291.84 | 5426.83 | 649.93 |
| 32 | 32 (1) | 4, 5, 2, | 5499.40 | 4784.76 | 6332.44 | 327.61 |
| | | 4, 3, 3, | 5618.60 | 5083.46 | 6290.76 | 318.02 |
| | | 4, 6, 2, | 5665.35 | 5105.20 | 6437.47 | 374.17 |
| 61 | 61 (1) | 3, 4, 4, 2 | 6911.32 | 6131.62 | 7488.99 | 311.65 |
| | | 4, 3, 4, 2 | 7011.18 | 6744.15 | 7313.50 | 153.61 |
| | | 4, 4, 4, | 7017.28 | 6573.07 | 7584.83 | 280.34 |
| 122 | 61 (2) | 8, 3, 4, 2 | 7175.09 | 6938.33 | 7388.28 | 90.85 |
| | | 8, 4, 4, | 7218.73 | 6795.64 | 7746.37 | 282.14 |
| | | 6, 6, 4, | 7221.19 | 6636.88 | 8028.26 | 483.75 |
| 183 | 61 (3) | 9, 4, 4, 2 | 7508.94 | 6666.81 | 8013.88 | 379.30 |
| | | 12, 4, 3, 2 | 7616.13 | 7166.18 | 8368.69 | 347.77 |
| | | 12, 3, 4, 2 | 7621.65 | 7316.50 | 7959.53 | 155.05 |

Table 6.3: Exploration of the SIMD Barrier using EPCC barrier with *balanced* thread policy. Clock cycles of the best three group size configurations of the SIMD barrier on the Intel Xeon Phi 7120 coprocessor using 1, 2, 3 and 4 threads/core. Balanced policy is equivalent to compact policy with 244 threads

dently from the number of threads per core, synchronizing around 4 cores is also the best configuration for the first level of the SIMD tree. In the best configurations of the second level, synchronizing 4 cores is very common for any number of threads, as well. On the contrary to the compact policy, centralized configurations are not the best configurations for a low number of threads. The reason is that the synchronization, even for a low number of threads, involves a larger number of cores in the balanced policy than in the compact policy.

These results confirm that performing a first individual in-core synchronization, as the Intel hierarchical barrier does, is not the best approach for this architecture. According to our experiments, faster synchronization involves the synchronization of multiple cores using the same cache line. Although sharing a cache line across multiple cores implies more memory traffic due to a potential ping-pong effect [127], the larger number of cores sharing a cache line the lower number of cache lines involved in the whole synchronization and the lower number of steps in the process. In this way, the best synchronization approach seems to be a trade-off between these two metrics: the number of threads per cache line and the total number of cache lines used.

The usage of SIMD instructions is crucial in achieving this trade-off. In the gather phase, the master thread only needs one vector load to check all the counters of the group. This reduces the execution time versus barrier approaches that have to iterate in a scalar way on a set of slave or child threads. In the release phase, releasing all the treads of a group with a single vector memory operation reduces the mentioned ping-pong effect on the involved cache line. The master thread of the group will request the cache line as exclusive to perform the release. This request will invalidate the cache line in all the cores of the slave threads that are waiting on the same cache line. After the vector store of the master thread, the cache line turns to be shared among the slave threads of the group. This results in a round trip memory transfer of the cache line. Proceeding in a scalar way would potentially entail more memory transfers because the master thread could lose the exclusivity of the cache line during the scalar writes. Other release scalar approaches normally requires several cache lines to synchronize multiple cores.

It is also important to note that groups smaller than or equal to 8 (64-bit group counters) would not require SIMD instructions as they could be implemented using a 64-bit integer. However, using SIMD instructions also benefit these smaller thread configurations because we can use streaming store instructions. These special SIMD stores also improve the performance of the barrier reducing cache memory transfers [22, 129].

The fact that four is the most appropriate number of cores to synchronize for most of the cases, regardless whether these cores are closer (fist level) or farther (second level) to each other, confirms that on-chip memory transfers between cores are independent from the distance between these cores. As stated by Ramos and Hoefler [127], memory transfers in the Intel Xeon Phi coprocessor mainly depend on the latency of accessing the distributed tag directory (DTD).

### SIMD Reduction Calibration

In the exploration of the best group size configuration for SIMD reductions, we conducted experiments with the Reduction Loop benchmark performing one and multiple reduction operations per loop with `float` and `double` data types. However, the resulting best group size configurations for all our experiments were approximately equivalent. The reason is that for one or multiple reductions, the computation scheme is the same. The only difference is that the computation of reductions happens one or multiple times according to the number of reductions to compute. The fact that the computation of reduction of `float` values is similar to `double` values can be justified as the involved cache lines do no vary significantly between both data types. For these reasons, we only include the best configurations obtained for a single `float` reduction operation per loop. Table 6.4 and Table 6.5 show the mentioned configurations.

As we can see in Table 6.4, the best configurations for the first level of the tree are significantly larger than those obtained for the barrier (see Table 6.2). A first group size of 16 threads (4 cores with compact affinity) is still present in some of the

| SIMD Tree Reduction | | | | | | |
|---|---|---|---|---|---|---|
| Num | Cores | Compact Binding | | | | |
| Thrds | (Thrds/Core) | Level Group Sizes | Avg | Min | Max | Sd |
| 8 | 2 (4) | 8, | 4694.29 | 4287.73 | 5133.49 | 210.32 |
| | | 4, 2 | 5587.85 | 5299.97 | 5824.69 | 124.00 |
| | | 2, 4 | 5916.56 | 5409.70 | 6301.93 | 241.40 |
| 16 | 4 (4) | 16, | 5680.49 | 5274.52 | 6093.27 | 227.50 |
| | | 12, 2 | 6242.90 | 5880.58 | 6982.90 | 255.22 |
| | | 8, 2 | 6728.39 | 6119.89 | 7290.19 | 296.10 |
| 32 | 8 (4) | 32, | 7367.59 | 6029.78 | 9002.66 | 866.15 |
| | | 24, 2 | 7495.35 | 6211.41 | 9220.70 | 870.78 |
| | | 16, 2 | 7812.75 | 6546.78 | 9090.34 | 831.84 |
| 61 | 16 (4*) | 12, 4, 2 | 9173.91 | 8016.48 | 10083.12 | 427.38 |
| | | 24, 3 | 9357.45 | 7733.44 | 10356.71 | 543.33 |
| | | 16, 4 | 9463.38 | 8243.45 | 10237.58 | 461.18 |
| 122 | 31 (4*) | 24, 6, | 10594.85 | 9889.36 | 11274.74 | 393.66 |
| | | 28, 3, 2 | 11058.86 | 10158.21 | 12301.82 | 543.93 |
| | | 32, 3, 2 | 11552.64 | 10153.14 | 13035.29 | 825.96 |
| 183 | 46 (4*) | 24, 6, 2 | 12591.62 | 11357.02 | 13835.63 | 555.32 |
| | | 24, 5, 2 | 13047.92 | 11689.02 | 14582.45 | 823.14 |
| | | 20, 6, 2 | 13094.36 | 11268.14 | 14742.34 | 1203.3 |
| 244 | 61 (4) | 16, 2, 7, 2 | 13016.31 | 12437.50 | 13889.34 | 320.22 |
| | | 16, 2, 6, 2 | 13073.61 | 12185.07 | 14596.92 | 531.01 |
| | | 16, 2, 3, 3 | 13082.40 | 11773.02 | 14230.36 | 625.48 |

Table 6.4: Exploration of the SIMD reductions using Reduction Loop with *compact* thread policy. Clock cycles (including SIMD barrier) of the best three group size configurations of the SIMD reduction on the Intel Xeon Phi 7120 coprocessor using 1, 2, 3 and 4 threads/core . *Last core has a fewer number of threads

configurations but many others have a group size of 24 threads (6 cores), or even up to 32 threads (9 cores). In addition, totally centralized configuration results in the best execution time for 8, 16 and also for 32 threads (2, 4 and 8 cores, respectively). It is important to note that this group size configurations are used not only for the computation of reductions but also for the synchronization barrier associated to the reduction computation.

This general increase in the first group size reduces the number of cache lines involved in the computation of the reduction operations. Even though a sub-optimal barrier configuration is being used, the reduction computation and the barrier results in the best aggregated execution time.

Despite the fact that a large first group may increase the ping-pong effect in the synchronization barrier, it is important to note that the effect on the computation of the SIMD reductions is different. In the latter case, a group of slave threads needs to write into the same cache line. This writing is a single-time operation per slave thread and there are no threads actively waiting on that cache line. In this way, the

| SIMD Tree Reduction | | | | | | |
|---|---|---|---|---|---|---|
| Num | Cores | Balanced Binding | | | | |
| Thrds | (Thrds/Core) | Level Group Sizes | Avg | Min | Max | Sd |
| 8 | 8 (1) | 5 2 | 5577.35 | 5029.89 | 6102.83 | 333.00 |
| | | 4 2 | 5755.55 | 4851.07 | 6407.26 | 394.25 |
| | | 3 3 | 5813.62 | 5112.14 | 6708.37 | 528.40 |
| 16 | 16 (1) | 3 4 2 | 7157.38 | 6208.24 | 7964.03 | 398.65 |
| | | 3 2 3 | 7193.89 | 6512.63 | 7927.64 | 367.77 |
| | | 5 4 | 7223.95 | 6349.05 | 8317.67 | 561.32 |
| 32 | 32 (1) | 5 5 2 | 8694.49 | 7437.26 | 9932.28 | 725.65 |
| | | 6 4 2 | 8813.52 | 7942.02 | 9758.17 | 478.75 |
| | | 6 3 2 | 8913.40 | 7951.56 | 10285.40 | 635.52 |
| 61 | 61 (1) | 3 4 4 2 | 10402.23 | 9623.69 | 11151.66 | 398.73 |
| | | 3 3 5 2 | 10501.92 | 9985.17 | 10996.37 | 202.52 |
| | | 3 3 7 | 10664.39 | 10183.35 | 11135.21 | 245.95 |
| 122 | 61 (2) | 6 3 7 | 11421.96 | 10769.61 | 11927.79 | 286.15 |
| | | 6 3 6 2 | 11606.13 | 10847.54 | 12417.74 | 318.35 |
| | | 6 5 3 2 | 11623.45 | 10849.83 | 12522.45 | 468.77 |
| 183 | 61 (3) | 9 3 5 2 | 12092.70 | 11315.82 | 12842.44 | 292.04 |
| | | 9 3 7 | 12193.19 | 11475.55 | 12806.87 | 289.42 |
| | | 9 5 3 2 | 12272.68 | 11563.00 | 13276.20 | 465.51 |

Table 6.5: Exploration of the SIMD reductions using Reduction Loop with *balanced* thread policy. Clock cycles (including SIMD barrier) of the best three group size configurations of the SIMD reduction on the Intel Xeon Phi 7120 coprocessor using 1, 2, 3 and 4 threads/core. Balanced policy is equivalent to compact policy with 244 threads

ping-pong effect only occurs among slave threads of the same group if these threads are in different cores. The intensity of this ping-pong effect will depend on the order of arrival of these threads to the barrier.

It is also interesting to note that large groups also cause a large standard deviation. However, the standard deviation values are not too high in comparison to the standard deviation of other barrier implementations that we will evaluate in the following section.

However, the best configurations for the balanced policy do not follow the same pattern. They are shown in Table 6.5. As we can see, these configurations are similar to, or in some cases smaller than, those obtained for the barrier (see Table 6.3). The synchronization barrier using this policy is more costly. For this reason, the best configuration for reductions is similar. Larger groups of threads would imply synchronizing a considerably larger number of cores than with the compact policy.

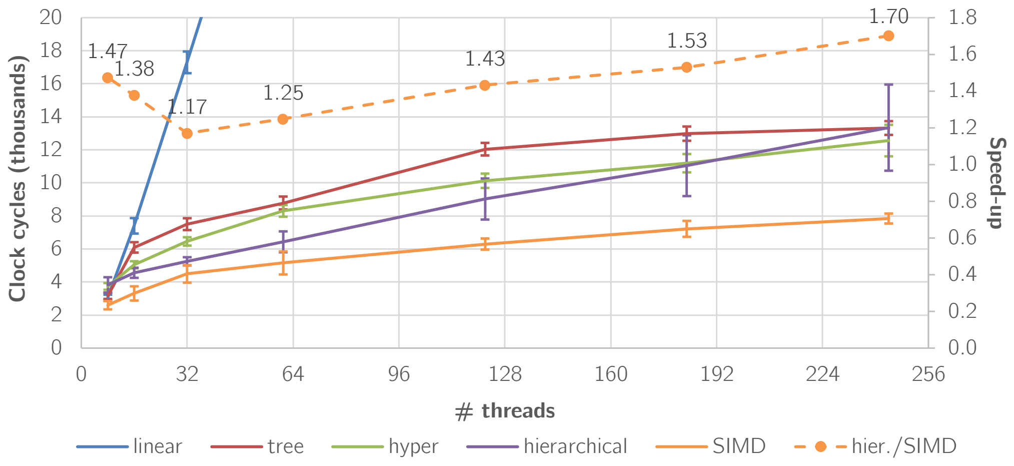


Figure 6.4: Performance of a single barrier on the EPCC barrier benchmark using compact affinity. Clock cycles with standard deviation whiskers (solid series, left axis) and speed-up (dashed series, right axis)

### 6.5.4 Performance Results

In this section, we show the performance results of our SIMD barrier and SIMD reduction algorithms and the *linear*, *tree*, *hyper* and *hierarchical* barrier and reduction approaches included by default in the Intel OpenMP Runtime Library. In our SIMD approaches, we used the best configuration found in the calibration phase for the barrier and the reduction algorithms, respectively (see Section 6.5.3).

#### EPCC Barrier

Figure 6.4 shows the performance results of the EPCC barrier benchmark for the compact thread affinity policy. Solid lines are the mean execution in clock cycles (left vertical axis) of a single barrier and the dashed line is the speed-up of the SIMD barrier over the best barrier version (hierarchical) of Intel. The linear barrier is clearly the barrier with the worst scalability. It scales linearly with the number of threads, although it offers a similar performance to other Intel barriers for 8 threads. From those Intel barriers that scale logarithmically, the tree barrier offers the worst performance. The hierarchical barrier outperforms the hyper barrier because the number of threads per core is maximum for any number of threads with compact affinity. This allows the hierarchical barrier to get the most from on-core synchronization. However, even so, the hyper barrier yields slightly better performance for 244 threads.

Our SIMD barrier greatly outperforms both the hierarchical and the hyper barrier for any number of threads. As the `hier./SIMD` speed-up series indicates, the SIMD approach scales better than Intel barriers with the number of threads, reaching a maximum speed-up of 1.66 with 244 threads. In addition, it is important to

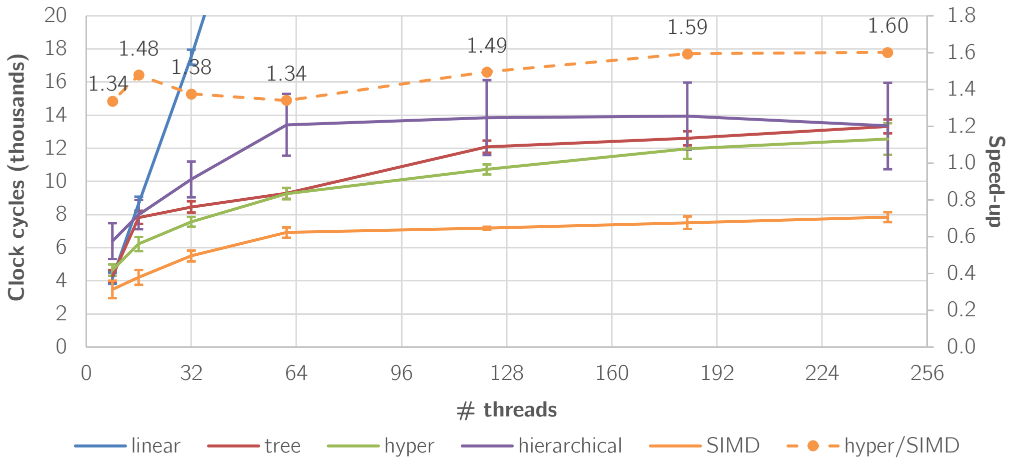


Figure 6.5: Performance of a single barrier on the EPCC barrier benchmark using balanced affinity. Clock cycles with standard deviation whiskers (solid series, left axis) and speed-up (dashed series, right axis)

note that the SIMD barrier shows a smaller standard deviation in comparison to the hierarchical barrier and the hyper barrier. Even though both the hierarchical and the SIMD barrier algorithms take advantage of the intra-core thread locality provided by the simultaneous multi-threading technology, the usage of SIMD instructions has a positive impact on the memory traffic. SIMD memory instructions allow synchronizing a larger number of threads per cache line and reducing the total number of cache lines used in the synchronization. In addition, releasing to all threads with a vector store reduces the ping-pong effect that usually leads to measurements with a higher dispersion.

Figure 6.5 depicts the results of the EPCC barrier benchmark for the balanced policy. In this case, the hyper barrier outperforms all other Intel barrier algorithms. The reason is that with a balanced policy, there are no multiple threads per core when the number of threads is small. Therefore, thread configurations smaller or equal to 61 threads do not exploit in-core synchronization in the hierarchical barrier. However, when the number of threads is larger than 61 threads, this in-core synchronization starts to be exploited. As we can see, the hierarchical barrier scales almost in a constant way, but with a high standard deviation, from 61 threads to 244 threads where we are adding more threads per core maintaining the same number of cores.

Our SIMD barrier outperforms the hyper barrier for any thread configuration, reaching a 1.60 speed-up for 183 threads. This improvement happens even when it is not possible to exploit in-core synchronization. This demonstrates that the benefit of using SIMD instructions in the synchronization process is greater than just taking advantage of the in-core thread locality. Exploiting SIMD instructions allow reducing the number of cache lines in the synchronization process by efficiently synchronizing multiple cores sharing a single cache line. In the same way as the hierarchical barrier, the scalability of the SIMD barrier is steeper when the thread configuration does not allow to exploit in-core synchronization. For thread configurations with multiple threads per core (number of threads larger than 61 one), the scalability is much more moderate. It is also important to note that the standard deviation of the SIMD barrier is notably smaller than in the hierarchical barrier and slightly smaller or similar than in the hyper barrier.

For a low number of threads, both with the compact policy (Figure 6.4) and the balanced policy (Figure 6.5), centralized configurations of the SIMD barrier greatly outperform all the Intel approaches, yielding up to approximately 47% of performance gain for both affinity policies.

### MCBench

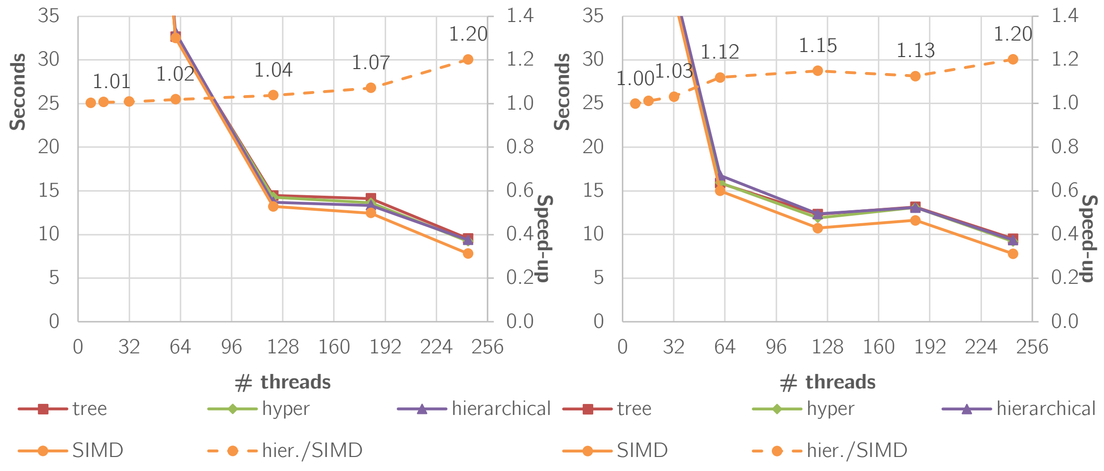
The MCBench benchmark offers an interesting evaluation scenario with high memory coherence traffic. Figure 6.6 plots the performance results of the MCBench for the compact (on the left hand side) and the balanced (on the right hand side) affinity policies. Even though this benchmark highly stresses memory hierarchy with memory coherence traffic, the weight of the barrier is enough to obtain different performance results depending on the barrier algorithm chosen.

In general terms, the hierarchical barrier offers the best performance among the Intel algorithms for both affinity policies, although difference with the hyper barrier is marginal. However, the SIMD barrier clearly outperforms any Intel barrier approach. The speed-up over the hierarchical barrier peaks at 20% for 244 threads which is where our barrier yields the best speed-up, according to the EPCC Barrier results. In addition, for a small number of threads, our approach also offers certain improvement, even though the amount of work per thread is higher for these thread configurations.

### Livermore Loop

The Livermore Loop benchmark offers an interesting evaluation scenario. The main loop nest of this code produces an incremental load imbalance: the number of iterations that do not have work to do in the innermost loop increases proportionally with the progress of the iterations of the outermost loop. Consequently, threads without work will arrive at the barrier considerably before threads with work to do.

Figure 6.7 shows the performance results of this benchmark for compact and balanced thread affinity policies. It is interesting to note how the tree barrier outperforms the hierarchical and the hyper barriers, for both affinity policies. This contradicts the results obtained in the EPCC Benchmark and other benchmarks used in our evaluation, where the tree barrier is never the best barrier approach in comparison with other Intel barriers. The SIMD barrier yields better performance than the hierarchical and the hyper barrier. This performance is close to the performance of tree barrier but the SIMD barrier is still slightly slower for both affinity



(a) Compact affinity (b) Balanced affinity

Figure 6.6: MCBench: Execution time (solid series, left axis) and speed-up (dashed series, right axis)

policies.

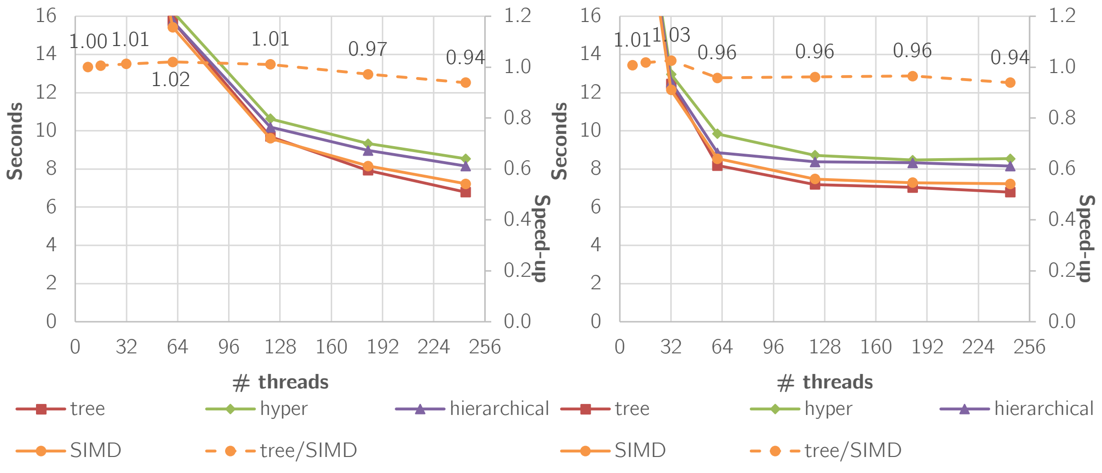
We speculate that the performance of the SIMD barrier and the tree barrier in this benchmark could be due to the load imbalance experimented in the execution. In the case of the SIMD barrier, thread 0 would be the last thread that reaches the barrier, the remaining threads could have enough time to progress until the last level of the structure of the tree. When thread 0 reached the barrier, the synchronization process would be in an advanced state and in a stable state (there should not be traffic of the barrier in the interconnection network). Therefore, the final part of the synchronization barrier should be faster. A similar situation could be happening for the Intel tree barrier that leads to the best barrier performance.

### MG

The MG benchmark has a high number of memory operations and computation other than the synchronization barrier. However, this version that exploits fine-grained parallelism is affected by the version of the barrier used.

Figure 6.8 depicts the performance results of the MG benchmark using the Class Y defined in Section 6.5.1. As we can see, the execution time increases for large number of threads. This means that a larger number of threads reduces the amount of computation per thread and it increases the impact of the barrier primitives on the overall execution time.

The best barrier from Intel is the hierarchical for the compact policy and the hyper for the balanced policy. This behavior is similar to the one observed in the EPCC Barrier benchmark. The MG version with our SIMD barrier notably outperforms the versions with the Intel barriers. It yields approximately 30% gain over



(a) Compact affinity (b) Balanced affinity

Figure 6.7: Livermore Loop: Execution time (solid series, left axis) and speed-up (dashed series, right axis)

the best Intel barrier for both policies.

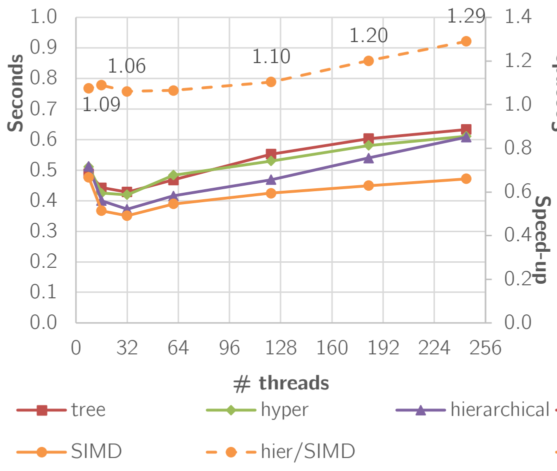
## Reduction Loop

For the Reduction Loop benchmark, apart from the different Intel algorithms, we include a version with the best Intel barrier algorithm for each affinity policy, configured to compute reduction operations using atomic instructions.

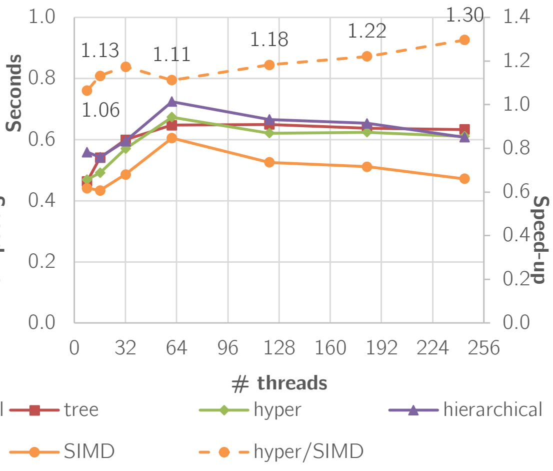
Figure 6.9 shows the performance results of the Reduction Loop benchmark using the compact affinity policy. These results include the execution of the whole benchmark, i.e., the reduction computation and the synchronization barrier. As in the EPCC barrier, the Intel linear algorithm yields the worst performance with a linear scalability. Using atomic instructions with the hierarchical algorithm also results in bad performance for a large number of threads. However, unlike the previous version, this approach scales in a logarithmic way. Regarding the tree, hyper and hierarchical versions, the first one notably offers less performance. The performance of the hyper and the hierarchical approaches are similar, although the hierarchical approach is consistently better. However, the standard deviation of the hierarchical version is considerably higher than in the hyper version which could equate the performance of both approaches for some executions.

Regarding the SIMD reduction version, it yields much better performance than the hierarchical version of the Intel compiler with a more reduced standard deviation. As in the SIMD barrier, the speed-up of the SIMD reduction algorithm increases with the number of threads over the best Intel implementation. It offers 1.56 maximum performance gain with 244 threads.

Figure 6.10 show the performance of the Reduction Loop benchmark for the



(a) Compact affinity



(b) Balanced affinity

Figure 6.8: MG: Execution time (solid series, left axis) and speed-up (dashed series, right axis)

balanced policy. For this thread affinity policy, it is interesting to note how the performance of the Intel tree, hyper and hierarchical reductions are specially similar for some configuration of threads. The tree version even outperforms the hierarchical version with 61 threads. The main reason, as in the EPCC barrier, is that the hierarchical barrier is not exploiting intra-core synchronization. In addition, this fact also favors the tree version which was not designed to take advantage of this intra-core synchronization. Unlike in the EPCC barrier, the hyper approach is clearly the best Intel version only when the number of threads is lower or equal than 122. For a larger number of threads, all versions yield similar performance.

If we observe the curves of the SIMD reduction for both thread policies, they are very similar to the curves of the SIMD barrier. This is expected since the computation of reductions are piggybacked to the barrier in OpenMP, as we described in Section 6.1.1. However, the improved performance results of the SIMD reduction algorithm over the Intel approaches reported in Figure 6.9 and Figure 6.10 could be only due to using a faster barrier algorithm. To clear this doubt, Figure 6.11 shows the performance of a single reduction without including the time of the synchronization barrier, both for compact and balanced affinity policies. As we can see, the performance of the SIMD reduction algorithm is significantly better than any other Intel approach.

In addition, we study how the three best Intel reduction algorithms and our SIMD reduction algorithm scale with the number of OpenMP reductions computed per loop. Figure 6.12 shows the clock cycles of versions from 0 (only the barrier) up to 5 `float` reductions per loop. As we stated before, the data type of the reduction operation does not impact significantly in the performance. All the Intel reduction approaches pack several reductions in the same cache line. This implementation

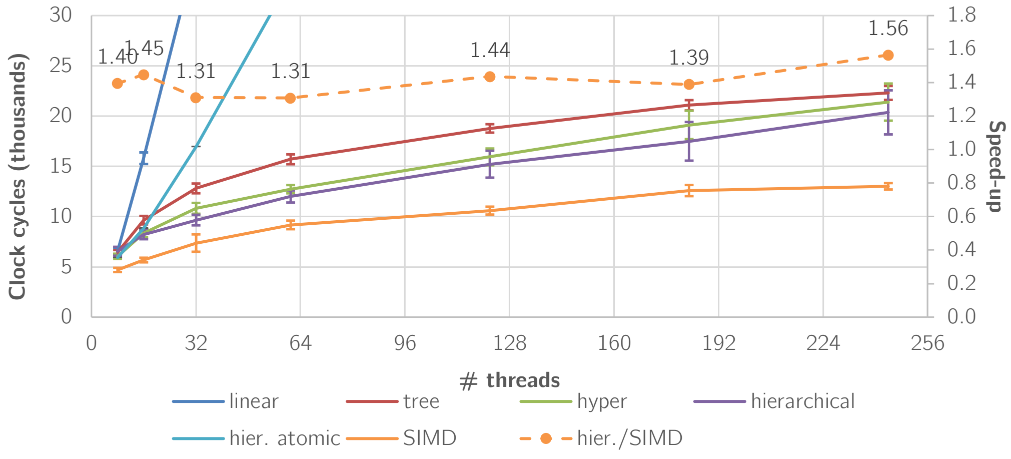


Figure 6.9: Performance of a single `float` reduction (barrier included) on the Reduction Loop benchmark using compact affinity. Clock cycles with standard deviation whiskers (solid series, left axis) and speed-up (dashed series, right axis)

design provides constant execution cycles for an increasing number of reductions if the data involved fits into the same cache line. On the contrary, our SIMD approach tries to maximize the use of the whole cache line with data of a single reduction. This provides much better performance that the previous approaches for a single reduction. The reason is that it reduces the number of cache lines used and it allows to exploit SIMD instructions in the reduction computation. However, the number of cache lines increases linearly with the number of reductions because our approach uses an independent buffer for each particular reduction. The computation of each reduction is also performed independently. As shown in Figure 6.12, despite the worse scalability of our approach, the SIMD reduction algorithm still outperforms Intel approaches up to with 3 reductions. Although it would be possible to tune our SIMD algorithm in order to share SIMD reduction buffers among different reductions to improve this issue, we think it is not worth increasing the complexity of our approach. According to our experience, the most common codes in OpenMP only contain one or two reductions per loop. A larger number of reductions is not used very often, except in synthetic or exceptional applications. For those cases, the compiler could advise the OpenMP runtime to use an implementation other than our SIMD approach.

## EPCC Reduction

The EPCC Reduction benchmark stresses the thread fork and join of parallelism of an OpenMP *parallel* directive that also computes a reduction operation in the join phase. In the Intel OpenMP Runtime Library, the fork and the join of threads that happen at the beginning and at the end of a parallel region, respectively, imply a barrier release and a barrier gather (*Fork-Join Barrier*, see Section 2.4). Therefore,

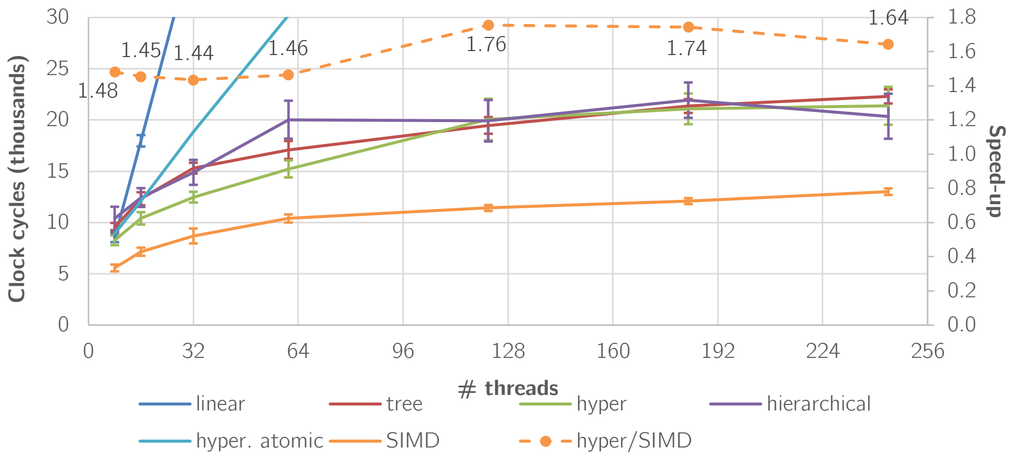


Figure 6.10: Performance of a single `float` reduction (barrier included) on the Reduction Loop benchmark using balanced affinity. Clock cycles with standard deviation whiskers (solid series, left axis) and speed-up (dashed series, right axis)

this benchmark measures not only the overhead of the reduction operation, but also the overhead of managing teams of threads in the creation, initialization and destruction of all the components involved in the parallel region.

Figure 6.13 shows the performance results of the EPCC Reduction benchmark for compact (on the left hand side) and balanced (on the right hand side) thread affinity policies. Intel barriers show a tendency similar to the Reduction Loop benchmark. The hierarchical barrier is the best approach for the compact policy whereas the hyper barrier outperforms the hierarchical barrier for most of the thread configurations of the balanced policy.

The results of our SIMD reduction scheme and barrier, however, are more difficult to analyze. In the compact policy, the performance of the SIMD approach is not the best for 61 and 122 threads. For these thread configurations, a cache miss happens for all threads in every iteration of the benchmark. This miss is located on the internal structure of the runtime that contains the information of the team of threads. Our barrier and reduction algorithms, however, do not perform any write on this structure at any point. We deeply investigated this issue without success. The particular cache line is not being written across benchmark iterations. We can guess that the miss occurs due to an unfortunate eviction of that cache line for these thread configurations. The issue does not take place for 183 and 244 threads, where our SIMD approach clearly outperforms the best Intel approach by up to 13%. In the balanced policy, our SIMD scheme outperforms Intel approaches for all the thread configurations, peaking at 31% with 183 threads.

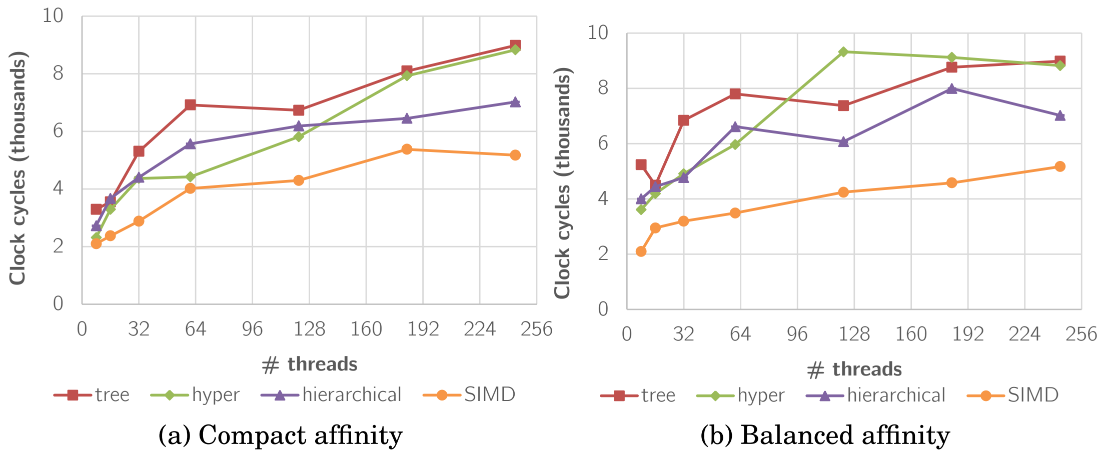


Figure 6.11: Clock cycles of a single `float` reduction without including the synchronization barrier on the Reduction Loop benchmark. Only the SIMD reduction and the best three Intel algorithms are shown

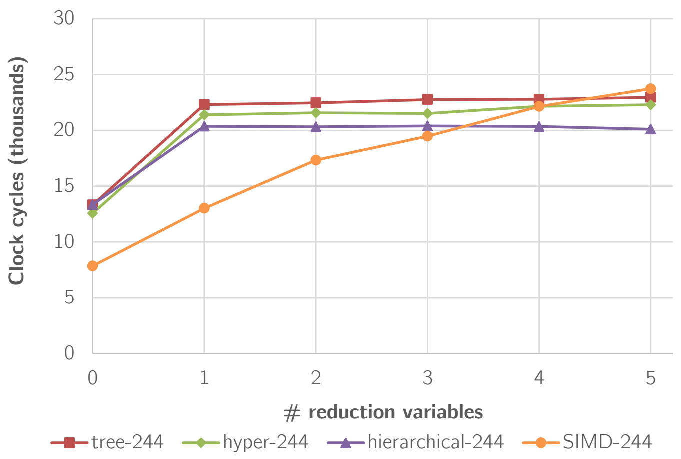
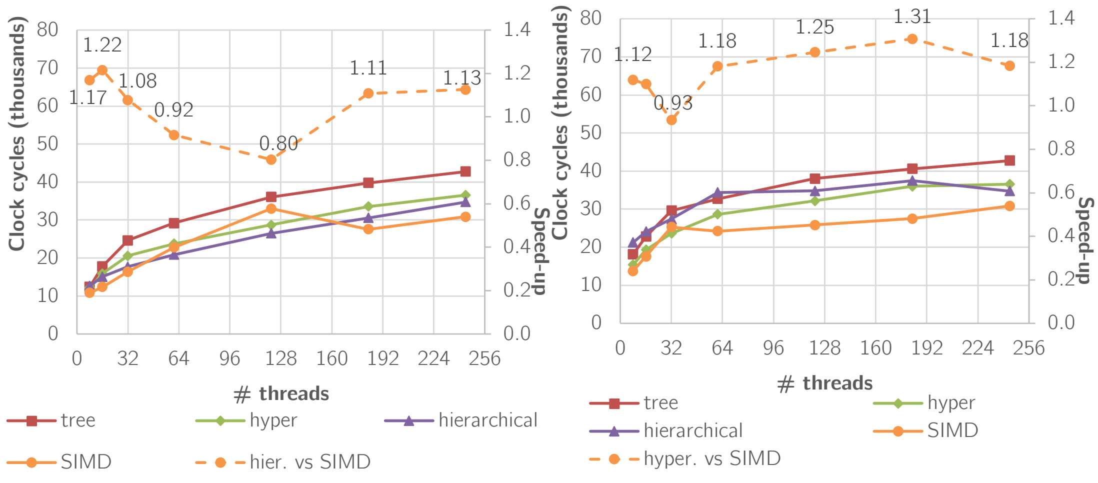


Figure 6.12: Scalability of the reduction algorithms (including barrier) with the number of `float` reductions per loop

## Cholesky

The Cholesky benchmark has a significant computational workload on a large 2D matrix. The computation involves non-contiguous memory data accesses and expensive square root operations. It contains some triangular loops that could lead us to results similar to those from the Livermore Loop benchmark. Nevertheless, as we can see in Figure 6.14, these results are different. In this case, load imbalance is less notorious than in the Livermore Loop benchmark since it only happens when the number of iterations of a triangular loop is not enough to provide work to all the threads. In addition, the imbalance does not occur in all loops at the same time,



(a) Compact affinity (b) Balanced affinity

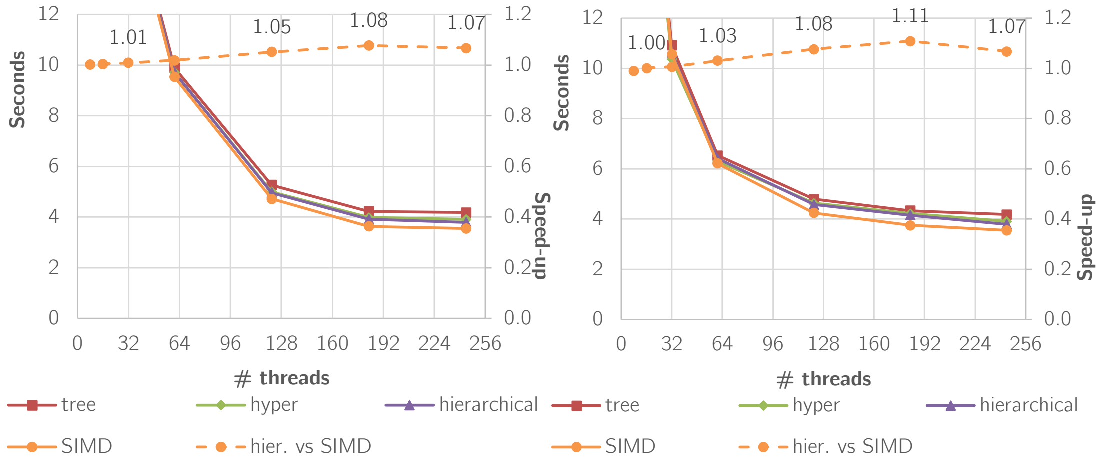
Figure 6.13: EPCC Reduction: Clock cycles (solid series, left axis) and speed-up (dashed series, right axis)

so its effect could be hidden by those triangular loops that do not experience such imbalance.

The weight of the synchronization barrier is less significant in this benchmark. The Intel hierarchical and the hyper schemes yield similar performance. They both slightly outperform the Intel tree barrier. Our SIMD barrier and reduction approach improves modestly the performance over the hyper and hierarchical approach, even though the time of these primitives are not dominant in this benchmark. This performance increases with the number of threads, as the parallelism turns finer and the computation of the reductions operations and the synchronization barrier become more important. However, the execution time scales appropriately with the number of threads. This means that the amount of computation still dominates the execution time even for 244 threads. We obtain up to 8% improvement with the compact affinity and up to 11% with the balanced affinity which is acceptable according to the characteristics of this benchmark.

## CG

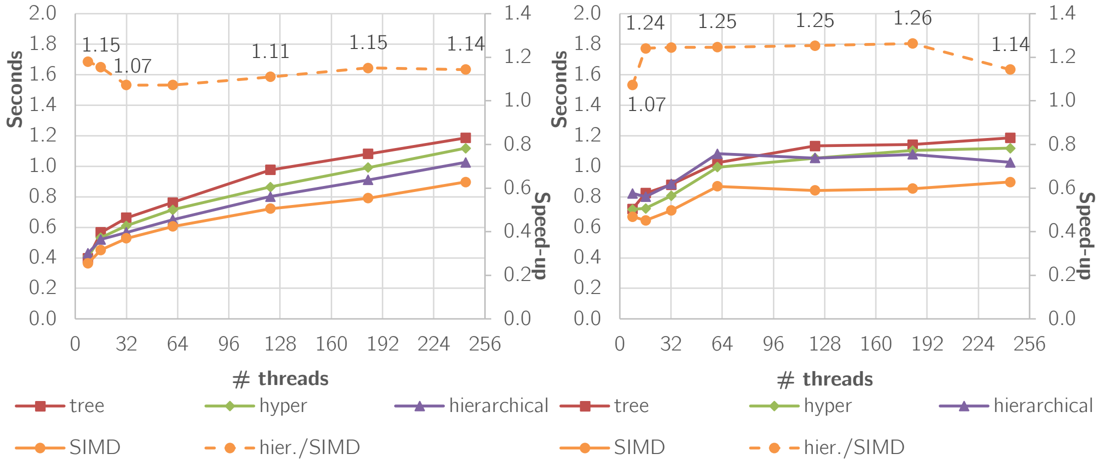
The CG benchmark performs multiple barriers and double-precision-floating-point reduction operations. We used the Class Y input, defined in Section 6.5.1, that exploits fine-grained parallelism. Figure 6.15 plots the performance results for the compact and the balanced affinity policies. As we can see, the overhead of the barrier and reduction primitives is considerable as different barrier and reduction approaches lead to performance differences. These results are similar to those obtained with the Reduction Loop which means that the barrier and reduction operations dominate the execution time.



(a) Compact affinity (b) Balanced affinity

Figure 6.14: Cholesky: Execution time (solid series, left axis) and speed-up (dashed series, right axis)

The best Intel approaches are the hierarchical for the compact policy and the hierarchical and the hyper, depending on the number of threads, for the balanced policy. The improvement of the SIMD scheme is significant, particularly for the balanced affinity. It reaches up to 36% gain over the hierarchical approach for 244 threads for this affinity policy. The speed-up for the compact policy is also relevant, leading up to 15% improvement for 122 threads over the hierarchical scheme.



(a) Compact affinity (b) Balanced affinity

Figure 6.15: CG: Execution time (solid series, left axis) and speed-up (dashed series, right axis)

## 6.6 Chapter Summary and Discussion

In this chapter, we propose a novel barrier and reduction algorithms that exploit SIMD instructions in the synchronization process and in the computation of the reduction operations. Our combined scheme of barrier and reduction is suitable for the OpenMP programming model which requires the computation of the reduction operations piggybacked to a split synchronization barrier scheme.

Our barrier algorithm takes advantage of the simultaneous multi-threading technology by grouping threads within the same core. In addition, the use of SIMD instructions in the synchronization process reduces the ping-pong effect that happens on the cache lines shared by these group of threads. This phenomenon occurs more intensively when a thread has to notify multiple threads in a sequential way that they can leave the barrier. On the contrary, the master threads of our SIMD barrier notify its group of several threads just using a single vector store. These facts explain that the best configuration found in our experiments is to first synchronize a set of four cores at the same time, instead of performing a first on-core synchronization step.

Our reduction scheme utilizes SIMD instructions to efficiently compute the reduction operations in a vertical way throughout the different levels of synchronization tree. Then, in the root of the tree, a SIMD instruction performs a horizontal reduction of the final vector register that contains the results of the vertical reduction phase. Moreover, our proposal is aimed at maximizing the use of the cache lines involved in a single reduction computation. This reduces the total number of cache lines used for a single reduction, in comparison to the reduction approaches implemented in the Intel OpenMP Runtime Library. However, the computation of multiple reductions per loop leads to increasing linearly this number of cache lines whereas this number is kept constant in the Intel approaches. Fortunately, applications normally contain one or two reductions per loop. Therefore, our approach is optimized for the most common use case.

Our implementation in the production-state Intel OpenMP Runtime Library greatly outperforms the *linear*, the *tree*, the *hyper* and the *hierarchical* barrier and reduction approaches implemented and tuned for the Intel Xeon Phi coprocessor. Particularly, our SIMD approach outperforms the *hierarchical* scheme that also conducts a first on-core synchronization step. Our improvement reaches up to 70% and 84% gain over the *hierarchical* barrier and reduction approaches, respectively. It also yields up to 35% speed-up on more realistic benchmarks that exploit fine-grained parallelism.

### 6.6.1 Dissemination Barrier versus SIMD Barrier

It is well-known that the dissemination barrier yields very good performance in bus interconnection networks, such as the one featured in the Intel Xeon Phi coprocessor [129]. The communication pattern of this barrier fits very well the network topology, keeping the number of communication steps low.

However, the dissemination barrier scheme has two important problems when it comes to its efficient integration into OpenMP, or at least into the Intel OpenMP Runtime Library.

The first problem is that this barrier does not have a natural scheme of two phases. There is no time where a single thread knows that all threads have reached the barrier. Consequently, an Intel *plain* barrier could be implemented using a single dissemination barrier, but an Intel *fork-join* barrier would require to perform two different synchronization barriers, once at the fork and once at the join of the parallel region. In addition, two dissemination barriers would also be needed if there are reduction operations to compute in a *plain* barrier or in the join phase of a *fork-join* barrier. A first dissemination barrier would be necessary to guarantee that all threads have reached the barrier and the partial reduction values of each thread is ready to be used. A second dissemination barrier would be necessary after the designated thread updated the shared reduction variable.

In the same direction, the second issue is related with the computation of reductions. In the barrier algorithms evaluated in this chapter (*linear*, *tree*, *hyper*, *hierarchical* and *SIMD*) the associated reduction computation follows the same tree internal structure of the barrier. In this way, the master thread is responsible for updating the shared reduction variable. However, the computation of a reduction operation following a dissemination barrier leads to replicating the reduction computation in all the threads. Thus, all the threads will have the final reduction value and only one will be designated to update the shared reduction variable. Then, the remaining threads should use the value of the shared reduction variable. This would turn the per-thread reduction computation useless.

From our point of view and based on the results of our experiments, our combined SIMD barrier and reduction scheme is the complete proposal that yields the fastest performance for the Intel Xeon Phi coprocessor, fulfilling the OpenMP implementation requirements, at least regarding the Intel OpenMP Runtime Library.

### 6.6.2 Alternative Tree Barrier Implementations

Split configurations for gather/release

Separate trees.

### 6.6.3 Packing Multiple SIMD Reductions per Cache Line

As we showed in Figure 6.12 in Section 6.5, our SIMD reduction approach does not scale well with the number of reduction operations computed per loop. We try to improve the use of every cache line and to reduce the overall number of cache lines involved in a single reduction computation. On the contrary, the Intel approach keeps the number of cache lines *constant* when computing multiple reductions, as values of several reductions are packed in the same cache line (until the cache line is full).

We could follow the same approach when the number of threads per group is low enough to place multiple reductions of the same group within the cache line. However, the implementation complexity could be very high. If two reductions packed together have different basic type and/or they have a different reduction operator, a much more complicated indexing and masking system would be necessary. Because the number of reductions in OpenMP applications tends to be low, we think the complexity introduced by this improvement might not be worth the benefit.

### 6.6.4 SIMD Reductions in User-defined Reductions

User-defined Reductions (UDRs) in OpenMP allows programmer to describe sophisticated reduction operations over basic data types or complex data structures. Our well-defined interface could allow programmers to benefit from SIMD reductions in UDRs if they provide the corresponding vertical and horizontal reduction functions, for example, using SIMD intrinsics or SIMD-annotated code.

However, the application of UDRs could only be feasible on basic data types. SIMD reductions on data structures with multiple basic fields, for example, would require to transform the data layout in order to place all the fields of the same basic data type and the same reduction operator contiguously in memory. Then, this layout transformation would have to be undone to place the result of the reduction in the shared reduction variable. Although an optimal solution for an arbitrary data structure would be difficult, we believe that new SIMD instructions could make feasible our SIMD reduction algorithm on some simple data structures. For example, new SIMD instructions that allow to parameterize the operations per vector lane will allow to apply our reduction algorithm on multi-field structures with the same basic data types and different reduction operators per field.

# Chapter 7

# Related Work

The related work of this thesis is structured in three sections to tackle specifically the state of the art of the main topics of our contributions. These topics are vectorization, vector codes optimizations related with the overlap optimization, and synchronization barrier and reduction algorithms.

## 7.1 Vectorization

The appearance of traditional vector processors aroused interest in automatic vectorization in the field of compilers. There were a considerable number of publications in the 70's, 80's and early 90's that were aimed at directly or indirectly improving compilers auto-vectorization capabilities.

As far as we know, the first publications that described methods to exploit vector parallelism date from the early 70's and they target the FORTRAN programming language. Yoichi Muraoka [106, 88] researched into compiling techniques for parallel and vector supercomputers, such as the ILLIAC IV [19]. Leslie Lamport [91] introduced *the hyperplane method* and *the coordinate method* to execute FORTRAN `DO` loops in a parallel fashion on massively parallel machines. Paul B. Schneck [137] described a parallelization/vectorization strategy which is based on flow information computed using interval analysis.

Later approaches [89, 5] introduced more sophisticated vectorization algorithms that placed more emphasis on data dependence analysis and strip-mining/unroll-and-jam loop transformations. This technology is the cornerstone in current production compilers [107, 16] and the base of new approaches that exploit new kinds of SIMD parallelism [92, 130, 80].

*Optimizing Compilers for Modern Architectures: A Dependence-based Approach* [81] and *High Performance Compilers for Parallel Computing* [155] are two remarkable current books that tackle vectorization in depth in the context of the FORTRAN language. They offer an extensive description of data-dependence analysis and loop transformations aimed at leveraging the vectorization of the code.

The advent of more modern programming languages such as C and C++ brought new challenges into auto-vectorizing compilers [4, 142]. The most significant issue, not present in FORTRAN array-based codes, is that vectorizing C/C++ compilers have to successfully deal with pointer aliasing in order to guarantee that vectorization is safe. In addition, the relatively recent introduction of short vector processing units in the latest CPUs and accelerators, and the increasing role that these units are playing in terms of performance [136, 65, 8], are bringing vectorization into prominence again. As a result, we find a lot of recent related work in the field that proposes valuable techniques on auto-vectorization with alignment constrains and operations with mixed data lengths [54, 110, 157, 158], vectorizing code with non-stride-one memory accesses [108, 112], and control flow divergences [26, 80].

In particular, our vectorization infrastructure is based on a strip-mining/unroll-and-jam loop vectorization approach similar to the vast majority of work presented by previous authors. We took many ideas related with vectorization from this state of the art and we adapted them to our high-level intermediate representation and to those constraints introduced by our source-to-source infrastructure.

### 7.1.1 Programming Technology for Vectorization

The automatic techniques of compilers do not always exploit SIMD resources efficiently. On the one hand, some codes are hard to be vectorized. They may require sophisticated compiler heuristics, analysis, and code transformations to provide better performance than the scalar version of the code [99]. On the other hand, in some codes there are several vectorization options that could yield different performance. Compilers find it hard to choose the most appropriate one. Codes with several loops in a loop nest that are suitable for vectorization are a good example of this scenario [111].

These limitations often force programmers to use low-level hardware-specific intrinsics or assembly code to exploit the SIMD resources of the architecture. This approach is time-consuming, cumbersome and not portable across architectures. To palliate this issue, there are several proposals that offer generic and higher level alternatives. For example, Wang et al. [153] and Estérie et al. [56] present libraries that provide an abstraction for programming different SIMD architectures in a generic way. These libraries pursue portability and programming efficiency across multiple SIMD architectures. However, they currently are in a very early stage and their applicability is limited. In addition, these approaches still require code rewriting.

New programming models and languages with intrinsic support for vector data and vector operations have emerged in the last years. Among them, the most popular are OpenCL [82] and CUDA (x86 implementation) [146] which define vector data types to describe vector operations that will be automatically vectorized by the compiler. In addition, ISCP [122] is a SPMD programming model that allows programmers to natively write applications with SIMD parallelism in mind. CHOR$\vec{u}$S

[128] extends C with the *map* and *fold* functions commonly used in Functional Programming [120] but they target them to vector operations. Although these approaches could yield good performance and more portability, sometimes not only a deeper code rewriting might be needed, but also a full redesign of the applications.

User-directed vectorization is another way of exploiting the power of SIMD instructions. There are several proposals based on compiler directives that allow programmers to propose loops that should be vectorized by the compiler [87, 86, 70, 147, 115]. In this way, programmers can guide the vectorization of their code and they can benchmark different vector versions by means of introducing simple notations in their applications. These approaches require little effort by the programmer whereas they still provide very good performance across platforms. Our work is also in this direction, but we define SIMD extensions in the context of the OpenMP programming model [83]. Our extensions also allow annotating a loop as safely vectorizable but we go a step further. We define the interaction between SIMD parallelism and fork-join parallelism in an integrated parallel approach.

Nowadays, OpenMP [117] includes SIMD extensions to express SIMD parallelism. These extensions are the result of our proposal and Intel's proposal. We collaborated in the definition of these extensions that are now part of the 4.0 version of the standard [83].

### 7.1.2 Source-to-source Vectorization Infrastructures

In this section, we describe other source-to-source vectorization infrastructures and compare them with our work done in the Mercurium compiler.

In comparison to vectorization approaches in full compilers that make use of a robust low-level intermediate representation (IR) [107, 145], vectorization at source level has different objectives and constraints. One of the most important requirements, at least in the Mercurium source-to-source compiler, is to preserve as much as possible the structure of the original code. The aim of this constraint is to provide programmers with a readable code easier to work with and debug. However, even though we took the liberty of transforming the *vector code* into three-address form (see Section 3.4 in Chapter 3), we cannot apply further code transformations or use a lower level intermediate representation due to this constraint. Consequently, this limits the analysis algorithms that we can use as many approaches are specifically designed for low-level representations in full three-address or static single assignment (SSA) [45] forms.

Despite these limitations, vectorization itself is a code transformation easily applicable following a source-to-source approach. The detailed high-level IRs used in these approaches contain the information needed in the vectorization process in a suitable way. For example, high-level IRs contain a more detailed information of loops and memory accesses. In low-level IRs, this information is lost in favor of a more generic representation and it has to be gathered by means of analysis. This fact might make source-to-source approaches the best option for vectorization.

Scout, CLVectorizer an CHOR$\vec{v}$S are practical example of source-to-source vectorization frameworks using available high-level source-to-source compilation infrastructures or even built-in infrastructures.

Scout [87, 86] is a source-to-source tool for user-directed vectorization with support for some Intel SIMD instruction sets, such as SSE, AVX and AVX2. It is implemented on the Clang front-end [144] that uses a high-level abstract syntax tree (AST) and a type system with support for vector types. The level of detail and abstraction of this IR is similar to the IR of Mercurium. However, in addition to vector types we introduce specific nodes for vector operations and vector masks. These nodes allow programming compiler phases targeting vector operations more efficiently at the expense of increasing the number of nodes of the IR. This is not considered a major problem in Mercurium.

Scout also relies on special C++ configuration files to describe the mapping between scalar operations of the IR and their hardware-specific SIMD intrinsic counterparts. In Mercurium, however, we follow an approach closer to a full compiler, where the generic vector IR is lowered to the characteristics of the target SIMD architecture using a legalization [145] and a back-end phase.

Moreover, to the best of our knowledge, Scout only support vectorization of loops. It does not support whole function vectorization [80] since function calls are inlined previously to the vectorization phase. This is an important limitation not present in Mercurium. However, Scout can benefit from code optimizations available in Clang that are not available in Mercurium, such as loop invariant code motion. Mercurium relies on the native compiler to optimize its output vector code.

CLVectorizer [77] is another source-to-source vectorization infrastructure that targets OpenCL kernels [82]. It provides two vectorization strategies: kernel inter-vectorization and kernel intra-vectorization. The inter-vectorization approach consists of vectorizing the kernel function in a similar way as the whole function vectorization implemented in our approach [80]. The intra-vectorization approach applies common loop vectorization to loops inside the kernel function. However, the generation of epilogue loops is not implemented yet.

In terms of compilation infrastructure, CLVectorizer is in an earlier stage as currently it is not implemented using any compiler framework. It uses OpenCL as IR which allows translating scalar OpenCL code to vector OpenCL code using the vector data types defined in the language. As stated before, the Mercurium vectorization infrastructure is closer to the vectorization infrastructure of a full compiler.

CHOR$\vec{v}$S [128] is another vectorization framework that brings whole function vectorization to the C language. They chose the ROSE source-to-source compiler [96] to implement the vector transformation using GCC's vector extensions [67], but extended to add the support of more sophisticated vector operations not included in these extensions. In addition, they used the Cocinelle engine [114] for pattern recognition instead of implementing this feature directly on ROSE. Using this engine they are able to detect more elaborated idioms such as dot-product and sum-of-absolute-differences. They support Intel AVX SIMD instructions [71].

The IR of ROSE is similar to the IR of Mercurium in level of detail and abstraction. However, we use our own generic vector IR instead of GCC's vector extensions, which allows us more flexibility in the representation of the vector code. This vector IR is later lowered to specific SIMD instructions or even it could be used to generate GCC's vector extensions. In addition, Mercurium does not features any sophisticated pattern recognition system. It relies on the expressiveness and the explicit analysis of the IR. Mercurium also introduces support for SSE, AVX2, IMCI and AVX-512 SIMD instructions although some of them are not in a production state.

## 7.1.3 Vectorization Techniques

In this section, we compare our vectorization technique targeting loops and whole functions with other state-of-the-art approaches. We consider some of the source-to-source vectorization infrastructures introduced in Section 7.1.2.

The Scout source-to-source vectorization tool [87, 86] follows a traditional unroll-and-jam approach for loop vectorization. This approach materializes the unrolling of the loop to later *jam* the isomorphic unrolled scalar instructions into SIMD intrinsics. This translation from scalar to SIMD is performed using a direct pattern matching approach. Vectorization of whole functions is not supported.

The vectorization algorithm of CLVectorizer [77] is also based on explicit unrolling and jamming of instructions. However, this framework also applies this technique to the vectorization of whole functions.

In contrast, Mercurium's algorithm follows an in-place vectorization approach that does not materialize the unrolling of the loop. This algorithm is also applied to the vectorization of whole functions. Explicit unroll-and-jam vectorization approaches allow the partial vectorization of codes. For example, limitations in the vectorization algorithm might not prevent the vectorization of the whole loop. However, keeping scalar instructions could require inserting vector packing and unpacking instructions in the code. Nevertheless, a more direct approach, such as the one implemented in our infrastructure, is potentially faster and more efficient as it does not require an explicit unrolling step. Unsupported vector operations could be unrolled specifically when this was necessary.

The work about vectorization in this thesis is not focused on the analysis of the code that determines the corresponding vector operation to use for each particular scalar counterpart. However, our vectorization algorithm is based on a set of attributes (see 3.5.3 in 3) that makes interesting the comparison with other proposals more focused on the analysis targeting vectorization.

Karrenberg and Hack [79, 80] described the vectorization of whole functions using SSA-based control flow graphs. Their algorithm is based on four attributes or properties: *consecutive*, *consecutive aligned*, *same value* and *same value aligned*.

The *same value* is equivalent to our *uniform* attribute and the *same value aligned* is equivalent to the combination of our *uniform* and *suitable* attributes. The *con-*

*secutive* attribute is similar to our *adjacent* attribute applied to memory accesses. When the *consecutive* attribute is applied to other expressions, it is equivalent to our *linear* attribute with step one. However, linear expressions with an arbitrary step do not have representation in their approach. This information might be useful to apply optimizations targeting interleaved memory accesses [112]. The *consecutive aligned* attribute is equivalent to the combination of our *adjacent* and the *aligned* attribute for memory accesses. When it is applied to expressions, the *consecutive aligned* could be represented with the *linear* attribute together with the *suitable* attribute applied to the lower bound of the linear expression. In general terms, both algorithms represent similar properties in a slightly different way, even though they are designed for IR with different levels of abstraction.

The main different between the algorithm described by Karrenberg and Hack approach is in the way of computing these attributes. Whereas they follow a forward data flow analysis to infer this information, our algorithm relies on information provided by programmers using our SIMD proposal (see Chapter 4). This information is complemented with a backward recursive approach that makes use of the static analysis techniques available in the Mercurium compiler. Our approach is probably less efficient because these analysis techniques have not been specifically designed for the purpose of vectorization.

Coutinho et al. [44] and Sampaio et al. [132] introduced divergence analysis in an attempt to reduce unnecessary flow and data divergences that may lead to less efficient predicated vector code or non-stride-one vector memory accesses. They follow a forward data-flow approach. Our vectorization infrastructure follows a basic and less sophisticated approach to deal with flow and data divergences. This approach is based on a backward recursive approach using reaching definition analysis [148] that allows to know if multiple definitions of a variable reach a particular region of code and study if they are equal or different. Although limited, this approach has not led to less efficient code in the scope of the evaluation of this thesis.

## 7.2 The Overlap Optimization

As we stated in Section 7.1, vectorization research in the field of compilers has been more intensive in the way of bringing vectorization to challenging codes that presented limitations or that could not be vectorized by simple vectorization strategies. However, the advent of new powerful and more sophisticated SIMD instructions in current and future processors, as register-to-register instructions, raises the question of how they might be exploited automatically by the compiler. Some of these instructions do not have a direct mapping from scalar instructions and their usage requires aggressive compiler analysis and code transformations.

In this context, inter-register vector shift instructions have been used to improve unaligned vector memory accesses and exploit data reuse in some way [141, 108, 54, 84, 112]. Nuzman et al. [112] also used the idea of caching data into vector registers. However, they applied this idea to deal with interleaved vector memory operations.

Kong et al. presented a polyhedral framework that tackles data locality optimizations, multi-core parallelism and efficient SIMD code generation [84]. They used the SSE/AVX `palignr` instruction, as we do, to reduce unaligned loads in stencil codes.

Eichenberger et al. [54] proposed a vector code transformation mainly focused on vectorizing codes with unaligned memory references in hardware without direct support for them. Although their approach exploits data reuse of the same vector load across iterations, they do not consider groups of overlapped vector loads in a generic way or a vector register cache, as we do.

The closest studies to ours are those from Naishlos et al. [108] and Shin et al. [141]. They both proposed code transformations that involve a vector register cache and vector shift instructions. However, their proposals are limited to certain loop nests where inner loop vectorization and outer loop unroll-and-jam can jointly be applied.

We present a more generic approach applicable to inner-loop and outer-loop vectorization scenarios where loop unrolling is not always necessary and loop tripcounts can be unknown at compile time. Furthermore, our OpenMP interface allows users to instruct the compiler to apply this aggressive code transformation and fine tune it. This could be a key factor in making this optimization adopted by production compilers. Moreover, we carry out the evaluation on highly-optimized kernels and workloads from real-world applications where the standalone speedup from our optimization is separately reported from that resulting from other colateral optimizations.

On the programmer side, there have been plenty of manual application-specific code optimization proposals [90, 135, 46]. Lacassagne et al. revisited some high-level transformations targeting SIMD and computer vision algorithms [90]. They considered using several vector inter-register data manipulation instructions to optimize unaligned vector loads and exploit data reuse.

Satish et al. performed a comparison of codes optimized meticulously at low-level by hand with versions optimized using compiler technology and high-level manual optimization, such as OpenMP thread parallelism, inner and outer loop vectorization, spatial and temporal blocking and data layout transformations [135]. Among their applications, they evaluated simple and small stencil codes.

Datta et al. [46] presented a set of optimizations targeting stencil computation: several spatial decomposition strategies, loop unrolling and interchange, padding and software prefetching, among others.

We use similar approaches to optimize our stencil codes. However, we found that some of the proposed solutions were too application-specific. Some of them were not applicable to our applications or they did not result in better performance when dealing with stencil codes with moderate/large order. Consequently, we had to develop and evaluate our own set of optimizations that better fitted each benchmark.

## 7.3 Synchronization Barrier and Reduction Algorithms

### 7.3.1 Synchronization Barriers

Synchronization barriers came into prominence with the advent of the first supercomputers. Initially, threads busy-waited on a shared variable for the synchronization process to be completed. The memory contention problem caused by many threads accessing to the same memory module at once is a well-known issue from that time [121]. Soon, the relevance of this issue brought expensive hardware-specific solutions, such as dedicated interconnection networks [59], that still had influence on relatively recent systems, such as the Cray T3E multiprocessor [139] or the IBM BlueGene/L system [7].

To tackle synchronization barrier primitives from a cheaper and more flexible perspective, many software barrier algorithms have been proposed. Mellor-Crummey and Scott collected several barrier approaches already available and improved the combining tree and tournament barriers spinning on separate locally-accessible flags [101]. Nanjegowda et al. evaluated these algorithms in OpenMP, concluding that the best algorithm is dependent on the number of threads used, the architecture and the application [109]. Hoefler et al. summarized nine barrier algorithms and analyzed them in the context of the InfiniBand* interconnection network [64]. Zhang et al. introduced a barrier algorithm based on distributed counters with locally-separated waiting flags, which considerably reduced overhead on IBM* POWER3 and IBM POWER4 systems [161].

Focusing on tree-based barrier algorithms similar to ours, Yew et al. presented the combining tree structure for the first time to palliate memory contention on shared variables [159]. They used a lock-based centralized counter per tree node and they determined an optimal uniform group size of four (fan-in) for each node in the tree. This is far from our heterogeneous lock-free distributed barrier approach.

Mellor-Crummey and Scott described several barrier approaches. They investigated on a new lock-free combining tree barrier with scalar flags where two tree structures (arrival and wake-up) are used [101]. Threads are statically linked to a unique node in both trees, not only to leaf nodes but also to nodes of any level. Only one thread is attached to non-leaf nodes and a group of threads are attached to leaf nodes. Unlike us, they use a uniform group size of four (fan-in) in the arrival tree and two (fan-out) in the wake-up tree. Their implementation is not reconfigurable and is bound to these fixed fan-in and fan-out parameters. On the contrary, we use a traditional combining tree structure that allows us to set a different tree degree per level. Furthermore, nodes contain reconfigurable distributed counters operated by SIMD instructions which allow checking and releasing a larger group of threads at the same time. In addition, all threads start from leaf nodes but some threads are designated to continue to the following levels. In these levels, threads are also placed in groups.

Gupta and Hill presented and adaptive combining tree with lock-based centralized counters which reshapes itself to move late threads towards the root of the tree [62]. A *fuzzy* [61] barrier version were also proposed and later improved by Scott and Mellor-Crummey [138]. Eichenberger and Abraham revisited the idea of placing slow threads close to the root [53], but their approach was based on the tree-based barrier structure published by Mellor-Crummey and Scott [101]. Eichenberger and Abraham pointed out that the optimal degree of combining trees could reach up to 128 in the presence of load imbalance.

Regarding current multi- and many-core architectures, research trends on barriers are dominated by hardware proposals. Abellán et al. developed a hardware-based barrier mechanism by means of an independent interconnection network based on global interconnection lines [1]. Sampson and González proposed a new cache-based barrier scheme that only requires additional hardware in the shared memory subsystem [133]. They also remarked the inefficiency of software barriers in the presence of fine-grained parallelism. Sartori and Kumar evaluated three hybrid hardware-software approaches that take advantage of the on-chip network with slight modifications [134]. They achieve performance comparable with dedicated synchronization networks. Villa et al. studied four hardware and software algorithms and evaluated them in a Network-on-Chip architecture [152]. It was demonstrated that, in many cases, simple networks can be more efficient than highly connected topologies.

In spite of the fact that an extensive variety of hardware proposals have been developed for chip multiprocessors (CMPs), they have not been widely adopted by industry because on-die hardware implementations would decrease the resources available for computational units. For this reason, proposals on new software barrier approaches like ours are specially relevant for improving this synchronization primitive in upcoming architectures.

### 7.3.2 Reductions

Reduction operations are common primitives in parallel algorithms. These operations consist of combining a set of distributed values using a specified combiner operation. There are several kinds of reduction operations. Simple reductions involve reducing simple scalar values with built-in combiner operations (addition, multiplication, maximum, minimum, etc.). More complex reductions can comprise reducing irregular arrays or data structures with sophisticated user-defined combiner operations [51, 104]. These operations entail computation and communication at all the levels of the parallel system, from micro architecture to distributed nodes.

Reductions have widely been studied by the community from very different perspectives and domains. At hardware level, Corbal et al. analyzed two important microarchitectural issues when computing reduction operations using SIMD instruction sets: maintaining precision of intermediate results and intra-register dependences [32]. They tackled both problems by studying the use of wider vector registers to store the intermediate reduction values. They also evaluated its proposal using a SIMD instruction set able to operates on a 2D data matrix at single instruction level. Dang et al. proposed specific non-coherent cache lines with special cache line fetch and eviction behavior. They are aimed at improving the reductions that are implemented by means of thread-private accumulation and global update. [58].

Some of the previous contributions also consider SIMD instructions in the computation of reductions. However, all of them are more oriented to solve microarchitectural problems. Our work framed in software reduction approaches.

Yu and Rauchwerger summarizes and evaluated advantages and disadvantages of the most relevant state-of-the-art parallel software reduction implementations applied to irregular reductions [160]. They classified the reduction approaches in direct update methods (critical sections [28, 55], local write or *owner computes* strategies [63], and software coherence protocols [52]), and private accumulation and global update methods (replicating private arrays [98]). They also proposed four new approaches or extensions to this methods for the computation of irregular reductions: replicated buffer with links (replicating private arrays per processor and adding links to the next processor private data), selective privatization (privatizing only data more referenced by processors), sparse reductions with privatization in hash tables (compacting the reduction references on every processor using hash tables and then applying selective privatization) and rescheduling iterations (similar to the local write approach but reordering iterations on processor to reduce contention on hot references).

Previous contributions mainly tackle irregular reductions from a generic viewpoint although some techniques are also used in approaches for regular reductions. Our work so far only address regular reductions in shared memory system and in the particular context of the OpenMP programming model.

The OpenMP programming model targets share memory systems and it defines reduction primitives. These primitives are tied to synchronization barrier primitive and it is very common to find piggybacked implementations of both of them. The Intel OpenMP Runtime Library is an example of those piggybacked implementations [75]. This library contains several reduction strategies that are used depending on the number of threads involved in the computation, the data type of the reduction operation and the combiner operator. It has support for reduction algorithms using atomic instructions, critical sections and several tree-based approaches that depend on the barrier algorithm used. In this context, Nanjegowda et al. evaluated a set of OpenMP benchmarks with barrier and reductions where they measure the performance of several piggybacked algorithms [109]: tree-based, dissemination, centralized and tournament approaches, as we mentioned in Section 7.3.1 before. In addition, we propose a novel reduction algorithm that exploit SIMD instructions.

Our work also evaluates a set of reduction algorithms. However, we used a production runtime in a newer architecture with a large number of cores and threads. We evaluated algorithms that are also optimized for on-chip multi-level hierarchy memory system. These algorithms take into account new hardware features, such as the simultaneous multi-processing (SMT) technology (hyper reduction and hierarchical reduction from the Intel OpenMP Runtime Library). In addition, we propose a novel reduction algorithm that exploits SIMD instructions.

On large system with non-uniform memory access (NUMA) or distributed memory, reduction operations turn critical because the communication among memory modules or nodes are particularly expensive. A lot of work have been done to improve reduction algorithms used in programming models for distributed systems, such as MPI. For instance, collective operations that involved reduction operations evolved from binary tree approaches to butterfly approaches in the MPICH library [143]. Butterfly communication seem to be a widely adopted communication pattern for collective communications on large system. Research here tends towards improving algorithms based on these communication patterns for more problematic cases, such as reductions with irregular communication patterns [126] or reductions with non-commutative operators [150].

Venkata et al. [151] proposed a hierarchical reduction design in the context of the MPI programming model that takes into account the different hardware *hierarchies* and communication networks of the system.

Whereas some of the previous approaches targeting distributed systems mainly focused on the communication network, our reduction approach is aimed at exploiting the on-chip computational resources. Our reduction approach also takes advantage of the memory hierarchy but only from the on-chip point of view. In addition, we use SIMD instructions in all the steps of the parallel reduction computation. This is something new from the best of our knowledge.

# Chapter 8

# Conclusions and Future Work

## 8.1 Conclusions of the Thesis

In this thesis, we present four contributions that tackle the underuse of SIMD instructions within the scope of the OpenMP programming model from three different perspectives: compiler algorithms, language constructions and runtime algorithms. These contributions are the following:

- A source-to-source vectorization infrastructure.
- A set of SIMD directives and clauses for extending OpenMP 3.1 with SIMD capabilities.
- A user-directed compiler vector code optimization that improves the memory efficiency of vector loads that overlap in memory.
- A combined barrier and reduction scheme specifically designed to exploit SIMD instructions.

Our first contribution introduces the basis of a source-to-source vectorization infrastructure as a programming tool to help out programmers in the exploitation of SIMD instructions. We demonstrate its usefulness with a use case where programmers vectorize an application with our tool, avoiding the hand-made vectorization. They further improve the performance by implementing simple optimizations over the vector code generated by the compiler. In addition, this vectorization infrastructure provides the basis for further research in the field of user-directed vectorization.

Our SIMD proposal to extend OpenMP 3.1 is aimed at improving the efficiency of programmers when dealing with vectorization. Our experiments demonstrate that these SIMD extensions allow the compiler to overcome the common limitations that usually prevent vectorization in fully automatic approaches. Therefore, by means of these SIMD annotations, programmers can take full advantage of the advanced vectorization technology of compilers without too much programming effort. For instance, targeting the Intel Xeon Phi coprocessor, we obtain up to 4.40 speed-up in the *Distq* benchmark versus the auto-vectorized version, and up to 12.90 speed-up in the Mandelbrot benchmark where the compiler is to able to auto-vectorized the code.

Regarding our compiler vector code optimization, our performance results show that this optimization improve overlapped vector memory loads, yielding up to 29% improvement over highly-optimized benchmarks in the Intel Xeon Phi coprocessor. In addition, our new clause *overlap* proposed in the context of our SIMD extensions allow programmers to fine tune this optimization difficult to apply automatically by the compiler.

On the runtime side, we demonstrate that redesigning algorithms with the aim of enabling the exploitation of SIMD instructions leads to relevant performance gains upon this software components. Our combined barrier and reduction scheme exploit SIMD instructions in the synchronization barrier process and in the computation of the reduction operations. Our implementation in the production-state Intel OpenMP Runtime Library reports up to 70% and 85% gain over the best barrier and reduction algorithms included in this library for the Intel Xeon Phi coprocessor, respectively. It also yields up to 35% speed-up on more realistic benchmarks that exploit fine-grained parallelism.

The fourth contributions presented in this thesis are a great step forward towards the exploitation of SIMD instructions in a more generic and efficient way. They also establish the basis to develop further research work in the field of programming models, compilers and runtime libraries.

## 8.2 Publications

The work conducted in this thesis resulted in four main publications plus another two pending publications. The first one is titled *Extending OpenMP with Vector Constructs for Modern Multicore SIMD Architectures* [83]. It was published in *the 8th International Workshop on OpenMP*, in 2012. This paper was written in collaboration with Intel and it describes the proposal to extend OpenMP with SIMD extensions.

The second article is called *An OpenMP Barrier Using SIMD Instructions for Intel® Xeon Phi™ Coprocessor* [22]. It was published in *the 9th International Workshop on OpenMP*, in 2013. In this paper we performed a preliminary evaluation of our barrier algorithm, exclusively. It does not contain our proposal on SIMD reductions.

The third article is *Optimizing Overlapped Memory Accesses in User-directed Vectorization* [24]. It was published in *the 29th ACM on International Conference on Supercomputing*, 2015. This paper introduces our vector code optimization for overlapped memory accesses.

The fourth article is called *Optimizing Fully Anisotropic Elastic Propagation on Intel Xeon Phi Coprocessors* [23]. It was published in *the 2nd EAGE Workshop on High Performance Computing for Upstream*. This paper is a use case of the Mercurium vectorizer infrastructure that we have developed for this thesis. We apply our vectorization proposal to a real scientific application that implements an elastic propagator.

Our fifth publication is still pending. We submitted an article with our SIMD reduction algorithm implemented in the Intel OpenMP Runtime Library to IPDPS 2016. We also plan to publish an article in a journal with the SIMD barrier and SIMD reduction algorithms and an extensive comparison with the dissemination barrier targeting the Intel Xeon Phi coprocessor.

In addition to the publications directly related with this thesis, I was involved in other two publications with my research group. These publications are related with the field of compilers.

The first publication is named *Mercurium: Design Decisions for a S2S Compiler* [57]. It was published in the *Cetus Users and Compiler Infrastructure Workshop* at PACT 2011. In this paper, we describe the Mercurium compiler infrastructure and implementation decision that we made throughout the years of its development.

The second paper is called *Compiler Analysis for OpenMP Tasks Correctness* [131]. It was published in *the 12th ACM International Conference on Computing Frontiers*, 2015. This contribution describes a set of compiler analysis techniques that detect potential inconsistencies or improvements on the OpenMP tasking model annotations introduced by programmers.

## 8.3 Impact

From our point of view, this thesis has had an outstanding impact on the field of programming models and compilers. OpenMP is used by thousands of programmers. Millions of people run applications parallelized with this programming model. Our proposal on extending OpenMP 3.1. with SIMD extensions influenced in the official extension finally included in the 4.0 version of the standard. We presented our proposal to the OpenMP Architecture Review Board (ARB) Committee and participated in its refinement.

Moreover, the Overlap vector code optimization arose the interest of some members of the Intel C/C++ compiler team. They gave us the chance of presenting them our work and we received and very valuable and positive feedback. This proposal was also introduced to the OpenMP ARB Committee and it could be considered for future versions of the standard.

Regarding our SIMD barrier and reduction algorithms, to the best of our knowledge, they are the fastest barrier and reduction combined implementations for the Intel Xeon Phi coprocessor (Knights Corner) that has demonstrate that can be applied in a production runtime for OpenMP. We aim to improve our implementation to be able to release it in the official Intel OpenMP Runtime Library, affecting in that way to its thousands of users.

Finally, the major software productization of this thesis is the vectorization infrastructure developed in the Mercurium compiler. This infrastructure can be used as a tool to efficiently vectorize codes. Its source-to-source nature allows programmers to improve the output vector code without vectorizing the code by hand. In this way, the Computer Applications in Sciences and Engineering (CASE) department at Barcelona Supercomputing Center is currently starting to use our vectorizer to optimize a production application in collaboration with Repsol. Moreover, our vectorizer has been considered as candidate vectorization infrastructure to do further work in the RoMoL [43] and Montblanc 3 [29] European projects. In the RoMoL project, the Mercurium source-to-source vectorization infrastructure could be used to carry out further work on vectorization targeting architectures with long vector instruction sets. In the Montblanc 3 project, our vectorization infrastructure could be used to generate vector code for ARM architectures, targeting the Neon SIMD instruction set.

## 8.4 Future Work

Our vectorizer infrastructure lays the basis for further research on vectorization techniques. This research can include new compiler vectorization algorithms and approaches that take advantage of new vector instructions to bring vectorization to more kinds of applications. In addition, our vectorization infrastructure allows keeping working on more extensions to exploit SIMD instructions from a programming model point of view. We also plan to extend our infrastructure to support research on new long vector architectures for the RoMoL project [43].

In the context of the SIMD extensions for OpenMP there is still room for improvement. Further extensions in the language of the programming model could be in the direction of user-directed optimizations that allow the compiler to use more sophisticated approaches. For example, a promising direction could be to guide the compiler in the exploitation of advanced vector permutation instructions.

Regarding the overlap optimization, we plan to evaluate our approach on past and future Intel SIMD extensions. In addition, it would be interesting to know how well the compiler is able to apply this optimization automatically. A sophisticated cost model to determine the worthiness of this optimization would be then necessary. In addition, this optimization could also be apply on non-stride-one vector loads with some kind of overlap. However, we have not yet found real cases where these memory access patterns occur naturally.

In the context of runtime services, it would be interesting to examine the applicability of SIMD instructions to other runtime components, such as the OpenMP dependence system, or other costly runtime algorithm. Regarding barrier and reductions algorithms, we would like to evaluate our SIMD proposal in other multicore and many-core architectures. For example, it could be interesting to evaluate this approach on architectures with narrower vector registers, such as SSE and AVX, or the next generation of the Intel Xeon Phi (Knights Landing) that features a different network topology.

The exploitation of SIMD instructions in the synchronization process could also be extended to other barrier algorithms. In addition, the applicability of the SIMD reduction algorithm to user-defined reductions could also be a relevant next step.

# Appendix A

# Benchmarks Source Code

This appendix contains the source code of some benchmarks used in this thesis.

```c
void kirchhoff_migration(float* traces /*Input traces*/,
    int width /*X-width*/, int height /*Z or time-height*/,
    float velocity /*Constant velocity*/, float dx /*Delta x*/,
    float dt /*Delta t*/, float dz /*Delta z*/, float* model /*Earth model*/)
{
    float r = 1.0f / (velocity * dt); // Calculate inverse velocity

    // For each point in the model space (x-z)
    for (int iz = 0; iz < height; ++iz) {
        float z = iz * dz;
        for (int ix = 0; ix < width; ++ix) {
            float x = ix * dx;
            float tsum = 0.0f;

    // Summation along hyperbolic travel-paths through the input traces space
            for (int icx = 0; icx < width; ++icx) {
                float cx = icx * dx;

                // Calculate corresponding travel-time
                int it = sqrtf((x - cx) * (x - cx) + z * z) * r + 0.5f;

                if (it < height) {                       // Gather and add
                    tsum += traces[it * width + icx];  // trace value to
                }                                        // the model
            }
            model[iz * width + ix] = tsum;
        }
    }
}
```

Listing A.1: Kirchhoff benchmark kernel

```
float min(float a, float b)
{
    return a < b ? a : b;
}

float distsq(float x, float y)
{
    float tmp = x - y;
    return tmp * tmp;
}

void compute (float *a, float *b, float *c, float *d, int N)
{
    for (int i=0; i<N; i++)
    {
        d[i] = min(distsq(a[i], b[i]), c[i]);
    }
}
```

Listing A.2: Distsq benchmark kernel

```
for (int j = 0; j < N; j++)             //#1
{
    for (int i1 = j + 1; i1 < N; i1++)  //#2
    {
        for (int k = 0; k < j; k++)     //#3
        {
            a[i1][j] -=  a[i1][k] * a[j][k];
        }
    }

    a_jj = a[j][j];         // a_jj is shared

    for (int i2 = 0; i2 < j; i2++)      //#4
    {
        a_jj -= a[j][i2] * a[j][i2];
    }

    a[j][j] = sqrtf(a_jj);

    for (int i3 = j + 1; i3 < N; i3++)  //#5
    {
        a[i3][j] = a[i3][j] / a[j][j];
    }
}
```

Listing A.3: Cholesky benchmark kernel

```c
void update_particle_soa(float* px, float* py, float* pz,
                         float* pvx, float* pvy, float* pvz,
                         float* fx, float* fy, float* fz,
                         float* pm, int i)
{
    const float t_div_m = time_interval / pm[i];
    const float half_time_interval = 0.5f * time_interval;

    const float vx_change = fx[i] * t_div_m;
    const float vy_change = fy[i] * t_div_m;
    const float vz_change = fz[i] * t_div_m;

    const float x_change = pvx[i] + vx_change * half_time_interval;
    const float y_change = pvy[i] + vy_change * half_time_interval;
    const float z_change = pvz[i] + vz_change * half_time_interval;

    pvx[i] += vx_change;
    pvy[i] += vy_change;
    pvz[i] += vz_change;

    px[i] += x_change;
    py[i] += y_change;
    pz[i] += z_change;
}

void update_particle_block(soa_particle_t * particles, soa_force_t * forces)
{
    float* px = particles->x;
    float* py = particles->y;
    float* pz = particles->z;
    float* vx = particles->vx;
    float* vy = particles->vy;
    float* vz = particles->vz;
    float* m = particles->m;
    float* fx = forces->x;
    float* fy = forces->y;
    float* fz = forces->z;

    for (int i = 0; i < bs; i++)
        update_particle_soa(px, py, pz,
                            vx, vy, vz,
                            fx, fy, fz,
                            m, i);
}
```

Listing A.4: Nbody benchmark: functions to compute the position of an AOSOA block of particles

```
void calculate_force_soa(float* p1x, float* p1y, float* p1z, float* p1m,
                         float* p2x, float* p2y, float* p2z, float* p2m,
                         float* fx, float* fy, float* fz,
                         int i, int j, int k, int l)
{
    const float diff_x = p2x[l] - p1y[k];
    const float diff_y = p2y[l] - p1y[k];
    const float diff_z = p2z[l] - p1z[k];
    const float dist_sqr = diff_x * diff_x + diff_y * diff_y +
                           diff_z * diff_z;

    const float inv_dist = 1.0f / sqrtf(dist_sqr);

    float f = (((p1m[k] * inv_dist) / dist_sqr) * (p2m[k] * G));

    if (j == i) f = 0.0f;

    *fx += f * diff_x;
    *fy += f * diff_y;
    *fz += f * diff_z;
}

void calculate_force_block(soa_particle_t * particle1,
                           soa_particle_t *  particle2,
                           soa_force_t *  forces, const int i, const int j)
{
    float*  p1x = particle1->x;
    float*  p1y = particle1->y;
    float*  p1z = particle1->z;
    float*  p1m = particle1->m;
    float*  p2x = particle2->x;
    float*  p2y = particle2->y;
    float*  p2z = particle2->z;
    float*  p2m = particle2->m;
    float*  fx = forces->x;
    float*  fy = forces->y;
    float*  fz = forces->z;

    for (int k = 0; k < bs; k++)
    {
        float x = fx[k];
        float y = fy[k];
        float z = fz[k];

        for (l = 0; l < bs; l++)
            calculate_force_soa(p1x, p1y, p1z, p1m,
                                p2x, p2y, p2z, p2m,
                                &x, &y, &z, i + k, j + l, k, l);

        fx[k] = x;
        fy[k] = y;
        fz[k] = z;
    }
}
```

Listing A.5: Nbody benchmark: functions to compute forces of an AOSOA block of particles

```c
void fwtCPU(float *h_Output, float *h_Input, int log2N)
{
    const int N = 1 << log2N;
    int pos, stride, base, j;

    for(pos = 0; pos < N; pos++)
        h_Output[pos] = h_Input[pos];

    //Cycle through stages with different butterfly strides
    for(stride = N / 2; stride >= 1; stride >>= 1){
        //Cycle through subvectors of (2 * stride) elements
        for(base = 0; base < N; base += 2 * stride)
        {
            //Butterfly index within subvector of (2 * stride) size
            for(j = 0; j < stride; j++){
                float T1 = h_Output[base + j + 0];
                float T2 = h_Output[base + j + stride];
                h_Output[base + j + 0] = T1 + T2;
                h_Output[base + j + stride] = T1 - T2;
            }
        }
    }
}
```

Listing A.6: Fastwalsh benchmark main kernel

```c
red = 0.0f;
#pragma omp parallel firstprivate(num_barrs, data, data_elements) shared(red)
{
    for(int i=0; i < num_barrs; i++)
    {
#pragma omp for reduction(+:red)
        for (int j = 0; j < data_elements; j++)
        {
            red += data[j];
        }
    }
}
```

Listing A.7: Code snippet of the Reduction Loop benchmark

```
int mandel(float c_re, float c_im, int count)
{
    float z_re = c_re, z_im = c_im;

    for (int i = 0; i < count; ++i)
    {
        float z_re2 = z_re*z_re;
        float z_im2 = z_im*z_im;

        if (z_re2 + z_im2 > 4.0f)
            break;

        float new_re = z_re2 - z_im2;
        float new_im = 2.0f * z_re * z_im;
        z_re = c_re + new_re;
        z_im = c_im + new_im;
    }

    return i;
}

void mandelbrot(float x0,  float y0, float x1,  float y1,
                     int width,  int height, int maxIterations,
                     int output[])
{
    float dx = (x1 - x0) / width;
    float dy = (y1 - y0) / height;

    for (int j = 0; j < height; j++)
    {
        float y = y0 + j * dy;

        for(int i = 0; i < width; i++)
        {
            float x = x0 + i * dx;
            output[j * width + i] = mandel(x, y, maxIterations);
        }
    }
}
```

Listing A.8: Mandelbrot benchmark main kernel

# Bibliography

[1] José L. Abellán, Juan Fernández, and Manuel E. Acacio. Efficient and scalable barrier synchronization for many-core cmps. In *Proceedings of the 7th ACM International Conference on Computing Frontiers*, CF '10, pages 73–74, New York, NY, USA, 2010. ACM.

[2] Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools (2nd Ed.)*. Addison-Wesley Longman Publishing Co., Inc., 2006.

[3] Frances E. Allen and John Cocke. A catalogue of optimizing transformations. In *Design and Optimization of Compilers*, 1972.

[4] Randy Allen and Stephen C. Johnson. Compiling c for vectorization, parallelization, and inline expansion. In *Proceedings of the ACM SIGPLAN 1988 Conference on Programming Language Design and Implementation*, PLDI '88, pages 241–249, New York, NY, USA, 1988. ACM.

[5] Randy Allen and Ken Kennedy. Automatic Translation of FORTRAN Programs to Vector Form. *ACM Trans. Program. Lang. Syst.*, 9(4):491–542, October 1987.

[6] Randy Allen and Ken Kennedy. Automatic loop interchange. *SIGPLAN Not.*, 39(4):75–90, April 2004.

[7] George Almási, Philip Heidelberger, Charles J. Archer, Xavier Martorell, C. Chris Erway, José E. Moreira, B. Steinmacher-Burow, and Yili Zheng. Optimization of mpi collective communication on bluegene/l systems. In *Proceedings of the 19th Annual International Conference on Supercomputing*, ICS '05, pages 253–262, New York, NY, USA, 2005. ACM.

[8] Dieter an Mey, Dirk Schmidl, Tim Cramer, and Michael Klemm. OpenMP Programming on Intel Xeon Phi Coprocessors: An Early Performance Comparison. In S. Lankes and C. Clauss, editors, *Proceedings of the Many-core Applications Research Community (MARC) Symposium at RWTH Aachen University*, pages 38–44, November 2012.

[9] Apple. The Swift Compiler. https://developer.apple.com/swift. Accessed: 20/09/2015.

[10] Mauricio Araya-Polo, Félix Rubio, Raúl de la Cruz, Mauricio Hanzich, José María Cela, and Daniele Paolo Scarpazza. 3D seismic imaging through reverse-time migration on homogeneous and heterogeneous multi-core processors. *Sci. Program.*, 17(1-2):185–198, 2009.

[11] ARM Ltd. ARM Compiler toolchain Version 5.03. Assembler Reference. http://infocenter.arm.com/help/topic/com.arm.doc.dui0489i/DUI0489I_arm_assembler_reference.pdf. Accessed: 16/08/2015.

[12] Eduard Ayguadé, Rosa M. Badia, Pieter Bellens, Javier Bueno-Hedo, Alejandro Duran, Yoav Etsion, Montse Farreras, Roger Ferrer, Jesús Labarta, Vladimir Marjanovic, Luis Martinell, Xavier Martorell, Josep M. Pérez, Judit Planas, Alex Ramirez, Xavier Teruel, Ioanna Tsalouchidou, and Mateo Valero. Hybrid/Heterogeneous Programming with OmpSs and its Software/Hardware Implications. In *Programming Multi-Core and Many-Core Computing Systems*. John Wiley & Sons, Inc., Wiley Series on Parallel and Distributed Computing edition, July 2012.

[13] Abdel-Hameed A. Badawy, Aneesh Aggarwal, Donald Yeung, and Chau-Wen Tseng. The efficacy of software prefetching and locality optimizations on future memory systems. *Journal of Instruction-Level Parallelism*, 6, 2004.

[14] David H. Bailey, Eric Barszcz, John T. Barton, D. S. Browning, Robert L. Carter, Leonardo Dagum, Rod A. Fatoohi, Paul O. Frederickson, Tom A. Lasinski, Robert S. Schreiber, Horst D. Simon, V. Venkatakrishnan, and Sisira K. Weeratunga. The NAS Parallel Benchmarks - Summary and Preliminary Results. In *Proceedings of the 1991 ACM/IEEE Conference on Supercomputing*, Supercomputing '91, pages 158–165, New York, NY, USA, 1991. ACM.

[15] Aart J. C. Bik, David L. Kreitzer, and Xinmin Tian. A Case Study on Compiler Optimizations for the Intel®Core$^{\text{TM}}$2 Duo Processor. *Int. J. Parallel Program.*, 36(6):571–591, December 2008.

[16] Aart J.C. Bik, Milind Girkar, Paul M. Grey, and Xinmin Tian. Automatic intra-register vectorization for the Intel® architecture. *Int. Journal of Parallel Programming*, 30(2):65–98, 2002.

[17] Andrey Bokhanko. Intel Clang-based C++ Compiler. http://llvm.org/devmtg/2014-04/PDFs/Posters/ClangIntel.pdf. Accessed: 20/09/2015.

[18] Koen Bosschere, Wayne Luk, Xavier Martorell, Nacho Navarro, Mike O'Boyle, Dionisios Pnevmatikatos, Alex Ramirez, Pascal Sainrat, André

Seznec, Per Stenström, and Olivier Temam. High-performance embedded architecture and compilation roadmap. In Per Stenström, editor, *Transactions on High-Performance Embedded Architectures and Compilers I*, pages 5–29. Springer-Verlag, Berlin, Heidelberg, 2007.

[19] W. Jack Bouknight, Stewart A. Denenberg, David E. McIntyre, J. M. Randall, Amed H. Sameh, and Daniel L. Slotnick. The Illiac IV system. *Proceedings of the IEEE*, 60(4):369–388, April 1972.

[20] J. Mark Bull, Fiona Reid, and Nicola McDonnell. A microbenchmark suite for openmp tasks. In *Proceedings of the 8th International Conference on OpenMP in a Heterogeneous World*, IWOMP'12, pages 271–274, Berlin, Heidelberg, 2012. Springer-Verlag.

[21] Diego Caballero. User-directed Vectorization in OmpSs. Master's thesis, Universitat Politècnica de Catalunya, Barcelona, Spain, September 2011.

[22] Diego Caballero, Alejandro Duran, and Xavier Martorell. An OpenMP Barrier Using SIMD Instructions for Intel® Xeon Phi® Coprocessor. In AlistairP. Rendell, BarbaraM. Chapman, and MatthiasS. Müller, editors, *Proceedings of the 9th International Workshop on OpenMP, IWOMP 2013. OpenMP in the Era of Low Power Devices and Accelerators*, volume 8122 of *Lecture Notes in Computer Science*, pages 99–113. Springer Berlin Heidelberg, 2013.

[23] Diego Caballero, Albert Farres, Alejandro Duran, Mauricio Hanzich, Santiago Fernández, and Xavier Martorell. Optimizing Fully Anisotropic Elastic Propagation on Intel Xeon Phi Coprocessors. In *Second EAGE Workshop on High Performance Computing for Upstream*, Dubai, United Arab Emirates, September 2015.

[24] Diego Caballero, Sara Royuela, Roger Ferrer, Alejandro Duran, and Xavier Martorell. Optimizing Overlapped Memory Accesses in User-directed Vectorization. In *Proceedings of the 29th ACM on International Conference on Supercomputing*, ICS '15, pages 393–404, New York, NY, USA, 2015. ACM.

[25] David Callahan, Ken Kennedy, and Allan Porterfield. Software prefetching. In *Proceedings of the Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS IV, pages 40–52, New York, NY, USA, 1991. ACM.

[26] Muhammad Omer Cheema and Omar Hammami. Application-specific SIMD Synthesis for Reconfigurable Architectures. *Microprocessors and Microsystems*, 30(6):398–412, 2006.

[27] Tong Chen, R. Raghavan, Jonathan N. Dale, and E. Iwata. Cell broadband engine architecture and its first implementation: A performance view. *IBM J. Res. Dev.*, 51(5):559–572, September 2007.

[28] Francis Y. Chin, John Lam, and I-Ngo Chen. Efficient parallel algorithms for some graph problems. *Commun. ACM*, 25(9):659–665, September 1982.

[29] European Commission. The MontBlanc 3 Project. `http://cordis.europa.eu/project/rcn/197943_en.html`. Accessed: 11/10/2015.

[30] Computer Architecture Deparment, Technical University of Catalonia. Doctoral program on computer architecture. `http://www.ac.upc.edu/en/phd-computer-architecture`, 2012. Accessed: 02/06/2014.

[31] Computing Community Consortium. 21st Century Computer Architecture, May 2012.

[32] Jesus Corbal, Roger Espasa, and Mateo Valero. On the efficiency of reductions in mu;-simd media extensions. In *Parallel Architectures and Compilation Techniques, 2001. Proceedings. 2001 International Conference on*, pages 83–94, 2001.

[33] IBM Corporation. *PowerPC Microprocessor Family: Vector/SIMD Multimedia Extension Technology Programming Environments Manual*, 2005.

[34] Intel Corporation. Balanced affinity type. Intel® C++ Compiler XE 13.1 User and Reference Guides. `http://software.intel.com/sites/products/documentation/doclib/iss/2013/compiler/cpp-lin/`, . Accessed: 2013-05-09.

[35] Intel Corporation. The Cilk Plus GCC. `https://www.cilkplus.org/build-gcc-cilkplus`, . Accessed: 20/09/2015.

[36] Intel Corporation. Intel OpenCL Compiler. `https://software.intel.com/en-us/intel-opencl`, . Accessed: 30/10/2015.

[37] Intel Corporation. Intel®Advanced Vector Extensions Programming Reference, 2011.

[38] Intel Corporation. Intel® Xeon Phi™ Coprocessor - The Architecture. http://software.intel.com/en-us/articles/intel-xeon-phi-coprocessor-codename-knights-corner, 2012. Accessed: 28/06/2014.

[39] Intel Corporation. Intel® Xeon Phi™ Coprocessor Instruction Set Architecture Reference Manual, 2012.

[40] Intel Corporation. Intel® Xeon Phi™ Core Micro-architecture. http://software.intel.com/sites/default/files/article/393195/intel-xeon-phi-core-micro-architecture.pdf, 2012. Accessed: 28/06/2014.

[41] Nvdia Corporation. NVIDIA GeForce GTX 980 Whitepaper. Featuring Maxwell, The Most Advanced GPU Ever Made. `http://international.download.nvidia.com/geforce-com/international/pdfs/GeForce_GTX_980_Whitepaper_FINAL.PDF`, 2014.

[42] NVIDIA Corporation. CUDA LLVM Compiler. `https://developer.nvidia.com/cuda-llvm-compiler`, . Accessed: 20/09/2015.

[43] European Research Council. Riding on Moore's Law (RoMoL). `http://erc.europa.eu/riding-moores-law`. Accessed: 11/10/2015.

[44] B. Coutinho, D. Sampaio, F.M.Q. Pereira, and W. Meira. Divergence analysis and optimizations. In *Parallel Architectures and Compilation Techniques (PACT), 2011 International Conference on*, pages 320–329, Oct 2011. doi: 10.1109/PACT.2011.63.

[45] Ron Cytron, Jeanne Ferrante, Barry K. Rosen, Mark N. Wegman, and F. Kenneth Zadeck. Efficiently computing static single assignment form and the control dependence graph. *ACM Trans. Program. Lang. Syst.*, 13(4):451–490, October 1991.

[46] Kaushik Datta, Mark Murphy, Vasily Volkov, Samuel Williams, Jonathan Carter, Leonid Oliker, David Patterson, John Shalf, and Katherine Yelick. Stencil computation optimization and auto-tuning on state-of-the-art multicore architectures. In *Proceedings of the 2008 ACM/IEEE Conference on Supercomputing*, SC '08, pages 4:1–4:12, Piscataway, NJ, USA, 2008. IEEE Press.

[47] Kaushik Datta, Samuel Williams, Vasily Volkov, Jonathan Carter, Leonid Oliker, John Shalf, and Katherine Yelick. Auto-tuning the 27-point stencil for multicore. In *Proceeding of the 4th International Workshop on Automatic Performance Tuning*, 2009.

[48] Raúl de la Cruz and Mauricio Araya-Polo. Algorithm 942: Semi-Stencil. *ACM TOMS*, 40(3):23:1–23:39, April 2014.

[49] Jim Demmel. A Look at Some Parallel Architectures. `http://www.cs.berkeley.edu/~demmel/cs267-1995/lecture10/lecture10.html`. Accessed: 20/09/2015.

[50] Gregory Diamos, Benjamin Ashbaugh, Subramaniam Maiyuran, Andrew Kerr, Haicheng Wu, and Sudhakar Yalamanchili. Simd re-convergence at thread frontiers. In *Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-44, pages 477–488, New York, NY, USA, 2011. ACM.

[51] Alejandro Duran, Roger Ferrer, Michael Klemm, BronisR. de Supinski, and Eduard Ayguadé. A proposal for user-defined reductions in openmp. In Mitsuhisa Sato, Toshihiro Hanawa, MatthiasS. Müller, BarbaraM. Chapman, and BronisR. de Supinski, editors, *Beyond Loop Level Parallelism in OpenMP: Accelerators, Tasking and More*, volume 6132 of *Lecture Notes in Computer Science*, pages 43–55. Springer Berlin Heidelberg, 2010.

[52] Sandhya Dwarkadas, Alan L. Cox, and Willy Zwaenepoel. An integrated compile-time/run-time software distributed shared memory system. In *Proceedings of the Seventh International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS VII, pages 186–197, New York, NY, USA, 1996. ACM.

[53] Alexandre E. Eichenberger and Santosh G. Abraham. Impact of load imbalance on the design of software barriers. In *in Proceedings of the 1995 International Conference on Parallel Processing*, pages 63–72, 1995.

[54] Alexandre E. Eichenberger, Peng Wu, and Kevin O'Brien. Vectorization for simd architectures with alignment constraints. In *Proceedings of the ACM SIGPLAN 2004 Conference on Programming Language Design and Implementation*, PLDI '04, pages 82–93, New York, NY, USA, 2004. ACM.

[55] Rudolf Eigenmann, Jay Hoeflinger, Z. Li, and David A. Padua. Experience in the automatic parallelization of four perfect-benchmark programs. In *Proceedings of the Fourth International Workshop on Languages and Compilers for Parallel Computing*, pages 65–83, London, UK, UK, 1992. Springer-Verlag.

[56] Pierre Estérie, Joel Falcou, Mathias Gaunard, and Jean-Thierry Lapresté. Boost.simd: Generic programming for portable simdization. In *Proceedings of the 2014 Workshop on Programming Models for SIMD/Vector Processing*, WPMVP '14, pages 1–8, New York, NY, USA, 2014. ACM.

[57] Roger Ferrer, Sara Royuela, Diego Caballero, Alejandro Duran, Xavier Martorell, and Eduard Ayguadé. Mercurium: Design Decisions for a S2S Compiler. In *Cetus Users and Compiler Infastructure Workshop. In conjunction with PACT 2011*, 2011.

[58] María Jesús Garzarán, Milos Prvulovic, Ye Zhangy, Josep Torrellas, Alin Jula, Hao Yu, and Lawrence Rauchwerger. Architectural support for parallel reductions in scalable shared-memory multiprocessors. In *Proceedings of the 2001 International Conference on Parallel Architectures and Compilation Techniques*, PACT '01, pages 243–, Washington, DC, USA, 2001. IEEE Computer Society.

[59] Allan Gottlieb, Ralph Grishman, Clyde P. Kruskal, Kevin P. McAuliffe, Larry Rudolph, and Marc Snir. The nyu ultracomputer&mdash;designing a mimd,

shared-memory parallel machine (extended abstract). In *Proceedings of the 9th Annual Symposium on Computer Architecture*, ISCA '82, pages 27–42, Los Alamitos, CA, USA, 1982. IEEE Computer Society Press.

[60] Toolchain Working Group. Linaro GCC. `https://launchpad.net/gcc-linaro`. Accessed: 20/09/2015.

[61] Rajiv Gupta. The fuzzy barrier: a mechanism for high speed synchronization of processors. *SIGARCH Comput. Archit. News*, 17(2):54–63, April 1989.

[62] Rajiv Gupta and Charles R. Hill. A Scalable Implementation of Barrier Synchronization Using an Adaptive Combining Tree. *Int. J. Parallel Program.*, 18(3):161–180, June 1990.

[63] Hwansoo Han and C.-W. Tseng. Improving compiler and run-time support for adaptive irregular codes. In *Proceedings of the 1998 International Conference on Parallel Architectures and Compilation Techniques*, PACT '98, pages 393–, Washington, DC, USA, 1998. IEEE Computer Society.

[64] Torsten Hoefler, Torsten Mehlan, Frank Mietke, and Wolfgang Rehm. A Survey of Barrier Algorithms for Coarse Grained Supercomputers, Dec. 2004.

[65] Johannes Hofmann, Jan Treibig, Georg Hager, and Gerhard Wellein. Comparing the performance of different x86 simd instruction sets for a medical imaging application on modern multi- and manycore chips. In *Proceedings of the 2014 Workshop on Programming Models for SIMD/Vector Processing*, WPMVP '14, pages 57–64, New York, NY, USA, 2014. ACM.

[66] Wei Huang, Mircea R. Stant, Karthik Sankaranarayanan, Robert J. Ribando, and Kevin Skadron. Many-core design from a thermal perspective. In *Proceed. of the 45th annual Design Automation Conference*, DAC '08, pages 746–749, New York, NY, USA, 2008. ACM.

[67] Free Software Foundation, Inc. Vector Extensions in GCC. `https://gcc.gnu.org/onlinedocs/gcc/Vector-Extensions.html`, .

[68] Free Software Foundation, Inc. GCC, the GNU Compiler Collection. `http://gcc.gnu.org`, .

[69] Free Software Foundation, Inc. Auto-vectorization in GCC. `http://gcc.gnu.org/projects/tree-ssa/vectorization.html`, .

[70] Intel Corporation. Intel® Cilk$^{TM}$ Plus Language Extension Specification Version 1.2. `https://www.cilkplus.org/sites/default/files/open_specifications/Intel_Cilk_plus_lang_spec_1.2.htm`, . Accessed: 24/06/2014.

[71] Intel Corporation. Intel Intrinsics Guide. `https://software.intel.com/sites/landingpage/IntrinsicsGuide`, . Accessed: 24/06/2014.

[72] Intel Corporation. Intel®SSE4 Programming Reference, 2007.

[73] Intel Corporation. Experiences in developing Seismic Imaging code for Intel® Xeon Phi$^{TM}$ Coprocessor. `https://software.intel.com/en-us/blogs/2012/10/26/experiences-in-developing-seismic-imaging-code-for-intel-xeon-phi-coprocessor`, 2012. Accessed: 2015-03-21.

[74] Intel Corporation. Intel®Architecture - Instruction Set Extensions Programming Reference, October 2014.

[75] Intel Corporation. The Intel® OpenMP* Runtime Library, July 2015. URL `https://www.openmprtl.org/`.

[76] James Jeffers and James Reinders. *Intel Xeon Phi Coprocessor High-Performance Programming*. Elsevier Science, 2013.

[77] Ana Lucia Varbanescu Jianbin Fang. CLVectorizer: A Source-to-Source Vectorizer for OpenCL Kernels. Technical report, Delft University of Technology, Delft, the Netherlands, November 2011. URL `http://www.pds.ewi.tudelft.nl/fileadmin/pds/homepages/fang/papers/PDS-2011-012.pdf`.

[78] Shoaib Kamil, Parry Husbands, Leonid Oliker, John Shalf, and Katherine Yelick. Impact of modern memory subsystems on cache optimizations for stencil computations. In *Proceedings of the 2005 Workshop on Memory System Performance*, MSP '05, pages 36–43, New York, NY, USA, 2005. ACM.

[79] Ralf Karrenberg. *Automatic SIMD Vectorization of SSA-based Control Flow Graphs*. Springer Vieweg, 2015. ISBN 3658101121, 9783658101121.

[80] Ralf Karrenberg and Sebastian Hack. Whole-function vectorization. In *Proceedings of the 9th Annual IEEE/ACM International Symposium on Code Generation and Optimization*, CGO '11, pages 141–150, Washington, DC, USA, 2011. IEEE Computer Society.

[81] Ken Kennedy and John R. Allen. *Optimizing Compilers for Modern Architectures: A Dependence-based Approach*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2002.

[82] Khronos OpenCL Working Group. The OpenCL Specification. Version 2.0, March 2014. `http://www.khronos.org/registry/cl/`.

[83] Michael Klemm, Alejandro Duran, Xinmin Tian, Hideki Saito, Diego Caballero, and Xavier Martorell. Extending OpenMP with Vector Constructs for Modern Multicore SIMD Architectures. In BarbaraM. Chapman, Federico Massaioli, MatthiasS. Müller, and Marco Rorro, editors, *Proceedings of the 8th International Workshop on OpenMP, IWOMP 2012. OpenMP in a Heterogeneous World*, volume 7312 of *Lecture Notes in Computer Science*, pages 59–72. Springer Berlin Heidelberg, 2012.

[84] Martin Kong, Richard Veras, Kevin Stock, Franz Franchetti, Louis-Noël Pouchet, and P. Sadayappan. When polyhedral transformations meet simd code generation. In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '13, pages 127–138, New York, NY, USA, 2013. ACM.

[85] Rakesh Krishnaiyer, Emre Kultursay, Pankaj Chawla, Serguei Preis, Anatoly Zvezdin, and Hideki Saito. Compiler-based data prefetching and streaming non-temporal store generation for the intel(r) xeon phi(tm) coprocessor. In *Proceedings of the 2013 IEEE 27th International Symposium on Parallel and Distributed Processing Workshops and PhD Forum*, IPDPSW '13, pages 1575–1586, Washington, DC, USA, 2013. IEEE Computer Society.

[86] Olaf Krzikalla, Kim Feldhoff, Ralph Müller-Pfefferkorn, and Wolfgang E. Nagel. Auto-Vectorization Techniques for Modern SIMD Architectures. In *Proceedings of the 16th Workshop on Compilers for Parallel Computing*, Padova, Italy, January 2012.

[87] Olaf Krzikalla, Kim Feldhoff, Ralph Müller-Pfefferkorn, and Wolfgang E. Nagel. Scout: A source-to-source transformator for simd-optimizations. In Michael Alexander, Pasqua D'Ambra, Adam Belloum, George Bosilca, Mario Cannataro, Marco Danelutto, Beniamino Di Martino, Michael Gerndt, Emmanuel Jeannot, Raymond Namyst, Jean Roman, StephenL. Scott, JesperLarsson Traff, Geoffroy Vallée, and Josef Weidendorfer, editors, *Euro-Par 2011: Parallel Processing Workshops*, volume 7156 of *Lecture Notes in Computer Science*, pages 137–145. Springer Berlin Heidelberg, 2012. doi: 10.1007/978-3-642-29740-3_17.

[88] David J. Kuck, Y. Muraoka, and Shyh-Ching Chen. On the number of operations simultaneously executable in fortran-like programs and their resulting speedup. *IEEE Trans. Comput.*, 21(12):1293–1310, December 1972.

[89] David J. Kuck, Robert H. Kuhn, Bruce Leasure, and Michael Wolfe. The structure of an advanced vectorizer for pipelined processors. In *Proceedings of the IEEE Computer Society 4th International Computer Software and Applications Conference*, 1980.

[90] Lionel Lacassagne, Daniel Etiemble, Ali Hassan Zahraee, Alain Dominguez, and Pascal Vezolle. High level transforms for simd and low-level computer vision algorithms. In *Proceedings of the 2014 Workshop on Programming Models for SIMD/Vector Processing*, WPMVP '14, pages 49–56, New York, NY, USA, 2014. ACM.

[91] Leslie Lamport. The parallel execution of do loops. *Commun. ACM*, 17(2): 83–93, February 1974.

[92] Samuel Larsen and Saman Amarasinghe. Exploiting superword level parallelism with multimedia instruction sets. In *Proceedings of the ACM SIGPLAN 2000 Conference on Programming Language Design and Implementation*, PLDI '00, pages 145–156, New York, NY, USA, 2000. ACM.

[93] Samuel Larsen, Emmett Witchel, and Saman P. Amarasinghe. Increasing and detecting memory address congruence. In *Proceedings of the 2002 International Conference on Parallel Architectures and Compilation Techniques*, PACT '02, pages 18–29, Washington, DC, USA, 2002. IEEE Computer Society.

[94] Chris Lattner. The LLVM Compiler Intrastructure. `http://llvm.org`.

[95] Chris Lattner and Vikram Adve. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *Proceedings of the 2004 International Symposium on Code Generation and Optimization (CGO'04)*, Palo Alto, California, Mar 2004.

[96] Lawrence Livermore National Laboratory. The ROSE Source-to-source Compiler. `http://rosecompiler.org/`. Accessed: 02/11/2015.

[97] Jaekyu Lee, Hyesoon Kim, and Richard Vuduc. When prefetching works, when it doesn&rsquo;t, and why. *ACM Trans. Archit. Code Optim.*, 9(1):2:1–2:29, March 2012.

[98] Yuan Lin and David A. Padua. On the Automatic Parallelization of Sparse and Irregular FORTRAN Programs. In *Selected Papers from the 4th International Workshop on Languages, Compilers, and Run-Time Systems for Scalable Computers*, LCR '98, pages 41–56, London, UK, UK, 1998. Springer-Verlag.

[99] Saeed Maleki, Yaoqing Gao, Maria J. Garzarán, Tommy Wong, and David A. Padua. An evaluation of vectorizing compilers. In *Proceedings of the 2011 International Conference on Parallel Architectures and Compilation Techniques*, PACT '11, pages 372–382, Washington, DC, USA, 2011. IEEE Computer Society.

[100] Frank H. McMahon. The Livermore FORTRAN kernels: A computer test of the numerical performance range, 1986.

[101] John M. Mellor-Crummey and Michael L. Scott. Algorithms for scalable synchronization on shared-memory multiprocessors. *ACM Trans. Comput. Syst.*, 9(1):21–65, February 1991.

[102] A. Mericas, N. Peleg, L. Pesantez, S.B. Purushotham, P. Oehler, C.A. Anderson, B.A. King-Smith, M. Anand, J.A. Arnold, B. Rogers, L. Maurice, and K. Vu. Ibm power8 performance features and evaluation. *IBM Journal of Research and Development*, 59(1):6:1–6:10, Jan 2015.

[103] MPI: A Message-Passing Interface Standard Version 3.0. Message passing interface forum. `http://www.mpi-forum.org/docs/mpi-3.0/mpi30-report.pdf`. Accessed: 26/06/2014.

[104] MPI Forum. MPI: Extensions to the Message-passing Interface, Version 2.2. Technical report, September 2009.

[105] Steven S. Muchnick. *Advanced Compiler Design and Implementation*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1997.

[106] Yoichi Muraoka. *Parallelism Exposure and Exploitation in Programs*. PhD thesis, University of Illinois at Urbana-Champaign, Champaign, IL, USA, 1971. AAI7121189.

[107] Dorit Naishlos. Autovectorization in GCC. *GCC Summit*, June 2004.

[108] Dorit Naishlos, Marina Biberstein, Shay Ben-David, and Ayal Zaks. Vectorizing for a simdd dsp architecture. In *Proceedings of the 2003 International Conference on Compilers, Architecture and Synthesis for Embedded Systems*, CASES '03, pages 2–11, New York, NY, USA, 2003. ACM.

[109] Ramachandra Nanjegowda, Oscar Hernandez, Barbara Chapman, and Haoqiang H. Jin. Scalability evaluation of barrier algorithms for openmp. In *Proceedings of the 5th International Workshop on OpenMP: Evolving OpenMP in an Age of Extreme Parallelism*, IWOMP '09, pages 42–52, Berlin, Heidelberg, 2009. Springer-Verlag.

[110] Dorit Nuzman and Richard Henderson. Multi-platform auto-vectorization. In *Proceedings of the International Symposium on Code Generation and Optimization*, CGO '06, pages 281–294, Washington, DC, USA, 2006. IEEE Computer Society.

[111] Dorit Nuzman and Ayal Zaks. Outer-loop vectorization: Revisited for short simd architectures. In *Proceedings of the 17th International Conference on Parallel Architectures and Compilation Techniques*, PACT '08, pages 2–11, New York, NY, USA, 2008. ACM.

[112] Dorit Nuzman, Ira Rosen, and Ayal Zaks. Auto-vectorization of interleaved data for simd. In *Proceedings of the 27th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '06, pages 132–143, New York, NY, USA, 2006. ACM.

[113] The Trustees of Indiana University. ConceptGCC. `http://www.generic-programming.org/software/ConceptGCC/`. Accessed: 20/09/2015.

[114] Mads Chr. Olesen, René Rydhof Hansen, Julia L. Lawall, and Nicolas Palix. Clang and coccinelle: Synergising program analysis tools for CERT C secure coding standard certification. In *4th International Workshop on Foundations and Techniques for Open Source Software Certification*, Pisa, Italy, September 2010.

[115] OpenACC Architecture Review Board. The openacc$^{TM}$ application programming interface version 2.0. `http://www.openacc.org/sites/default/files/OpenACC.2.0a_1.pdf`, June 2013.

[116] OpenMP Architecture Review Board. OpenMP application program interface version 3.1. `http://www.openmp.org/mp-documents/OpenMP3.1.pdf`, July 2011.

[117] OpenMP Architecture Review Board. OpenMP application program interface version 4.0. `http://www.openmp.org/mp-documents/OpenMP4.0.0.pdf`, July 2013.

[118] David A. Padua and Michael J. Wolfe. Advanced compiler optimizations for supercomputers. *Commun. ACM*, 29(12):1184–1201, December 1986.

[119] J. C. H. Park and M. S. Schlansker. *On Predicated Execution*. HP Laboratories Technical Report HPL-91-58, 1991.

[120] Simon Peyton Jones et al. The Haskell 98 Language and Libraries: The Revised Report. *Journal of Functional Programming*, 13(1):0–255, Jan 2003. `http://www.haskell.org/definition/`.

[121] Gregory F. Pfister and V. Alan Norton. "Hot Spot" contention and combining in multistage interconnection networks. In Isaac D. Scherson and Abdou S. Youssef, editors, *Interconnection Networks for High-performance Parallel Computers*, pages 276–281. IEEE Computer Society Press, Los Alamitos, CA, USA, 1994.

[122] Matt Pharr and William R. Mark. ispc: A SPMD compiler for high-performance CPU programming. In *Innovative Parallel Computing (InPar), 2012*, pages 1–13, May 2012.

[123] Fred J. Pollack. New Microarchitecture Challenges in the Coming Generations of CMOS Process Technologies. In *Proceedings of the 32Nd Annual ACM/IEEE International Symposium on Microarchitecture*, MICRO 32, pages 2–, Washington, DC, USA, 1999. IEEE Computer Society.

[124] Programming Models Group, Barcelona Supercomputing Center. The Programming Models Group site. `http://pm.bsc.es`. Accessed: 02/06/2014.

[125] Fernando Magno Quintao Pereira, Raphael Ernani Rodrigues, and Victor Hugo Sperle Campos. A fast and low-overhead technique to secure programs against integer overflows. In *Proceedings of the 2013 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, CGO '13, pages 1–11, Washington, DC, USA, 2013. IEEE Computer Society.

[126] Rolf Rabenseifner and JesperLarsson Träff. More efficient reduction algorithms for non-power-of-two number of processors in message-passing parallel systems. In Dieter Kranzlmüller, Péter Kacsuk, and Jack Dongarra, editors, *Recent Advances in Parallel Virtual Machine and Message Passing Interface*, volume 3241 of *Lecture Notes in Computer Science*, pages 36–46. Springer Berlin Heidelberg, 2004.

[127] Sabela Ramos and Torsten Hoefler. Modeling communication in cache-coherent smp systems: A case-study with xeon phi. In *Proceedings of the 22Nd International Symposium on High-performance Parallel and Distributed Computing*, HPDC '13, pages 97–108, New York, NY, USA, 2013. ACM.

[128] G. Rapaport, A. Zaks, and Y. Ben-Asher. Streamlining Whole Function Vectorization in C Using Higher Order Vector Semantics. In *Parallel and Distributed Processing Symposium Workshop (IPDPSW), 2015 IEEE International*, pages 718–727, May 2015.

[129] Andrey Rodchenko, Andy Nisbet, Antoniu Pop, and Mikel Luján. Effective barrier synchronization on intel xeon phi coprocessor. In Jesper Larsson Träff, Sascha Hunold, and Francesco Versaci, editors, *Euro-Par 2015: Parallel Processing*, volume 9233 of *Lecture Notes in Computer Science*, pages 588–600. Springer Berlin Heidelberg, 2015.

[130] Ira Rosen, Dorit Nuzman, and Ayal Zaks. Loop-aware slp in gcc. *GCC Summit*, July 2007.

[131] Sara Royuela, Roger Ferrer, Diego Caballero, and Xavier Martorell. Compiler Analysis for OpenMP Tasks Correctness. In *Proceedings of the 12th ACM International Conference on Computing Frontiers*, CF '15, pages 7:1–7:8, New York, NY, USA, 2015. ACM.

[132] D. Sampaio, R. Martins, S. Collange, and F.M.Q. Pereira. Divergence analysis with affine constraints. In *Computer Architecture and High Performance Computing (SBAC-PAD), 2012 IEEE 24th International Symposium on*, pages 67–74, Oct 2012. doi: 10.1109/SBAC-PAD.2012.22.

[133] Jack Sampson, Ruben Gonzalez, Jean-Francois Collard, Norman P. Jouppi, Mike Schlansker, and Brad Calder. Exploiting fine-grained data parallelism with chip multiprocessors and fast barriers. In *Proceedings of the 39th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO 39, pages 235–246, Washington, DC, USA, 2006. IEEE Computer Society.

[134] John Sartori and Rakesh Kumar. Low-overhead, high-speed multi-core barrier synchronization. In *Proceedings of the 5th International Conference on High Performance Embedded Architectures and Compilers*, HiPEAC'10, pages 18–34, Berlin, Heidelberg, 2010. Springer-Verlag.

[135] Nadathur Satish, Changkyu Kim, Jatin Chhugani, Hideki Saito, Rakesh Krishnaiyer, Mikhail Smelyanskiy, Milind Girkar, and Pradeep Dubey. Can traditional programming bridge the ninja performance gap for parallel computing applications? In *Proceedings of the 39th Annual International Symposium on Computer Architecture*, ISCA '12, pages 440–451, Washington, DC, USA, 2012. IEEE Computer Society.

[136] Dirk Schmidl, Tim Cramer, Sandra Wienke, Christian Terboven, and MatthiasS. Müller. Assessing the performance of openmp programs on the intel xeon phi. In Felix Wolf, Bernd Mohr, and Dieter an Mey, editors, *Euro-Par 2013 Parallel Processing*, volume 8097 of *Lecture Notes in Computer Science*, pages 547–558. Springer Berlin Heidelberg, 2013.

[137] Paul B. Schneck. Automatic recognition of vector and parallel operations in a higher level language. In *Proceedings of the ACM Annual Conference - Volume 2*, ACM '72, pages 772–779, New York, NY, USA, 1972. ACM.

[138] Michael L. Scott and John M. Mellor-Crummey. Fast, contention-free combining tree barriers for shared-memory multiprocessors. *Int. J. Parallel Program.*, 22(4):449–481, August 1994.

[139] Steven L. Scott. Synchronization and communication in the T3E multiprocessor. *SIGPLAN Not.*, 31(9):26–36, September 1996.

[140] Jaewook Shin. Introducing control flow into vectorized code. In *Proceedings of the 16th International Conference on Parallel Architecture and Compilation Techniques*, PACT '07, pages 280–291, Washington, DC, USA, 2007. IEEE Computer Society.

[141] Jaewook Shin, Jacqueline Chame, and Mary W. Hall. Compiler-controlled caching in superword register files for multimedia extension architectures.

In *Proceedings of the 2002 International Conference on Parallel Architectures and Compilation Techniques*, PACT '02, pages 45–55, Washington, DC, USA, 2002. IEEE Computer Society.

[142] Lauren L. Smith. Vectorizing c compilers: How good are they? In *Proceedings of the 1991 ACM/IEEE Conference on Supercomputing*, Supercomputing '91, pages 544–553, New York, NY, USA, 1991. ACM.

[143] Rajeev Thakur and WilliamD. Gropp. Improving the performance of collective operations in mpich. In Jack Dongarra, Domenico Laforenza, and Salvatore Orlando, editors, *Recent Advances in Parallel Virtual Machine and Message Passing Interface*, volume 2840 of *Lecture Notes in Computer Science*, pages 257–267. Springer Berlin Heidelberg, 2003.

[144] The LLVM Project. clang: a C language family frontend for LLVM. `http://http://clang.llvm.org/`, . Accessed: 02/11/2015.

[145] The LLVM Project. The llvm target-independent code generator. `http://www.llvm.org/docs/CodeGenerator.html#legalize-selectiondag-types`, . Accessed: 25/08/2015.

[146] The Portland Group, Inc. PGI CUDA-X86. `https://www.pgroup.com/resources/cuda-x86.htm`.

[147] Xinmin Tian, Hideki Saito, Milind Girkar, Serguei V. Preis, Sergey S. Kozhukhov, Aleksei G. Cherkasov, Clark Nelson, Nikolay Panchenko, and Robert Geva. Compiling c/c++ simd extensions for function and loop vectorizaion on multicore-simd processors. In *Proceedings of the 2012 IEEE 26th International Parallel and Distributed Processing Symposium Workshops & PhD Forum*, IPDPSW '12, pages 2349–2358, Washington, DC, USA, 2012. IEEE Computer Society.

[148] Linda Torczon and Keith Cooper. *Engineering A Compiler*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2nd edition, 2011. ISBN 012088478X.

[149] Konrad Trifunovic, Dorit Nuzman, Albert Cohen, Ayal Zaks, and Ira Rosen. Polyhedral-model guided loop-nest auto-vectorization. In *Proceedings of the 2009 18th International Conference on Parallel Architectures and Compilation Techniques*, PACT '09, pages 327–337, Washington, DC, USA, 2009. IEEE Computer Society.

[150] JesperLarsson Träff. An improved algorithm for (non-commutative) reduce-scatter with an application. In Beniamino Di Martino, Dieter Kranzlmüller, and Jack Dongarra, editors, *Recent Advances in Parallel Virtual Machine and Message Passing Interface*, volume 3666 of *Lecture Notes in Computer Science*, pages 129–137. Springer Berlin Heidelberg, 2005.

[151] Manjunath G. Venkata, Pavel Shamis, Rahul Sampath, Richard L. Graham, and Joshua S. Ladd. Optimizing blocking and nonblocking reduction operations for multicore systems: Hierarchical design and implementation. In *Cluster Computing (CLUSTER), 2013 IEEE International Conference on*, pages 1–8, Sept 2013.

[152] Oreste Villa, Gianluca Palermo, and Cristina Silvano. Efficiency and Scalability of Barrier Synchronization on NoC Based Many-core Architectures. In *Proceedings of the 2008 International Conference on Compilers, Architectures and Synthesis for Embedded Systems*, CASES '08, pages 81–90, New York, NY, USA, 2008. ACM.

[153] Haichuan Wang, Peng Wu, Ilie Gabriel Tanase, Mauricio J. Serrano, and José E. Moreira. Simple, portable and fast simd intrinsic programming: Generic simd library. In *Proceedings of the 2014 Workshop on Programming Models for SIMD / Vector Processing*, WPMVP '14, pages 9–16, New York, NY, USA, 2014. ACM.

[154] Michael Wolfe. Iteration space tiling for memory hierarchies. In *Proceedings of the Third SIAM Conference on Parallel Processing for Scientific Computing*, pages 357–361, Philadelphia, PA, USA, 1989. Society for Industrial and Applied Mathematics.

[155] Michael Joseph Wolfe. *High Performance Compilers for Parallel Computing*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1995.

[156] H'sien Jie Wong, J. Cai, Alistair P. Rendell, and Peter E. Strazdins. Micro-benchmarks for cluster openmp implementations: Memory consistency costs. In *Proceedings of the 4th International Conference on OpenMP in a New Era of Parallelism*, IWOMP'08, pages 60–70, Berlin, Heidelberg, 2008. Springer-Verlag.

[157] Peng Wu, Alexandre E. Eichenberger, and Amy Wang. Efficient simd code generation for runtime alignment and length conversion. In *Proceedings of the International Symposium on Code Generation and Optimization*, CGO '05, pages 153–164, Washington, DC, USA, 2005. IEEE Computer Society.

[158] Peng Wu, Alexandre E. Eichenberger, Amy Wang, and Peng Zhao. An integrated simdization framework using virtual vectors. In *Proceedings of the 19th Annual International Conference on Supercomputing*, ICS '05, pages 169–178, New York, NY, USA, 2005. ACM.

[159] P.-C. Yew, N.-F. Tzeng, and D. H. Lawrie. Distributing hot-spot addressing in large-scale multiprocessors. *IEEE Trans. Comput.*, 36(4):388–395, April 1987.

[160] Hao Yu and Lawrence Rauchwerger. Adaptive reduction parallelization techniques. In *Proceedings of the 14th International Conference on Supercomputing*, ICS '00, pages 66–77, New York, NY, USA, 2000. ACM.

[161] Guansong Zhang, Francisco Martínez, Arie Tal, and Bob Blainey. Busy-wait barrier synchronization using distributed counters with local sensor. In *Proceedings of the OpenMP Applications and Tools 2003 International Conference on OpenMP Shared Memory Parallel Programming*, WOMPAT'03, pages 84–98, Berlin, Heidelberg, 2003. Springer-Verlag.