Design of Energy-Efficient Vector Units for In-order Cores

Milan Stanić

Department of Computer Architecture

Universitat Politècnica de Catalunya - BarcelonaTech

A thesis submitted for the degree of

Doctor of Philosophy in Computer Architecture

October, 2016

Director: Dr. Oscar Palomar
Codirector: Prof. Mateo Valero

UNIVERSITAT POLITÈCNICA DE CATALUNYA
BARCELONATECH
Escola de Doctorat

Acta de calificación de tesis doctoral

Curso académico:

Nombre y apellidos

Programa de doctorado

Unidad estructural responsable del programa

Resolución del Tribunal

Reunido el Tribunal designado a tal efecto, el doctorando / la doctoranda expone el tema de su tesis doctoral titulada _____
_____.

Acabada la lectura y después de dar respuesta a las cuestiones formuladas por los miembros titulares del tribunal, éste otorga la calificación:

☐ NO APTO ☐ APROBADO ☐ NOTABLE ☐ SOBRESALIENTE

(Nombre, apellidos y firma) Presidente/a	(Nombre, apellidos y firma) Secretario/a

(Nombre, apellidos y firma) Vocal	(Nombre, apellidos y firma) Vocal	(Nombre, apellidos y firma) Vocal

_____, _____ de _____ de _____

El resultado del escrutinio de los votos emitidos por los miembros titulares del tribunal, efectuado por la Comisión Permanente de la Escuela de Doctorado, otorga la MENCIÓN CUM LAUDE:

☐ SÍ ☐ NO

(Nombre, apellidos y firma) Presidente/a de la Comisión Permanente de la Escuela de Doctorado	(Nombre, apellidos y firma) Secretario/a de la Comisión Permanente de la Escuela de Doctorado

Barcelona, _____ de _____ de _____

Abstract

In the last 15 years, power dissipation and energy consumption have become crucial design concerns for almost all computer systems. Technology feature size scaling leads to higher power density and therefore to complex and costly cooling. While power dissipation is critical for high-performance systems such as data centers due to large power usage, for mobile systems battery life is a primary concern. In the low-end mobile processor market, power, energy and area budgets are significantly lower than in the server/desktop/laptop/high-end mobile markets. The ultimate goal in low-end systems is also to increase performance, but only if area/energy budget is not compromised.

Vector architectures have been traditionally applied to the supercomputing domain with many successful incarnations. The energy efficiency and high performance of vector processors, as well as their applicability in other emerging domains, encourage pursuing further research on vector architectures. However adding support for them using conventional design incurs area and power overheads that would not be acceptable for low-end mobile processors and also there is a lack of appropriate tools to perform this research.

In this thesis, we propose an integrated vector-scalar design for the ARM architecture that mostly reuses scalar hardware to support the execution of vector instructions. The key element of the design is our proposed block-based model of execution that groups vector computational instructions together to execute them in a coordinated manner. We complement this with an advanced integrated design which features three energy-performance efficient ideas: (1) chaining from the memory hierarchy, (2) direct result forwarding and (3) memory shape instructions.

This thesis also presents two tools for measuring and analyzing an application suitability for vector microarchitectures. The first tool is VALib, a library that enables hand-crafted vectorization of applications and its main purpose is to collect data for detailed instruction level characterization and to generate input traces for the second tool. The second tool is SimpleVector, a fast trace-driven simulator that is used to estimate the execution time of a vectorized application on a candidate vector microarchitecture.

The thesis also evaluates characteristics of Knights Corner processor with simple in-order SIMD cores. Acquired knowledge is applied in the integrated design.

Keywords: Computer Architecture, Microarchitecture, Data-Level Parallelism, Vector Processors, Single Instruction Multiple Data, Energy Efficiency, Instruction Level Characterization, Tools, Simulator, Xeon Phi

Acknowledgements

Pursuing a PhD is a multi-year endeavour that can turn into a tedious and never ending journey. That was not my case, and I am thankful to a lot of people without whom I would not have been able to complete my PhD studies. While it is not possible to make an exhaustive list of names, I would like to mention a few. Apologies if I forget to mention any name below.

Firstly and foremost, I would like to express my sincere gratitude to my director - Dr. Oscar Palomar for all the help and guidance he provided during my PhD studies. You have been an outstanding mentor for me and it has been a privilage to work with you. I would also like to express my appreciation to my supervisors - Dr. Osman Ünsal, Dr. Adrián Cristal, and Prof. Mateo Valero. Their support, confidence, and sound technical advise have played a major role shaping my research ideas into the contributions expressed in this thesis. I appreciate the great opportunity that you have given to me.

A big thank you to the members of my thesis defence committee - Prof. Victor Viñals, Prof. Lasse Natvig, Prof. Eduard Ayguadé, Dr. Petar Radojković and Dr. Juan Manuel Cerbrián. I would also like to give a shot-out the the excellent staff at the Barcelona Supercomputing Center and UPC's Departament d'Arquitectura de Computadores, especially Joana Munuera and Dr. Xavier Masip.

I would also like to acknowledge all my colleagues from the office in Barcelona Supercomputing Center that helped me throughout my PhD; for their insights and expertise in technical matters, and for their unconditional support that has been crucial to keep me sane. My special thanks go to my colleagues from the "Vector Group", Ivan Ratković,

Milovan Đurić, Timothy Hayes, and Nikola Bežanić for the stimulating discussions, for the sleepless nights we were working together before deadlines, and for all the fun we have had in the last seven years. I acknowledge to my colleague Nikola Marković and my ex-flat mate Aleksandar Branković for opportunity to talk and discuss with them whenever I had an issue.

Last but not least, I would like to thank to my family and friends for supporting me during this endeavour. I especially thank my wife Suzana and daughter Nikolina for their unconditional love and care. You have been unimaginably patient waiting for this time to arrive. I also thank my parents and sister for their unconditional support during my PhD.

This thesis has been supported by the cooperation agreement between the Barcelona Supercomputing Center and Microsoft Research, by the Agency for Management of University and Research Grants - AGAUR (FI-DGR 2014), by the European Union (FEDER funds) under contract TIN2015-65316-P, and by the European Union's Seventh Framework Programme (FP7/2007- 2013) under the ParaDIME project (GA no. 318693) and the RoMol ERC Advanced Grant (GA no. 321253).

Contents

1 Introduction

In this chapter, we first introduce vector processors, present their historical overview and describe their advantages and limitations. We then discuss the motivation behind our work and present the objectives that we address in this thesis. Finally, we provide an overview of our contributions and thesis organization.

1.1 Vector Processors

Vector processors [6, 31] are known to be very energy efficient and yield high performance whenever there is enough data-level parallelism (DLP) [47]. They typically operate with vector registers that hold multiple values instead of single-value registers as in super-scalar processors. They provide vector instructions that operate on all the values of the registers. For example, a scalar addition instruction would take values from two scalar registers A and B, and produce a result stored in scalar register C, as Figure 1.1 (a) shows. A vector addition instruction would take two vectors A and B of vector length (VL) elements, and produce a resulting vector C of the same size, as in Figure 1.1 (b)

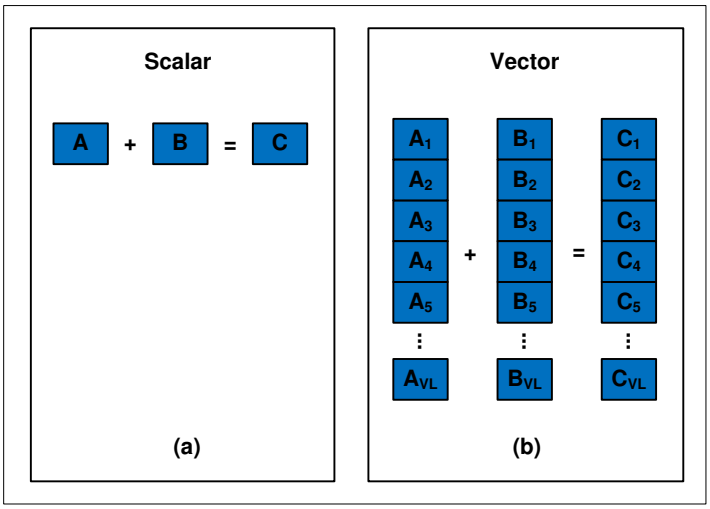


Figure 1.1: Comparison of a scalar instruction and a vector instruction.

1.1.1 Vector Processors in Supercomputers

Vector processors have a long and successful history in supercomputers where they are used for large scientific and engineering applications. The first vector architectures in early 70s were memory based with instructions that operate on memory-resident vectors [32, 84]. Cray [69], register-based vector machines were the first commercially successful supercomputers [20]. They provided arithmetic instructions that operate on vector registers, while separate vector load and store instructions move data between vector registers and memory. Several modest mini-vector supercomputers [18, 61] were released in the mid 80s.

The Japanese manufacturers, Fujitsu (VP50, VP100, VP200, VP400), Hitachi (S810) and NEC (SX) have been very successful is building vector processors for supercomputing [71].

Supercomputers that are built from commodity microprocessors have been dominant from the 1990s [23], but since then there have been still several vector supercomputers on the market. The Earth Simulator (ES) was a highly parallel vector supercomputer system based on NEC SX-6 architecture. It was the fastest supercomputer in the world from 2002 to 2004. Black Widow (Cray X2) [72] is a vector processing node for the Cray XT5h supercomputer launched in 2007. ES was replaced by the Earth Simulator 2 (ES2) in 2009, that is based on the NEC SX-9

architecture [85]. In 2013, NEC released SX-ACE vector supercomputers [51] and NEC roadmaps indicate that the successor to the SX-ACE will be released around 2017.

1.1.2 Vector Microprocessors

It has been proposed to adopt vector units in designs of microprocessors to support execution of vector instructions [7, 9, 24, 26, 41, 42, 44]. While early vector supercomputers were built from very expensive low-density ECL logic chips coupled with several thousand BiCMOS SRAM memory chips, vector microprocessor are built on a single high-density CMOS die. Torrent-0 [7], a vector microprocessor designed for multimedia, neural networks, and other digital signal processing tasks, and VIRAM [41], a scalable processor based on vector architecture and IRAM technology, are the first examples of vector microprocessors developed as part of academic research. CODE [42] is a successor of VIRAM that overcomes the limitations of conventional vector processors such as the complexity of a multiported centralized register file, the difficulty of implementing precise exceptions for vector instructions, and the high cost of on-chip vector memory system.

Espasa [20] showed that vector processors can improve their performance and hide latency by applying techniques such as decoupling, out-of-order execution and multithreading. Espasa et al. [24] developed Tarantula, a vector extension to the Alpha architecture.

The vector-thread (VT) [44] unifies the vector and multithreaded compute models and provides good performance for all-purpose computing. Maven [9] is successor of VT that explores a new approach to build data-parallel accelerators based on simplifying the instruction set, microarchitecture, and programming methodology for a VT architecture.

Virtual Vector Architecture (ViVA) [26] provides an effective and practical approach to hide latency by combining the memory semantics of vector computers with a software-controlled scratchpad memory.

1.1.3 Multimedia Extensions

Multimedia extensions, such as MAX [46], MMX [60], SSE [80], AVX [14], AltiVec [25], 3DNow! [58], etc., are very popular in desktop processors of all of the major vendors. They implement instructions that use subword parallelism to accelerate data-parallel applications. These multimedia extensions do not implement all traditional vector instructions and operate on much shorter vectors than in old vector architectures. Current trend is to increase width of single instruction multiple data (SIMD) registers. The width of SIMD arithmetic logic unit (ALU) units is equal to the width of SIMD registers. They also provide weaker memory units. Most of multimedia extensions do not support gather/scatter type memory operations as vector machines usually do and they have issues with alignment when access memory. Multimedia extension ISAs tend to be less general-purpose, less uniform, and more diversified [68].

Most recently, the Xeon Phi is a recent massively parallel x86 microprocessor designed by Intel and is based on the Larrabee [74] GPU that contains a 512-bit SIMD vector processing unit in each core.

1.1.4 Advantages and Limitations of Vector Processors

As it is emphasized in previous work [6, 31, 38], vector processors and vector Instruction Set Architectures (ISA)s have several advantages:

- A single vector instruction specifies N operations, where N represents tens or hundreds of operations. It dramatically reduces instruction fetch bandwidth, which is a bottleneck of conventional processors, particularly in terms of power consumption [59, 52].
- These N operations are independent. It allows simultaneously execution of all operations in an array of parallel functional units, or in a single very deeply pipelined functional unit, or in any intermediate configuration of parallel and pipelined functional units.
- Reduced control logic complexity. Hardware needs only to check for data hazards between two vector instructions once per vector operand, not once

for every element within the vectors. Therefore, the dependency checking logic required between two vector instructions is approximately the same as that required between two scalar instructions, but now many more elemental operations can be in flight and it will keep functional units much more time busy.

- Vector instructions that access memory have a known access pattern. A memory system can implement important optimizations if it has accurate information on the address stream. In particular, a stream of unit-stride accesses can be performed very efficiently using a large block transfer. Also vector memory instructions can amortize a high overall latency, because a single access is initiated for the entire vector rather than for a single word.

The vector processors have several well known limitations:

- Cost. Vector processors are specialized processors and work well with data that can be processed in highly parallel manner. If a workload does not contain a significant amount of DLP most of the time the vector unit will be idle. For example, ALU units that execute vector instructions will be idle if scalar code is executed.

- Vector Registers. Vector registers are fast memory but area expensive and power hungry structure [48]. They also have limited vector length.

1.1.5 Power and Energy Efficiency

Driven with the issue of power and energy consumption, researchers and industry have been proposed a lot of architectural level energy and power efficient optimization techniques for microprocessors [66]. These techniques (clock gating, power gating, gated-Vdd, etc.) help computer architects to optimize their low power designs.

Lemuet at al. [48] discussed the potential of energy efficiency of vector processors as accelerators for high performance computing systems. Lee at al. [47] confirmed that vector-based microarchitectures are more area and energy efficient than scalar-based microarchitectures, even for fairly irregular data-level parallelism. They

also explored a series of microarchitectural optimizations to improve performance, area, and energy efficiency of baseline vector cores.

Low-power techniques such as clock gating [49], power gating [35, 81], etc. could be combined with vector microarchitectures and possibly further reduce power consumption and increase energy efficiency. For example, if a vector processor implementation has four lanes maybe at some point of the execution two lanes are enough to ensure efficient execution and the remaining two can be powered off. The execution of vector instructions will make busy some parts of processors such as ALUs for a while and it can allow switching off inactive parts of the processor such as the instruction cache.

1.2 Motivation

Increasing performance has been traditionally the ultimate goal of processor designers since the first computers appeared. Technology scaling of CMOS [53] has provided exponential growth of transistor density for the last four decades. Combined with frequency scaling, it also provided improved performance of every next processor generations. Increased transistor density allowed for implementing on-chip implementation of solid computer architecture ideas such as out-of-order execution, pipelining, caching, multithreading, chip-multiprocessing, etc. that further improved performance.

Various forms of parallelism have been exploited in computer architecture to increase performance. The three major categories are: **instruction-level parallelism (ILP)**, **thread-level parallelism (TLP)** and **DLP**. ILP allows simultaneous execution of multiple instructions from one instruction stream (superscalar processors and out-of-order execution are examples of techniques that exploit ILP). TLP allows simultaneous execution of multiple instruction streams (simultaneous multi threading and multiprocessors are examples of techniques that exploit TLP). DLP allows simultaneous execution of the same operations on arrays of elements (multimedia extensions are an example of technique that exploits DLP).

In the last 15 years, power dissipation became the primary design concern for almost all computer systems. Processor designers realized that further technology

feature size scaling, would lead to higher power density and cooling such processor could became extremely difficult and very costly (since around 2005-2007 Dennard scaling appears to have broken down [12]). Power density limits have already impacted potential planned speed-ups by Moore's law, resulting in a slow down of CMOS scaling [79]. It has also caused that commodity microprocessors have ceased to increase frequency in each generation [78]. As technology feature size scaling goes further, power density is getting higher. Mike Muller, ARMs CTO, also claimed [5] that if device power budgets are kept at present levels, and the transistor density on chips increases in line with Moore's Law, mobile system designers could find themselves with computing power they cannot afford to use, unless innovative ways are found to circumvent the problem.

There are also many other reasons why processor designers should worry about power dissipation and energy consumption. Power and energy efficiency are equally important for low-end and high-end processors. For example, there is an increasing need data centers for huge amount of data storage and computational resources with the substantial associated power usage. While high-performance systems focus more about power dissipation than energy consumption, mobile systems are different. In battery operated, devices battery life becomes a major concern. Lowering the microprocessor energy as much as possible without spoiling performance is the main design goal. The need for green electronics is also a concern that has come consistently to the foreground in recent years [37].

Due to the above described facts, power and energy consumption are currently one of the most important issues faced by the computer architecture community. Driven with this goal, researchers have been trying to make better processor solutions by proposing new and improving existing architectures. Vector processors are an example of an energy efficient architecture.

Besides the long and successful history of vector processors in supercomputers, vector units have been proposed in microprocessor design [9, 24, 42]. Recent works on vector processors shows that they can be a good match even for workloads with complex and irregular DLP [47] or applications from others domains such as column-store data bases [30]. Knight Landing [76] is a recent, second generation of the Xeon Phi processor. It is massively parallel x86 microprocessor designed by Intel and based on the Larrabee [74] GPU that contains a 512-bit SIMD processing

unit in each core. Also, SIMD multimedia extensions [14, 80] are often included in modern microprocessors. While vector processors and SIMD extensions both exploit DLP, they differ in the way the data operand elements are handled at the execution stage. While SIMD extensions process all elements at once, vector processors execute elements in a pipelined fashion. Although vector processors are energy efficient, they still have high power and area overheads for low-end mobile processors. This is mostly due to the strictly power and area budget of embedded systems.

Before a vector microprocessor design is proposed it is very important to have detailed knowledge of the low-level characteristics of the vectorized code (e.g. degree of vectorization, distribution of vector lengths, etc.) as it is done in [6, 20, 64, 82]. In addition, it is also important to perform preliminary analysis of microarchitectural choices and implementation alternatives to narrow down the vast design space. Finally, it is common in computer architecture research to evaluate changes in the vector ISA that better fit the needs of the new workloads (for example, experimentation with new vector instructions or novel ways of vectorizing loops [17]). For these kind of studies, there is a need for tools to vectorize the application, characterize it, estimate performance of microarchitectural alternatives and experiment with new instructions. Actual vector processors, and associated vector compilers, could be used for the first two cases but they cannot be used to perform vector architecture exploration studies.

1.3 Thesis Objectives

Thesis objectives are listed bellow.

Integrated vector-scalar processing for low-end mobile processor. An important characteristic of microprocessor vector architectures is that a vector processing unit is designed as an separate unit or coprocessor to a scalar core that is used to run data-parallel applications. It requires additional hardware that is idle most of the time in scalar intensive applications. Power and area overheads of such additional hardware could be not acceptable for low-end mobile processors.

The main goal of the thesis is to propose an energy-efficient design of processor extension for the low-end mobile market that will satisfy the strictly power and

area budget of embedded processors and at the same time improves performance for applications with potential high DLP.

Development of tools for initial research on vector architectures. We consider that good research practice of proposing a new feature in computer architecture consists of several steps. They include obtaining detailed knowledge of the low-level characteristics of target applications, performing preliminary analysis of microarchitectural choices and implementation alternatives that narrows down the vast design space, and evaluation of changes in the ISA that better fit the needs of the target workloads.

Since there is no access to any tool that will allow for achieving previously mentioned steps, second goal of the thesis is to develop tools that will analyze vectorization characteristic of target workloads and perform preliminary design space exploration of vector microarchitecture alternatives.

Evaluation of existing Xeon Phi Knights Corner processor. Even though the Xeon Phi is designed for high-performance computing, Knights Corner has simple in-order core design. Deeper insight into Knights Corner capabilities during execution of vectorized applications could provide fruitful informations for future processor design.

Our research center has access to machines with Knights Corner processor and as complementary goal, another contribution of the thesis is to evaluate characteristics of the Knights Corner and use the acquired knowledge in the process of making design decisions for our low-end mobile extension.

1.4 Thesis Contributions

This thesis makes the following contributions:

- An energy-efficient integrated vector-scalar design on an in-order ARM core.
- Two tools that allow for rapid initial research on vector architectures.
- An evaluation of Xeon Phi capabilities that can be useful for processor design decisions.

Next we highlight the most important concepts of each contribution.

1.4.1 Integrated Vector-Scalar Design for an In-order Core

The main contribution of this thesis is a method to increase the performance of the low-power low-end embedded systems in an energy-efficient way. Energy efficiency is attained by modifying a scalar core to execute vector instructions on the existing infrastructure. In particular, the thesis proposes an integrated vector-scalar design that combines scalar and vector processing mostly using existing resources of an energy-efficient scalar processor (in our evaluation environment it is based on the ARM Cortex A7). In addition to a design that uses a conventional vector execution model, the thesis also contribute a novel block-based model of execution for vector computational instructions. Performance, power, area and energy evaluation results of this integrated design are also presented in the thesis.

Additionally, the thesis proposes an advanced integrated design which features three energy-performance efficient ideas: (1) chaining from the memory hierarchy, (2) direct result forwarding and (3) memory shape instructions. Two novel techniques that chain from the cache are proposed and implemented with the goal of further improving the performance of our integrated design. They can be applied to a conventional vector unit as well. We design and implement a novel result forwarding mechanism which complements the block-based execution and does not require writing to the vector register file. We design a vector memory unit with support for complex memory instructions including memory shape and scatter/-gather instructions.

1.4.2 Tools for Rapid Initial Research on Vector Microarchitectures

The thesis also contributes two tools, the vector architecture library (VALib) and the SimpleVector simulator: 1) VALib is a library that implements vector instructions and allows the rapid manual vectorization and characterization of applications, and 2) SimpleVector is a simple and very fast trace-based simulator which helps to estimate the performance of a vector processor. Both have been designed to be easily extended with new instructions or implementation alternatives.

The thesis includes use cases of the tools with six vectorized applications taken from distinct application domains. They illustrate how to perform the instruction-level characterization by using VALib or how to analyze the impact in performance of the cache hierarchy, functional unit configuration and memory decoupling in a vector processor.

This thesis also presents a case study to show how the tools can be used to add a new uncommon vector memory instruction to the ISA and then analyze the performance benefits with relative ease. In one interesting finding of our case study, we show that our emerging applications benefit from this new instruction which can provide up to 1.45x speedup.

1.4.3 Evaluation of Knights Corner Capabilities

The Knights Corner cores are in-order, which makes it very sensitive to cache misses. The thesis includes a study of the impact of prefetching on the performance of the Knights Corner and presents two observations that can be useful Knights Corner users or processor architects that aim to use in-order architecture: the initial results for vectorization can be misleading and prefetching is of maximum importance if data is not cache resident. We also prove that the combination of vectorization and parallelization is beneficial for workloads running on the Knights Corner.

1.5 Thesis Organization

The rest of the documents is organized as follows: Chapter 2 presents two tools for rapid initial research on vector vector microarchitectures.

Chapter 3 analyzes the capabilities of Knights Corner processor that consists of SIMD in-order cores.

Chapter 4 presents energy-efficient integrated design that allows for execution of vector computational instructions mostly reusing resources of an ARM in-order core. Three energy-performance efficient ideas are also presented in this Chapter and they additionally improves energy-efficiency or performance of the integrated design.

Chapter 5 concludes this thesis and points to future research directions.

2

VALib and SimpleVector: Tools for Rapid Initial Research on Vector Architectures

Vector architectures have been traditionally applied to the supercomputing domain with many successful incarnations [18, 51, 61, 69, 72, 85]. High performance of vector processors whenever there is enough DLP, as well as their applicability in other emerging domains (not just high-performance computing domain), encourage pursuing further research on vector architectures. However, there is a lack of appropriate tools to perform this research. Particularly in our case, we are missing tools to perform detailed low-level characterization of the vectorized code (e.g. degree of vectorization, distribution of vector lengths, instruction mix, etc.), to vectorize and analyze six applications with potential high DLP, to perform preliminary analysis of microarchitectural choices and implementation alternatives that narrow down the vast design space, and to evaluate changes in the vector ISA that better fit the needs of the new workloads (for example, experimentation with new vector instructions).

Our goal in this chapter is to develop tools that will allow rapid initial research on vector architectures, characterize at instruction level target applications, evaluate several alternative properties of the vector microarchitecture, and experiment with new or uncommon vector instructions. In summary, the main contributions of this chapter are:

- Two tools for measuring and analyzing an application's suitability for vector microarchitectures. The first tool is VALib, a library that enables hand-crafted

vectorization of applications and its main purpose is to collect data for detailed instruction level characterization and to generate input traces for the second tool. The second tool is SimpleVector, a fast trace-driven simulator that is used to estimate the execution time of a vectorized application on a candidate vector microarchitecture.

- Evaluation of six vectorized applications taken from distinct application domains. It includes the instruction-level characterization performed by using VALib and analysis of the impact in performance of the cache hierarchy, functional unit configuration and memory decoupling in a vector processor. The results show high degrees of vectorization for these applications, ranging from 91% to 63.2%. An application is dominated by short vectors of 16 elements but the rest have much longer average vector lengths, ranging from 31 to 59.9, for a maximum vector length of 64 elements. Memory decoupling provides in average for all applications 16% improvement in the execution time using the maximum vector length of 128 elements with three functional units and three load/store units.

- A case study to show how the tools can be used to add a new uncommon vector memory instruction to the ISA and then analyze the performance benefits with relative ease. In one interesting finding of our case study, we show that our emerging applications benefit from this new instruction which can provide up to 1.45x speedup.

The rest of this chapter is organized as follows: Section 2.1 introduces VALib, Section 2.2 presents the characterization of six applications vectorized using VALib and Section 2.3 discusses alternatives to the use of VALib. Section 2.4 describes SimpleVector. Results for different vector microarchitectures are presented in Section 2.5. Section 2.6 shows a case study of adding a new vector instruction. Section 2.7 discusses related work. Finally, Section 2.8 presents the summary of this chapter.

2.1 VALib

VALib offers a vector instruction set to the programmer and emulates the execution of the instructions. Target applications are vectorized by hand by adding calls to VALib similarly to programming using intrinsics. This enables executing vectorized applications in the absence of a vector processor which in turn allows for initial research on their applicability for different vector architecture design options. The main purpose of VALib is to collect data for detailed instruction level characterization of vectorized applications and to generate input traces for SimpleVector. VALib is not bound to any specific vector ISA and can be easily extended to offer new instructions. This section introduces the features and implementation details of VALib.

2.1.1 Vector ISA

The default ISA offered is inspired by classic register-based vector machines, e.g. CRAY and CONVEX [2, 18, 31] with some additional uncommon instructions (see 2.6) that are useful or required to vectorize our target applications. They can be grouped into the following classes:

Arithmetic and logical. These instructions are common operations such as addition, multiplication, logical bitwise operations, etc. They can operate in vector-vector or vector-scalar mode. Some instructions support masking [75].

Memory. There are instructions for unit-stride, strided and indexed access. Some variants support masking, which is useful to access elements conditionally. The library also implements shape instructions for 2D strided accesses, based on those introduced by RSVP [15]. See section 2.6.

Reduction. This class includes *sum*, *max* and *min*.

Bit/element manipulation. Examples of this class include instructions to read or write individual elements of a vector register, or to compress a register according to a vector mask.

VALib implements a vector register file and a vector mask register file. The number of registers and the number of elements per vector register are parameterizable. This allows us to experiment with different maximum vector length

(MVL). There is also vector length register and corresponding instructions to read and write it. Several data types are supported: signed or unsigned integer (16, 32 and 64 bits), single and double precision floating-point (FP) (32 and 64 bits) or char (8 bits).

2.1.2 API

VALib offers the following types of function described below and provides wrapper functions in C, C++ and FORTRAN for all the functions of the library.

Instruction functions are the core of VALib and each one implements a vector instruction from the ISA described above. The function updates the state according to the semantics of the instruction it implements. State includes memory and the vector register file. The identifiers of the source and destination vector register are input parameters of the function.

Annotation functions can be optionally used to provide additional information useful for characterization. For example, we use them to indicate the presence of blocks of scalar instructions or that a vectorized loop begins or ends. Such information is used by VALib and SimpleVector to estimate the amount of executed scalar instructions.

Initialization sets up the internal structures of the library such as the vector register file and clears all statistics.

Finalization performs the opposite task. It closes the opened files, dumps the statistics and frees all allocated memory and internal objects.

2.1.3 Results and Statistics

A main objective of VALib is to generate statistics and characterize the vectorized applications. It collects the following statistics:

Percentage of vectorized code provides the degree of vectorization of an application; it is calculated using the number of vector operations and the number of scalar instructions executed.

Vector instruction mix shows the distribution of the vector instructions; the number of instructions of each type executed and the number of operations per instruction.

Distribution of vector lengths indicates how many instructions used a given vector length and helps to determine the efficient utilization of the vector registers.

Histogram of strides helps to determine the dominant memory access patterns.

2.1.4 Instruction and address traces

VALib can generate traces of the vector instructions executed as well as traces of the addresses accessed by memory vector instructions. SimpleVector uses the traces as inputs to estimate the execution time of the vectorized application (see Section 2.4). The instruction trace encodes the vector instructions and the number of scalar instructions between two vector instructions.

VALib supports two ways of counting scalar instructions, with performance PAPI counters [54] and with annotation functions. The first approach is more precise, but the second is much faster. Reasons for precise scalar info are following: (i) we need to calculate percentage of vectorization for an application, and (ii) sometimes there are still scalar instructions inside vectorized code (e.g. scalar code inside a loop that performs strip-mining) and their precise count is important to estimate the execution time in SimpleVector. However, PAPI function calls add additional overhead that slows down an application. Calls to PAPI functions also require additional scalar instructions, therefore we need to calibrate PAPI counters (determine number of scalar instructions associated with calls to PAPI functions). The annotation functions just indicate the approximate size of the scalar block. When counters are used, the wrappers include the appropriate code to disable and enable counting before and after the call to VALib.

2.1.5 Extensibility

VALib has been designed with extensibility in mind. It is simple to include new instructions. Templates and auxiliary classes are used to implement versions of each instruction for all supported data types automatically. The wrappers for the different data types are created automatically once a new instruction is included in VALib. It is also simple to incorporate new statistics and traces.

```
for (i = 0; i < veclen; i++) {
    diff1 = x[i] - m1[i];
    dval1 -= diff1 * diff1 * v1[i];
}
```

(a)

```
LDV R3, x              // load vector x
LDV R1, m1             // load vector m1
SUBV R4, R3, R1        // vector-vector subtraction
MULV R5, R4, R4        // vector multiplication
LDV R2, v1             // load vector v1
MULV R5, R5, R2        // vector multiplication
VREDADD R5, temp       // vector reduction
dval1 =- temp
```

(b)

```
MVL = get_mvl();
set_vl(MVL);
for (k=0; k < veclen; k+=MVL) {
    if ((k + MVL) > veclen) {
        MVL = veclen – k; set_vl(MVL);
    }
    ldv_fl(R3, x+k);
    ldv_fl(R1, (m1+k));
    subv_fl_fl_fl(R4, R3, R1);
    mulv_db_fl_fl(R5, R4, R4);
    ldv_fl(R2, (v1+k));
    mulv_db_db_fl(R5, R5, R2);
    vredadd_db(&temp_db, R5);
    dval1 -= temp_db;
}
```

(c)

Figure 2.1: Example of vector library usage: a) source code of kernel, b) vectorized pseudo-code, c) vectorized code using VALib.

For example, we have extended VALib to generate a trace of all operands of addition/subtraction vector instructions. The traces, coupled with timing information generated by SimpleVector, were used as input to a circuit-level simulator to measure performance and estimate power of a vector addition unit [67].

2.1.6 Example of Vector Library Usage

The following example shows how to use VALib to vectorize a kernel. The code in the Figure 2.1.a represents one part of a kernel from Sphinx3. It consists of one loop which computes a scalar from three input arrays (*x*, *m1* and *v1*).

The pseudo-code in the Figure 2.1.b shows how the example can be vectorized. Vectors are accessed using vector memory load instructions from VALib and then computation is performed using vector subtraction and multiply functions. Finally, a reduction instruction generates the scalar.

Once we have the pseudo-code of the vectorized kernel it is straightforward to replace this code with function calls from VALib (Figure 2.1.c). This resembles the way SSE/AVX intrinsics are used to manually SIMDize code. Function names are composed of two parts. The first one indicates the opcode of the vector instruction (e.g. ldv, subv, etc.) and the second indicates the data type or types used by the instruction. Since strip-mining is typically applied, we also did it in our example (the loop and first three rows in the loop in the Figure 2.1.c).

2.2 Characterization of Vectorized Applications

This section presents the instruction-level characterization that is possible to perform with VALib on six applications. First, the methodology applied to vectorize the applications is described and then the results of the characterization are presented.

Since the goal of our tools is to evaluate differences between vector designs (not to measure speed-up over scalar) we chose applications with high potential DLP, as subsection 2.2.2 shows. The applications are described in Table 2.1. Four applications are from SPEC benchmark suites. Sphinx3, H264ref and Hmmer are from the SPEC2006 benchmark suite. Sphinx3 performs speech recognition, H264ref does video coding while Hmmer feature hidden markov models which are used in machine learning. Facerec is from the SPEC2000 benchmark suite. Even though SPEC benchmarks are typically used to evaluate general purpose processors, applications such as speech recognition or face recognition are widely used in mobiles. The Graph500 [55, 56] benchmark is a data intensive, high performance graph processing application but we choose this application because it is highly cache unfriendly and it is a good example to evaluate our ideas for cache unfriendly scenarios. ECLAT [57] is an application from the data-mining realm. In particular, it implements a known algorithm for frequent itemset mining.

Table 2.1: Vectorized applications.

Application	Description
FaceRec	Face recognition system.
Sphinx3	Speech recognition system.
H264ref	Implementation of H.264 /AVC.
Hmmer	Search a gene sequence database.
ECLAT	Frequent itemset mining.
Graph500	BFS for undirected graphs.

2.2.1 Methodology

The methodology that we used during the process of vectorization has the following steps:

Profiling is performed to identify the functions that dominate the application's execution time. We used gprof and Intel's VTune profilers.

Kernel analysis is performed for kernels that consume most of the execution time. They are examined for vectorization (e.g. does the kernel contain loops, what kinds of dependencies appear, what is the size of loop, etc.). The kernels that are successfully identified as vectorizable are selected for the next step.

Kernel vectorization is done by writing pseudo-code and inserting calls to VALib. The actual vector length required by an algorithm (the number of iterations of a loop) is usually larger than the maximum vector length (MVL) supported by the architecture. Also, we want to experiment with several MVLs. Furthermore, the actual vector length is often unknown at compile time. Strip-mining is applied in all vectorized kernels that have a number of loop iterations greater than MVL or when it is unknown/variable (as vectorized example shows in the Figure 2.1.c). It also allows changing the MVL without modifying the vectorized source code.

Result analysis is performed after the vectorized application is run and its statistics are collected. The results are analyzed using the instruction-level characterization described below.

Table 2.2: Instruction-level characterization.

Application	Scalar instructions	Vector instructions	Vector operations	Percentage of vectorization	Average VL
FaceRec	2.1×10^{10}	2.4×10^{9}	9.4×10^{10}	81.8	38.7
Sphinx3	3.6×10^{11}	3.7×10^{10}	1.7×10^{12}	82.5	46.0
ECLAT	1.7×10^{8}	3.8×10^{7}	1.7×10^{9}	91.1	59.1
Hmmer-I	2.2×10^{11}	8.5×10^{9}	5.1×10^{11}	70.1	59.9
Hmmer-II	5.2×10^{11}	2.1×10^{10}	1.1×10^{12}	66.8	49.8
H264ref-I	1.5×10^{11}	1.6×10^{10}	2.5×10^{11}	63.2	15.3
H264ref-II	1×10^{11}	9.5×10^{9}	2.1×10^{11}	67.7	22.2
H264ref-III	8.9×10^{10}	7.7×10^{11}	2.1×10^{12}	73.2	23.4
Graph500	1.1×10^{10}	3×10^{9}	9.5×10^{10}	89.4	31.1

2.2.2 Instruction-Level Characterization

This section illustrates the kind of detailed characterization at the instruction level that can be performed using VALib.

Degree of Vectorization

Table 2.2 presents some statistics for vectorized applications described in Section 2.1.3. The first column lists the applications. *Hmmer* and *H264ref* are presented with multiple data sets. The next two columns contain the total number of executed scalar and vector instructions. The next column presents the number of operations performed by the vector instructions. A scalar instruction performs a single operation while a vector instruction performs a number of operations determined by the value of the vector length register. The fifth column is the percentage of vectorization of each application, defined as the ratio of vector operations and total operations (scalar instructions plus vector operations) [20]. The last column presents the average vector length (AVL) (vector operations divided by vector instructions) with an MVL of 64.

The first interesting point from Table 2.2 is the degree of vectorization which shows if a vector processor is a suitable choice for the application. All our appli-

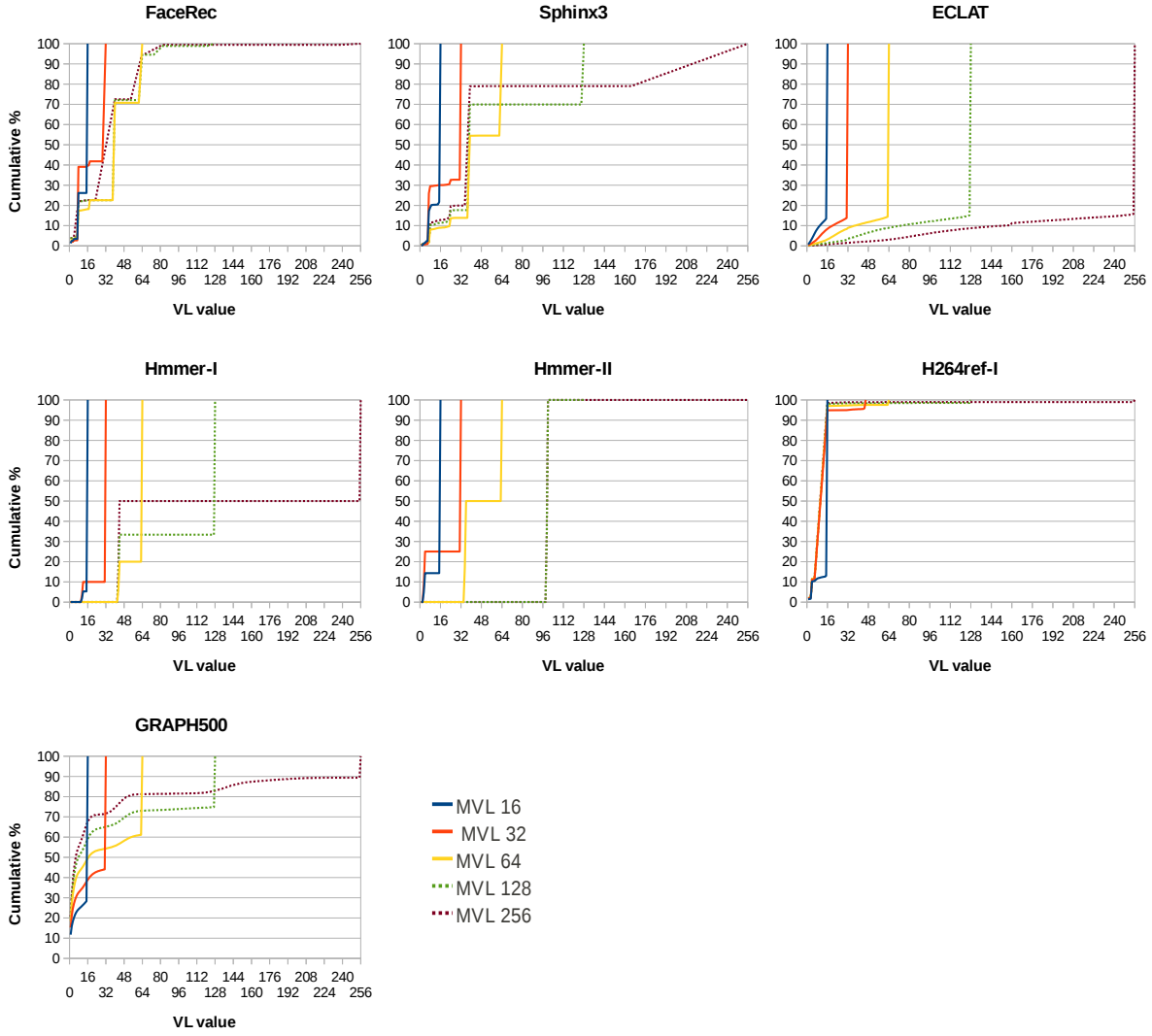


Figure 2.2: Distribution of vector lengths. X-axis represents the VL and Y-axis is a cumulative %.

cations have a high degree of vectorization and it is an expected result from the selected applications, that have a lot of potential DLP. It ranges from 62.90% for *H264ref* up to 91.06% for *ECLAT*. It is important to mention that we just vectorized the most executed kernels and these numbers could be improved by vectorizing other kernels in the applications. The degree of vectorization also depends on the input data set. We can observe that *Hmmer* has different degree of vectorization for its two different input data sets, 70.08% and 66.76%. We investigated *Hmmer* and found that the most executed loop has only 100 iterations in *Hmmer-I* while 300 in *Hmmer-II*.

Vector Lengths

The actual vector length used by vector instructions is an important characteristic of vectorized applications and one convenient way to present it is the AVL. The AVL observed in the vectorized applications is presented in Table 2.2. Even though these applications are highly vectorizable, their AVLs vary considerably. Only *Hmmer* and *ECLAT* have AVLs very close to the MVL. All other applications also have relatively long AVLs except *H264ref* and *Graph500*. *H264ref* has a short AVL, and in *Graph500*, the AVL is half of the MVL.

While the AVL is important when long memory latencies are considered, the effective usage of vector registers is important in achieving high performance. For this reason, we further analyze vector lengths and study the effective usage of vector registers with different MVLs. Vector length distribution is crucial to understand the interaction between latencies and performance. The MVL is a key design decision of a vector unit and this statistic shows the actual vector length used by the applications. In Figure 2.2, we plot the cumulative percentage of vector instructions executed with each vector length. The X-axis plots the vector length value and the Y-axis plots the cumulative percentage of instructions that have used a given vector length. Each line corresponds to a different MVL. For example, for *Sphinx3* we can see that for an MVL of 64 about 15% of all vector instructions were executed with a vector length that was lower than 39, about 40% were executed with vector length 39, and 45% were executed using the MVL (64).

Our applications present a very wide range of behavior with respect to the vector length distribution. Most instructions in *ECLAT* use the MVL or a close value for any MVL, due to a very long loop. *H264ref* has a dominant vector length of 16. The distribution of vector lengths in *Hmmer-I* and *Hmmer-II* shows a step in many cases that is determined by the number of iterations of the vectorized kernel (i.e. 300 and 100, respectively). For example, when the MVL is 256, *Hmmer-I* executes instructions with vector lengths 256 and 44. *FaceRec* and *Sphinx3* have a distribution that follows a staircase with several dominant vector lengths. The number of instructions that use the MVL is higher for shorter MVLs. *Graph500* executes instructions with all possible vector lengths from one to the MVL and many instructions have a very short length, which lowers the AVL.

This data shows that the utilization of vector registers depends both on the application and the input data set. While some of the applications almost always use the whole vector register, the other could also run at a similar performance using shorter MVLs (*H264ref*). Results for *Hmmer* and *H264ref* show how the input data sets used to run the applications have an impact in the vector lengths used. Moreover, the way these kernels are vectorized has a lot of influence in the usage of vector registers as is shown in Section 2.6. VALib is useful to easily study how larger MVLs could be used by vectorized applications.

It can also be interesting to know what would happen if the ISA offered an infinite MVL. The annotation functions of VALib can be used to perform such a study by counting the number of iterations in the vectorized loops. The most interesting results are that *Graph500* and *ECLAT* could use longer MVLs because their lengths are much longer than 256 elements.

Vector Instruction Mix

The instruction mix indicates which vector instructions types are executed in the vectorized applications. We can determine the most executed instructions, as well as the ratio between memory and computation instructions. This helps to identify potential bottlenecks. Table 2.3 shows the distribution of four categories of vector instructions: arithmetic and logical, memory, reduction, and bit/element manipulation instructions. The vector library counts the number of executions for each

Table 2.3: Instruction mix.

Application	Vector instructions	Instructions %			
		Arithmetic & Logical	Memory	Reduction	Bit & Element
FaceRec	2.4×10^9	41.0	34.4	13.1	11.5
Sphinx3	3.7×10^{10}	46.5	37.8	11.3	4.3
ECLAT	3.8×10^7	7.3	21.9	0.0	70.8
Hmmer	8.5×10^9	39.5	42.1	2.6	15.8
H264ref-I	1.6×10^{10}	31.8	48.8	13.4	6.0
H264ref-II	9.3×10^9	26.6	57.0	11.2	5.1
H264ref-III	7.7×10^{11}	26.3	56.0	10.8	6.9
Graph500	3×10^9	16.7	58.4	0.0	24.9

instruction individually. We grouped them here to simplify the table and provide a general view of the applications. The library also counts the operations per instruction that aren't included here because the overall numbers are very similar.

We can see that 35% to 58% of all vector instructions are of the memory type, except for *ECLAT*, in which this percentage is lower, about 22%. Memory instructions are the most executed type in *Hmmer*, *H264ref* and *Graph500*. These results stress the importance of the memory system. The characteristics of these instructions are further studied in the next section. In the following chapters we pay special attention to the memory unit/instructions. The dominant computation category are arithmetic and logical instructions, except for *ECLAT* and *Graph500* where bit/element manipulation instructions represent 70.8% and 24.9% of the vector instructions respectively. Reduction instructions are significant in *Sphinx3*, *FaceRec*, *H264ref-I* and *H264ref-II* applications with the range between 10% and 15%. The table shows the results just for one input data set of *Hmmer* because we observed that the distribution is the same for the different input data sets.

Memory access patterns

Stride distribution shows which memory access patterns are used by the applications. This helps to identify which accesses should be optimized in the memory

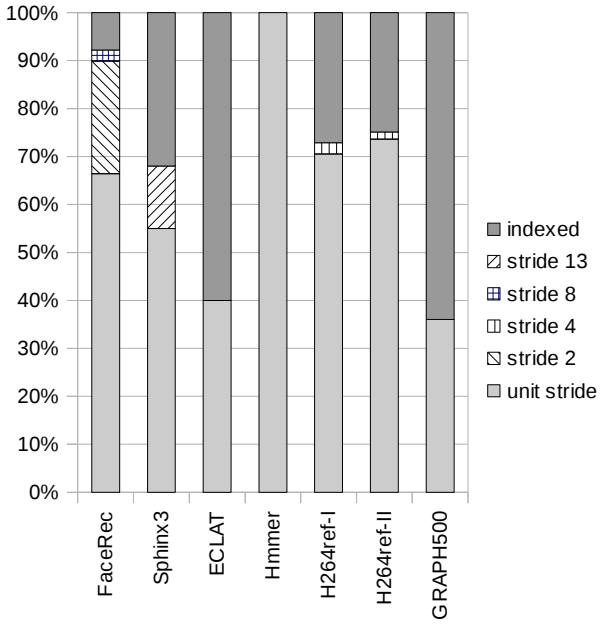


Figure 2.3: Distribution of memory access patterns.

system, a critical part of any vector machine. Figure 2.3 presents the distribution of memory access patterns, including unit-stride, stride *n* and indexed. *HM* executes almost only unit-stride instructions. The vector processor can take advantage of the spatial locality and the burst nature of unit-stride accesses and implement a faster support for this instruction type, retrieving a whole cache line per access. On the contrary, the memory easily becomes the bottleneck for applications that use strides greater than one or indexed memory accesses. *FaceRec* and *Sphinx3* have a relevant percentage of strided memory accesses. Index memory instructions are dominant in *ECLAT* and *Graph500* and are very important for *Sphinx3* and *H264ref*.

2.3 Possible Alternatives to VALib

One important initial motivation to create VALib was the lack of an available vector machine native compiler. Compilers for vector architectures have been very successful at vectorizing code automatically. However, vector compilers are bound to a specific vector ISA and machine and are available only to users of the system. Another problem is that they are not open source; for this reason, they lack the flex-

ibility to extend them to support new instructions, a common study in computer architecture research. On the contrary, VALib does not support auto-vectorization capabilities, and it requires vectorization by hand, but it is flexible to extend the implemented vector ISA with new instructions (see Section 2.6) and it is not bound to any specific vector ISA.

An alternative would be to use a tool such as the Intel SPMD Programing Compiler (ispc) [62] as an alternative to avoid vectorization by hand. The ispc is an open-source compiler that provides a high-level programming interface for programming SIMD in an OpenMP-like style. The compiler can generate C++ code with a set of generic SIMD intrinsics, which can then be mapped to platform specific SIMD intrinsics (that implements the particular generic SIMD intrinsic interface). Another option is to extend an existing compiler such as gcc or LLVM [45] with a vector back-end and use its existing auto-vectorization capabilities or to employ auto-vectorization capabilities of a source-to-source compiler and implement a backend that inserts calls to VALib. However, ispc or gcc is intended for generating code that uses SIMD instructions. Although SIMD and vector have similarities, they have important differences that would require modifying the ispc substantially to adapt it to a vector ISA. For example, SIMD instructions operate with a fixed vector length while vector architectures have a variable vector length. Another issue is that it may not be trivial to extend the compiler to utilize the new instructions and to extend the compiler optimizations to account for the new instructions. For this reason, we would need first to establish the usefulness of the new instructions with VALib to ensure that it is worth the effort to modify the compiler.

Auto-vectorization capabilities cannot always capture the DLP available. For this reason, typically intrinsics are employed to use SIMD instructions effectively for highly optimized code. We consider that hand vectorization with VALib is similar to coding using intrinsics. The main difference is that existing intrinsics do not implement a vector ISA but SIMD extensions, so they cannot be used for research on vector architectures.

Recently, we have managed to connect our VALib with Mercurium compiler [8]. Mercurium is able to vectorize part of a code that is annotated with pragma directives. Instead of calling SIMD intrinsics we managed to insert call to functions from our VALib. It means that the compiler is able to automatically vectorize a code

using our VALib. For example, a loop below is annotated with pragma directive *simd*.

```
#pragma simd
for (i = 0; i < LENGTH; i++) {
    z[i] = x[i] + y[i];
}
```

After compiling the loop, Mercurium compiler generates the following code:

```
valib_setvl(64);
for (i = 0; i <= -64 + n; i = 64 + i) {
    valib_ld_int32(VR0, &b[i]);
    valib_ld_int32(VR1, &c[i]);
    valib_add_int32_int32_int32(VR2, VR0, VR1);
    valib_st_int32(VR2, &a[i]);
}
valib_ld_int32(VR0, _vliteral_0);
valib_set_int32(VR1, n - i);
valib_le_int32_int32(MR0, VR0, VR1);
if (!valib_mask_is_zero(MR0)) {
    valib_ldm_int32(VR0, &b[i], MR0);
    valib_ldm_int32(VR1, &c[i], MR0);
    valib_addm_int32_int32_int32(VR2, VR0, VR1, MR0);
    valib_stm_int32(VR2, &a[i], MR0);
}
```

The loop is vectorized using corresponding functions from our VALib. In this particular example, the remaining part of the loop (epilogue) is vectorized using masked vector instructions. Mercurium compiler also supports approach which sets new vector length for the epilogue (like in Figure 2.1).

2.4 SimpleVector

SimpleVector is a simple trace-driven simulator for vector processors based on a generic vector processor, such as the described by Hennessy and Patterson [31]. The model consists of a component that models the microarchitecture of the desired vector processor, a memory hierarchy and methods that apply chaining and

other implementation specific features. The model uses an instruction trace, optionally an address trace and the Instruction Per Cycle (IPC) metric of scalar code as inputs to estimate execution time of the vectorized application. The traces are generated by VALib (see section 2.1.4). SimpleVector generates detailed statistics of the resource usage. These statistics help us to better understand the obtained results and the behavior of the vectorized applications.

Detailed microarchitectural simulators are a very common way to evaluate performance. Although it can be accurate this method is time-consuming both to create the simulator and to run the simulations. Our goal was to develop a tool that provides results fast and allows a preliminary evaluation and an early parameter exploration. In chapter 4, we build a simulator for a specific vector microarchitecture.

2.4.1 Simulated Microarchitecture

The structure of SimpleVector is presented in Figure 2.4. It takes as input two files, the instruction trace file and the memory address trace file. It models functional units (FU), vector load/store units (LD/ST), a vector register file, a memory hierarchy, a fetch/decode unit and an execution unit. The fetch/decode unit loads instructions from the instruction trace. The execution unit checks for available destination and source registers as well as a FU or LD/ST unit. After that it computes instruction's start and end of execution, and updates states of units that are affected by execution of that instruction. For memory instruction, the execution unit generates accesses to the memory hierarchy using the memory address trace.

SimpleVector is parameterizable in several ways. It can model in-order and decoupled execution. The number of FUs and LD/ST units, as well as the number of vector lanes, are parameters. Each unit is parameterizable too, including: 1) the number and length of vector registers for the vector register file, 2) instruction and data types supported by each FU (e.g. FP multiply, logical, etc.), 3) memory instruction types supported by each vector memory unit and 4) latencies. The parameters can be provided either at compile or run time.

As mentioned above, SimpleVector simulates a memory hierarchy. Even though class vector supercomputers accessed main memory directly, it has been shown that

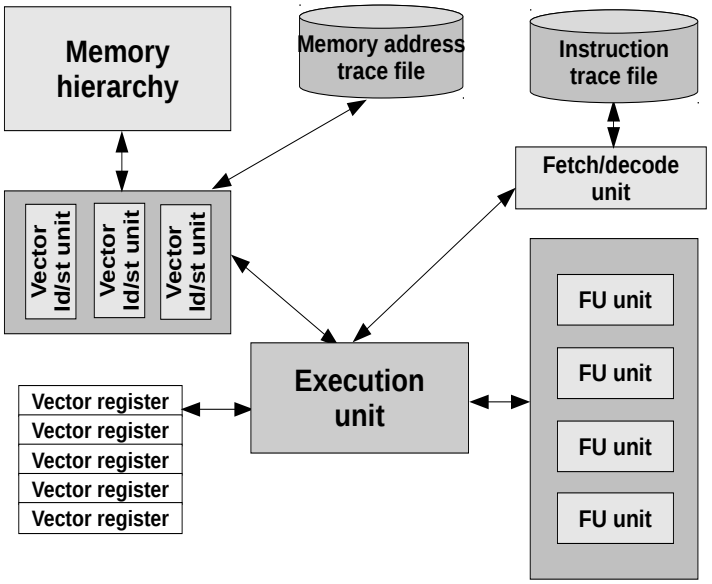


Figure 2.4: The basic structure of SimpleVector.

vector processors benefit from accessing caches [24, 40, 65] and we expect any future vector processor to access a memory hierarchy. A basic cache simulator based on a modified memory model of SimpleScalar [13] is attached to SimpleVector. Access to main memory is simulated in a more accurate way than in SimpleScalar. We model a bus and therefore have limited bandwidth with a fixed memory latency. On top of that, we use DRAMsim [83] to calculate a representative accurate memory latency. The cache simulator takes as input the address trace generated by VALib. For each vector memory instruction, the address trace contains all necessary information to generate the addresses accessed. For unit-stride memory instructions, we can load/store a whole L1 cache line with only one access, while for strided, indexed and shape memory instructions, only one element per access is loaded/stored.

Currently SimpleVector models the execution unit for in-order and decoupled vector architectures. In the in-order model, all instructions are executed in program order. Decoupled vector architectures [22] improve in-order by separating the execution of computation instructions and data movement, and allowing memory instructions to be executed ahead of computation instructions. For both in-order and decoupled architecture, SimpleVector implements several techniques commonly

found in vector processors [31] such as multilaning, pipelined instruction start-up (dead time or recovery time) and chaining for arithmetic vector instructions. SimpleVector does not support chaining from the memory hierarchy. In chapter 4, we propose techniques that allow for chaining from the memory hierarchy with their detailed description and evaluation.

All the parameters mentioned above help us to analyze a broad range of different configurations of vector processors, from very simple vector processors with only one lane, one vector load/store unit and a small number of functional units, to more complex vector processor architectures with multiple lanes and a rich set of load/store and functional units as shown in Section 2.5.

2.4.2 Extensibility

SimpleVector has been also designed with extensibility in mind. For example if an instruction is added to VALib, SimpleVector will automatically decode it correctly from the instruction trace because VALib and SimpleVector share files where the new instruction is defined. Only the semantics of the new instruction has to be implemented in SimpleVector. By semantics we mean the specification of how the instruction uses the resources of the simulated processor (e.g. when are FU used, registers accessed or memory requests initiated).

In order to implement a new vector processor model, the already defined vector instruction functionality can be reused as well as structures such as the vector register file, FU units, etc. Only the new execution model of vector instructions has to be implemented.

2.4.3 Accuracy Testing

In order to increase our confidence in the correctness of the results of SimpleVector, we decided to compare the output of SimpleVector with known and predictable results. We used as reference published results of Cray-1 M [71] for three microbenchmarks. We chose microbenchmarks because we do not have access to a existing vector machine and we cannot use published results because many variables are unknown. The first microbenchmark adds two vectors and stores the

Table 2.4: Execution cycles for three microbenchmarks.

Benchmark	SimpleVector			Cray-1 M		
ID	10	100	1000	10	100	1000
First	56	352	3442	121	416	3508
Second	60	356	3471	147	444	3563
Third	70	460	4480	116	508	4531

result to a third one, the second microbenchmark performs vector-scalar multiplication and then vector-vector addition while the third performs vector-vector multiplication and then vector-vector addition. We configure SimpleVector as close as Cray-1 M as possible, following the Cray-1 M hardware reference manual [1].

Table 2.4 presents results for SimpleVector and Cray-1 M. The results are presented in cycles and three different vector lengths (10, 100 and 1,000) are used in all microbenchmarks. The MVL of Cray-1 is 64 elements. The differences in the results of SimpleVector and Cray-1 M are small. For example, in the first microbenchmark the difference between our results and CRAY-1 M is around 65 cycles for all different lengths. Even though this is large for short vector lengths, for typical vector array lengths the difference is small, e.g. 2.7% for 1,000 elements. Since this difference is constant across different array lengths, it seems that some undocumented kernel initialization is the likely source of the discrepancy. We did not take this into account in our experiments. The results for the other two microbenchmarks show similar trends.

2.5 Evaluation of Microarchitectural Alternatives

In this section, we evaluate several alternatives of the microarchitecture by using SimpleVector. In particular, we study the memory hierarchy and compare in-order (IO) and decoupled (DC) architectures. The vectorized applications presented in Section 2.2 are used to perform the evaluation.

As it is well known, long vectors are useful to hide the memory latency. We want to study the behavior of the set of vectorized applications using different cache configurations and MVLs.

Table 2.5: Cache hierarchy configurations.

#	C2K	C16K	C64K	C128K	C1M
L1 cache	2KB	16KB	64KB	128KB	1MB
L2 cache	32KB	256KB	1MB	2MB	16MB

We also evaluate two different vector architectures: IO and DC. These two architectures are interesting alternatives. IO processors are overwhelmingly used in the embedded domain because they are more energy-efficient compared to bulky out-of-order architectures, while DC architecture is an efficient way to improve performance without the complexity of out-of-order execution. It is also not trivial to determine the best configuration for the functional units. We want to see what is the impact on execution time for the set of vectorized applications when using different configurations of functional units for both IO and DC.

2.5.1 Memory Hierarchy

In the experiments of this section, we use five different configurations of the cache hierarchy presented in Table 2.5. They are ranging from small caches (L1 2 KB, L2 32 KB) to larger caches (L1 1 MB, L2 16 MB). The L2 cache is 16 times larger than L1 in all configurations.

We performed an analysis using four different L1 hit latencies (1, 2, 4 and 8 cycles) and three different L2 hit latencies (12, 16 and 20 cycles) before we fixed their values. We run experiments using all possible combinations of hit latencies and the *C64K* configuration for cache sizes. Results show that the difference in execution time is less than 5% for all applications, except for *H265ref-I* when L1 hit latency is 8. In this case, the execution time is around 15% worse than the best configuration. We chose 4 and 12 cycles for L1 and L2 hit latencies respectively since they are common in modern microprocessors [3]. All caches are 4-way set associative and have 64-byte lines.

Two FU and two memory units are used in all experiments of this section. The FU support all non-memory vector instructions and the memory units support all memory vector instructions. Instruction and address traces with MVLs of 32, 64,

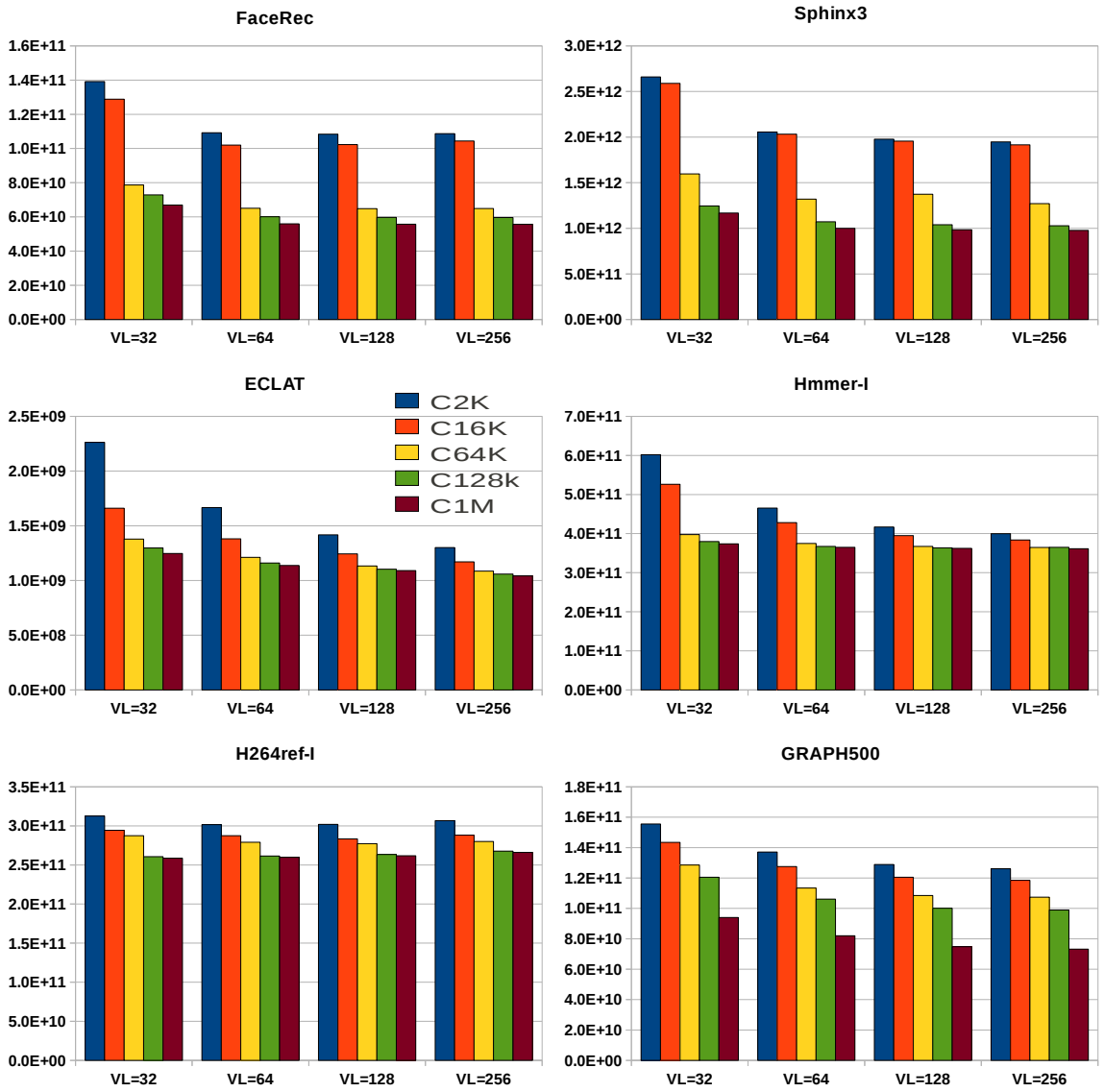


Figure 2.5: Execution time for different MVLs and configurations of cache hierarchies.

128 and 256 are used in the experiments. All experiments in this section were performed using the IO vector architecture. Performance counters are used to count the scalar instructions. Average memory access time is obtained using trace-based DRAMsim for each application and for all configurations of the cache hierarchy and MVLs separately. The measured average access time is used as the main memory latency in SimpleVector. Figure 2.5 presents the execution time in cycles of the applications using different cache sizes and MVLs. A lower number means a better result.

The first and expected observation is that we have better execution time for longer MVLs for smaller cache configurations except *H2-I* which has almost the same execution time for all MVLs because most instructions use a vector length around 16 as Figure 2.2 shows. In *FaceRec*, the execution time is almost the same for all MVLs greater or equal to 64. The reason is the small number of instructions that are executed with vector lengths larger than 64 (see Figure 2.2). The very high hit rates for larger cache configurations explain why we have almost the same execution time in *Hmmer-I* for different MVLs.

The second observation is that we have better execution time for larger caches for all MVLs. One of the main benefits of long MVLs is that it is easier to hide memory latency. However, when data set fits in the cache, memory latency is short, so long MVLs add little performance improvement. The best execution time is obtained for *C1M* in all cases. However, it is interesting to notice that there is very small difference between *C128K* and *C1M*, except for *Graph500* which has a large data set. For the rest of applications, the data set fits in the *C128K* configuration with an L2 cache of 2 MB, and there is no need for larger caches. Moreover, for some applications (*ECLAT* and *Hmmer-I*), caches larger than C64K are not useful.

A preliminary conclusion from these experiments with the cache hierarchy is that a *C128K* cache configuration is enough to provide sustainable performance for all applications for MVLs larger or equal to 64 elements. Longer MVLs also provide better performance for smaller caches *if* the application can exploit longer MVLs (*Sphinx3*, *ECLAT* and *Hmmer-I*).

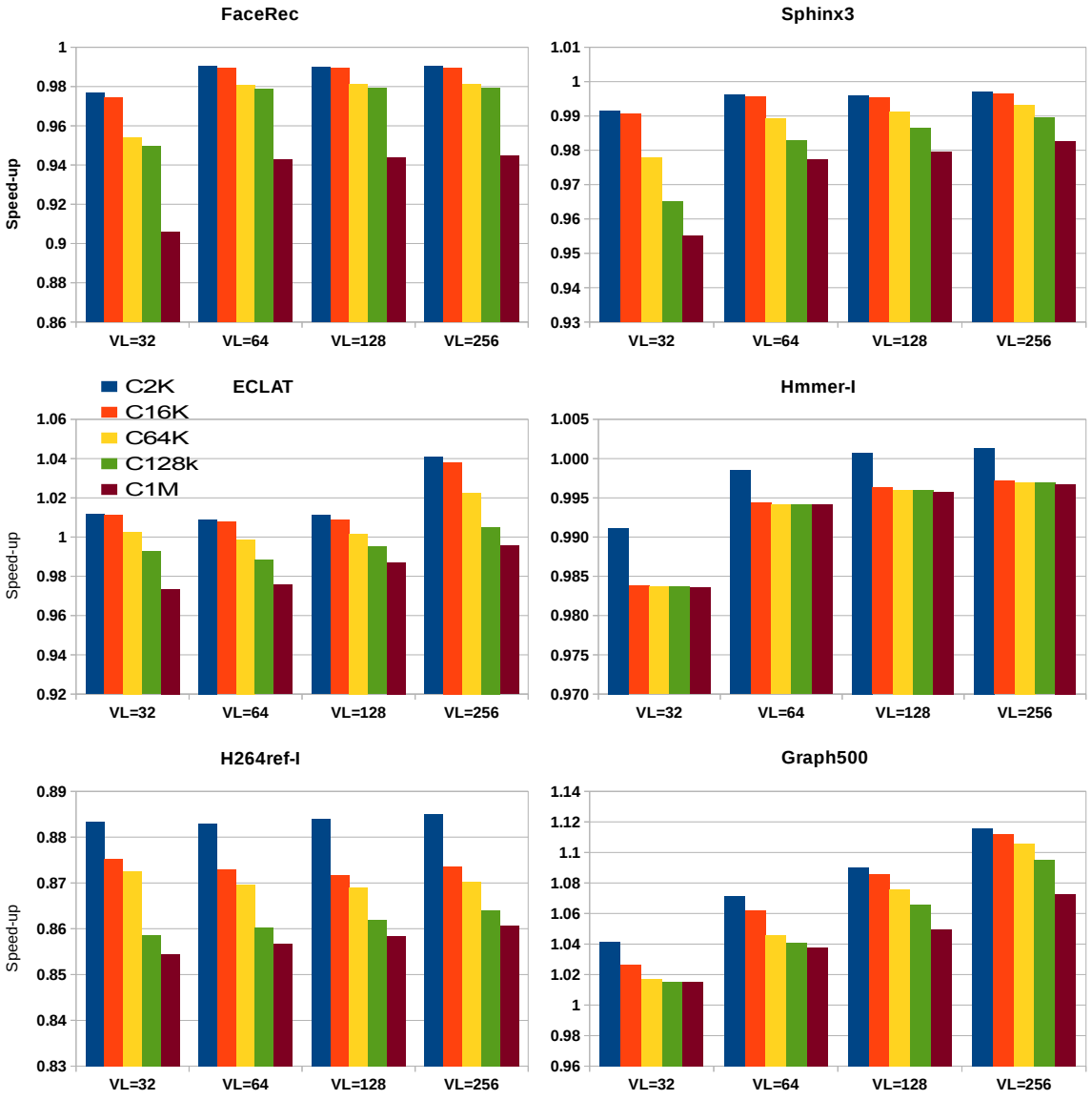


Figure 2.6: Results for direct L2 access.

Direct L2 access

Direct L2 access is an interesting solution to integrate a vector unit with the cache hierarchy of a microprocessor [24, 30, 65]. The idea is that vector memory instructions directly load/store data from/to L2 cache. We performed the same experiments described above but with direct L2 access. Figure 2.6 shows speedups for direct L2 access over the results presented in Figure 2.5. In most cases, accessing the L2 directly does not have a big impact on performance. In the baseline configuration, an L2 hit pays a higher latency than the same access in direct L2 because it has to add the latency of accessing and missing in the L1. On the contrary, when an access hits in the L1 the latency is smaller. For this reason, in larger cache configurations where the L1 hit rate is higher, the baseline configuration is usually faster than the direct L2 configuration while for smaller configurations direct L2 often outperforms the baseline. The difference in performance is less than 5% for all applications except for *H264ref* which degrades performance about 15% with direct L2 access and *Graph500* that sees a speedup of roughly 10%. The reason is that for *H264ref*, the data set fits in L1 even with the smallest configuration, and for *Graph500*, it does not fit even with the largest.

2.5.2 In-Order vs Decoupled

We perform two kinds of experiments in this section: we compare the IO and DC execution models and search for optimal configuration of functional and memory units using vectorized applications and different MVLs. We consider four MVLs as in the previous section and four different configurations of functional units ranging from two FUs and two memory units to four FUs and three memory units. The FU units support all vector arithmetic and logic instructions with any data types for both models. In the IO model, the memory units support all types of vector memory instructions while in the DC model, a given memory unit supports either load or store instructions. This approach allows parallel execution of independent load and store instructions. In configurations with three memory units, two units support vector load instructions and one unit support store instructions. In other configurations, memory units are divided equally. The *C128K* cache configuration is used in all experiments.

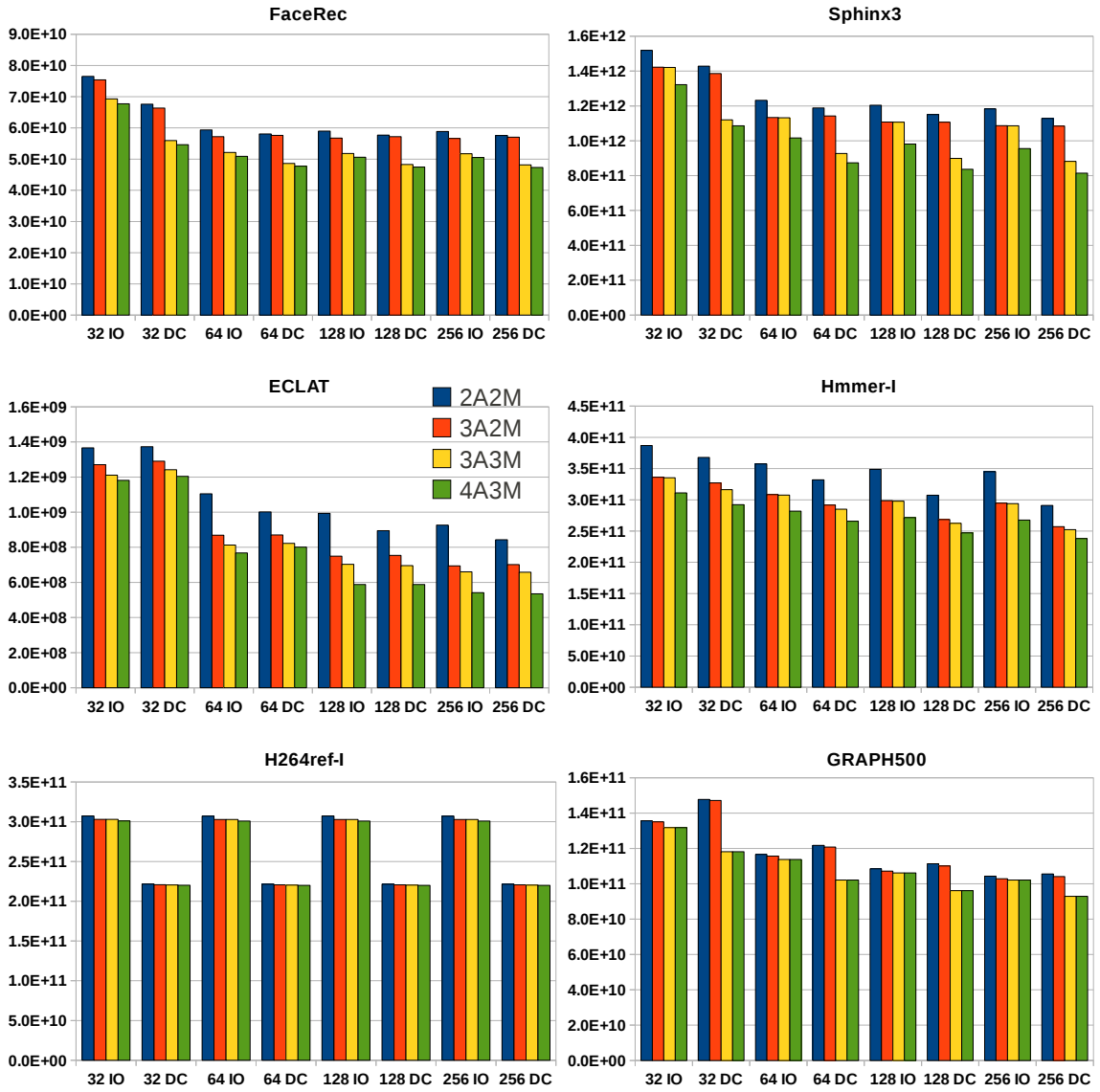


Figure 2.7: Execution time of applications for in-order and decoupled vector architectures. *jAkM* stands for a configuration with *j* FUs and *k* memory units.

Figure 2.7 presents the execution time for the vectorized applications evaluated on the two models using different MVLs and configurations of functional units. As it is expected, the best results are obtained for the DC architecture for all applications because vector memory instructions can be executed ahead of computation instructions. Regarding the IO architecture, the results show that the memory instructions limit the performance of the applications. Thus, it is beneficial to start executing them as soon as possible. The main exception is *Graph500* for the configurations that have two memory units where IO is faster than DC. The reason is that in DC memory units are dedicated to either vector loads or stores while, in IO, they are universal. This allows IO to execute two load instructions in parallel from the main kernel. This outperforms the advantage of DC to overlap memory with computation because there are not many computation instructions in the main kernel. This also happens for some other applications, but the difference is much smaller. We performed an experiment with two dedicated memory units (one for loads, another for stores) for IO. DC outperforms IO in this case.

A large number of FUs provides smaller execution time for all applications, except *H264ref-I* which has almost the same execution time for all configurations. An increased MVL provides smaller execution time for *Sphinx3* and *FaceRec* for MVLs up to 64. It is not the case for *Hmmer-I* and *H264ref-I* because the L1 cache hit rates are very high as mentioned in the previous section.

Adding a third memory unit in the IO model yields very small improvements in execution time for all applications, while a third memory unit in the DC model improves the execution time for *Sphinx3* and *FaceRec*.

The results show that decoupling improves execution time for our applications over an in-order architecture. For the configuration with three FUs and three LD/ST units and an MVL of 128 elements, the average speed-up is 1.16x and it ranges from just 1.01x for *ECLAT* up to 1.37x for *H264ref-I*.

2.6 Case Study: New instruction

Vector ISAs do not provide all possible vector instructions and the lack of a given instruction can decrease performance or even prevent vectorization of a kernel by the compiler as it was shown in Section 2.3. Our tools make it very easy to add

new instructions to the ISA of VALib and then analyze the performance using SimpleVector. VALib can also be used to study the impact in the average vector length and other instruction-level metrics.

In the first vectorized version of *FaceRec* that we wrote, the average vector length for the complete application was only 18 because it loaded many short vectors in several kernels. Figure 2.8 shows how a particular kernel accesses the elements of a matrix. There are three cases (a, b and c). All three patterns are used the same number of times in a cyclic fashion. The kernel always accesses 64 elements. In case *a* all 64 elements can be loaded with a single vector load instruction with a stride of 2 while, in the other two cases, several vector load instructions have to be executed with a shorter vector length. However, as the figure shows, there is a regularity in the accesses that would make possible to load all the elements at once even in cases *b* and *c*.

In this kernel, there is the possibility of applying memory shape instructions [15], in which the vector is described by the address of the first element and three scalar values: *stride*, *span* and *skip*. Stride describes the spacing between each loaded/stored element (inclusive of element). Span describes how many elements to access at stride spacing before applying the second-level skip offset. Memory shape instructions can be seen as an extension of stride instructions to 2-D patterns.

We implemented the memory shape instructions in VALib and SimpleVector and modified the vectorized kernel of *FR* to use them. All instructions in the new version of the kernel have a vector length of 64 (unless the MVL is smaller). The average vector length for the whole application becomes 38 for an MVL of 64.

The results with SimpleVector show that memory shape instructions provide up to 1.45x speed-up over the original vectorized kernel, with small differences depending on the MVL and configuration chosen. The memory shape instructions are implemented in the same way as strided memory instructions (one access to memory per element). It means that this speed-up is achieved from longer vector lengths and reduced number of executed vector instructions. The memory shape instructions have been used to vectorize several kernels. They are used in *FaceRec* and *H264ref* applications, accounting for 8.1% and 4.3% of all vector memory instructions, respectively.

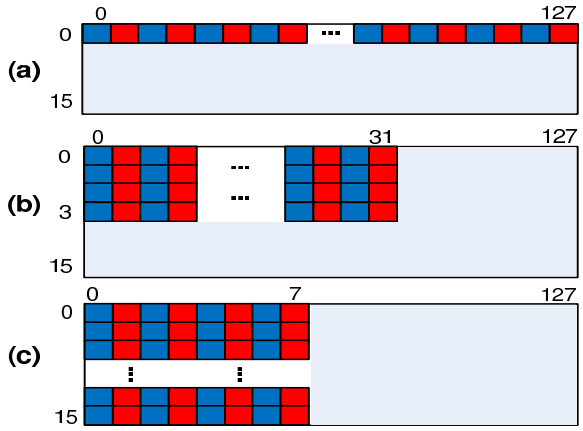


Figure 2.8: Memory access patterns of the case study.

We also implemented several other uncommon instructions that are useful or required to vectorize our target applications. *Select* instruction selects elements from one or other source vector register depending on the value in vector mask register. The library also implements new reduction instruction called *sub-reduction* add or *sub-sum*. This instruction performs the sum for sub-sets in a vector register. Some vector memory instructions (including indexed vector load or gather) are also implemented with support for masking. These instructions are useful in kernels where we have to store or load some elements of a stream depending on some condition.

2.7 Related Work

Valero [82], Espasa [20] and Quintana [64] presented a detailed instruction-level characterization of selected programs from the PERFECT club and SPECfp92 benchmarks. Quintana also included several benchmarks for the Mediabench suite, modified some benchmarks to vectorize them and manually strip-mined all programs. A trace-driven approach was used to characterize the programs. Binaries generated by Convex C3400's compiler were instrumented to produce traces [21]. Asanovic [6] presented the design, implementation, and evaluation of the first single-chip vector microprocessor (T0). He presented the results and statistics for several applications evaluated with T0.

Previous work mentioned above includes statistics similar to the ones presented in this paper, e.g. distribution of vector lengths, vector register usage, etc. However, the methodology to obtain the results is very different since instead of using a vector compiler we developed VALib that allows us to vectorize applications and explore the potential of new vector instructions for them. It is hard to compare results because we vectorized different applications, there is no general trend for all applications and input data sets are much larger compared to those used in their experiments.

Corbal [19] implemented a library to emulate a multimedia extension for Alpha architectures. The code was manually modified to insert calls to the library. An instrumentation tool was used to detect the calls to the library and feed a simulator. The instruction set implemented was completely different to ours and targeted multimedia extensions instead of vector architectures. Moreover, the use of an instrumentation tool to detect the calls can slow down a lot the target application.

Like us, Janin [38] developed a library that implements many common instructions that are present in a register-based vector architecture. However, the library does not allow the simulation of any particular architecture nor microarchitectural details (e.g. memory hierarchy, chaining, etc.) and is not useful to estimate execution time. The library was specifically designed to vectorize speech recognition algorithms. It is not publicly available so it cannot be used by the research community.

2.8 Summary

This chapter presented two tools, VALib that enables rapid manual vectorization and characterization of applications and SimpleVector, a simple and fast simulator to estimate the execution time in many different vector processor configurations. The potential and applicability of the two tools was demonstrated on six applications with potentially high DLP. VALib can easily be used to vectorize and characterize an application at the instruction level.

SimpleVector allows the evaluation of several alternative properties of the microarchitecture such as the memory hierarchy and in-order versus decoupled architectures. Moreover, a case study shows how VALib and SimpleVector can be used

to evaluate the performance speed-up achieved with the addition of new instructions to a vector ISA. VALib and SimpleVector have been designed to be extended easily with new features and instructions.

In this chapter we saw that the memory side is also important. Some applications are memory bound where vector memory instructions are dominant (see Section 2.2.2). There is also need to support complex memory access patterns including gather/scatter, 2D-access, masking, etc. In a future vector microprocessor, special attention should be given to a design and implementation of the memory side.

3

Evaluation of Vectorization Potential of Graph500 on Intel's Xeon Phi

In the previous chapter we performed a detailed analysis of instruction-level characteristics of our target applications. Most of them have a high degree of vectorization and this suggests that vector processors could be a good match. We also analyzed the impact in performance of the cache hierarchy and functional unit configuration in in-order and decoupled vector architectures. Since our targets are low-end embedded systems it seemed reasonable to evaluate them rather than more power hungry alternatives like out of order.

Our research center has access to machines with the Xeon Phi "Knights Corner" processor. The Xeon Phi processor is a massively parallel x86 microprocessor designed by Intel. Since it has simple in-order cores and energy-efficient characteristics, deeper insight into Xeon Phi capabilities during execution of vectorized applications could provide fruitful informations for our processor design.

Our goal in this chapter is to evaluate of Xeon Phi characteristics that can be useful in the process of making design decisions for our low-end embedded processor extension. We decided to use the Graph500 [56] benchmark as an example during evaluation. The main contributions of this chapter are:

- A study of the impact of prefetching on the performance of the Xeon Phi. The Xeon Phi cores are in-order, which makes it very sensitive to cache misses.
- We prove that the combination of vectorization and parallelization is beneficial for our workload running on the Xeon Phi.

- Two observations that can be useful for Xeon Phi users or processor architects: the initial results for vectorization can be misleading and prefetching is of maximum importance if data is not cache resident.

The rest of this chapter is organized as follows: Section 3.1 gives an overview of Graph500 and Xeon Phi. Section 3.2 describes parallel implementations of Graph500. Section 3.3 shows how we vectorized Graph500 while Section 3.4 outlines methodology that we used during experiments. Section 3.5 presents experimental results and finally, Section 3.6 concludes the chapter.

3.1 Background

The Graph500 benchmark is a data intensive, high performance computing application. It complements TOP500, the performance evaluation metric used to rank supercomputers, and targets five graph-related business areas: Cybersecurity, Medical Informatics, Data Enrichment, Social Networks, and Symbolic Networks. Graph500 is used for performance ranking and has attracted attention in recent years. The Graph500 list ranks the first 500 supercomputers with highest performance running the Graph500 benchmark.

Graph500 aims to implement three kernels: concurrent search, optimization (single source shortest path), and edge-oriented (maximal independent set). They access a single data structure representing a weighted, undirected graph. The main kernel of Graph500, Breadth First Search (BFS), contains a lot of irregular memory accesses. Most SIMD ISAs do not provide support for indexed memory accesses and this lack has prevented vectorization of BFS.

The Xeon Phi is a recent massively parallel x86 microprocessor designed by Intel and is based on the Larrabee GPU [74]. Its new features with respect to some multimedia extensions, such as scatter/gather instructions, satisfy missing requirements that prevent vectorization of Graph500. It has a large number of cores and each core contains a wide SIMD unit. Each core in Xeon Phi [36] uses a short in-order pipeline and is capable of supporting 4 threads in hardware. It contains a vector processing unit (VPU) that implements a novel 512-bit SIMD instruction set. A mask register was added to allow predicated execution. The VPU can execute

8 operations per cycle with 64-bit data or 16 operations per cycle with 32-bit data. It delivers substantial performance and has been designed for energy efficiency when executing highly parallel applications that can benefit from parallelization and vectorization. These characteristics make the Xeon Phi an attractive processor for building supercomputers. The most powerful supercomputer according to the TOP500 list from November 2013[1], Tianhe-2 (MilkyWay-2), is built using Xeon Phi co-processors.

Currently there are six supercomputers on the Graph500 list[2] (November 2015) that are built using the Xeon Phi. However, in their implementation of the Graph500 they just exploit the parallelization features of the Xeon Phi, following the work done by Saule et al. [70] and ignore the vector features offered by the Xeon Phi that allows for more efficient parallelization.

3.2 Parallel BFS Implementations

BFS is the main kernel of Graph500. BFS begins at a random node called start node and inspects all its neighboring nodes. Then, for each of those neighbor nodes, it inspects their neighbor nodes which were yet unvisited, and so on. In Graph500, there is freedom to change the algorithm, the implementation and the data structures used. In order to be able to compare the performance of BFS implementations across a variety of architectures, programming models, languages and frameworks, the performance metric traversed edges per second (TEPS) is used. It is defined as the ratio of the number of edges in the input graph to the execution time. In this section we describe the current default implementation (V1.2) that we have used in our experiments, as well as an overview of recently proposed alternative BFS implementations.

3.2.1 Current BFS Implementation

The default parallel implementation of BFS is queue-based with local per-thread next-level queues. Figure 3.1 shows the pseudo-code of that implementation. It is a

[1] http://top500.org/lists/2013/11/
[2] http://www.graph500.org/results_nov_2015

level synchronous BFS algorithm with two optimization techniques borrowed from [4]: *'test and test-and-set'* operation and use of local next-level queues. It means that the search of unvisited neighbor nodes (*neighbors*) is done in parallel (*partition* per thread) and there is synchronization before the algorithm starts to search their neighbors. The algorithm manages two sets of nodes: the visited (*parent*) set and the next-nodes (*next*) set. The *next* set is implemented using a queue (global queue). BFS starts searching from the *start* node. In each iteration, the algorithm visits all the nodes in the *next* set in parallel and for each node, the *'test and test-and-set'* operation (lines 8-9) is used to check if the neighbors have not been visited already (line 8). If this is true, the *parent* node is assigned to the parent set (line 9). Unvisited nodes are first stored in the per-thread local queue (*local_queue*) until they are bulk inserted into the global queue or the *next* set (lines 12, 20). The per-thread local queue is used to avoid synchronization for each element added in the global queue, and therefore to provide better performance. Figure 3.2 shows an example of the graph traversed using this implementation. The *start* node is A. In the first iteration, all its neighbors are visited (gray nodes - B, C, D) and added to the *next* set. In the next iteration, the neighbors of gray nodes are visited and only unvisited nodes (yellow nodes - E, F, G, H) are added to the *next* set.

3.2.2 Other BFS Implementations

In this subsection we describe alternative implementations of BFS. Since all these alternatives are still not included in official Graph500 benchmark we did not consider their vectorization.

Bitmap

A bitmap is used to compactly represent the visited set as an optimization to the default algorithm that has been discussed above. The main benefit is to reduce the size of the structure, thus reducing cache misses. Since this structure is the most frequently accessed data in the algorithm, the use of a bitmap in shared-memory machines with large last-level caches is effective.

```
1.  function breadth-first-search(vertices, source)
2.      next ← {source}
3.      parents ← [-1,-1,...,-1]
4.      parents[source] = 0
5.      while next ≠ {} do
6.          for v ∈ next.partition[tid] do
7.              for n ∈ neighbors[v] do
8.                  if parents[n] = -1 then
9.                      if parents[n].atomicSet(v) = 1 then
10.                         local_queue[tid] <- n
11.                         if local_queue[tid].isFull() then
12.                             next.safeBulkPush(local_queue[tid])
13.                             local_queue[tid] ← {}
14.                         end if
15.                     end if
16.                 end if
17.             end for
18.         end for
19.         if local_queue[tid].notEmpty() then
20.             next.safeBulkPush(local_queue[tid])
21.             local_queue[tid] ← {}
22.         end if
23.     end while
24.     return parents
```

Figure 3.1: Pseudo-code for the Graph500 BFS algorithm.

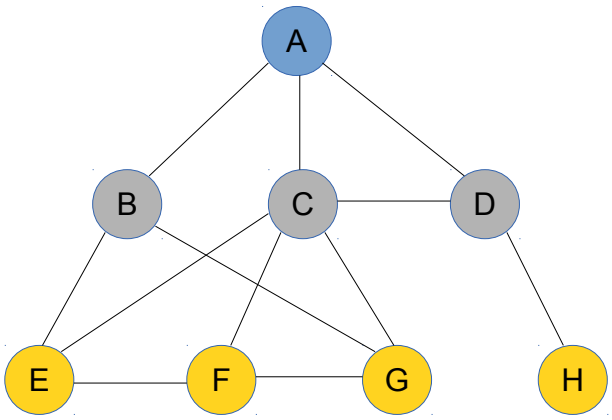


Figure 3.2: An example of graph traversed by the BFS algorithm.

Read-based

Read-based BFS is proposed in [33] and it also keeps the visited set as a bitmap. In contrast to the current BFS implementation, it keeps a node's level in the *next* set. During the search, if a node belongs to the current level, it means that its neighbors should be searched in the current iteration. For example, in Figure 3.2, the blue node will have level 0, the gray nodes will have level 1 and the yellow nodes will have level 2 instead of having id of parent node. The advantages of this method are the elimination of queue overhead (it removes atomic instructions and also saves cache and memory bandwidth) and the replacement of some indexed memory accesses with sequential memory accesses.

Bottom-Up

Bottom-up BFS [10] proposes searching in the opposite direction compared to previous methods. Instead of each node in the *next* set attempting to become the parent of *all* of its neighbors, each unvisited node attempts to find *any* parent among its neighbors. A neighbor can be a parent if the neighbor is a member of the *next* set. For example, in the first iteration for the graph in Figure 3.2 only node A is in the *next* set. It means that only the gray nodes (B, C, D) can find a valid parent in the first iteration. This approach is more efficient when the *next* set is large. It reduces a lot the total number of edges examined.

Hybrid

Hybrid BFS [10] is the state-of-the-art BFS implementation and it is part of pre-released version of Graph500 benchmark. It combines top-down and bottom-up approaches because they are complementary. It uses the top-down approach when the *next* set is small and the bottom-up approach when the *next* set is large.

3.3 Vectorization of Graph500

We vectorized by hand two versions of BFS: sequential and parallel OpenMP versions based on the current implementation. Both versions are vectorized practically

in the same way using intrinsics. The part of code that is actually vectorized is the loop where a node searches for its neighbors (lines 7-17 in Figure 3.1). All neighbors of the node are loaded using vector load instructions (line 7). Line 8, where the algorithm checks if a neighbor is not visited, is vectorized using an indexed vector load from the *parent* set and comparing loaded values with -1. The neighbors that are loaded in the previous step are used as indices. The two previous steps are exactly the same for the sequential and parallel versions. The next step, where the *parent* and *next* sets have to be updated, is different. In the parallel version, unvisited neighbors are stored to a temporal array using the vector packstore instruction to store selected elements to consecutive memory locations using a vector mask. Then scalar atomic instructions are used to ensure that all the threads correctly update the unvisited nodes to the *parent* and *next* sets. This is done with scalar code. In the sequential version, the *parent* set is updated using an indexed vector store, while the *next* set is updated using the vector packstore instruction.

As mentioned above, the Xeon Phi operates on 512-bit vectors. Our implementation of Graph500 uses 32-bit data, thus each Xeon Phi vector instruction operates on 16 elements at a time. This is known as the vector length. If the number of iterations in a loop is longer than the vector length, the strip-mining technique [31] is applied during the process of vectorization. Strip-mining is a technique that allows for operating on "stripes" of data of length less than or equal to the vector length. If the number of iterations is not a multiple of the vector length there is a remaining part, called epilogue. The epilogue can be vectorized in the Xeon Phi using vector masked instructions or it can be left as scalar code. Please note that when the number of iterations is smaller than the vector length, the whole loop is considered the epilogue. Xeon Phi has support for gather vector memory instruction that has an index vector with elements of a maximum of 32 bits. Since elements that are loaded from memory and used as index vector, we are forced to use 32-bit elements for graph representation instead of 64-bit elements like the original implementation of Graph500.

3.4 Methodology

This section presents our experimental setup. We measured the performance of our various vectorized BFS implementations and we compared them against the original Graph500 implementations. *Edgefactor* and *SCALE* are the main input parameters of the graph generator. *SCALE* is the log_2 of the number of nodes in the graph. This parameter determines the graph size and consequently the size of data structures needed to store it. *Edgefactor* is the ratio of the graph's edge count to its node count (i.e., half the average degree of a node in the graph). For higher *edgefactor*, the average size of the adjacency list is also higher. In particular, this parameter has a direct effect on the length of the vectorized loop and, consequently, on the amount of vectorized code when the epilogue is executed with scalar instructions.

We use an *edgefactor* of 8 and a *SCALE* of 23 in our experiments unless specified otherwise. This is the highest value of SCALE that can be used in our system. The graphs are stored in Compressed Sparse Row (CSR) format. It merges the adjacency lists of all nodes into a single array, with the initial location of each node's adjacency list stored in a separate array. We used Intel's compiler version 13.0.1 with the -O3 optimization level.

We run our experiments on a compute node that contains two Intel Xeon CPU E5-2670 @ 2.60GHz processors (8 cores/processor), 64 GB of RAM memory and two Intel Xeon Phi 5110P (60 cores and four hardware threads per core). In our experiments we use a single Xeon Phi processor.

As mentioned above, TEPS is used to benchmark performance and it represents the ratio of the number of edges in the input graph to the runtime. Graph500 performs 64 searches (starting from a random node) per run. We use the harmonic mean of TEPS of all 64 searches (*harm TEPS*).

For the sake of clarity, we label the experiments in the following way: the name of experiment has three parts; the first part indicates whether it is vectorized (*vect*) or sequential (*seq*) code; the second part tells if sequential prefetching is applied (*spf*), vectorized (*vpf*), both are combined (*vspf*) or there is no prefetching (*npf*); this is followed by a number that indicates if it is a single-thread execution (1) or multi-thread execution (*n* being the number of threads used). For example,

seq_npf_1 is a sequential single-thread execution without prefetching. Additional information is appended to the name in some cases.

Two modes of execution are used to run experiments on the Xeon Phi: "offload" and "native" mode. In "offload" mode, an application runs on the host machine and the parts of the code that are specified to be executed on the Xeon Phi are "offloaded" during execution to the co-processor. Data has to be copied to the co-processor memory before and after "offloading". In our experiments in "offload" mode, the whole BFS kernel is executed on the Xeon Phi. In "native" mode, the application is executed completely on the Xeon Phi. By comparing the two, we can see the cost of "offloading".

3.5 Experimental Results

In the first part of our experiments, we evaluated our vectorized implementation of BFS against the sequential implementation for single-thread execution. In the second part, we focused on parallel OpenMP implementations. Our main goal was to check if there is any benefit of applying vectorization on Graph500 using the Xeon Phi.

3.5.1 Single-Thread Results

For the single-thread execution, we focused on three types of experiments. We compared results when only vectorization is applied, then we applied prefetching and finally we measured hardware counters using the PAPI library [54] to better understand the results that we obtained in the previous experiments.

Manual vectorization and memory alignment

We compared our sequential implementations of Graph500 against the original sequential CSR implementation from the benchmark suite.

The CSR format is used to store the graph. It merges the adjacency lists of all nodes into a single array, and the offset to the initial location of each node's adjacency list is stored in a separate array. By storing the list consecutively, the lists are

typically unaligned at the 512-bit boundary. This implies that two vector memory instructions are needed to load/store a 512-bit vector with a node's adjacency list in the Xeon Phi. We have implemented a padded version of this structure to enforce 512-bit aligned accesses, thus reducing the access to a single vector memory instruction.

Figure 3.3 shows the speedup over the original implementation for different single-threaded versions of Graph500: sequential original CSR (*seq_npf_1*), a vectorized version with aligned accesses (*vect_npf_1+alig*), a vectorized version with unaligned accesses to the original (non-padded) data structure (*vect_npf_1+unalig*), and *vect_npf_1+unalig+epil*, which is the same but with a vectorized epilogue using masked operations. The epilogue is executed in scalar fashion in *vect_npf_1+alig* and *vect_npf_1+unalig* versions. The speedups are computed for *harmonic TEPS* over *seq_npf_1*.

In Figure 3.3, one may notice that vectorized implementations yield small speedups, ranging from 3% for unaligned+epilogue up to 7% for unaligned. Another interesting point is that we did not get better results when aligned memory accesses are enforced. The main reason is the increased cache miss rate due to the use of a larger structure to store the input graph. We also run experiments using the "native" mode of execution and we almost got the same results for the sequential and aligned versions. We experimented with an *edgefactor* of 16 to increase the length of the vectorized loop. However, we saw only a 3.5% speedup for the unaligned version. Subsequent experiments assume vectorized code with unaligned memory accesses and scalar epilogue.

Vectorization Prefetching vs Sequential Prefetching

Since the results of vectorization were not satisfying, we decided to apply prefetching to the original sequential code and our implementation. The Intel compiler has good automatic prefetching capabilities but in our experiments it does not provide significant speedup. The main reason is the abundance of indexed memory accesses, which the compiler does not prefetch automatically. Therefore, we decided to manually insert prefetching intrinsics. We used scalar prefetch instructions for

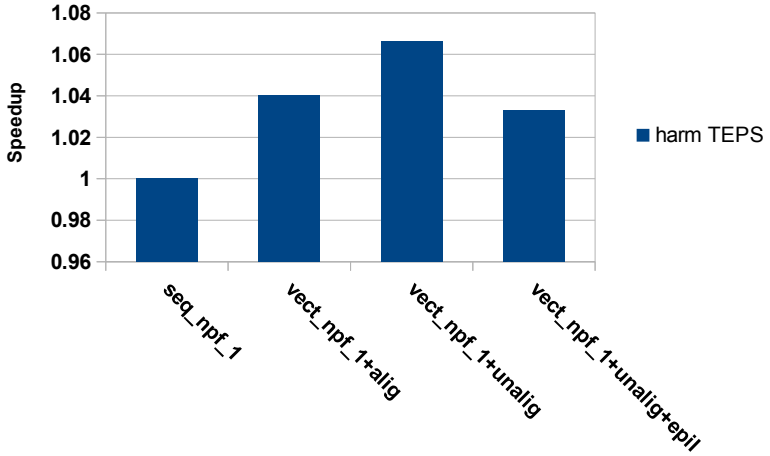
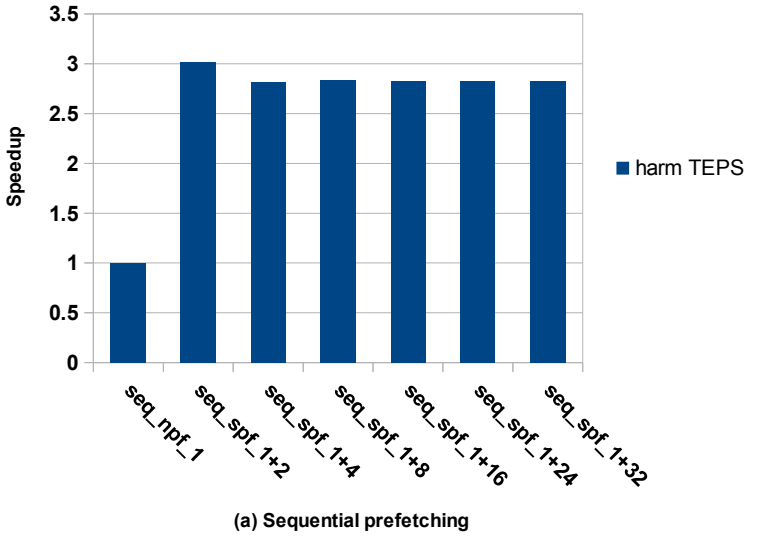


Figure 3.3: Results for different implementations using single-thread run and no prefetching.

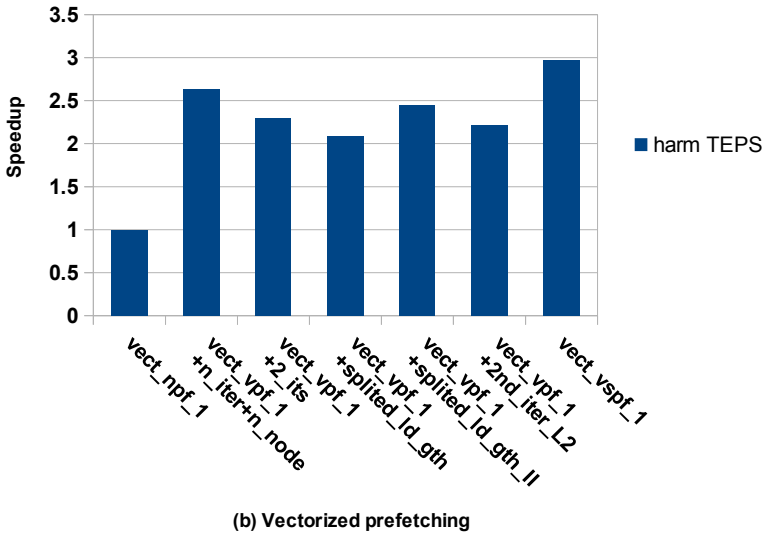
the sequential version and vector gather prefetch instructions for our vectorized implementation with unaligned accesses to memory. The results are presented only for the "native" mode but the results for "offload" are very similar.

For sequential prefetching, we experimented with six different prefetch distances. Figure 3.4 (a) shows results for these experiments. The y-axis presents the speedups over the original implementation (*seq_npf_1*) and the x-axis shows experiments with different prefetch distances. Scalar prefetching allows for significant speedups. A prefetch distance of two provides the maximum speedup, 3.01x.

We implemented six variants for vectorized prefetching. Prefetching is applied to the vectorized *for* loop in line 7 of Figure 3.1. In the first variant (*n_iter+n_node*), we prefetched the next iteration of the vectorized loop (neighbor's nodes) for the current node in the *next* set or the first iteration of the next node in the *next* set if we are in the last iteration of the vectorized *for* loop. In the second variant (*2_its*), we prefetched the next two iterations if possible. In order to implement prefetching, we need to load the index vector for the gather vector prefetch instruction. We separated the vector load and the vector gather prefetch instructions in the third (*splited_ld_gth*) and fourth (*splited_ld_gth_II*) variants and placed them in different places of the code. The fifth variant (*2nd_iter_L2*) is similar to the second one, the only difference is that the second iteration is prefetched to the L2 cache. The last variant (*vect_vspf_1*) combines the first variant for vectorized code and sequential



(a) Sequential prefetching



(b) Vectorized prefetching

Figure 3.4: Results with prefetching for single-thread run.

prefetching for the epilogue. As explained above, the epilogue is implemented with scalar instructions.

Results for prefetching in the vectorized code are presented in Figure 3.4 (b). Speedups are computed over the vectorized implementation without prefetching. Prefetch again provides significant speedups, up to 2.99x for the sixth variant that combines sequential and vector prefetching. It is interesting to notice that the best results are obtained when sequential prefetching is included. The speedup is 1.07x over the best vector-only prefetching variant (first variant). The combination of both prefetching (*vspf*) schemes in the vectorized version provides a speedup of 1.04x over the best sequential variant (prefetch distance of two). The next section aims to explain these results by analyzing measurements obtained with hardware counters.

PAPI Profiling Results

For further analysis of the results that we obtained using vectorization and vector prefetching, we used PAPI [54] hardware counters. We run experiments for five different implementations: sequential (*seq_npf_1*), vectorized (*vect_npf_1*), sequential with prefetching (*seq_spf_1+2*), vectorized with vector prefetching (*vect_vpf_1+n_iter+n_node*) and vectorized version with combined sequential and vector prefetching (*vect_vspf_1*). The experiments were run using the "native" mode of execution. Table 3.1 summarizes the results that we collected with PAPI counters, while aggregating the results for 64 BFS calls (the benchmark performs searches for 64 random start nodes). The first column lists the different implementations of BFS. The second column presents the percentage of vector instructions executed in the benchmark. Third and fourth columns are L1 and L2 cache miss rates, respectively. The fifth column is the number of vector instructions executed. Finally, the last column is the number of execution cycles.

Table 3.1 shows that the results for the original and vectorized versions are similar. Provided that the epilogue is executed with scalar instructions, one may guess that there is a small amount of vector instructions and there are no benefits of vectorization, but it is actually not the case. 42.71% of all executed instructions are vector and the main reason why we observe similar performance for sequential

Table 3.1: Obtained results using hardware counters.

Implementation	% of vector instructions	L1 cache miss rate	L2 cache miss rate	# of instructions	# of cycles
Sequential	0.07	44.98	89.34	11.3×10^8	47.8×10^9
Vectorized	42.71	66.78	89.99	6.36×10^8	47.5×10^9
Seq_prefetching	0.01	1.96	15.87	30.1×10^8	12.0×10^9
Vect_prefetching	53.78	12.68	47.14	9.83×10^8	15.9×10^9
Seq_vect_prefetching	41.55	4.54	19.34	12.7×10^8	11.8×10^9

and vectorized versions is cache behavior. The use of gather/scatter instructions increases the L1 cache miss rate and therefore reduces the benefit of vectorization.

As can be expected, the use of prefetch instructions (rows three and four) decreases a lot L1 and L2 cache miss rates and this is the main reason for the significant speedups shown in Figure 3.4. However, another interesting point is that L1 and L2 cache miss rates for the vectorized version with prefetching are again higher than L1 and L2 cache miss rates for sequential prefetching. This is the reason why experiments with sequential prefetching achieve higher speedups in Figure 3.4. L2 cache miss rate is always higher than L1 cache miss rate because the data set is not L2 cache resident.

The combination of vector and scalar prefetching provides better cache utilization, allowing the vectorized version to outperform the best sequential version by 4%.

3.5.2 Results for OpenMP Implementation

The Xeon Phi provides the best performance when both, parallelization and vectorization, are applied together. In this section we explore the effect of vectorization on the CSR version of Graph500 parallelized using OpenMP.

Manual vs. Automatic Vectorization

Figure 3.5 presents results for three different parallel versions of Graph500. *seq_npf_n* is the original parallel version, *seq_npf_n-vect* is the same but with auto-vectorization disabled and *vect_npf_n* is our vectorized version of parallel Graph500. All three

version are compiled using the -O3 optimization level without prefetching as described in section 3.5.1. The y-axis presents measured performance in harmonic TEPS and the x-axis represents the number of threads used (n).

The results show that vectorization does not have significant impact on the measured performance. The difference between *seq_npf_n-vect* and *seq_npf_n* is also negligible because the compiler is not able to vectorize any part of BFS code. While vectorization seems inefficient, the increased number of threads provides better performance. For example, the parallel vectorized version with 40 threads in "offload" mode is 14 times faster than the best single-thread implementation. The results scale well for "native" mode of execution while they saturate for "offload" mode if we use more than 160 threads. The main reason for better results and scalability in "native" mode is substantial data transfer overhead before and after "offloading".

Vectorization Prefetching vs Sequential Prefetching

We further explore the effects of prefetching on the parallel version of Graph500. We applied prefetching on the original OpenMP implementation using sequential prefetch instructions while we used vector scatter/gather prefetch instructions for our implementation. We experimented with several different prefetch distances for sequential prefetching as well as six different variants for the vectorized version. These are the same prefetching schemes presented in subsection 3.5.1. All experiments were performed using 160 threads.

Figure 3.6 shows the speedup for the original OpenMP implementation with sequential prefetch instructions that use different prefetch distances. The speedup is computed over the original OpenMP implementation. Prefetching increases the performance of Graph500 for all prefetch distances in both modes, "offload" and "native". For example, the highest speedups are 2.07x and 2.36x for "offload" and "native" modes respectively, for a prefetch distance of eight.

The results for vectorized prefetching are presented in Figure 3.7. The y-axis presents speedups computed over our vectorized OpenMP implementation and the x-axis shows our different prefetching implementations. Again prefetching

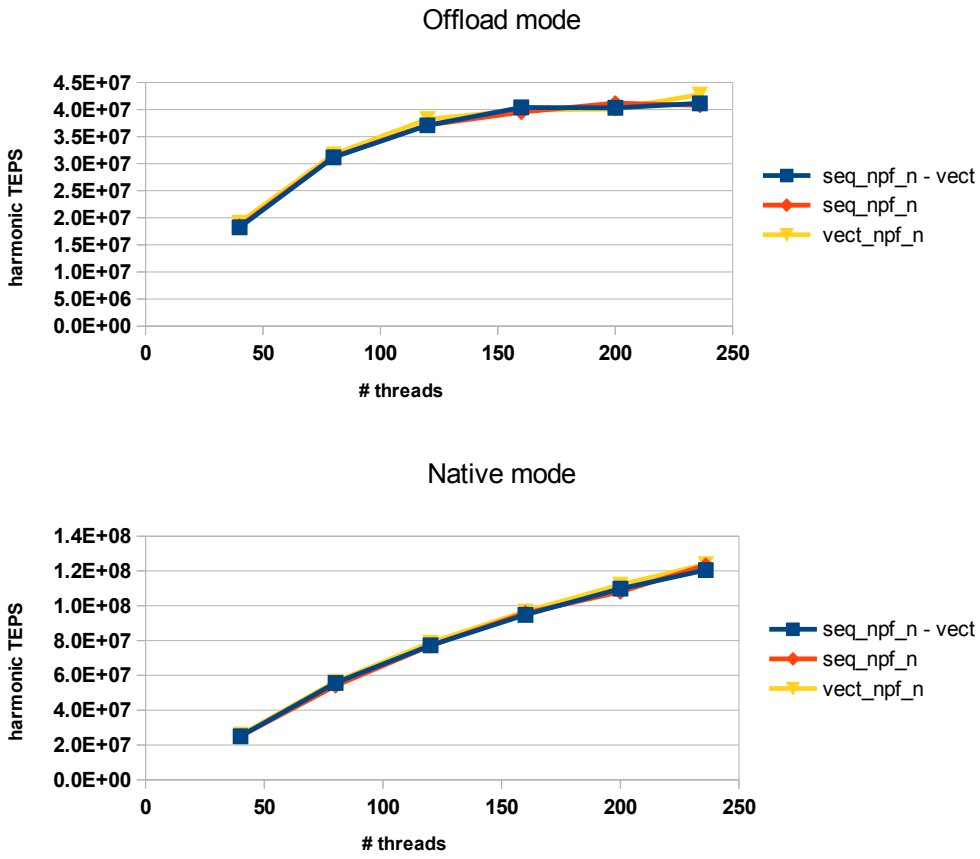


Figure 3.5: Results for hand-written vectorization, auto-vectorization and no vectorization.

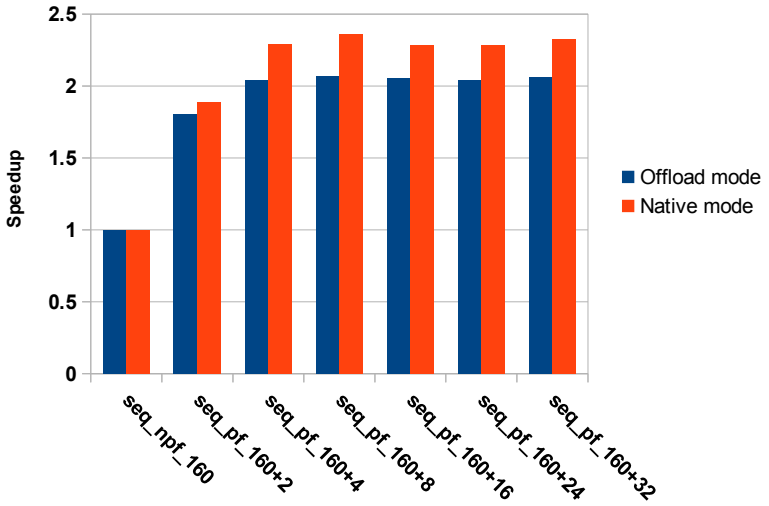


Figure 3.6: Results for OpenMP version with sequential prefetching.

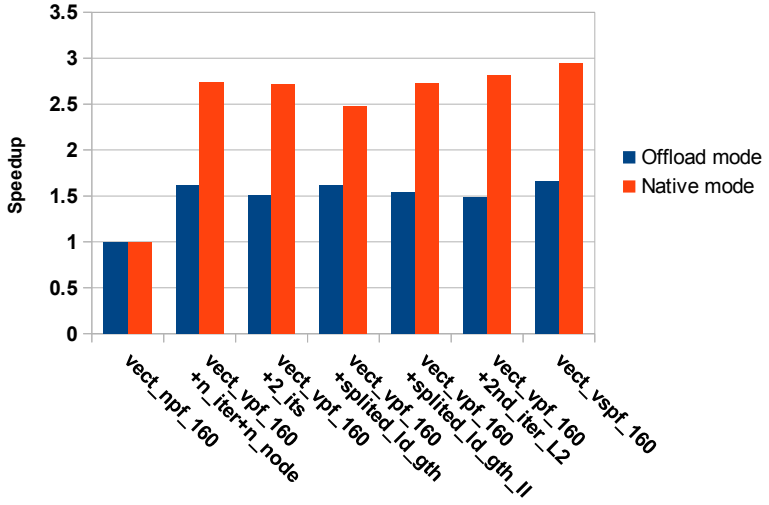


Figure 3.7: Results for vectorized version with gather/scatter prefetching.

increases performance significantly and the best results are obtained for the combined vector and sequential prefetching scheme. Speedups are 1.66x and 2.94x for "offload" and "native" modes, respectively.

Our implementation outperforms the best implementation from Figure 3.6 providing 10% and 27% higher harmonic TEPS in "offload" and "native" modes respectively. The Xeon Phi provides the best performance when both, parallelization and vectorization are applied and Graph500 clearly can benefit from it.

Scalability

Figure 3.8 shows the results for different number of threads for the OpenMP implementation with sequential prefetching and the vectorized version with combined vector-sequential prefetching. It can be seen that "native" mode performance scales better than the results for the "offload" mode due to substantial data transfer overhead in the "offload" mode. In the "offload" mode, the harmonic TEPS seem to saturate after 120 threads, while performance still grows, albeit slower, for "native" execution. It is also noticeable that the vectorized version consistently outperforms the sequential version, for any number of threads and both execution modes.

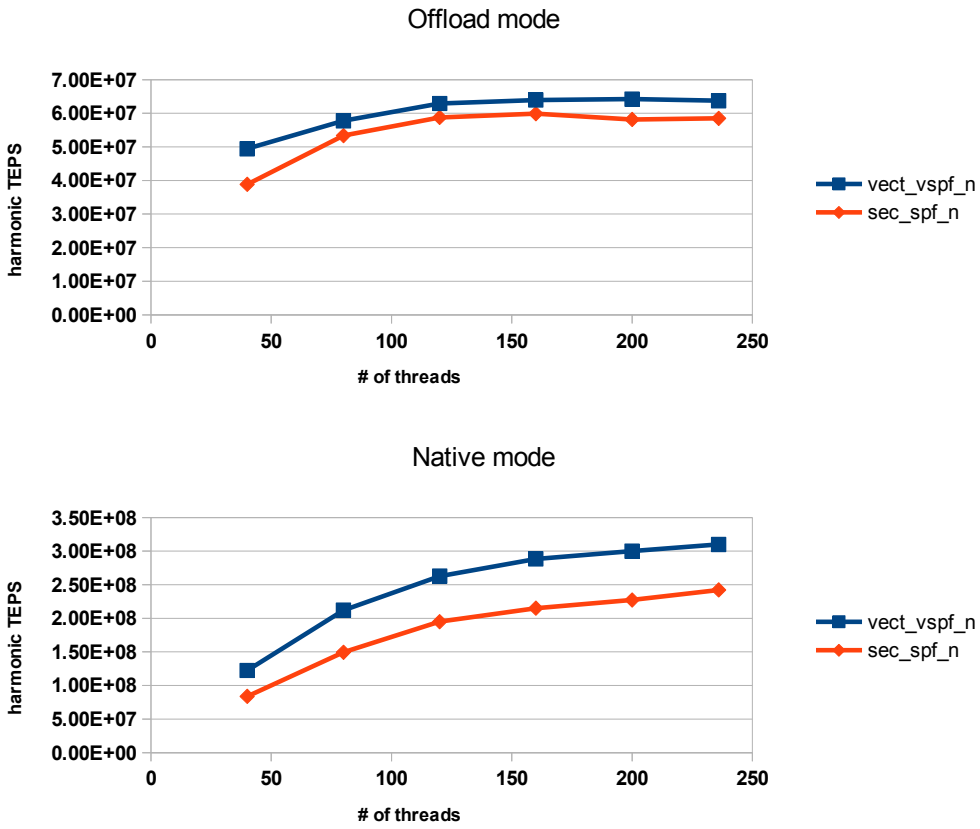


Figure 3.8: Results for prefetching using different number of threads.

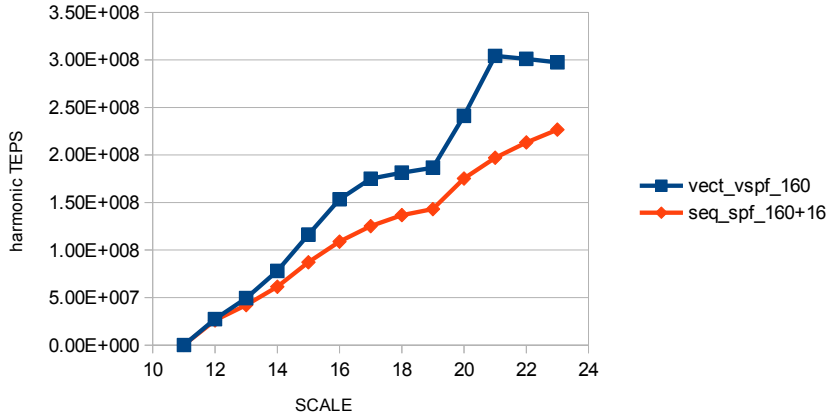


Figure 3.9: Impact of *SCALE* on performance in native mode.

Scale and Edgefactor

Finally, we experimented with different values for *SCALE* and *edgefactor*. We compared again the OpenMP implementation with sequential prefetching and the vectorized version with combined vector-sequential prefetching. We used a fixed number of threads (160) for all experiments and run them in "native" mode.

Figure 3.9 shows results for different *SCALE* values. The y-axis presents measured performance in harmonic TEPS and the x-axis shows different values for *SCALE*. It can be seen that the vectorized version consistently outperforms the sequential version. The vectorized version achieves the highest performance when *SCALE* is 21 and performance saturates for higher numbers. For *SCALE* lower than 14 the vectorized version achieves nearly no speedup and even has slowdown, for a *SCALE* of 11. This *SCALE* is extremely small since Graph500 aims to represent applications with very large graphs. In this case, the generated graph is small enough to fit into the L1 cache and be cache resident.

Figure 3.10 shows results for different *edgefactor* values. The y-axis presents measured performance in harmonic TEPS and the x-axis shows different values for *edgefactor*. The vectorized version again outperforms the sequential version. As it is expected, the vectorized version benefits from a higher *edgefactor* while this gain is smaller for the sequential version. To quantify this, we have measured how often nodes are processed with vector instructions. For an *edgefactor* of 8, the vectorized loop processes 87% of the nodes, while the rest is executed in the scalar epilogue.

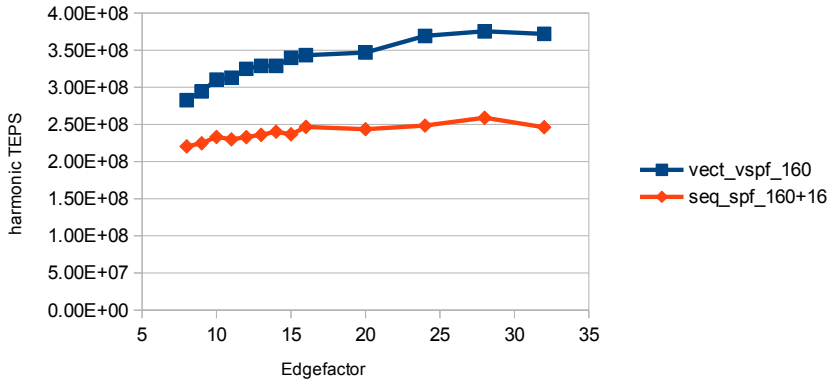


Figure 3.10: Impact of *edgefactor* on performance in native mode.

For *edgefactor* of 32, 94.2% of the nodes are processed with vector instructions due to the increased vector lengths.

3.6 Summary

In this chapter we evaluated the capabilities of the Xeon Phi by vectorizing and running the Graph500. Applying vectorization on a single-threaded implementation provides negligible improvement and a hasty conclusion would be that Xeon Phi and in-order architecture can not be good choice for an irregular data intensive application. This conclusion is misleading because we achieve higher speedups when prefetching is combined with vectorization. Prefetching is more important for the Xeon Phi when running applications like Graph500, because the data is not cache resident. The combination of parallelization with vectorization and prefetching is very important for achieving higher performance. Use of parallelization and four hardware threads per core is a way to hide latency of memory instructions and keep busy wide SIMD units in in-order core. We achieve 27% of speedup for the best vectorized version with applied vector prefetch instructions over the best scalar implementation with sequential prefetching in "native" mode.

4 An Integrated Vector-Scalar Design on an In-order ARM Core

In chapter 2 we vectorized six applications and performed a detailed instruction-level characteristics. We analyzed the impact in performance of the cache hierarchy and functional unit configuration in in-order and decoupled vector architectures. We also experimented with new uncommon vector instructions. In chapter 3 we evaluated one of these applications on the Xeon Phi and results show that even simple in-order cores can provide decent speed-up for highly irregular data-intensive application. The performance of the memory system is crucial. In chapter 3 we used prefetching and evaluate its impact. Alternatively, new methods could be used to keep necessary data close to the CPU when there is need for them.

Our goal in this chapter is to propose and implement a method that will increase the performance of the low-end embedded systems in an energy-efficient way (see chapter 1) by using the gained knowledge and experience from previous chapters. The energy efficiency is attained by modifying a scalar core to execute vector instructions on the existing infrastructure. A set of vectorized kernels from applications analyzed in chapter 2 will be used to evaluate the proposed ideas in this chapter. In summary, the main contributions of this chapter are:

- We propose an integrated vector-scalar design that combines scalar and vector processing mostly using existing resources of an energy-efficient scalar processor (in our evaluation environment it is based on the ARM Cortex A7). In addition to a design that uses a conventional vector execution model, we also

contribute a novel block-based model of execution for vector computational instructions.

- Additionally, we also propose an advanced integrated design which features three energy-performance efficient ideas: (1) chaining from the memory hierarchy, (2) direct result forwarding and (3) memory shape and memory unified instructions. We propose and implement two novel techniques that chain from the cache with the goal of further improving the performance of our integrated design. They can be applied to a conventional vector unit as well. We design and implement a novel result forwarding mechanism which complements the block-based execution and does not require writing to the vector register file. We design a vector memory unit with support for complex memory instructions including memory shape, scatter/gather and unified instructions.

- We present performance, power, area and energy evaluation results of t integrated design. The results show that all vector designs significantly reduce energy over the scalar baseline for most of the considered kernels with a small area overhead. We report up to 5x energy reduction for our block-based execution model over the scalar baseline. Additionally, we found that the block-based execution model provides better results (up to 26% of energy saving) than a conventional vector unit with dedicated units. Regarding performance gains, we report more than a 6x speed-up compared to the scalar baseline. Moreover, our block-based execution model is up to 1.4x faster than the conventional vector unit for floating-point kernels.

- Our results regarding the advanced integrated design reveal that additional speed-up (up to 20%) is achieved with our chaining techniques over the integrated design without chaining. Direct forwarding reduces energy/power consumption of the vector register file by more than 50% for three evaluated kernels while the vector memory shape instruction increases speed-up of a vectorized kernel from 1.77x to 2.66x for the block-based model of execution.

Section 4.1 describes our integrated design. Section 4.2 outlines the experimental methodology and evaluates our integrated design regarding performance, area,

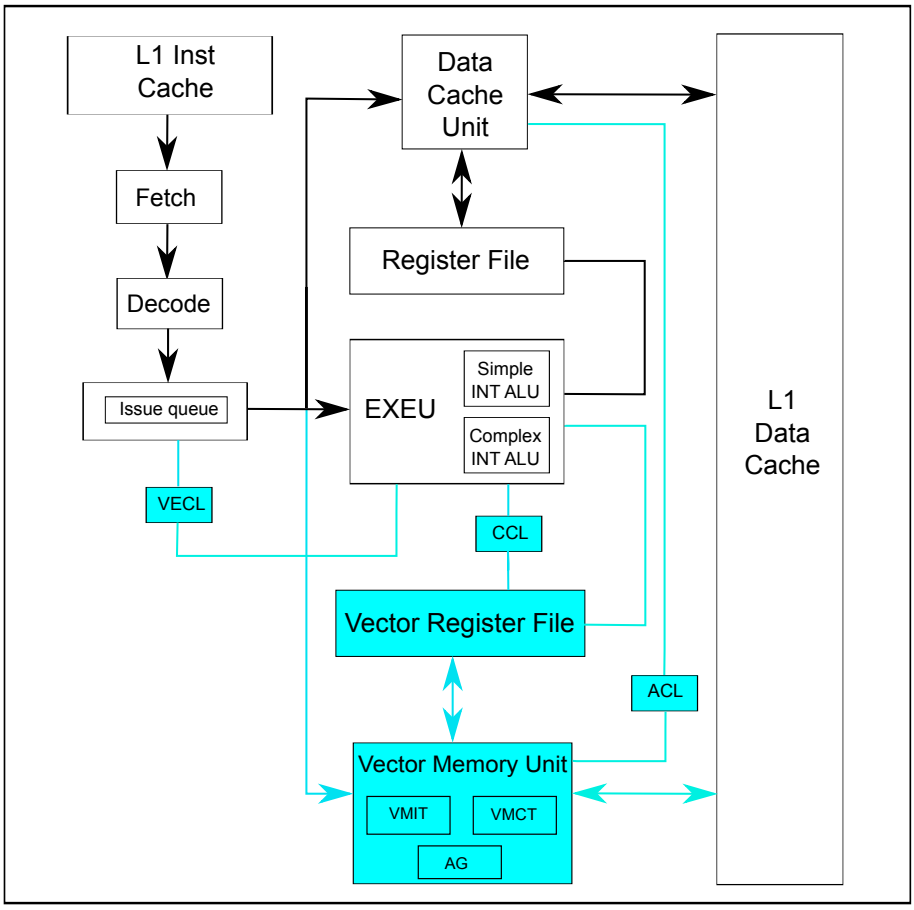


Figure 4.1: Block diagram of the integrated design.

power and energy. In Section 4.3 we propose techniques that further improve the integrated design and we evaluate them in Section 4.4. Finally, Section 4.6 concludes the chapter.

4.1 Integrated Design

As a baseline, we use a scalar core based on the highly energy-efficient ARM Cortex-A7. It is an in-order, dual-issue processor that implements the ARM v7 architecture with an 8-stage pipeline (non-highlighted gray blocks in Figure 4.1).

In our proposed integrated vector-scalar design, we attempt to maximize the reuse of resources already present in the baseline scalar core (white blocks in Figure 4.1) while adding support for vector instructions. While the front-end of the

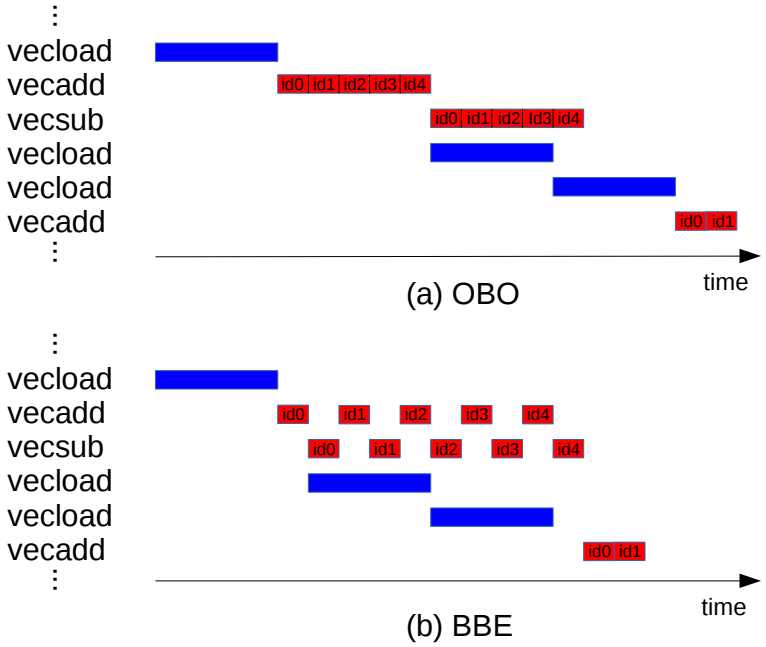


Figure 4.2: An example of code with vector instructions executed with one ALU assuming (a) the one-by-one model and (b) the block-based execution model.

pipeline is the same (fetch and decode[1] stages), in the back-end we added two structures to support the execution of vector instructions on the scalar core: a vector register file, and a vector memory unit (blue blocks in Figure 4.1). There is also additional logic that controls the execution of vector instructions: the vector execution control logic (VECL) is added in the issue stage to support the execution of computational vector instructions. Aliasing control logic (ACL) exchanges information between the vector memory and the data cache unit and forces scalar and vector memory instructions to be executed in-order. We implement support for chaining [69], a well-known concept in vector processors. Similar to result forwarding in scalar processors, chaining allows starting the execution of a dependent vector instruction as soon as the first element of the vector is generated by the previous computational instruction. Chaining control logic (CCL) is responsible for the execution of chained dependent computational instructions.

[1]With the obvious exception of the decode logic, which needs to be extended to support the new vector instructions.

4.1.1 Execution of Vector Computational Instructions

For executing the vector computational instructions on the existing scalar FUs, we study two alternatives: 1) the One-By-One model of execution (OBO), in essence the classic vector execution model, in which a vector instruction is executed to completion once it starts execution in a functional unit, i.e. for all the operations of the vector; and 2) a novel execution model called Block-Based Execution (BBE). In this model, for a block of consecutive vector computational instructions, first all operations on the first element of the vectors are executed, then the operations of the second element, and so on. Figure 4.2 shows an example with a sequence of vector instructions, illustrating the difference of the two execution models. For this example, we assume that vector instructions operate on FP data by using a single FP unit and a single data cache port. The first *vecload* instruction is executed in the same way and at the same time on both models, since the models refer only to computational instructions. Regarding computational vector instructions, in the first case (*OBO*, Figure 4.2 (a)) all operations of one vector computational instruction (*vecadd*) are executed, and then we move on to the next vector instruction (*vecsub*). In the second case (*BBE*, Figure 4.2 (b)), several consecutive vector computational instructions form a block of vector instructions, and we execute one operation from each instruction of the block and repeat this for each operation in the block of vector instructions. In the example, we execute one operation from *vecadd* and then one operation from *vecsub*. The process ends once all operations are computed. The next subsection describes the *BBE* model in more detail.

Block-Based Execution

In order to support this model of execution, we added simple control logic and a small table that keeps the information of the instructions of the block. In the design presented in this thesis, the blocks of vector computational instructions are formed dynamically in a very simple way: once a computational vector instruction is ready for execution, the control logic examines the next instruction in the issue queue and adds it to the block if it is a vector computational instruction. This process stops when the next instruction in the issue queue is of another type (a scalar or vector memory instruction) or the block table is full.

The number of vector instructions that can be executed in parallel or with chaining using the *OBO* model is restricted by the number of available FUs. *BBE* does not have this limitation, allowing for execution of more vector instructions in parallel. Inherently, more dependent instructions can be chained (scalar bypass logic can be reused) since one vector instruction does not occupy the ALU for all its elements in continuous cycles, and thus it can be interleaved with other instructions using the same ALU. An important advantage of *BBE* over *OBO* or a classic vector unit is the following: while a block of vector computational instructions is under execution, *BBE* allows for the execution of subsequent scalar or vector memory instructions if they are ready for execution and there are free functional units that can execute them. In Figure 4.2 (b), the second *vecload* instruction can start execution just after the *vecsub* started with execution of the first operation.

BBE has some drawbacks as well: the number of vector computational instructions dynamically included in a block is sensitive to the placement of the vector instructions inside vectorized code (e.g. if the second and third loads in Figure 4.2 are moved before *vecadd* and *vecsub*, there will be three vector computational instruction in the block). Smaller blocks present less opportunities for forwarding/bypassing that can be achieved with longer blocks. In order to keep the design simple, the implementation presented in the paper does not allow overlapping of execution between two consecutive blocks, which is a potential solution to alleviate this problem. This means that the first block needs to finish execution before the second block can start. Another limitation is that dealing with multi-cycle instructions and especially groups of dependent instructions with different latencies inside a block requires additional control logic support. In our model, we support execution of multi-cycle instructions as well as execution of dependent instructions with different latencies, but restricted to the case where they are executed on different ALUs. The benefit of *BBE* can be further increased if we can chain vector memory instructions in an efficient way. The next section describes the vector memory unit of the proposed system, and the subsection 4.3.1 presents our proposal to implement chaining from the memory hierarchy.

4.1.2 Vector Memory Unit

The vector memory unit holds and controls the execution of vector memory instructions. There are two tables that hold the necessary information to execute vector memory instructions, as shown in Figure 4.3. The vector memory instruction table (VMIT) keeps information for each instruction (instruction, start address, stride, number of elements, current element and number of completed operations). The vector memory control table (VMCT) controls the exchange of packets with the L1 cache. Each entry in this table has the following fields: instruction ID, packet/request ID and a valid bit. Additional fields are used to identify the corresponding element(s) of the source/destination vector register.

The address generator (AG) performs the address generation for vector memory instructions. The information about the instruction stored in the VMIT, namely the opcode, start address, stride, number of elements and current element, is enough to generate all requests to the cache hierarchy. We can decode the type of vector memory instruction (unit-strided, strided or indexed) and destination/source register from the instruction opcode field. The stride field can hold the stride for strided memory instructions or the ID of a vector register that holds the index vector for indexed memory instructions. We load/store whole cache lines for unit-stride vector memory instructions with a single access.

ACL controls the proper execution of vector and scalar memory instructions. In our model, we support a very simple aliasing policy to limit the complexity of the control logic: a scalar load instruction waits until all older vector stores are finished and vice versa.

4.2 Integrated Design Evaluation

We have extended the gem5 simulator [11] to model an in-order ARM core, a classic vector unit (CVU) and our two models of execution: *OBO* and *BBE*. *CVU* can be seen as a co-processor or accelerator to the scalar core [24]. The *OBO* model is very similar to the ARM's own VFP mode, which reuses FP registers as short vector registers [73], when executing vector computational instructions with FP data. There are still some differences that are discussed in Section 4.5. Simulations

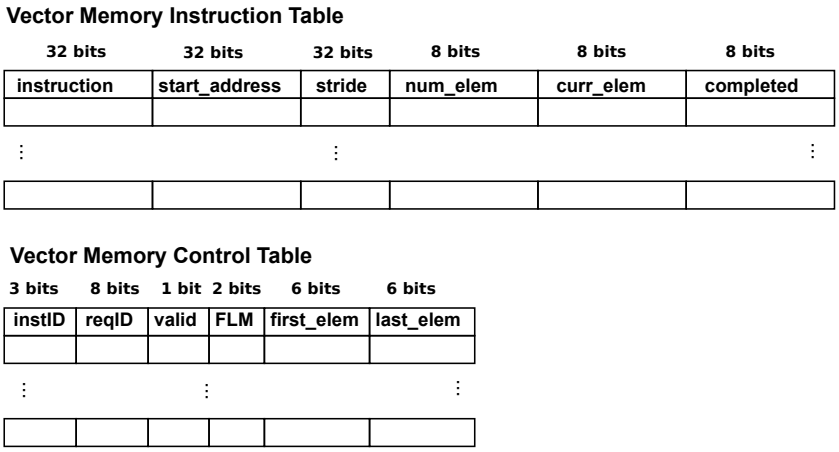


Figure 4.3: Vector memory unit.

Table 4.1: Microarchitectural parameters.

Parameter	Value
Instruction Width	32-bits
L/S Queue	16 entries
VMIT Entries	8 entries
VRF	16 registers
L1 D-Cache	32KB, 64B/line, 4-cycle latency
L2 Cache	256KB, 64B/line, 12-cycle latency
ALU units	One simple, one complex int ALU and one floating-point ALU
L1 D-Cache ports	One read/one write

are performed using system call emulation mode which avoids the need to model devices or an operation system by emulating most system-level services [11].

Table 4.1 summarizes the micro-architectural parameters used in our experiments. *CVU* has two additional integer ALUs (one simple and one complex) and one FP ALU to execute vector computational instructions. The vector register file has 16 vector registers. We use four different MVLs: 16, 32, 64 and 128 elements, with 32 being the default length. Gem5's model of the bus between CPU and L1

Table 4.2: Vectorized kernels.

kernel name	benchmark	access pattern	com/mem ratio	# of vec insts per iteration	loop count	data type	vector/scalar op ratio
sphinx-a	Sphinx3	strided	2:1	12	128	FP	56:1
sphinx-b	Sphinx3	unit-stride	4:3	7	32	FP	10:1
sphinx-c	Sphinx3	indexed	1:1	4	20.5	INT	2:1
saxpy	-	unit-stride	1:1	4	512	FP	53:1
h264ref	h264ref	unit-stride	3:1	4	16	INT	1:2
hmmer	hmmer	unit-stride	11:8	38	256	INT	3:1
graph500	graph500	indexed	1:3	4	28	INT	4:1
facerec	Facerec	strided	17:11	56	25.3	FP	6:1

data cache is extended to model bandwidth and bus contention. We used 16 bytes for the bus width. L2 bus bandwidth is one cache-line per cycle. L2 also has a simple strided hardware prefetcher. *OBO* has a similar configuration to *CVU* except it uses scalar ALUs to compute vector computational instructions. Broadcast logic (including broadcast bus) is also different. The differences between *OBO* and *BBE* are a table that holds vector computational instructions (four instructions) for *BBE* and logic that controls execution of vector computational instructions. We used a latency of one cycle for all instructions that use the simple int ALU, three cycles for the complex int ALU and four cycles for all instructions that use the FP ALU.

We have modified McPAT [50] to evaluate power, energy and area of these micro-architecture variants. We modeled additional structures using the same approach and borrowing the parameters from existing structures in McPAT or CACTI if it is suitable. For example, in order to model a decoder for chaining logic from the memory hierarchy we took a decoder from decode stage with new parameters (e.g. number of bits to compare). We assume a 40nm technology for embedded processor with low operating power for energy, power and area evaluation. We experimented with frequencies of 1 GHz and 2 GHz and results show similar trends with both frequencies.

Table 4.2 lists the kernels that are used to evaluate our design: saxpy micro-kernel, the three most time consuming kernels from Sphinx3, one from H264ref, one from Hmmer, one from Facerec and one from Graph500. The first three benchmarks are from the SPEC2006 benchmark suite. We have chosen these applica-

tions because they represent typical mobile applications: Sphinx3 performs speech recognition, H264ref does video coding while Hmmer feature hidden markov models which are used in machine learning. Facerec is from the SPEC2000 benchmark suite. Even though SPEC benchmarks are typically used to evaluate general purpose processors, applications such as speech recognition or face recognition are widely used in mobiles. The `saxpy` microkernel is not as representative of mobile workloads but we chose it as a a paradigmatic example of a vectorized kernel. Moreover, its simplicity and characteristics help to reason about the results. The Graph500 [56] benchmark is a data intensive, high performance graph processing application but we choose this application because it is highly cache unfriendly and it is a good example to evaluate our ideas for cache unfriendly scenarios. Vectorization potential of the Graph500 is evaluated on Xeon Phi in chapter 3. `sphinx-a`, `sphinx-b`, `saxpy` and `facerec` operate on FP data point while the rest of kernels use integers. Table 4.2 also presents characteristics for each kernel. Our eight kernels exploit several different memory access patterns. Four kernels have a unit-stride, `sphinx-a` and `facerec` have strided while `sphinx-c` and `graph500` have indexed memory access patterns. The ratio between computational and memory instructions varies across kernels as well as the number of vector instructions inside a vectorized loop. Kernels also have different loop counts. There are kernels with very short loop count (`sphinx-c`[1], `facerec`[2], `h264ref`, `sphinx-b` and `graph500`[3]) and kernels with a longer loop count (`sphinx-a`, `saxpy` and `hmmer`). The last column shows the ratio between vector and scalar operations. Most of the kernels have a high percentage of executed operations inside vector instructions (`sphinx-a`, `sphinx-b`, `saxpy`, `sphinx-b` and `facerec`). `sphinx-c`, `hmmer` and `graph500` have between 66% and 80% of all operations executed in vector instructions, while `h264ref` is the only kernel with dominant scalar operations (around 66%). More detailed instruction-level characterization that includes selected benchmarks is performed is chapter 2. We tried to cover as many scenarios and aspects as possible with these eight

[1]This is the average vector length for all iterations. Most of iterations are short, the maximum count is 166.

[2]This is the average vector length for all iterations. Three vector lengths (4, 18 and 64) are repeated in cyclic fashion.

[3]This is the average vector length.

kernels. We extracted the input data from the applications when running the ref input data set and used them to initialize the data structures before simulation.

We extended the ARM ISA with a set of 27 vector instructions that we used to vectorize the kernels. Nine instructions are vector memory instructions: unit-stride load, strided load, indexed load, shape load, unified load, unit-stride store, unit-stride store over vector mask register, strided store and indexed store. There are 11 computational instructions: addition of two vectors, vector-scalar addition, subtraction of two vectors, scalar-vector subtraction, scalar-vector subtraction over vector mask register, multiplication of two vectors, vector-scalar multiplication, comparison of two vectors, vector-scalar comparison, vector-vector select instruction and vector-scalar select instruction. We also implemented two reduction instructions: sum and maximum. There are also three instructions that perform element manipulation. the first instruction performs cast to corresponding data type. The second instruction set the first element of a vector register to a value provided in the first source register and each next element of the vector register is equal to the value of the previous element incremented by value provided in the second source register. The third instruction sets all elements of a vector register to the same value provided in the source register. Finally, we implemented two instructions that set and get value of a register that hold current vector length.

4.2.1 Performance Evaluation

Figure 4.4 shows the speed-ups for *CVU*, *OBO* and *BBE* over the scalar baseline for all kernels. As expected, *CVU* outperforms the integrated designs (*OBO* and *BBE*) for integer data (`sphinx-c`, `h264ref`, `hmmer` and `graph500`) because it has two additional ALUs for computational vector instructions while ALUs are shared between scalar and vector computational instructions in the integrated designs. In the vectorized loops, there are still many scalar instructions for bookkeeping, which then interfere with the execution of vector instructions in the integrated designs. The integrated designs provide good speed-up over the scalar baseline for `hmmer` and `graph500`, while there is little speed-up over the scalar for `sphinx-c`. The main reasons are the use of index memory accesses, the small amount of computational vector instructions and the short vector lengths for the majority of iterations. `h264ref`

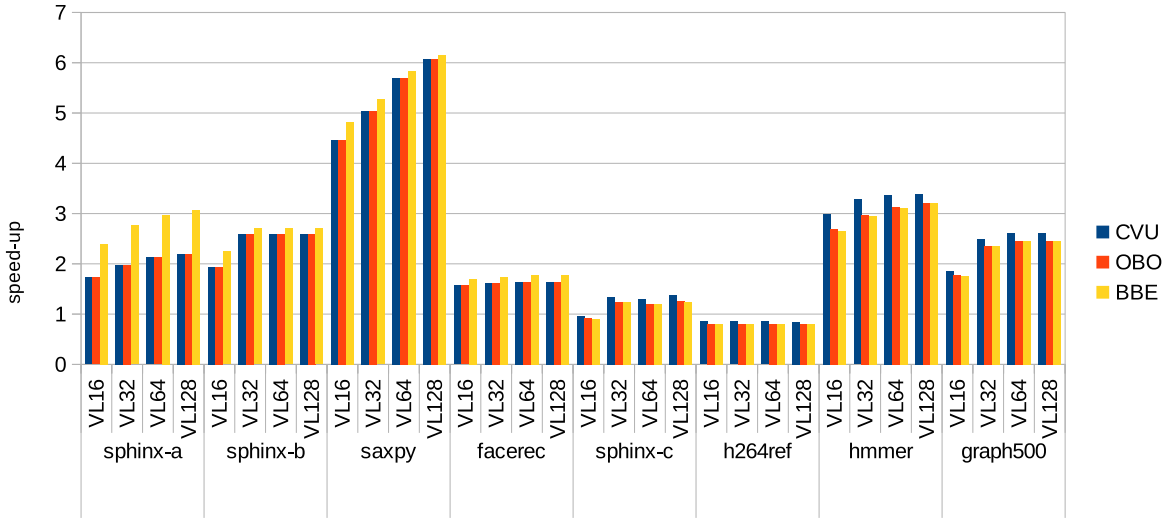


Figure 4.4: Speed-up for CVU, OBO and BBE over the scalar baseline.

is an example where vectorization is inefficient. We obtained slow down over the scalar for all vector models. Despite this kernel has a short vector length (only 16), the main reason for slow down is that in order to vectorize this kernel, we require twice the amount of computational operations than the scalar version.

Regarding FP kernels, vector models are always faster than the scalar. Speed-ups for `saxpy` are extremely high. It is a very simple kernel with a unit-strided memory access pattern and highly regular DLP. A whole cache line can be accessed with one access (16 elements per access) and for this particular experiment the input data set fits into L1 cache. `sphinx-a` and `sphinx-b` have decent speed-ups. `sphinx-a` uses strided vector memory instructions to load data from the cache and it is the limiting factor for higher speed-ups. `sphinx-b` uses unit-stride vector memory instructions but it is limited by the number of iterations in the vectorized loop (only 32 iterations). `facerec` has the smallest speed-up and the main reasons are the presence of strided memory accesses and a short vector length in most of cases (only four or eight). *CVU* and *OBO* have the same execution time (one FP unit) while *BBE* outperforms them due to the advantage of executing subsequent integer scalar (loop overhead instructions) or vector memory instructions in parallel with the current block of vector computational instructions. In-order execution blocks the vector instructions and subsequent instructions in *CVU* and *OBO*, but

this does not happen in *BBE*, as shown above in Figure 4.2 with an example of code with vector instructions executed in *OBO* and *BBE* models of execution. This difference is especially noticeable for `sphinx-a` where the speed-up for *BBE* over *CVU* and *OBO* is around 1.4x for all MVLs. `Sphinx-b`, `saxpy` and `facerec` exploit a benefit from *BBE* execution inside single iteration of a vectorized loop, while `sphinx-a` is a kind of kernel that is able to exploit the benefit across multiple iterations. There is usually a vector memory store instruction at the end of a vectorized loop (`sphinx-b`, `saxpy` and `facerec`). This instruction needs to wait until all elements of a vector source register are computed before it starts with execution. Then we can proceed with the next iteration. `Sphinx-a` does not have a vector memory store instruction in the inner loop and it is able to overlap execution of a vector load instruction in next iteration with vector computational instructions in the previous iteration. For this particular case, vector computational instructions are completely overlapped with vector memory instructions across all iterations of inner loop and it is the main reason for significant speed-up of *BBE* over *CVU* and *OBO*.

Increasing the MVL from 16 to 32 elements provides better execution time for all kernels except `h264ref` (only 16 iterations in the vectorized loop) and `facerec` (in most of cases only four or eight iterations in the vectorized loop). Further increasing MVL to 64 is beneficial for `sphinx-b`, `saxpy`, `hmmer` and `graph500` while vector registers with 128 elements only provide marginal speed-up for `sphinx-a` and `saxpy`. `Sphinx-b` can not exploit the benefits of longer MVLs than 32 because the vectorized loop has only 32 iterations. `Sphinx-c` is limited by short loop counts and does not scale with longer MVLs.

We also performed a sensitivity analysis using several different latencies for vector arithmetic instructions. For example, we used four different latencies for vector arithmetic FP instructions: four, five, six and eight cycles. If we increase latency from four to eight cycles, speed-up over scalar baseline is decreases by 5.9% using MVL 16 in `sphinx-a`, while this number is only 1.1% MVL 128. Results are similar for other kernels.

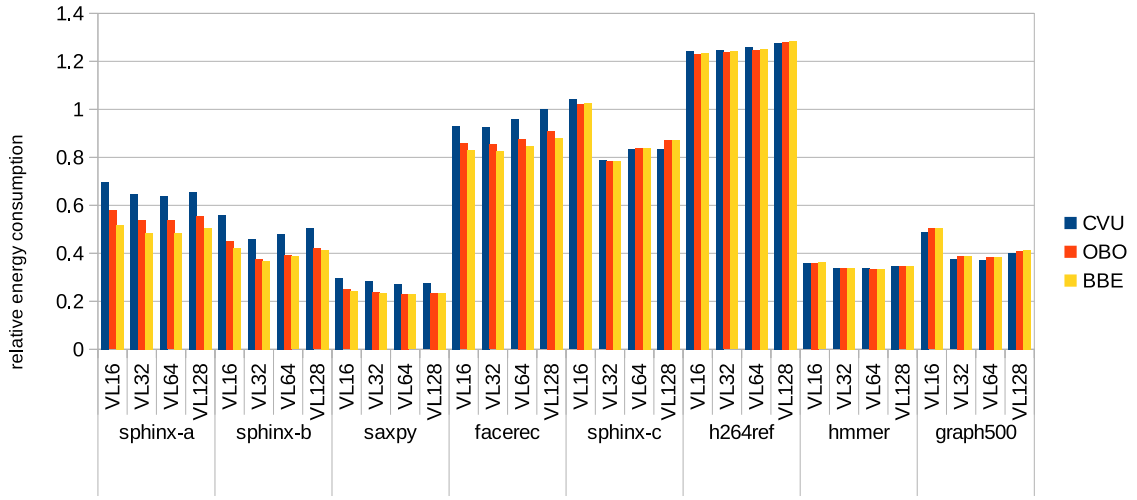


Figure 4.5: Normalized energy consumption for CVU, OBO and BBE over the scalar baseline.

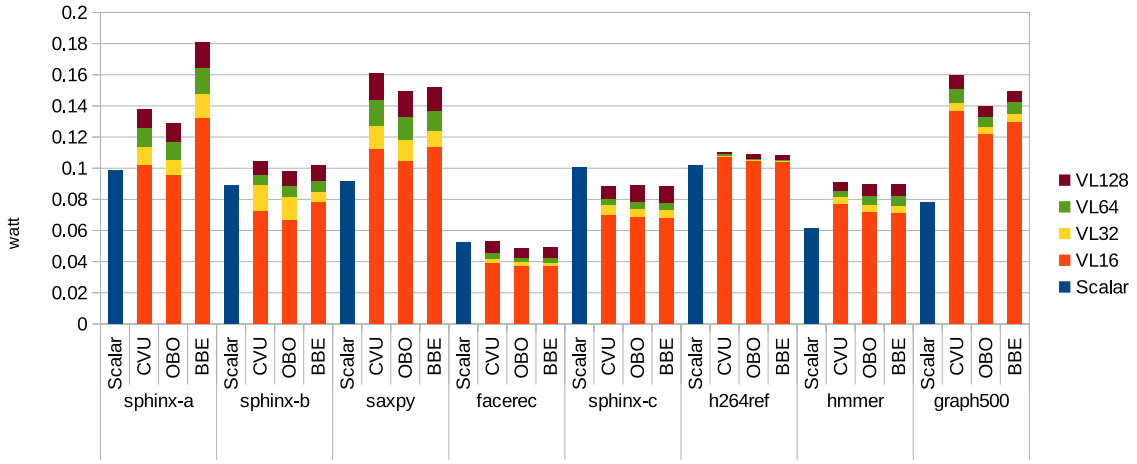


Figure 4.6: Dynamic power.

4.2.2 Area, Power and Energy

Table 4.3 shows the area and leakage numbers for the scalar baseline and the *CVU*, *OBO* and *BBE* models of execution. We present results for vector registers with 16 elements. The area is computed in mm^2. Leakage is presented in watts.

The area overhead of *OBO* and *BBE* is only 4.66% compared to the scalar baseline (it is around 3.3% for vector registers with 16 elements and it goes up to 12.2%

Table 4.3: Area and leakage.

	Scalar	CVU		OBO/BBE
		w/o FP	w/ FP	
Area [mm^2]	2.831	3.065	4.040	2.925
Leakage [W]	0.057	0.069	0.085	0.060

for vector registers with 128 elements). *CVU* without a FP unit increases the area of the baseline for 9.6%. When we include a FP unit, the area overhead of adding *CVU* is significant, around 44% with vector registers of 32 elements.

Energy consumption of *CVU*, *OBO* and *BBE* is presented in Figure 4.5 and it is normalized to the scalar baseline. Energy consumption is significantly lower than in the baseline for `sphinx-a`, `sphinx-b`, `saxpy`, `hmmer` and `graph500` (kernels with decent or high speed-ups over the scalar), showing how adding a vector unit is an energy-efficient way to increase performance. As can be expected, the exceptions are `facerec`, `h264ref` and `sphinx-c` due to slow-down or small speed-up. Energy consumption for `sphinx-c`, `h264ref`, `hmmer` and `graph500` is very similar for *CVU*, *OBO* and *BBE*. Regarding FP kernels, *OBO* and *BBE* clearly outperform *CVU*. *BBE* is also better than *OBO* for `sphinx-a`, `sphinx-b` and `facerec`. If we consider different MVLs for vector register and energy consumption, Figure 4.5 shows that 32 elements is the optimal size for the vector register.

Figure 4.6 shows dynamic power for scalar, *CVU*, *OBO* and *BBE*. Results are stacked for vector models using four different MVLs. *OBO* and *BBE* have slightly lower dynamic power than *CVU* for integer kernels. For FP kernels, *OBO* has always lower dynamic power than *CVU*, while *BBE* has the highest dynamic power for `sphinx-a`. This is a direct consequence of the speed-up achieved with *BBE* over *CVU* and *OBO* (Figure 4.4) combined with its larger reduction of consumed energy (Figure 4.5). For the rest of kernels this difference is smaller and we observed that the most power-consuming model changes with the MVL. *OBO* and *BBE* also have lower dynamic power than the scalar baseline for `sphinx-c` and `facerec` for all MVLs and `sphinx-b` with shorter MVLs (16 and 32) due to significant savings in the front-end of processor (instruction fetch and dispatch).

This evaluation indicates that if only performance is important and only integer data is used then *CVU* should be chosen. If area or power are important then one of our proposed integrated models can be a good match. For FP kernels, our *BBE* model is clearly the best choice from the performance, area and energy consumption perspectives.

4.3 Advanced Integrated Design

A basic integrated design featuring the block-based execution (*BBE*) is presented and evaluated in previous Sections. In this Section, we present four techniques that improve this design. The techniques have negligible area overhead and increase performance or reduce energy and/or power with respect to the basic integrated design.

4.3.1 Chaining from the Memory Hierarchy

Chaining from vector memory instructions was a feature typical implemented in classic vector supercomputers [69, 71]. In those systems, the vector processor accessed main memory directly. The lack of a cache hierarchy made memory access time completely predictable, so given an instruction it was simple to determine exactly in which cycle every element would arrive from memory. Therefore, additional control logic to support chaining from vector memory instructions did not require substantial resources. The first issues arose with vector microprocessors and cache hierarchy. Due to the nature of the cache hierarchy and cache misses it was difficult to predict when the requested data will be arrive from the cache. Processor architects realized that supporting chaining from the memory hierarchy is expensive and required complex control logic, and did not implement this feature in vector microprocessors (e.g. [42]).

We consider that chaining from cache can be particularly fruitful for our *BBE* model due to the opportunity to chain larger number of vector arithmetic instructions in a block. This technique works not only for *BBE*, but for *OBO* and *CVU* as well. Therefore, we decided to design and evaluate support for this feature. Figure 4.7 shows the logic and structures we propose to allow chaining from the

memory hierarchy to one vector register. The main idea is to track the last written element (*lastWritten*) of a vector register, and use it in the VECL (the Vector Execution Control Logic, as explained in Section 4.1). Then, the VECL can determine if the current vector operation is ready for execution, simply by comparing the last written element and the current operation (*currElem*): if *lastWritten* is greater than or equal to *currElem* then VECL can execute the current vector operation. Otherwise it needs to wait until *currElem* is written in the vector register and *lastWritten* updated. The most complex operation of this design is updating the *lastWritten* register. Due to the presence of caches, the elements of a vector load can arrive out of order to the vector register, so we need to track all elements that have arrived. To this purpose, we added a bitmap. The size of the bitmap is equal to the MVL of the vector registers. In order to decode which element is written, a decoder is added. Subsequently, a priority encoder is used to determine the new value of the *lastWritten* register: the position of an element in the vector register, in which all prior elements are written already, as well as itself, and its immediate successor is not written yet.

Figure 4.8 illustrates an example of how the *lastWritten* register is updated. Current state is presented in Figure 4.8 (a). The *lastWritten* register has value two because, as can be seen in the bitmap, elements 0 and 1 of the vector register are already written while element 3 has still not arrived from the cache (i.e. element 0 and element 1 have value one and element 2 has value zero in the bitmap). Figure 4.8 (b) shows what happens when element 2 in the vector register is written and two is the input of the decoder to update the *lastWritten*. The decoded value is *00100000* and the corresponding element in the bitmap is set to one (the light gray box). Then the content of the bitmap is send to the input of the priority encoder. Value five is generated since it is the last consecutive one in the bitmap (dark gray box) before the first zero. Therefore, five is written to the *lastWritten*.

In an initial design, we first assume that each vector register has dedicated chaining logic. This design tracks the last written element for each vector load even though there is no ready dependent computational vector instruction that can be chained. This decision leads to high area, power and energy overheads. Aiming to decrease the number of chaining elements and track the last written element only for vector loads that can be chained from, we also propose a restricted

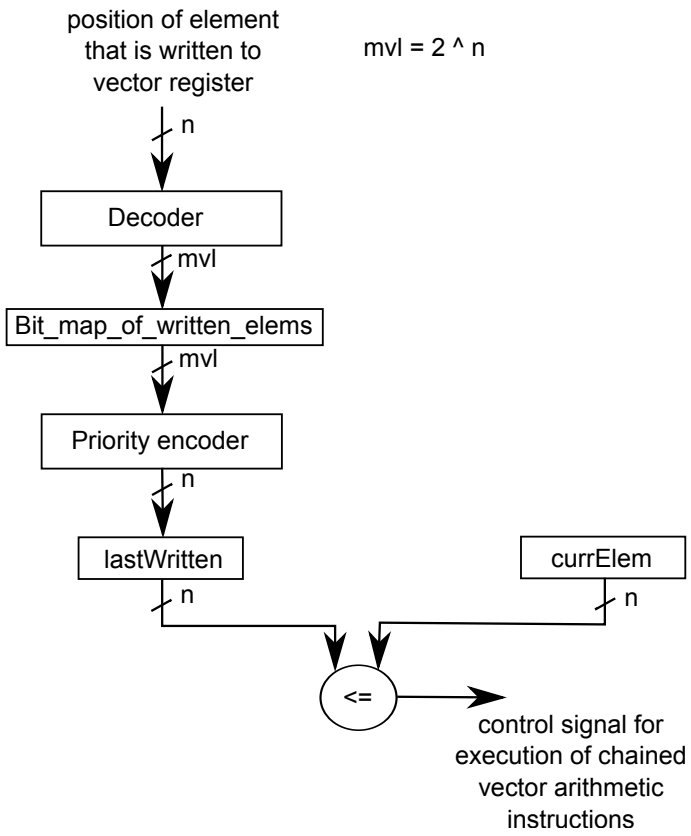


Figure 4.7: Chaining from memory hierarchy.

chaining from memory with few chaining elements. In this design, the chaining control logic needs to know if it will track the last written element for a vector load, i.e. if there is any dependent computational vector instruction that is ready for execution. Since there is enough time to resolve if there is any ready dependent instruction between the moment a vector load starts execution and the moment the first element arrives to a vector register, we decided to use this as a requirement to apply chaining from memory. This means that a dependent instruction will be chained only if it is in the issue queue and ready for execution before the first element arrives to the vector register. Otherwise, the dependent instruction stalls until the vector load is completed.

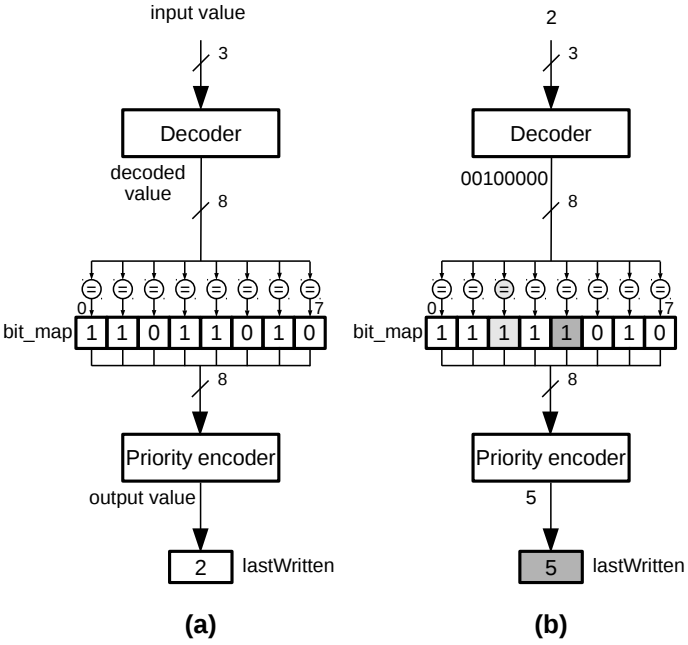


Figure 4.8: An example of how to update the *lastWritten* register for chaining from the memory hierarchy.

```
vecloadst  VR2
vecloadst  VR3
vecsubsv   VR4, scalar, VR3
vecmul     VR5, VR4, VR4
vecmul     VR6, VR5, VR2
vecsub     VR7, VR7, VR6
```

Figure 4.9: A sequence of vector instructions where writing to the vector register file can be avoided.

4.3.2 Direct Forwarding

While we are vectorizing our evaluated kernels, we realized that there are cases in which data are computed and then used only once. Figure 4.9 shows an example from Sphinx3 kernel. Instruction *vecsubsv* stores the result of its computation in vector register *VR4* and only the subsequent instruction *vecmul* uses it as input operand. The scenario is the same for vector registers *VR5* and *VR6*. They are only used once as input operands by subsequent instructions. Therefore, if these

Table 4.4: Possible reduction in number of writes and reads to/from the vector register file.

kernel name	current		with forwarding	
	reads	writes	reads	writes
sphinx-a	92	82	42	41
sphinx-b	7	6	3	3
sphinx-c	3	3	3	3
saxpy	4	4	3	3
h264ref	3	3	2	2
hmmer	24	17	10	9
graph500	4	2	4	2
facerec	72	48	60	36

instructions are chained and we can take advantage of the fact that they are read only once, we could save three (x VL) writes and three reads (x VL) to/from the vector register file in this particular example. Moreover, forwarding from vector memory instructions we could avoid writes and reads to vector registers *VR2* and *VR3* as well. It means that we would only need to read data from vector register *VR7* and to write the final result there.

To further analyze the potential applicability for result forwarding without writing to the vector register file (direct forwarding), we examined all vectorized kernels. Table 4.4 shows how many vector registers are read or written per iteration (we just counted one access per vector register of each instruction, not VL times). These numbers show that we can reduce the number of reads and writes by more than two for `sphinx-a`, `sphinx-b` and `hmmer`, but there are also kernels that can not benefit from this technique.

Since there is already logic for scalar result forwarding/bypassing we decided to reuse it with small additional logic that will allow for direct forwarding. The key idea is to somehow identify if the result of an instruction is used in only one subsequent instruction. In many situations, the compiler can relatively easy identify the previously mentioned case and annotate that instruction (e.g. a bit

set to one). Forwarding logic detects if the bit is set to one and forwards results to corresponding FU without writing to the vector register. The consequence of this approach is that the ISA must be extended to allow for annotating of vector instructions.

Direct forwarding can be applied to all three models: *CVU*, *OBO* and *BBE* but it is much more suitable for *BBE* because it allows for execution of larger number of vector computational instructions in parallel and therefore, it increases the benefit of direct forwarding. In *CVU* and *OBO* it is only useful for chained instructions, which depends on the number and types of the instructions executed.

4.3.3 Vector Memory Shape Instruction

Even though `facerec` is highly vectorizable, we did not obtain high speed-ups over the scalar baseline (up to 1.77x for *BBE*). The main reason is that it performs a complex memory access pattern to a matrix. Elements are loaded from the matrix using three memory access patterns that are repeated in a cyclic fashion. Each memory pattern loads 64 elements. In the first pattern, 64 elements are loaded using a single strided vector memory load instruction with stride two. In the second pattern, we need four strided vector load instructions with stride two to load all 64 elements while in the third pattern we need 16 instructions (each one loads only four elements). We realized that there is a regularity in the accesses for the second and third case that cannot be expressed with strided memory instructions, but it would be possible to load all the elements by providing a more complex vector memory instruction that supports this pattern: we found in the literature the "vector memory shape instruction" [15]. Vector memory shape instruction uses a base address and three scalar values: stride, span and skip to describe a vector. Stride has the same role like in strided vector memory instructions (spacing between each accessed element). Span describes how many elements to access at stride spacing before applying the second-level skip offset. Memory shape instructions can be called 2-D strided vector memory instruction because they are an extension of stride instructions to 2-D patterns.

In order to support the execution of vector memory shape instructions, we slightly modified the vector memory unit (Figure 4.3). VMIT is extended with

span (8 bits), skip (32 bits) and a field that counts the number of elements before skip is applied (8 bits). An additional circuitry is added to increment the third field and compare with span. AG is also extended to increment an address for stride or skip depending on the output of the previously mentioned comparator.

4.3.4 Unified Indexed Vector Load

We achieved decent speed-up for `graph500` even though it is data-intensive, cache unfriendly application. In the first part of the BFS kernel, algorithm loads all neighboring nodes of the current node and checks if they are visited (see section 3.2.1). In the current implementation we used three vector instructions to perform this task: 1) unit-stride vector load gets indices, 2) indexed vector load gets neighboring nodes, and 3) compare vector-scalar instruction checks if they are already visited. Since we already implemented chaining from memory instructions, we were wondering what will be benefit for `graph500` if we somehow merge first and second instructions. The idea is to have single instruction and once we receive data from data cache for indices (index values), automatically initiate accesses for indexed vector load instruction (operations that correspond to loaded indices - final values). Loading an array and use it as index vector is also used in other applications.

We profiled `graph500` using 3,000 nodes and 50,000 edges as input parameters for graph and three different cache configurations: small (L1 8KB and L2 128KB), medium (L1 16KB and L2 256KB) and big (L1 32KB and L2 256KB). Our findings were that unit-stride vector load misses a lot for all cache sizes (2/3 of all requests misses L1 data cache) while miss rate for indexed vector load depends on number of nodes and cache size (as table 4.5 shows). The desired scenario for us would be if one of the requests of the unit-stride vector load hits L1 data cache and the corresponding elements of the indexed vector load miss L1 data cache. In this particular case we will receive earlier values for indexed vector load compared to the current implementation where indexed vector load needs to wait until unit-stride vector load is finished. Figure 4.10 shows a simplified example how elements are loaded from the memory system using unit-stride and indexed vector load (a) and unified vector load (b). Requests to load index values are blue squares and loaded index values are blue cycles. Requests to load final values are red

Table 4.5: Number of L1 data cache misses for unit-stride and indexed vector loads.

cache size	unit-stride load		indexed load	
	hits	misses	hits	misses
small	2254	4029	65970	1361
medium	2226	4057	56325	11006
big	2223	4060	33776	33555

squares and loaded final values are red cycles. Lets assume that we have the desired scenario for the second element: request for the index value hits L1 while request for the final value misses L1. The first approach that uses unit-stride and indexed vector load (Figure 4.10 (a)) has to wait with sending requests for final values until all index values are loaded, while in the case of the unified vector load requests for final values are initiate immediately after the index values are received (Figure 4.10 (b)). Therefore, we can see in the presented example that the second final value is loaded much earlier using the unified vector load as well as all final values. If we assume that a hit to L1 is four and miss twelve cycles, all elements will be loaded in 33 cycles in the first case, while we will need 22 cycles with the unified vector load. We further profiled `graph500` and we found that there is always at least one request inside unit-stride vector load that hits L1 data cache. We can start with indexed load accesses for those cases while waiting for the rest. Our profiling results shown that it could be beneficial to implement unified indexed vector load instruction.

We had several dilemmas regarding implementation of unified indexed vector load instruction. The initial idea was to add hardware that will detect packets with indices (when the unit-stride vector load), buffer them and initiate accesses for indexed vector load at the cache side. We realized that we will need to communicate with the TLB in some cases (index value is higher than the TLB page size) in order to perform the address translation. We also realized that the indexed vector loaded by unit-stride vector load is used by several subsequent vector instructions and that we will need to transfer those values to the vector register file. Therefore, we decided to move additional hardware in the vector memory unit (see subsection 4.1.2) where most of the existing hardware can be reused to support execution of unified indexed vector load. If the index values are just used in the unified instruction then

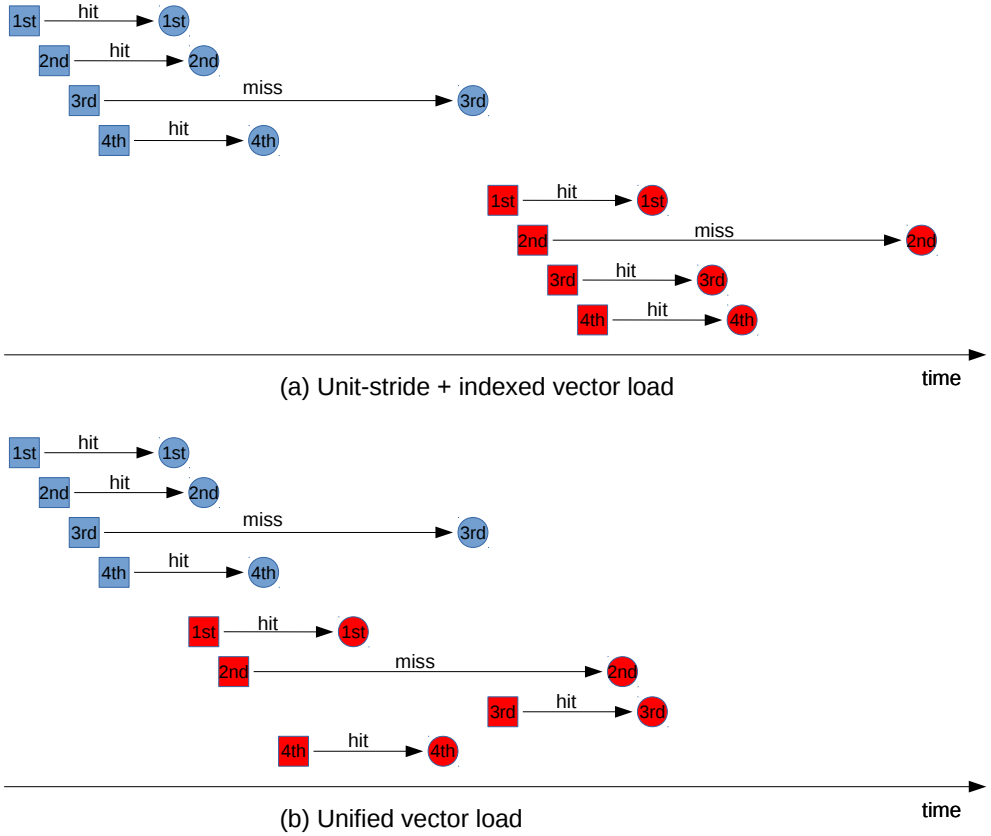


Figure 4.10: An example that shows execution of (a) unit-stride and indexed vector load and (b) unified vector load.

moving hardware to the data cache size could be a good design option.

Once the unified indexed vector load instruction is dispatched to the vector memory unit, two entries in VMIT are reserved. The first entry controls loading of index values, while the second entry is responsible for getting final values. As additional hardware we need only two buffers. The first buffer holds index values that are returned from data cache. Since packets that contain index values can return out of order due to cache misses, we need the second buffer where each value indicates the position in vector register for each index value in the first buffer. As it is explained above, index values are stored in a vector register. Once a corresponding element in vector register is selected to be written, the ID number is written in the second buffer. This information is important for the second part of the unified instruction in-order to write final values on the right position in the destination

vector register. The size of buffers is equal to the size of vector registers.

Our current implementation supports forwarding between older stores and subsequent loads based on load addresses. Since we will not have the address of the final value until we receive corresponding index value from the data cache, we decided to apply a conservative approach during the execution of the unified instruction. Our unified instruction is waiting all older vector stores to complete it before starts execution. In our particular kernel `graph500`, it is possible that there is a conflict between older vector stores and subsequent vector loads. It means that subsequent vector loads could access a value that is written by older vector stores. Therefore, our conservative approach is required. If we are sure that there is no collision between stores and loads, a weak ordering mechanism as in the latest NEC SX-ACE machine could be applied.

4.4 Advanced Integrated Design Evaluation

In this Section we evaluate the advanced integrated design and present results for the techniques described in the previous Section.

4.4.1 Chaining from the Memory Hierarchy

Figure 4.11 presents the speed-ups achieved with our two proposed approaches over the scalar baseline: Full Chaining Support (FCS) and Restricted Chaining Support (RCS), while Figure 4.12 presents the speed-ups achieved over the same models of execution without chaining. We used 16 chaining elements for FCS and four for RCS.

There are several interesting points in these figures. The difference in speed-up for FCS and RCS is negligible or the speed-up is the same for almost all kernels in all three models, except for `sphinx-a` and `sphinx-b` with MVL 16 for *CVU* and *OBO*. Therefore, if area and power are the most important constraints, RCS can provide good results for the most of kernels.

Each kernel has different trends and we analyze them case by case. FCS provides up to 17% improvement in `sphinx-a` for *CVU* and *OBO*, while *BBE* does not benefit a lot from chaining for this kernel. *BBE* without chaining already has

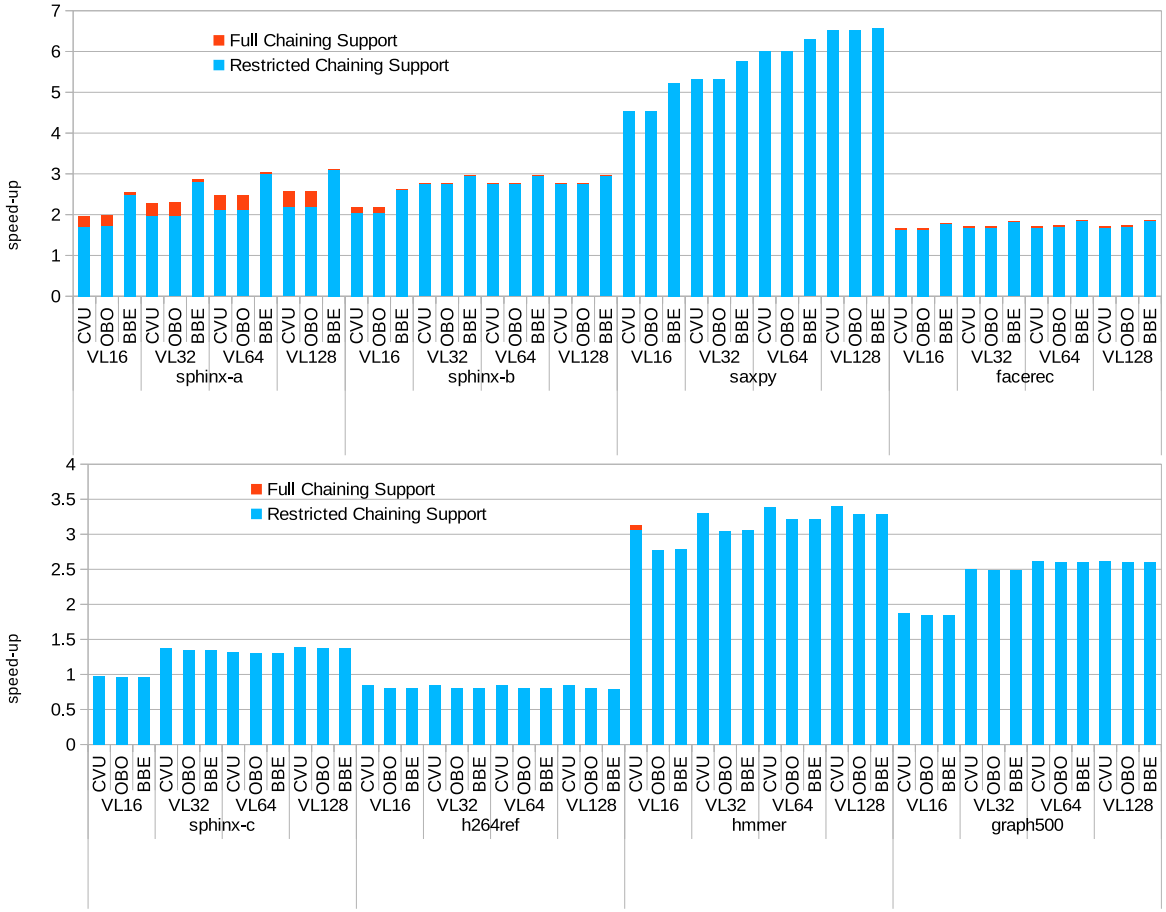


Figure 4.11: Speed-up for CVU, OBO and BBE with full (FCS) and restricted (RCS) chaining support from memory hierarchy over the scalar baseline.

significant speed-up over *CVU* and *BBE* (see subsection 4.2.1). This speed-up is mainly due to overlapped execution of vector memory and computational instructions in *BBE*. Single L1 D-Cache port is already almost fully utilized, and therefore chaining is not able to provide additional significant performance improvements. The speed-up of *BBE* over *CVU* and *OBO* is reduced from 40% to 20% with chaining. `Sphinx-a` does not benefit at all from RCS for *CVU* and *OBO*, while there is small speed-up for *BBE* and shorter MVLs. Applications with strided and indexed vector loads can benefit more from chaining because memory instructions have longer execution time. This is the reason why we obtained the highest speed-up with chaining for `sphinx-a`. *BBE* benefits the most from FCS in `sphinx-b`, around

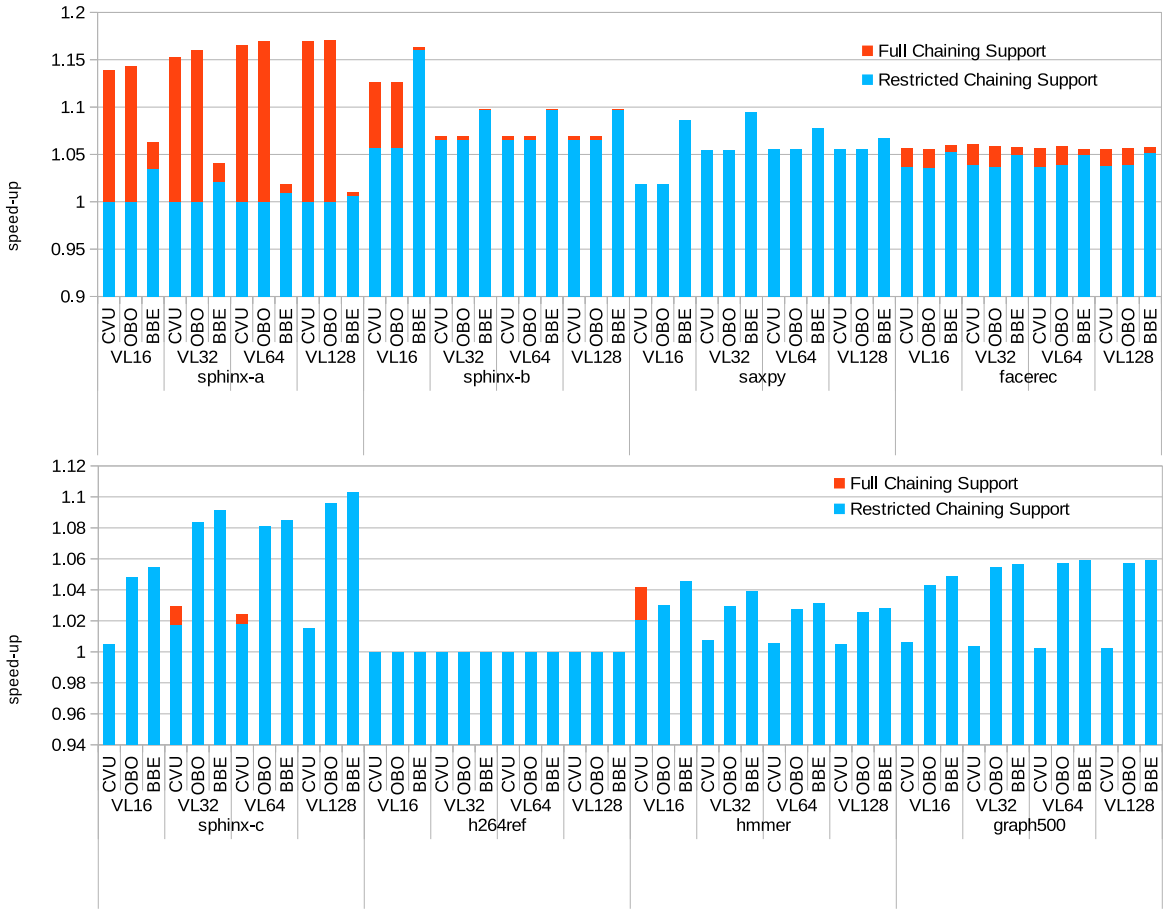


Figure 4.12: Speed-up for CVU, OBO and BBE with full (FCS) and restricted (RCS) chaining support from memory hierarchy over the same models without chaining.

16% for MVL 16. *CVU* and *OBO* also experience improvements higher than 10% and trends are consistent across different MVLs. We knew that unit-stride vector loads are fast (whole cache line per access) but we still expected better results for `saxpy`, speed-up increases around 5% for *CVU* and *OBO*, while *BBE* has little better results, up to 9.5% for MVL 64. The data set for `saxpy` is cache resident, so we decided to perform a test to pollute the L1 data cache before `saxpy` starts computation. The results that are then obtained for *CVU* show 20.5% of improvement for MVL 32. This means that applications with poor cache locality can benefit a lot from chaining from memory hierarchy, similarly to the applications with strided and indexed memory instructions. Table 4.6 shows L1 data cache miss rates for

Table 4.6: L1 data cache miss rates for MVL 32.

kernel name	L1 miss rate in %
sphinx-a	8.0
sphinx-b	25.1
sphinx-c	3.0
saxpy	7.8
h264ref	6.0
hmmer	7.2
graph500	9.8
facerec	17.2

current data input sets and MVL 32. Speed-ups are the same for FCS and RCS in `saxpy`. *BBE* increases the speed-up over *CVU* and *OBO* for `sphinx-b` and `saxpy`. FCS provides around 5% of improvement in `facerec` for all models, while using of RCS slightly decreases the speed-up achieved with FCS.

Even though `sphinx-c` has indexed vector loads, *CVU* does not benefit a lot from chaining mainly due to the use of short vectors, while chaining provides up to 10% improvements for *OBO* and *BBE* with MVL 128. The difference in execution time between *CVU* and *OBO* or *BBE* with chaining is only 1.4% for MVL 128 while it was 9.5% without chaining. `h264ref` does not benefit at all from chaining. The reason is the short vector length of the loop, combined with the unit-stride access. The vector load is able to bring all 16 elements from data cache with one access. Therefore, chaining does not play any role in this case. We also expected higher speed-ups for `hmmer`, but we observed that the execution of consecutive vector computational instructions is serialized. The reason is that a unit-stride vector load is followed by several vector computational instructions that are all of the same type. Since there is only one ALU for that operation that they must share, the first instruction can start execution a few cycles earlier while the rest of instructions still need to wait for the ALU. Adding another ALU could overcome this issue. *CVU* does not benefit from chaining in `graph500` while chaining provides almost 6% of improvement for *OBO* and *BBE*. *OBO* and *BBE* provides almost the same speed-ups

Table 4.7: Total number and number of chained vector loads per iteration.

kernel name	# of vector loads per iteration	# of chained vector loads per iteration
sphinx-a	27	26
sphinx-b	3	2
sphinx-c	2	2
saxpy	2	2
h264ref	1	0
hmmer	14	9
graph500	2	1
facerec	14	4

with chaining as *CVU* for `graph500`.

Table 4.7 shows how often chaining is used. The first column lists kernels. The second column shows number of vector loads per single iteration in vectorized kernel, while the last column shows how many vector loads are chained. We can observe that the chaining from vector loads is quite often used in most of the vectorized kernels, except `h264ref` and `facerec`.

The area overhead of adding chaining logic from the memory hierarchy is presented in table 4.8 and depends on the size of vector register. Area for FCS ranges from 0.4% for MVL 16 up to 2.6% for MVL 128 in *OBO* or *BBE*. It is around 19% of total area overhead for *OBO* or *BBE* for MVL 128. This percentage is lower for *CVU* since it has a larger total area. RCS contributes four times less area overhead and it is around 5% of total area overhead for *OBO* or *BBE* for MVL 128.

Dynamic power contribution of FCS or RCS to the total dynamic power is always lower than 1% for all kernels. Total dynamic power is increased up to 18% for `sphinx-a` in *OBO* for MVL of 128. Relative energy consumption is presented in Figure 4.13. Results are presented as relative energy consumption for CVU, OBO and BBE with FCS and RCS over the same models without chaining support. Regarding energy consumption for FCS, there is up to 6% savings for `sphinx-a` and `saxpy` in *CVU*, while the rest of kernels are in range of 2%. Savings are smaller for RCS in *CVU*, up to 2.8% for `sphinx-b`. Savings are similar for *OBO* and *BBE*. Some

Table 4.8: Area overhead of supporting chaining from the memory hierarchy.

	w/o chaining		w/ FCS		w/ RCS	
	CVU	OBO/BBE	CVU	OBO/BBE	CVU	OBO/BBE
VL16 - Area [mm^2]	4.040	2.925	4.045	2.937	4.036	2.928
VL32 - Area [mm^2]	4.077	2.964	4.094	2.986	4.077	2.969
VL64 - Area [mm^2]	4.158	3.044	4.194	3.086	4.162	3.054
VL128 - Area [mm^2]	4.292	3.178	4.369	3.261	4.307	3.199

kernels have higher energy consumption over models without chaining for higher MVLs and RCS but it is 1.6% in the worst case for `sphinx-a` with MVL 128 in *BBE* and around 7% for `graph500` with MVL 16 in *CVU*.

All these numbers suggest that chaining from the memory hierarchy is fruitful for most of the kernels in terms of performance and energy savings with small area overheads. The contribution of chaining logic to the total dynamic power is also negligible.

4.4.2 Direct Forwarding

We evaluated this technique in terms of energy and power in McPAT and results are presented in Figure 4.14. Obtained results show that we can save more than 50% of energy/power of the vector register file in `sphinx-a`, `sphinx-b` and `hmmer`. `h264ref`, `saxpy` and `facerec` have decent savings between 33% and 16%. Depending on the size of the vector registers, savings at the level of CPU range from 2.5% for short vector registers (16 elements) up to 11% for MVL of 128 elements for `sphinx-b`.

The results presented in this subsection show the benefit of applying this technique between vector arithmetic instructions. It would be possible to achieve even further savings in the vector register file if this technique is combined with chaining from the memory hierarchy to provide direct forward for vector memory instructions as well. This idea is more challenging because it will require more complex logic to implement it and the proposed solution should not affect the performance results.

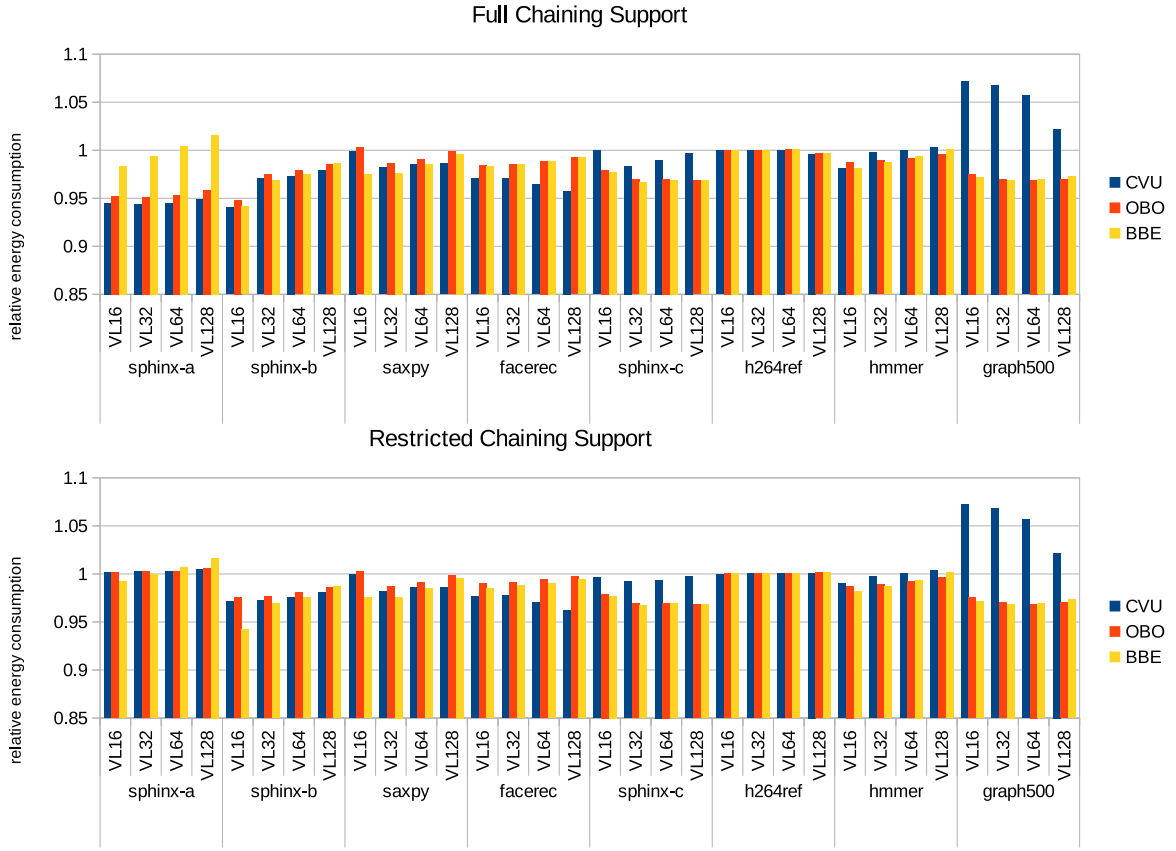


Figure 4.13: Relative energy consumption for CVU, OBO and BBE with full (FCS) and restricted (RCS) chaining support from memory hierarchy over the same models without chaining.

4.4.3 Vector Memory Shape Instruction

Figure 4.15 shows the speed-ups for *CVU*, *OBO* and *BBE* over the scalar baseline for `facerec` with the vector memory shape instruction. We presented results both with and without chaining from the memory hierarchy. The vector memory shape instruction increases speed-up from 1.63x to 2.4x for *CVU* and *OBO* using MVL 64 while the speed-up is increased from 1.77x to 2.66x for *BBE*. Chaining from the memory hierarchy also additionally improves performance (from 2.4x to 2.68x for *CVU* and *OBO*). It is also interesting that performance scales better with increased MVL from 16 to 64 elements compared to the implementation without support for vector memory shape instruction (Figure 4.4). We also modeled the additional

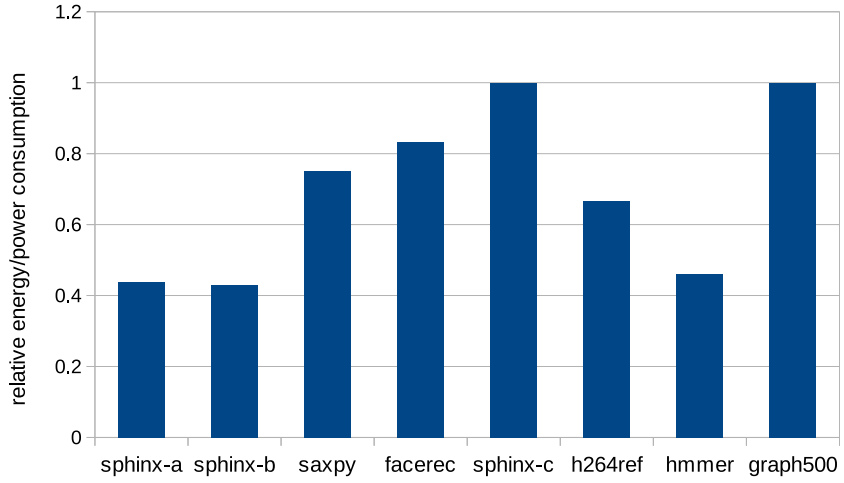


Figure 4.14: Normalized energy/power consumption for BBE model with direct forwarding over the same model without direct forwarding.

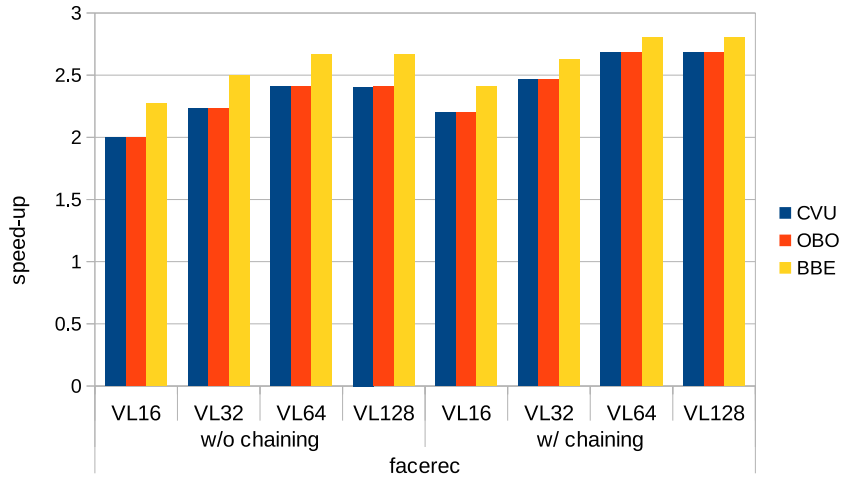


Figure 4.15: Speed-up for CVU, OBO and BBE over the scalar baseline when using vector memory shape instruction.

hardware in McPAT and results show negligible area overheads (less than 0.2%) while energy savings are almost proportional to the achieved speed-up (from 9% for MVL 16 up to 18% for longer MVLs).

4.4.4 Unified Indexed Vector Load

Figure 4.16 shows the speed-ups for the vectorized version of `graph500` using the unified vector load over the version with the indexed vector load. Results are presented with and without chaining from the memory hierarchy for all three models of execution: *CVU*, *OBO* and *BBE*. The unified vector load instruction increases speed-up only up to 1.08x for *CVU* without chaining from memory. Results are similar with chaining from the memory hierarchy. We profiled execution of both vectorized versions and we found that the unified vector load completes execution faster than the indexed vector load. The speed-up per single instruction goes from 1.1x to 1.22x. Since the rest of the kernel is the same, overall speed-up goes only up to 1.08x. We also noticed that loading all neighboring nodes and their current state (visited or not visited) takes a few hundreds cycles (300-400) for 30 neighboring nodes on average. Therefore, the benefit of having single instruction and saving some cycles in the front-end of the processor is also small in the particular example. Area overhead of the additional hardware is small, less than 0.3% in the worst case when two buffers with 128 elements are added. Energy savings are also small and proportional to the speed-ups, less than 4.1%, while dynamic power increases up to 8%.

Even though current results are not promising, the unified indexed vector load could be further improved. For example, if the index vector is used only in the unified vector load instruction, we could avoid writing to the vector register. We also noticed that a cache line that holds only neighbors of one node (index values) is accessed only once. We could avoid caching these lines in L1 or L2 and provide a small victim cache or streaming buffer for those lines. Therefore, L1 and/or L2 caches would not be polluted with cache lines that are used only once.

4.5 Related Work

In this section we discuss differences between our integrated design and several works that integrated vector and scalar processing. We also discuss alternatives to vectors and justify design decisions.

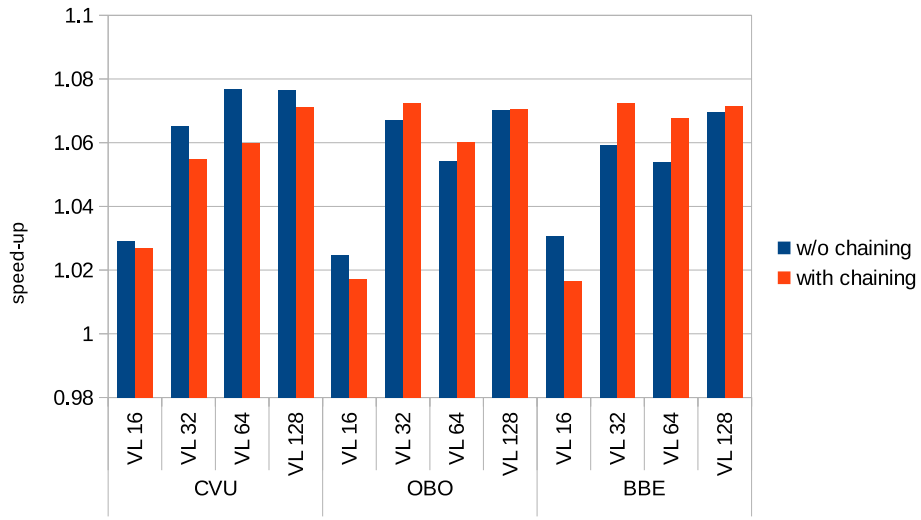


Figure 4.16: Speed-up with the unified vector load over the indexed vector load in `graph500` for CVU, OBO and BBE.

An important characteristic of most microprocessor vector architectures is that the vector processing unit is designed as an extension or co-processor to a scalar core, but there are few works that combine vector and scalar processing on the same substrate. Quintana et al. [65] added a vector unit to a superscalar core. Since their research focused on high performance, this processor design resembles a classic vector with multiple lanes and direct L2 access which is completely different from our design. This work also does not include any evaluation of power and energy of the proposed design, even though their approach requires additional hardware that is idle most of the time in scalar intensive applications. Gebis et al. [26] proposed the first integrated solution that combines scalar and vector processing (ViVA). However, ViVA adds support only for vector memory instructions while regular scalar instructions are used for computation. We have observed the advantages of using vector computation instructions, e.g. the energy reduction due to reduced front-end activity. Soliman [77] proposed a low-complexity vector core that has a common execution data-path for executing scalar/vector instructions, but using the scalar register file limits the design to support only short vector (up to 8 elements). PTX [16] supports vector instructions, but they are transformed to scalar instructions for Nvidia SIMT microarchitectures, which behave like many independent scalar units. This can be seen an instance of using scalar hardware to

execute vector instructions.

Integrated vector-scalar design is also proposed in CELL and power processors [27, 28, 29]. Cell and power processors target high-performance and it is crucial for them to have parallel hardware that can exploit available DLP and provide high performance. Since they realized the importance of having integrated vector-scalar processing, support for scalar execution is added in the parallel hardware (SIMD vector unit). In our work we are targeting the low-end embedded market, where power, energy and area are important design concerns. So we used *the opposite* approach: we added minimal hardware to support execution of traditional vector instructions (not SIMD) on a scalar in-order core.

The Imagine processor [39] implements a somewhat similar model of execution to our *BBE*. They called it compound stream operations. They perform multiple computational operations on each stream element. Our *BBE* model is more flexible, as instructions in the block can be independent. The Imagine targets streaming applications with little data reuse, while our approach is more general. Their model of execution is implemented on specific accelerators while *BBE* works on a general purpose core and does not have any specific constraint. It can use regular vectorized code and execute in a block-based fashion without additional support from the compiler. The SCALE processor [43] also implements a kind of block-based execution at the level of virtual processor (VP). VP instructions (RISC-like) are grouped into atomic instruction blocks (AIBs), which is the unit of work issued to a VP at one time. Albeit the common use of the term “block", our *BBE* blocks are completely distinct from AIBs. In our case, it is a block of vector instructions that is dynamically formed during execution. Like for the Imagine, *BBE* can execute regular vector code without additional compiler support.

We would like to emphasize that we use *traditional vector execution* (Cray-I inspired) in our integrated design rather than SIMD implementations found in commodity processors such as Intel/AMD X86, Cell, Power, etc. SIMD processing units operate in parallel requiring multiple functional units and thus they are less efficient solutions in area and energy for embedded systems. Additionally, even when focusing on high-performance, execution of vector instructions in a pipelined form is still a relevant design point. For example the modern NEC SX-ACE vector processor (2013) has 16 vector pipelines in each core; each pipeline can execute up

to four operations per cycle but each vector register has 256 elements; thus each instruction is pipelined serially in the functional units unlike SIMD-based architectures.

NEON is a SIMD unit incorporated in ARM processors. We predict that area overhead of NEON unit would be similar or higher to CVU in the case of 128-bit SIMD FP unit. Regarding performance, NEON should provide better performance for cache-friendly applications with unit-stride access and short vectors: NEON is able to process two or four operations in parallel, while we chose to provide a single vector lane. The advantages of our design would be apparent in applications with long vectors (due to the smaller code size and reduced dynamic instruction count, resulting in less front-end activity) and irregular memory access patterns - strided or indexed memory accesses (NEON only supports vectors that are stored consecutively in memory and therefore code with indexed or strided memory access pattern cannot be vectorized). Cache-unfriendly applications would also benefit greatly from using long vectors because vector memory instructions are able to hide long latencies.

OBO mode of execution is similar to the ARM's VFP mode [73] for floating-point computation but there are still some differences. Our *OBO* model supports longer vector lengths while VFP is restricted to shorter vector lengths. Also, we provide support for gather/scatter vector memory instructions. We applied chaining for computational vector instructions as well as chaining from the memory hierarchy. Moreover, the execution of vector instructions with integer data is also different, since our model uses scalar ALUs.

Our baseline is an energy-efficient in-order ARM core and we wanted to improve it in terms of energy and performance. Since vector processors are by default energy-efficient [48], we added support for vector instructions. We did not consider adding multiple lanes due to the following reasons: (1) we wanted to do minimal changes to the existing scalar processor in order to keep the area/power envelope and reuse the existing processor resources in the most efficient way; (2) adding one additional lane increases area of scalar baseline by 44%; adding four lanes will more than double the area of the processor and it is not acceptable in highly constrained low-end devices.

Early vector processors had single lane architectures (CRAY-1) and they were successful compared to others at that era such as IBM370. Efficiency was the factor. By analogy, in-order processors are overwhelmingly used in the embedded domain because they are more energy-efficient compared to bulky out-of-order architectures. Recent work demonstrates that single-lane vector units offer significant performance gains over commodity processors [30]. It shows quite clearly that the advantages come from a reduction of fetch/decode/rename/commit and consolidating memory requests together. Our results in the paper also confirmed that we can achieve good speed-ups over scalar baseline.

As we mentioned above, chaining from the memory hierarchy was popular in classic vector machines [69, 71]. Since access time to the memory hierarchy was constant, it was not so difficult to implement it. To the best of our knowledge there is no published work that discusses how to implement chaining from memory hierarchy with caches.

Regarding direct forwarding, it can be seen as a extreme case of short-lived registers [34, 63] because we are using only once produced value. Short-lived registers or values are used only for a short period of time after they are written, meaning that the destination registers targeted by these values are renamed by the time the results are written back. Writing to the register file is avoided by caching in a small dedicated register file [63] or by using a small structure which sits between the functional units and the register file that buffer and filter accesses to the register file [34].

4.6 Summary

In this chapter we proposed the integrated vector-scalar design that mostly reuses scalar hardware of in-order ARM core to support the execution of vector instructions. Our integrated design has several advantages. It has a small area and power overheads (only 4.66% when using a vector register with 32 elements) while at the same time the *BBE* model provides even better performance results (up to 1.4x) than *CVU* for FP data. As a result, not only a significant reduction of energy over the scalar is achieved (up to 5x), as expected due to the energy efficiency of vector architectures, the integrated design also consumes less energy than *CVU* (up to

26% of reduction). Direct forwarding is applied to *BBE* and it provides additional power/energy saving in the vector register file (up to 57%).

5 Conclusion and Future Work

This chapter presents the conclusions of the research done during this thesis work. It also provides suggestions for future research directions.

5.1 Conclusion

Using a vector processor is one of the most energy efficient ways of achieving high performance for a wide number of applications that contain significant DLP. However, even in vector-heavy workloads, the vector execution hardware goes underutilized most of the time and therefore, the vector ALU sits idle, burning static power, being idle a substantial portion of the time.

In this thesis, we proposed the integrated vector-scalar design that allows for execution of vector computational instructions mostly reusing resources of an ARM in-order core. We showed that good performance and energy-efficiency improvements can be achieved with minimal hardware overhead for applications with decent or high DLP. We also showed how additional improvements could be achieved by applying several energy-performance efficient ideas: (1) chaining from the memory hierarchy, (2) direct result forwarding and (3) memory shape and unified instructions. The key advantages are that our integrated design (a) apply vector-scalar processing automatically in hardware, (b) gain the advantages of a vector ISA, and (c) apply the best ideas from prior vector architectures to reduce power and increase performance in a mobile CPU.

We still think it is a good idea to reuse the scalar ALUs to execute vector computational instructions. Even though we did not exploit the benefit of parallel hardware and vector computational instructions, our results in the Chapter 4 confirmed that we can achieve good speed-ups over scalar baseline. This is mainly due to a reduction of instructions in the front-end of the processor (instruction cache, fetch and decode stages) and consolidating memory requests together. We did not consider adding multiple lanes due to the following reasons: (1) we wanted to do minimal changes to the existing scalar processor in order to keep the area/power envelope and reuse the existing processor resources in the most efficient way; (2) adding one additional lane increases area of scalar baseline by 44%; adding four lanes will more than double the area of the processor and it is not acceptable in highly constrained low-end devices.

Even though BBE model of execution did not provide any performance improvement over OBO model of execution, its nature to execute more vector computational instructions in coordinated manner opened a new opportunities to improve performance or energy-efficiency of our integrated design in combination with ideas such as chaining from the memory hierarchy or direct result forwarding. Chaining from the memory hierarchy is more complex than directly from main memory, but we have seen it is possible to implement it without very complex hardware. Direct result forwarding is simple idea but in the combination with BBE can decrease the energy consumption of the vector register file by factor of two. Even additional savings could be achieved if we consider direct result forwarding from vector memory instructions.

Special attention should be paid to the performance of the memory system in in-order architectures. The Knight Corner deals with this using four hardware threads, extensive prefetching and gather/scatter instructions. Beside the use of long vector memory instructions (they are able to hide long memory latencies) in our integrated design, we have seen it is possible to improve performances for some kernels by implementing specific vector memory instructions (the vector shape instruction, gather/scatter, the unified instruction) with simple hardware extension and small area overhead. Instead of using multiple hardware threads like in the Knight Corner to deal with the memory system in in-order architecture, we were able to provide good speed-ups over the scalar baseline by providing support for

long vector memory instructions, applying chaining from the memory hierarchy and implementing specific vector memory instructions.

In this thesis, we also developed two tools that allow for rapid initial research on vector architectures: VALib and SimpleVector. We vectorized six applications using VALib, performed their detailed instruction-level characterization and evaluated several alternative properties of the vector microarchitecture. It helped us to define the set of vector instructions that we implemented in the integrated design as well as to pay special attention to the memory side. It guided us to propose techniques for chaining from the memory hierarchy and special vector memory instructions. Since vector architectures will become even more popular in the future due to their potential for energy-efficient high performance execution, we think that these tools will also help the community to study the potential of vectorization for target applications, characterization of the vectorized code at the instruction level and make a preliminary evaluation on a broad range of vector microarchitectures.

We also performed an evaluation of Knights Corner capabilities that were later useful in making design decisions for our integrated vector-scalar core. Even though the Knights Corner is designed for high-performance computing and it implements SIMD processing, it was very interesting for us because its architecture is in-order. Our evaluation again stressed the importance of the memory system for achieving high performance. In particular case, we achieved good performance for Graph500 when vectorization is combined with prefetching and parallelization. In our integrated design, use of longer vector memory instructions, chaining from the memory hierarchy and unified vector memory instruction provided decent speed-up over a scalar version of Graph500.

5.2 Future Research Directions

Obviously, not all aspects of the integrated design are perfect and there are a few open issues. Even though McPAT is widely used by computer architecture community, it is not as accurate as a design of an architecture at circuit level. Therefore, an interesting direction would be an evaluation of *BBE*'s control logic at the circuit level to accurately study the complexity/power/performance trade-offs of its design. An implementation at the circuit level will determine if the changes to

the hardware affect the critical path and maximum operating frequency. Another concern is that *CVU*, *OBO* and *BBE* are extremely sensitive to the order of the instructions in the program because they are all in-order. This could be partially relieved with decoupling [22], but careful power evaluation should be performed to ensure it does not compromise the savings achieved with the proposed architecture. Since we think that our proposed instructions are valuable, another direction would be to put a lot of effort on compiler support.

We were able to reduce the energy consumption of the vector register file with direct result forwarding but still there is an area overhead of adding the vector register file. The area overhead is more significant if we want to support longer vector register. An alternative would be to use part of a 3D stacked memory to implement the vector register file. It would allow for having support for long vector registers with fast access and hight bandwidth at low cost in therm of area.

Another possible direction would be a research that will investigate how our integrated design will deal with multi-core or multi-thread architecture or out-of-order execution. If there is a good way to provide support for long vectors using 3D stacked memory, an idea would be to support execution of single vector instruction across multiple threads. Since this kind of processor that combines our integrated design with out-of-order execution, multi-core or multi-thread architecture will not most likely target low-end mobile system, it will also open the opportunity of adding multiple lanes.

6 Publications

The work of the thesis has resulted in the following publications.

6.1 Publications from the thesis

- Milan Stanić, Oscar Palomar, Ivan Ratković, Milovan Duric, Osman Unsal, Adrian Cristal, and Mateo Valero, "VALib and SimpleVector: Tools for Rapid Initial Research on Vector Architectures", ACM International Conference on Computing Frontiers, May 2014, Cagliari, Italy.

- Milan Stanić, Oscar Palomar, Ivan Ratković, Milovan Duric, Osman Unsal, Adrian Cristal, and Mateo Valero, "Evaluation of Vectorization Potential of Graph500 on Intel Xeon Phi", International Conference on High Performance Computing & Simulation (HPSC), July 2014, Bologna, Italy.

- Milan Stanić, Oscar Palomar, Timothy Hayes, Ivan Ratković, Osman Unsal, Adrian Cristal, and Mateo Valero, "Towards Low-Power Embedded Vector Processor", ACM International Conference on Computing Frontiers, May 2016, Como, Italy.

- Milan Stanić and Oscar Palomar, "Block-Based Execution on an Integrated Vector-Scalar In-Order Core", BSC International Doctoral Symposium, May 2016, Barcelona, Spain.

- Milan Stanić, Oscar Palomar, Timothy Hayes, Ivan Ratković, Osman Unsal, Adrian Cristal, and Mateo Valero, "POSTER: An Integrated Vector-Scalar Design on an In-order ARM Core", In Proceedings of the 2016 International Conference on Parallel Architectures and Compilation (pp. 447-448), September 2016, Haifa, Israel.

- Milan Stanić, Oscar Palomar, Timothy Hayes, Ivan Ratković, Osman Unsal, Adrian Cristal, and Mateo Valero, "An Integrated Vector-Scalar Design on an In-order ARM Core", In submission at ACM Transactions on Architecture and Code Optimization (TACO).

6.2 Related publications not included in the thesis

- Ivan Ratković, Oscar Palomar, Milan Stanić, Osman Unsal, Adrian Cristal, and Mateo Valero, "On the Selection of Adder Unit in Energy Efficient Vector Processing", Proceedings of the 2013 The International Symposium on Quality Electronic Design (ISQED), March 2013, Santa Clara, USA.

- Ivan Ratković, Oscar Palomar, Milan Stanić, Osman Unsal, Adrian Cristal, and Mateo Valero, "Design of Energy-Efficient Adder Units for Vector Processors", Proceedings of the 2013 Advanced Computer Architecture and Computation for Embedded Systems (ACACES), July 2013, Fiuggi, Italy.

- Ivan Ratković, Oscar Palomar, Milan Stanić, Osman Unsal, Adrian Cristal, and Mateo Valero, "Physically vs. Physically-Aware Estimation Flow: Case Study of Design Space Exploration of Adders", In Proceedings of IEEE Computer Society Annual Symposium on VLSI (ISVLSI), July 2014, Tampa, USA.

- Milovan Duric, Oscar Palomar, Aaron Smith, Milan Stanic, Osman S. Unsal, Adrian Cristal, Mateo Valero, Doug Burger, Alexander V. Veidenbaum, "Dynamic-vector execution on a general purpose EDGE chip multiprocessor", International Conference on Embedded Computer Systems: Architectures, Modeling and Simulation (SAMOS), July 2014, Samos, Greece.

- Ivan Ratković, Oscar Palomar, Milan Stanić, Milovan Duric, Djordje Pešić, Osman Unsal, Adrian Cristal, and Mateo Valero, "Joint Circuit-System Design Space Exploration of Multiplier Unit Structure for Energy-Efficient Vector Processors", In Proceedings of IEEE Computer Society Annual Symposium on VLSI (ISVLSI), July 2015, Montpelier, France.

- Milovan Duric, Milan Stanić, Ivan Ratković, Oscar Palomar, Osman Unsal, Adrian Cristal, and Mateo Valero, "Imposing Coarse-Grain Reconfiguration to General Purpose Processors", International Conference on Embedded Computer Systems: Architectures, Modeling and Simulation (SAMOS), July 2015, Samos, Greece.

- Ivan Ratković, Oscar Palomar, Milan Stanić, Osman Unsal, Adrian Cristal, and Mateo Valero, "A Fully Parameterizable Low Power Design of Vector Fused Multiply-Add Using Active Clock-Gating Techniques", 2016 IEEE/ACM International Symposium on Low Power Electronics and Design (ISLPED), August 2016, San Francisco, USA.

Appendices

A Examples of Graph500 Codes

In this appendix we present different versions of Graph500 implementations used in chapter 3. All versions use compress sparse row (CSR) representation for an input graph. It includes the following implementations of Graph500:

- Original sequential version.
- Vectorized sequential version.
- Sequential version with scalar prefetching.
- Vectorized sequential version with vector prefetching.
- Vectorized sequential version with vector and scalar prefetching.

A.1 Original Sequential Version

```
static int64_t * restrict xoff; /* Length 2*nv+2 */
#define XOFF(k) (xoff[2*(k)])
#define XENDOFF(k) (xoff[1+2*(k)])

int
make_bfs_tree (int64_t *bfs_tree_out,
               int64_t *max_vtx_out,
               int64_t srcvtx, int64_t nv_scale)
```

```
{
  int32_t * restrict __attribute__((target(mic))) \
                                  bfs_tree = bfs_tree_out;
  int err = 0;
  int64_t * restrict vlist = NULL;
  int64_t k1, k2;

  *max_vtx_out = maxvtx;

  vlist = xmalloc_large (nv * sizeof (*vlist));
  if (!vlist) return -1;

  for (k1 = 0; k1 < nv; ++k1)
    bfs_tree[k1] = -1;

  #pragma offload target(mic:MIC_DEV) \
            in(xoff : length(2*nv+2)) \
            in(xadj : length((XOFF(nv)))) \
            inout(vlist : length(nv)) \
            inout(bfs_tree : length(nv_scale) \
            alloc_if(1) free_if(1))
  {
    vlist[0] = srcvtx;
    bfs_tree[srcvtx] = srcvtx;
    k1 = 0; k2 = 1;
    while (k1 != k2) {
      const int64_t oldk2 = k2;
      int64_t k;
      for (k = k1; k < oldk2; ++k) {
        const int64_t v = vlist[k];
        const int64_t veo = XENDOFF(v);
        int64_t vo;
        for (vo = XOFF(v); vo < veo; ++vo) {
          const int64_t j = xadj[vo];
          if (bfs_tree[j] == -1) {
            bfs_tree[j] = v;
```

```
            vlist[k2++] = j;
          }
        }
      }
      k1 = oldk2;
    }
  //offload end
  }

  xfree_large (vlist);

  return err;
}
```

A.2 Vectorized Sequential Version

```
[
    basicstyle=\small,
]
static int64_t * restrict xoff; /* Length 2*nv+2 */
#define XOFF(k) (xoff[2*(k)])
#define XENDOFF(k) (xoff[1+2*(k)])

int
make_bfs_tree (int32_t *bfs_tree_out,
               int64_t *max_vtx_out,
               int64_t srcvtx, int64_t nv_scale)
{
  int32_t * restrict __attribute__((target(mic))) \
                                bfs_tree = bfs_tree_out;
  int err = 0;

  int32_t * restrict vlist = NULL;
  int32_t kk1, kk2;

  *max_vtx_out = maxvtx;

  vlist = xmalloc_large (nv * sizeof (*vlist));
  if (!vlist) return -1;
```

```
  #pragma offload target(mic:MIC_DEV) \
              in(xoff : length(2*nv+2)) \
              in(xadj : length((XOFF(nv)))) \
              inout(vlist : length(nv)) \
              inout(bfs_tree : length(nv_scale) \
              alloc_if(1) free_if(1))
  {

    for (kk1 = 0; kk1 < nv; ++kk1)
      bfs_tree[kk1] = -1;

    vlist[0] = srcvtx;
    bfs_tree[srcvtx] = srcvtx;
    kk1 = 0; kk2 = 1;
    while (kk1 != kk2) {
      const int32_t oldk2 = kk2;
      int32_t k;
      for (k = kk1; k < oldk2; ++k) {
        const int32_t v = vlist[k];
        const int32_t veo = XENDOFF(v);
        int32_t vo, i;

#ifdef __MIC__

        int32_t int_mask, cnt;

        vo = XOFF(v);
        if (veo - vo >= 16){
          // set vector zmm3 to -1
          __m512i zmm3 = _mm512_set1_epi32(-1);
          // set vector zmm4 to v
          __m512i zmm4 = _mm512_set1_epi32(v);

          for ( ; vo <= (veo-16); vo+=16){
#ifdef ALIGNED_ACCESS
            //load connected nodes from xadj
            __m512i zmm1 = _mm512_load_epi32( &(xadj[vo]));
#else
            __m512i zmm1 = _mm512_loadunpacklo_epi32 \
            (zmm1, &(xadj[vo]));
```

```
            zmm1 = _mm512_loadunpackhi_epi32 \
            (zmm1, &(xadj[vo+16]));
#endif
            // gather elements from bfs_tree
            __m512i zmm2 = _mm512_i32gather_epi32 \
            (zmm1, bfs_tree, 4);

            __mmask16 k1 = _mm512_cmpeq_epi32_mask(zmm2, zmm3);
            int_mask = _mm512_mask2int(k1);
            cnt = _mm_countbits_32(int_mask);
            if (cnt) {
              // scatter store over mask register to bfs_tree
              _mm512_mask_i32scatter_epi32 \
              (bfs_tree, k1, zmm1, zmm4, 4);

              // store over mask to vlist
              _mm512_mask_packstorelo_epi32 \
              (&(vlist[kk2]), k1, zmm1);
              _mm512_mask_packstorehi_epi32 \
              (&(vlist[kk2+16]), k1, zmm1);
              kk2 += cnt;
            }
          }
        }

#ifdef VECT_REMAINING_PART
        int32_t temp_mask;
        int32_t rem = veo - vo;
        if (rem > REMAINING_PART){
           switch(rem){
             case 15: temp_mask = 0b111111111111111;
                      break;
             case 14: temp_mask = 0b11111111111111;
                      break;
             case 13: temp_mask = 0b1111111111111;
                      break;
             case 12: temp_mask = 0b111111111111;
                      break;
             case 11: temp_mask = 0b11111111111;
                      break;
             case 10: temp_mask = 0b1111111111;
```

```
                    break;
            case  9: temp_mask = 0b111111111;
                    break;
            case  8: temp_mask = 0b11111111;
                    break;
            case  7: temp_mask = 0b1111111;
                    break;
            case  6: temp_mask = 0b111111;
                    break;
            case  5: temp_mask = 0b11111;
                    break;
            case  4: temp_mask = 0b1111;
                    break;
            case  3: temp_mask = 0b111;
                    break;
            case  2: temp_mask = 0b11;
                    break;
            case  1: temp_mask = 0b1;
                    break;
          }

        // set vector zmm3 to -1
        __m512i zmm3 = _mm512_set1_epi32(-1);
        // set vector zmm4 to v
        __m512i zmm4 = _mm512_set1_epi32(v);

        __mmask16 k1 = _mm512_int2mask(temp_mask);

#ifdef ALIGNED_ACCESS
          //load connected nodes from xadj
          __m512i zmm1 = _mm512_mask_load_epi32 \
          (zmm1, k1, &(xadj[vo]));
#else
          __m512i zmm1 = _mm512_mask_loadunpacklo_epi32 \
          (zmm1, k1, &(xadj[vo]));
          zmm1 = _mm512_mask_loadunpackhi_epi32 \
          (zmm1, k1, &(xadj[vo+16]));
#endif
          // gather elements from bfs_tree
          __m512i zmm2 = _mm512_mask_i32gather_epi32 \
          (zmm2, k1, zmm1, bfs_tree, 4);
```

```
            __mmask16 k2 = _mm512_mask_cmpeq_epi32_mask \
            (k1, zmm2, zmm3);
            int_mask = _mm512_mask2int(k2);
            cnt = _mm_countbits_32(int_mask);
            if (cnt) {
              // scatter store over mask register to bfs_tree
              _mm512_mask_i32scatter_epi32 \
              (bfs_tree, k2, zmm1, zmm4, 4);

              // store over mask to vlist
              _mm512_mask_packstorelo_epi32 \
              (&(vlist[kk2]), k2, zmm1);
              _mm512_mask_packstorehi_epi32 \
              (&(vlist[kk2+16]), k2, zmm1);
              kk2 += cnt;
            }
          } else {
#endif

            for (; vo < veo; ++vo) {
              const int32_t j = xadj[vo];
              if (bfs_tree[j] == -1) {
                bfs_tree[j] = v;
                vlist[kk2++] = j;
              }
            }
#ifdef VECT_REMAINING_PART
          }
#endif
#else
          for (vo = XOFF(v); vo < veo; ++vo) {
            const int32_t j = xadj[vo];
            if (bfs_tree[j] == -1) {
              bfs_tree[j] = v;
              vlist[kk2++] = j;
            }
          }
#endif
        }
        kk1 = oldk2;
```

```
    }
//offload end
  }

  xfree_large (vlist);
  return err;
}
```

A.3 Sequential Version with Scalar Prefetching

```
static int64_t * restrict xoff; /* Length 2*nv+2 */
#define XOFF(k) (xoff[2*(k)])
#define XENDOFF(k) (xoff[1+2*(k)])
#define USE_PREFETCH
#define DIST 32

int
make_bfs_tree (int64_t *bfs_tree_out,
               int64_t *max_vtx_out,
               int64_t srcvtx, int64_t nv_scale)
{
  int32_t * restrict __attribute__((target(mic))) \
                                bfs_tree = bfs_tree_out;
  int err = 0;
  int64_t * restrict vlist = NULL;
  int64_t k1, k2;

  *max_vtx_out = maxvtx;

  vlist = xmalloc_large (nv * sizeof (*vlist));
  if (!vlist) return -1;

  for (k1 = 0; k1 < nv; ++k1)
    bfs_tree[k1] = -1;

  #pragma offload target(mic:MIC_DEV) \
            in(xoff : length(2*nv+2)) \
```

```
            in(xadj : length((XOFF(nv)))) \
            inout(vlist : length(nv)) \
            inout(bfs_tree : length(nv_scale) \
            alloc_if(1) free_if(1))
  {
    vlist[0] = srcvtx;
    bfs_tree[srcvtx] = srcvtx;
    k1 = 0; k2 = 1;
    while (k1 != k2) {
      const int64_t oldk2 = k2;
      int64_t k;
#ifdef __MIC__
#ifdef USE_PREFETCH
        int32_t new_node = 0;
    int32_t temp;
#endif
#endif
      for (k = k1; k < oldk2; ++k) {
        const int64_t v = vlist[k];
        const int64_t veo = XENDOFF(v);
        int64_t vo;
#ifdef __MIC__
#ifdef USE_PREFETCH
          int32_t v1 = vlist[k+1];
          int32_t vo1 = XOFF(v1);
          int32_t veo1 = XENDOFF(v1);
          int32_t distance = DIST;
          new_node = 0;
#endif
#endif
        for (vo = XOFF(v); vo < veo; ++vo) {
#ifdef __MIC__
#ifdef USE_PREFETCH
            if (!new_node){
              if ((veo-vo) < DIST){
                new_node = 1;
```

```
                if ((veo1-vo1) < DIST)
                  distance = veo1-vo1;
                for (temp = 0; temp < distance; ++temp)
                  _mm_prefetch \
                  (&bfs_tree[xadj[vo1+temp]], _MM_HINT_T0);
              } else {
                _mm_prefetch \
                (&bfs_tree[xadj[vo+DIST]], _MM_HINT_T0);
              }
            }
#endif
#endif
          const int64_t j = xadj[vo];
          if (bfs_tree[j] == -1) {
            bfs_tree[j] = v;
            vlist[k2++] = j;
          }
        }
      }
      k1 = oldk2;
    }
  //offload end
  }

  xfree_large (vlist);

  return err;
}
```

A.4 Vectorized sequential version with vector prefetching

```
static int64_t * restrict xoff; /* Length 2*nv+2 */
#define XOFF(k) (xoff[2*(k)])
#define XENDOFF(k) (xoff[1+2*(k)])
#define USE_PREFETCH
```

```
int
make_bfs_tree (int32_t *bfs_tree_out,
               int64_t *max_vtx_out,
               int64_t srcvtx, int64_t nv_scale)
{
  int32_t * restrict __attribute__((target(mic))) \
                           bfs_tree = bfs_tree_out;
  int err = 0;
  int32_t * restrict vlist = NULL;
  int32_t kk1, kk2;
  *max_vtx_out = maxvtx;
  vlist = xmalloc_large (nv * sizeof (*vlist));
  if (!vlist) return -1;

  #pragma offload target(mic:MIC_DEV) \
            in(xoff : length(2*nv+2)) \
            in(xadj : length((XOFF(nv)))) \
            inout(vlist : length(nv)) \
            inout(bfs_tree : length(nv_scale) \
            alloc_if(1) free_if(1))
  {
    for (kk1 = 0; kk1 < nv; ++kk1)
      bfs_tree[kk1] = -1;

    vlist[0] = srcvtx;
    bfs_tree[srcvtx] = srcvtx;
    kk1 = 0; kk2 = 1;
    while (kk1 != kk2) {
      const int32_t oldk2 = kk2;
      int32_t k;
      for (k = kk1; k < oldk2; ++k) {
        const int32_t v = vlist[k];
        const int32_t veo = XENDOFF(v);
        int32_t vo, i;
```

```
#ifdef __MIC__
        const int32_t v1 = vlist[k+1];
        const int32_t veo1 = XENDOFF(v1);
        int32_t vo1 = XOFF(v1);

        __m512i zmm7;
        int32_t int_mask, cnt;

        vo = XOFF(v);
        if (veo - vo >= 16){
          // set vector zmm3 to -1
          __m512i zmm3 = _mm512_set1_epi32(-1);
          // set vector zmm4 to v
          __m512i zmm4 = _mm512_set1_epi32(v);

          for ( ; vo <= (veo-16); vo+=16){
#ifdef USE_PREFETCH
            if ((vo + 32) < veo){
#ifdef ALIGNED_ACCESS
              zmm7 = _mm512_load_epi32( &(xadj[vo+16]));
#else
              zmm7 = _mm512_loadunpacklo_epi32 \
              (zmm7, &(xadj[vo+16]));
              zmm7 = _mm512_loadunpackhi_epi32 \
              (zmm7, &(xadj[vo+32]));
#endif
              _mm512_prefetch_i32gather_ps \
              (zmm7, bfs_tree, 4, _MM_HINT_T0);
            }
#endif

#ifdef ALIGNED_ACCESS
            //load connected nodes from xadj
            __m512i zmm1 = _mm512_load_epi32  \
            (&(xadj[vo]));
#else
```

```
            __m512i zmm1 = _mm512_loadunpacklo_epi32 \
            (zmm1, &(xadj[vo]));
            zmm1 = _mm512_loadunpackhi_epi32 \
            (zmm1, &(xadj[vo+16]));
#endif
            // gather elements from bfs_tree
            __m512i zmm2 = _mm512_i32gather_epi32 \
            (zmm1, bfs_tree, 4);
            __mmask16 k1 = _mm512_cmpeq_epi32_mask \
            (zmm2, zmm3);
            int_mask = _mm512_mask2int(k1);
            cnt = _mm_countbits_32(int_mask);
            if (cnt) {
              // scatter store over mask register to bfs_tree
              _mm512_mask_i32scatter_epi32 \
              (bfs_tree, k1, zmm1, zmm4, 4);

              // store over mask to vlist
              _mm512_mask_packstorelo_epi32 \
              (&(vlist[kk2]), k1, zmm1);
              _mm512_mask_packstorehi_epi32 \
              (&(vlist[kk2+16]), k1, zmm1);
              kk2 += cnt;
            }
          }
        }

#ifdef VECT_REMAINING_PART
        int32_t temp_mask;
        int32_t rem = veo - vo;
        if (rem > REMAINING_PART){
           switch(rem){
             case 15: temp_mask = 0b111111111111111;
                      break;
             case 14: temp_mask = 0b11111111111111;
                      break;
```

```
            case 13: temp_mask = 0b1111111111111;
                     break;
            case 12: temp_mask = 0b111111111111;
                     break;
            case 11: temp_mask = 0b11111111111;
                     break;
            case 10: temp_mask = 0b1111111111;
                     break;
            case  9: temp_mask = 0b111111111;
                     break;
            case  8: temp_mask = 0b11111111;
                     break;
            case  7: temp_mask = 0b1111111;
                     break;
            case  6: temp_mask = 0b111111;
                     break;
            case  5: temp_mask = 0b11111;
                     break;
            case  4: temp_mask = 0b1111;
                     break;
            case  3: temp_mask = 0b111;
                     break;
            case  2: temp_mask = 0b11;
                     break;
            case  1: temp_mask = 0b1;
                     break;
        }

        // set vector zmm3 to -1
        __m512i zmm3 = _mm512_set1_epi32(-1);
        // set vector zmm4 to v
        __m512i zmm4 = _mm512_set1_epi32(v);

        __mmask16 k1 = _mm512_int2mask(temp_mask);

#ifdef ALIGNED_ACCESS
```

```
            //load connected nodes from xadj
            __m512i zmm1 = _mm512_mask_load_epi32 \
            (zmm1, k1, &(xadj[vo]));
#else
            __m512i zmm1 = _mm512_mask_loadunpacklo_epi32 \
            (zmm1, k1, &(xadj[vo]));
            zmm1 = _mm512_mask_loadunpackhi_epi32 \
            (zmm1, k1, &(xadj[vo+16]));
#endif
            // gather elements from bfs_tree
            __m512i zmm2 = _mm512_mask_i32gather_epi32 \
            (zmm2, k1, zmm1, bfs_tree, 4);

            __mmask16 k2 = _mm512_mask_cmpeq_epi32_mask \
            (k1, zmm2, zmm3);
            int_mask = _mm512_mask2int(k2);
            cnt = _mm_countbits_32(int_mask);
            if (cnt) {
              // scatter store over mask register to bfs_tree
              _mm512_mask_i32scatter_epi32 \
              (bfs_tree, k2, zmm1, zmm4, 4);

              // store over mask to vlist
              _mm512_mask_packstorelo_epi32 \
              (&(vlist[kk2]), k2, zmm1);
              _mm512_mask_packstorehi_epi32 \
              (&(vlist[kk2+16]), k2, zmm1);
              kk2 += cnt;
            }
          } else {
#endif

#ifdef USE_PREFETCH
            if ((vo1 + 16) < veo1){
#ifdef ALIGNED_ACCESS
              zmm7 = _mm512_load_epi32( &(xadj[vo1]));
```

```
#else
            zmm7 = _mm512_loadunpacklo_epi32 \
            (zmm7, &(xadj[vo1]));
            zmm7 = _mm512_loadunpackhi_epi32 \
            (zmm7, &(xadj[vo1+16]));
#endif
            _mm512_prefetch_i32gather_ps \
            (zmm7, bfs_tree, 4, _MM_HINT_T0);
          }
#endif
          for (; vo < veo; ++vo) {
            const int32_t j = xadj[vo];
            if (bfs_tree[j] == -1) {
              bfs_tree[j] = v;
              vlist[kk2++] = j;
            }
          }
#ifdef VECT_REMAINING_PART
        }
#endif
#else
        for (vo = XOFF(v); vo < veo; ++vo) {
          const int32_t j = xadj[vo];
          if (bfs_tree[j] == -1) {
            bfs_tree[j] = v;
            vlist[kk2++] = j;
          }
        }
#endif
      }
      kk1 = oldk2;
    }
//offload end
  }

  xfree_large (vlist);
```

```
  return err;
}
```

A.5 Vectorized sequential version with vector and scalar prefetching

```
static int64_t * restrict xoff; /* Length 2*nv+2 */
#define XOFF(k) (xoff[2*(k)])
#define XENDOFF(k) (xoff[1+2*(k)])
#define USE_PREFETCH

int
make_bfs_tree (int32_t *bfs_tree_out,
               int64_t *max_vtx_out,
               int64_t srcvtx, int64_t nv_scale)
{
  int32_t * restrict __attribute__((target(mic))) \
                          bfs_tree = bfs_tree_out;
  int err = 0;
  int32_t * restrict vlist = NULL;
  int32_t kk1, kk2;
  *max_vtx_out = maxvtx;

  vlist = xmalloc_large (nv * sizeof (*vlist));
  if (!vlist) return -1;

  #pragma offload target(mic:MIC_DEV) \
            in(xoff : length(2*nv+2)) \
            in(xadj : length((XOFF(nv)))) \
            inout(vlist : length(nv)) \
            inout(bfs_tree : length(nv_scale) \
            alloc_if(1) free_if(1))
  {
    for (kk1 = 0; kk1 < nv; ++kk1)
      bfs_tree[kk1] = -1;
```

```
    vlist[0] = srcvtx;
    bfs_tree[srcvtx] = srcvtx;
    kk1 = 0; kk2 = 1;
    while (kk1 != kk2) {
      const int32_t oldk2 = kk2;
      int32_t k;
      for (k = kk1; k < oldk2; ++k) {
        const int32_t v = vlist[k];
        const int32_t veo = XENDOFF(v);
        int32_t vo, i;

#ifdef __MIC__
        const int32_t v1 = vlist[k+1];
        const int32_t veo1 = XENDOFF(v1);
        int32_t vo1 = XOFF(v1);

        __m512i zmm7;

        int32_t int_mask, cnt;
        int32_t distance, temp;

        vo = XOFF(v);
        if (veo - vo >= 16){
          // set vector zmm3 to -1
          __m512i zmm3 = _mm512_set1_epi32(-1);
          // set vector zmm4 to v
          __m512i zmm4 = _mm512_set1_epi32(v);

          for ( ; vo <= (veo-16); vo+=16){
#ifdef USE_PREFETCH
            if ((vo + 32) < veo){
#ifdef ALIGNED_ACCESS
              zmm7 = _mm512_load_epi32( &(xadj[vo+16]));
#else
              zmm7 = _mm512_loadunpacklo_epi32 \
```

```
                (zmm7, &(xadj[vo+16]));
                zmm7 = _mm512_loadunpackhi_epi32 \
                (zmm7, &(xadj[vo+32]));
#endif
                _mm512_prefetch_i32gather_ps \
                (zmm7, bfs_tree, 4, _MM_HINT_T0);
              } else {
                distance = veo-vo;
                for (temp = 16; temp < distance; ++temp)
                  _mm_prefetch \
                  (&bfs_tree[xadj[vo+temp]], _MM_HINT_T0);
              }
#endif

#ifdef ALIGNED_ACCESS
              //load connected nodes from xadj
              __m512i zmm1 = _mm512_load_epi32( &(xadj[vo]));
#else
              __m512i zmm1 = _mm512_loadunpacklo_epi32 \
              (zmm1, &(xadj[vo]));
              zmm1 = _mm512_loadunpackhi_epi32 \
              (zmm1, &(xadj[vo+16]));
#endif
              // gather elements from bfs_tree
              __m512i zmm2 = _mm512_i32gather_epi32 \
              (zmm1, bfs_tree, 4);

              __mmask16 k1 = _mm512_cmpeq_epi32_mask \
              (zmm2, zmm3);
              int_mask = _mm512_mask2int(k1);
              cnt = _mm_countbits_32(int_mask);
              if (cnt) {
                // scatter store over mask register to bfs_tree
                _mm512_mask_i32scatter_epi32 \
                (bfs_tree, k1, zmm1, zmm4, 4);
```

```
                // store over mask to vlist
                _mm512_mask_packstorelo_epi32 \
                (&(vlist[kk2]), k1, zmm1);
                _mm512_mask_packstorehi_epi32 \
                (&(vlist[kk2+16]), k1, zmm1);
                kk2 += cnt;
              }
            }
          }

#ifdef VECT_REMAINING_PART
          int32_t temp_mask;
          int32_t rem = veo - vo;
          if (rem > REMAINING_PART){
             switch(rem){
               case 15: temp_mask = 0b111111111111111;
                        break;
               case 14: temp_mask = 0b11111111111111;
                        break;
               case 13: temp_mask = 0b1111111111111;
                        break;
               case 12: temp_mask = 0b111111111111;
                        break;
               case 11: temp_mask = 0b11111111111;
                        break;
               case 10: temp_mask = 0b1111111111;
                        break;
               case  9: temp_mask = 0b111111111;
                        break;
               case  8: temp_mask = 0b11111111;
                        break;
               case  7: temp_mask = 0b1111111;
                        break;
               case  6: temp_mask = 0b111111;
                        break;
               case  5: temp_mask = 0b11111;
```

```
                    break;
            case  4: temp_mask = 0b1111;
                    break;
            case  3: temp_mask = 0b111;
                    break;
            case  2: temp_mask = 0b11;
                    break;
            case  1: temp_mask = 0b1;
                    break;
          }

        // set vector zmm3 to -1
        __m512i zmm3 = _mm512_set1_epi32(-1);
        // set vector zmm4 to v
        __m512i zmm4 = _mm512_set1_epi32(v);


         __mmask16 k1 = _mm512_int2mask(temp_mask);

#ifdef ALIGNED_ACCESS
         //load connected nodes from xadj
         __m512i zmm1 = _mm512_mask_load_epi32 \
         (zmm1, k1, &(xadj[vo]));
#else
         __m512i zmm1 = _mm512_mask_loadunpacklo_epi32 \
         (zmm1, k1, &(xadj[vo]));
        zmm1 = _mm512_mask_loadunpackhi_epi32 \
         (zmm1, k1, &(xadj[vo+16]));
#endif
         // gather elements from bfs_tree
         __m512i zmm2 = _mm512_mask_i32gather_epi32 \
         (zmm2, k1, zmm1, bfs_tree, 4);

         __mmask16 k2 = _mm512_mask_cmpeq_epi32_mask \
         (k1, zmm2, zmm3);
         int_mask = _mm512_mask2int(k2);
```

```
            cnt = _mm_countbits_32(int_mask);
            if (cnt) {
              // scatter store over mask register to bfs_tree
              _mm512_mask_i32scatter_epi32 \
              (bfs_tree, k2, zmm1, zmm4, 4);

              // store over mask to vlist
              _mm512_mask_packstorelo_epi32 \
              (&(vlist[kk2]), k2, zmm1);
              _mm512_mask_packstorehi_epi32 \
              (&(vlist[kk2+16]), k2, zmm1);
              kk2 += cnt;
            }
          } else {
#endif

#ifdef USE_PREFETCH
            if ((vo1 + 16) < veo1){
#ifdef ALIGNED_ACCESS
              zmm7 = _mm512_load_epi32( &(xadj[vo1]));
#else
              zmm7 = _mm512_loadunpacklo_epi32 \
              (zmm7, &(xadj[vo1]));
              zmm7 = _mm512_loadunpackhi_epi32 \
              (zmm7, &(xadj[vo1+16]));
#endif
              _mm512_prefetch_i32gather_ps \
              (zmm7, bfs_tree, 4, _MM_HINT_T0);
            } else {
              distance = veo1-vo1;
              for (temp = 0; temp < distance; ++temp)
              _mm_prefetch \
              (&bfs_tree[xadj[vo1+temp]], _MM_HINT_T0);
            }
#endif
            for (; vo < veo; ++vo) {
```

```
            const int32_t j = xadj[vo];
            if (bfs_tree[j] == -1) {
              bfs_tree[j] = v;
              vlist[kk2++] = j;
            }
          }
#ifdef VECT_REMAINING_PART
        }
#endif
#else
        for (vo = XOFF(v); vo < veo; ++vo) {
          const int32_t j = xadj[vo];
          if (bfs_tree[j] == -1) {
            bfs_tree[j] = v;
            vlist[kk2++] = j;
          }
        }
#endif
      }
      kk1 = oldk2;
    }
//offload end
  }

  xfree_large (vlist);

  return err;
}
```

List of Figures

List of Tables

Bibliography

[1] (1981). *Cray-1 S Series Hardware Reference Manual HR-0808*. Cray. 32

[2] (2002). *Cray Assembly Language CAL for Cray X1 Systems Ref. Manual*. 15

[3] (2012). *Intel 64 and IA-32 Architectures Optimization Reference Manual*. Intel. 33

[4] Agarwal, V., Petrini, F., Pasetto, D. & Bader, D.A. (2010). Scalable graph exploration on multicore processors. In *Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*, 1–11, IEEE Computer Society. 48

[5] ARM (2013). EE Times Virtual Conference. www.eetimes.com/arm/. 7

[6] Asanović, K. (May, 1998). *Vector Microprocessors*. Ph.D. thesis, University of California, Berkeley. 1, 4, 8, 41

[7] Asanović, K. & Beck, J. (1997). T0 Engineering Data. Tech. Rep. CSD-97-931. 3

[8] Balart, J., Duran, A., Gonzàlez, M., Martorell, X., Ayguadé, E. & Labarta, J. (2004). Nanos mercurium: a research compiler for OpenMP. In *Proceedings of the European Workshop on OpenMP*, vol. 8, 56. 27

[9] Batten, C.F. (2010). *Simplified vector-thread architectures for flexible and efficient data-parallel accelerators*. Ph.D. thesis, Citeseer, Cambridge, MA, USA. 3, 7

[10] Beamer, S., Asanović, K. & Patterson, D. (2013). Direction-optimizing breadth-first search. *Scientific Programming*, **21**, 137–148. 50

[11] Binkert, N., Beckmann, B., Black, G., Reinhardt, S.K., Saidi, A., Basu, A., Hestness, J., Hower, D.R., Krishna, T., Sardashti, S., Sen, R., Sewell, K., Shoaib, M., Vaish, N., Hill, M.D. & Wood, D.A. (2011). The Gem5 simulator. *SIGARCH Comput. Archit. News*, **39**, 1–7. 71, 72

[12] Bohr, M. (2007). A 30 year retrospective on Dennard's MOSFET scaling paper. *Solid-State Circuits Society Newsletter, IEEE*, **12**, 11–13. 7

[13] Burger, D. & Austin, T.M. (1997). The SimpleScalar tool set, version 2.0. *SIGARCH Comp. Arch. News*, **25**, 13–25. 30

[14] Buxton, M., P. Jinbo, K.N. & N. Firasta (2008). Intel AVX: New frontiers in performance improvements and energy efficiency. White paper. 4, 8

[15] Ciricescu, S., Essick, R., Lucas, B., May, P., Moat, K., Norris, J., Schuette, M. & Saidi, A. (2003). The reconfigurable streaming vector processor (RSVPTM). In *Proceedings of the 36th annual IEEE/ACM International Symposium on Microarchitecture*, 141, IEEE Computer Society. 15, 40, 85

[16] Compute, N. (2010). PTX: Parallel thread execution ISA version 2.3. *http://developer. download. nvidia. com/compute/cuda/3*, **1**. 98

[17] Cong, J., Ghodrat, M.A., Gill, M., Huang, H., Liu, B., Prabhakar, R., Reinman, G. & Vitanza, M. (2012). Compilation and architecture support for customized vector instruction extension. In *Design Automation Conference (ASP-DAC), 2012 17th Asia and South Pacific*, 652–657, IEEE. 8

[18] Convex Press (1992). *CONVEX Architecture Reference Manual (C Series)*. 6th edn. 2, 13, 15

[19] Corbal, J., Espasa, R. & Valero, M. (2002). Three-dimensional memory vectorization for high bandwidth media memory systems. In *Proceedings of the 35th annual ACM/IEEE international symposium on Microarchitecture*, 149–160. 42

[20] Espasa, R. (1997). *Advanced Vector Architectures*. Ph.D. thesis, Universitat Politècnica de Catalunya. 2, 3, 8, 21, 41

[21] Espasa, R. & Martorell, X. (1994). Dixie: a trace generation system for the C3480. Tech. Rep. CEPBA-RR-94- 08, Univ. Politècnica de Catalunya. 41

[22] Espasa, R. & Valero, M. (1996). Decoupled vector architectures. In *High-Performance Computer Architecture, 1996. Proceedings., Second International Symposium on*, 281–290. 30, 106

[23] Espasa, R., Valero, M. & Smith, J.E. (1998). Vector architectures: past, present and future. In *Proceedings of the 12th international conference on Supercomputing*, 425–432, ACM. 2

[24] Espasa, R., Ardanaz, F., Emer, J., Felix, S., Gago, J., Gramunt, R., Hernandez, I., Juan, T., Lowney, G., Mattina, M. *et al.* (2002). Tarantula: a vector extension to the alpha architecture. In *Computer Architecture, 2002. Proceedings. 29th Annual International Symposium on*, 281–292, IEEE. 3, 7, 30, 37, 71

[25] Fuller, S. (1998). Motorola's AltiVec technology. White paper. 4

[26] Gebis, J.J. (2008). *Low-complexity vector microprocessor extension*. Ph.D. thesis, Berkeley, CA, USA. 3, 98

[27] Gschwind, M. (2006). Chip multiprocessing and the cell broadband engine. In *Proceedings of the 3rd conference on Computing frontiers*, 1–8, ACM. 99

[28] Gschwind, M. (2016). Workload acceleration with the IBM POWER vector-scalar architecture. *IBM Journal of Research and Development*, **60**, 14–1. 99

[29] Gschwind, M., Hofstee, H.P., Flachs, B., Hopkins, M., Watanabe, Y. & Yamazaki, T. (2006). Synergistic processing in Cell's multicore architecture. *IEEE micro*, **26**, 10–24. 99

[30] Hayes, T., Palomar, O., Unsal, O., Cristal, A. & Valero, M. (2012). Vector extensions for decision support DBMS acceleration. In *Proceedings of the 45th Annual ACM/IEEE International Symposium on Microarchitecture*, 166–176. 7, 37, 101

[31] Hennessy, J.L. & Patterson, D.A. (2006). *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann Publishers Inc., 4th edn., appendix F. 1, 4, 15, 28, 31, 51

[32] Hintz, R.G. & Tate, D.P. (1972, IEEE). Control data STAR-100 processor design. In *Compcon 72*, 1–4. 2

[33] Hong, S., Oguntebi, T. & Olukotun, K. (2011). Efficient parallel graph exploration on multi-core CPU and GPU. In *Parallel Architectures and Compilation Techniques (PACT), 2011 International Conference on*, 78–88, IEEE. 50

[34] Hu, Z. & Martonosi, M. (2000). Reducing register file power consumption by exploiting value lifetime. In *Proceedings of WCED in conjunction with ISCA*, vol. 27. 101

[35] Hu, Z., Buyuktosunoglu, A., Srinivasan, V., Zyuban, V., Jacobson, H. & Bose, P. (2004). Microarchitectural techniques for power gating of execution units. In *Proceedings of the 2004 international symposium on Low power electronics and design*, ISLPED '04, 32–37, ACM, New York, NY, USA. 6

[36] Intel (2012). *Intel Xeon Phi TM Coprocessor Instruction Set Architecture Reference Manual*. 46

[37] Irimia-Vladu, M. (2014). Green electronics: biodegradable and biocompatible materials and devices for sustainable future. *Chemical Society Reviews*, **43**, 588–610. 7

[38] Janin, A.L. (2004). *Speech Recognition on Vector Architecture*. Ph.D. thesis, Univ. of California, Berkeley. 4, 42

[39] Khailany, B., Dally, W.J., Kapasi, U.J., Mattson, P., Namkoong, J., Owens, J.D., Towles, B., Chang, A. & Rixner, S. (2001). Imagine: Media processing with streams. *IEEE Micro*, **21**, 35–46. 99

[40] Kobayashi, H., Egawa, R., Takizawa, H., Okabe, K., Musa, A., Soga, T. & Shimomura, Y. (2009). First experiences with NEC SX-9. In *High Performance Computing on Vector Systems 2008*, 3–11, Springer. 30

[41] Kozyrakis, C. (2002). Scalable vector media-processors for embedded systems. Tech. rep., Berkeley, CA, USA. 3

[42] Kozyrakis, C. & Patterson, D. (2003). Overcoming the limitations of conventional vector processors. In *Computer Architecture, 2003. Proceedings. 30th Annual International Symposium on*, 399–409. 3, 7, 80

[43] Krashinsky, R., Batten, C., Hampton, M., Gerding, S., Pharris, B., Casper, J. & Asanović, K. (2004). The vector-thread architecture. In *Proceedings of the 31st Annual International Symposium on Computer Architecture*, ISCA '04, 52–64. 99

[44] Krashinsky, R.M. (2007). *Vector-thread architecture and implementation*. Ph.D. thesis, Massachusetts Institute of Technology, Cambridge, MA, USA. 3

[45] Lattner, C. & Adve, V. (2004). LLVM: A compilation framework for lifelong program analysis & transformation. In *Code Generation and Optimization, 2004. CGO 2004. International Symposium on*, 75–86, IEEE. 27

[46] Lee, R.B. (1995). Accelerating multimedia with enhanced microprocessors. *IEEE Micro*, **15**, 22–32. 4

[47] Lee, Y., Avizienis, R., Bishara, A., Xia, R., Lockhart, D., Batten, C. & Asanović, K. (2011). Exploring the tradeoffs between programmability and efficiency in data-parallel accelerators. In *Proceedings of the 38th Annual International Symposium on Computer Architecture*, ISCA '11, 129–140, ACM, New York, NY, USA. 1, 5, 7

[48] Lemuet, C., Sampson, J., Collard, J.F. & Jouppi, N. (2006). The potential energy efficiency of vector acceleration. In *Proceedings of the 2006 ACM/IEEE conference on Supercomputing*, SC '06, ACM, New York, NY, USA. 5, 100

[49] Li, H., Bhunia, S., Chen, Y., Vijaykumar, T.N. & Roy, K. (2003). Deterministic clock gating for microprocessor power reduction. In *Proceedings of the 9th International Symposium on High-Performance Computer Architecture*, HPCA '03, 113–, IEEE Computer Society, Washington, DC, USA. 6

[50] Li, S., Ahn, J.H., Strong, R., Brockman, J., Tullsen, D. & Jouppi, N. (2009). McPAT: An integrated power, area, and timing modeling framework for multicore and manycore architectures. In *Proceedings of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture*, 469–480. 73

[51] Momose, S., Hagiwara, T., Isobe, Y. & Takahara, H. (2014). The brand-new vector supercomputer, SX-ACE. In *Supercomputing*, 199–214, Springer. 3, 13

[52] Montanaro, J., Witek, R.T., Anne, K., Black, A.J., Cooper, E.M., Dobberpuhl, D.W., Donahue, P.M., Eno, J., Hoeppner, W., Kruckemyer, D. *et al.* (1996). A 160-mhz, 32-b, 0.5-w CMOS RISC microprocessor. vol. 31, 1703–1714, IEEE. 4

[53] Moore, G.E. (2000). Readings in computer architecture. chap. Cramming more components onto integrated circuits, 56–59, Morgan Kaufmann Publishers Inc., San Francisco, CA, USA. 6

[54] Mucci, P.J., Browne, S., Deane, C. & Ho, G. (1999). PAPI: A portable interface to hardware performance counters. In *Proceedings of the department of defense HPCMP users group conference*, 7–10. 17, 53, 57

[55] Murphy, R., Berry, J., McLendon, W., Hendrickson, B., Gregor, D. & Lumsdaine, A. (2006). DFS: a simple to write yet difficult to execute benchmark. In *2006 IEEE International Symposium on Workload Characterization*, 175–177, IEEE. 19

[56] Murphy, R.C., Wheeler, K.B., Barrett, B.W. & Ang, J.A. (2010). Introducing the Graph 500. *Cray Users Group (CUG)*. 19, 45, 74

[57] Narayanan, R., Ozisikyilmaz, B., Zambreno, J., Memik, G. & Choudhary, A. (2006). Minebench: A benchmark suite for data mining workloads. In *2006 IEEE International Symposium on Workload Characterization*, 182–188, IEEE. 19

[58] Oberman, S., Favor, G. & Weber, F. (1999). AMD 3DNow! technology: Architecture and implementations. *IEEE Micro*, **19**, 37–48. 4

[59] Palacharla, S., Jouppi, N.P. & Smith, J.E. (1997). Complexity-effective superscalar processors. In *Proceeding of the 24th International Symposium on Computer Architecture*, ISCA 97, 206–218. 4

[60] Peleg, A. & Weiser, U. (1996). MMX technology extension to the Intel architecture. *IEEE Micro*, **16**, 42–50. 4

[61] Perron, R. & Mundie, C. (March, 1986). The architecture of the Alliant FX/8 computer. *Compcon 86*, 390–394. 2, 13

[62] Pharr, M. & Mark, W.R. (2012). ispc: A SPMD compiler for high-performance CPU programming. In *Innovative Parallel Computing (InPar), 2012*, 1–13, IEEE. 27

[63] Ponomarev, D., Kucuk, G., Ergin, O. & Ghose, K. (2003). Reducing datapath energy through the isolation of short-lived operands. In *Proceedings. 12th International Conference on Parallel Architectures and Compilation Techniques, 2003. PACT 2003.*, 258–268, IEEE. 101

[64] Quintana, F. (2001). *Aceleradores Vectoriales para Procesadores Superescalaress*. Ph.D. thesis, Universitat Politècnica de Catalunya. 8, 41

[65] Quintana, F., Corbal, J., Espasa, R. & Valero, M. (1999). Adding a vector unit to a superscalar processor. In *Proceedings of the 13th international conference on Supercomputing*, ICS '99, 1–10. 30, 37, 98

[66] Ratković, I. (2011). *An Overview of Architecture Level Power and Energy Efficient Techniques*. Master's thesis, Faculty of Electrical Engineering. 5

[67] Ratković, I., Palomar, O., Stanić, M., Ünsal, O.S., Cristal, A. & Valero, M. (2013). On the selection of adder unit in energy efficient vector processing. 143–150. 18

[68] Ren, G., Wu, P. & Padua, D. (2003). A preliminary study on the vectorization of multimedia applications for multimedia extensions. In *Languages and Compilers for Parallel Computing*, 420–435, Springer. 4

[69] Russell, R.M. (1978). The CRAY-1 computer system. *Commun. ACM*, **21**, 63–72. 2, 13, 68, 80, 101

[70] Saule, E. & Çatalyürek, Ü.V. (2012). An early evaluation of the scalability of graph algorithms on the Intel MIC architecture. In *Parallel and Distributed Processing Symposium Workshops & PhD Forum (IPDPSW), 2012 IEEE 26th International*, 1629–1639, IEEE. 47

[71] Schönauer, W. (1987). *Scientific computing on vector computers*. Elsevier Science Inc. 2, 31, 80, 101

[72] Scott, S., Abts, D., Kim, J. & Dally, W.J. (2006). The BlackWidow high-radix clos network. In *Proceedings of the 33rd annual international symposium on Computer Architecture*, ISCA '06, 16–28, IEEE Computer Society, Washington, DC, USA. 2, 13

[73] Seal, D. (2000). *ARM Architecture Reference Manual*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2nd edn. 71, 100

[74] Seiler, L., Carmean, D., Sprangle, E., Forsyth, T., Dubey, P., Junkins, S., Lake, A., Cavin, R., Espasa, R., Grochowski, E., Juan, T., Abrash, M., Sugerman, J. & Hanrahan, P. (2009). Larrabee: A many-core x86 architecture for visual computing. vol. 29, 10–21, IEEE Computer Society, Los Alamitos, CA, USA. 4, 7, 46

[75] Smith, J.E., Faanes, G. & Sugumar, R. (2000). Vector instruction set support for conditional operations. In *Proceeding of the 27th International Symposium on Computer Architecture*, ISCA 00, 260–269. 15

[76] Sodani, A. (2015). Knights landing: 2nd generation intel xeon phi processor. In *August issue of Proceedings of Hot Chips: A Symposium on High Performance Chips*. 7

[77] Soliman, M. (2011). LcVc: Low-complexity vector-core for executing scalar/vector instructions. In *Computer Engineering Conference (ICENCO), 2011 Seventh International*, 19–24. 98

[78] Sutter, H. (2005). The free lunch is over: A fundamental turn toward concurrency in software. *Dr. Dobbs journal*, **30**, 202–210. 7

[79] Taur, Y. (2002). CMOS design near the limit of scaling. *IBM Journal of Research and Development*, **46**, 213–222. 7

[80] Thakkar, S. & Huff, T. (1999). Internet streaming SIMD extensions. *Computer*, **32**, 26–34. 4, 8

[81] Usami, K., Goto, Y., Matsunaga, K., Koyama, S., Ikebuchi, D., Amano, H. & Nakamura, H. (2011). On-chip detection methodology for break-even time of power gated function units. In *Proceedings of the 17th IEEE/ACM international symposium on Low-power electronics and design*, 241–246, IEEE Press. 6

[82] Valero, M. & Espasa, R. (1995). Instruction level characterization of the Perfect Club programs on a vector computer. In *SCCC 15*, 198–209. 8, 41

[83] Wang, D., Ganesh, B., Tuaycharoen, N., Baynes, K., Jaleel, A. & Jacob, B. (2005). DRAMsim: a memory system simulator. *ACM SIGARCH Computer Architecture News*, **33**, 100–107. 30

[84] Watson, W. (1972). The TI-ASC, a highly modular and exible super computer architecture. In *AFIPS '72 (Fall, part I)*, 221–228. 2

[85] Zeiser, T., Hager, G. & Wellein, G. (2009). The world's fastest CPU and SMP node: Some performance results from the NEC SX-9. *Parallel and Distributed Processing Symposium, International*, **0**, 1–8. 3, 13

Acronyms

ACL Aliasing control logic. 54, 58

AG address generator. 57, 70

ALU arithmetic logic unit. 6, 8, 9, 55–59, 61

AVL average vector length. 35–37

BBE Block-Based Execution. 55–59, 61–65, 69, 71–75, 78

BFS Breadth First Search. 12–18, 22, 24

CCL Chaining control logic. 55

FCS Full Chaining Support. 70–73

CSR Compressed Sparse Row. 17, 19, 24

CVU classic vector unit. 58, 59, 61–65, 69, 71–75

DC decoupled. 44, 45, 48, 49

DLP data-level parallelism. 1, 4, 6, 7, 33, 35, 40, 51, 62, 75

FP floating-point. 30, 42, 55, 58–60, 62–65, 75

FU functional units. 41–44, 46, 48, 49, 55, 56

ILP instruction-level parallelism. 1

IO in-order. 44–46, 48, 49

IPC Instruction Per Cycle. 41

ISA Instruction Set Architectures. 3, 6, 8, 10, 12, 29–31, 37, 40, 41, 49–51, 53, 61

ispc Intel SPMD Programing Compiler. 40

LD/ST load/store units. 41, 42, 49

MVL maximum vector length. 30, 33–37, 43–50, 58, 62–66, 71–74

OBO One-By-One model of execution. 55, 56, 58, 59, 61–65, 69, 71–75, 77

RCS Restricted Chaining Support. 70–73

SIMD single instruction multiple data. 3, 6, 12, 16, 32, 40, 41

TEPS traversed edges per second. 13, 18, 19, 24, 26–28

TLP thread-level parallelism. 1

VALib vector architecture library. 10, 29–34, 37, 40–43, 50, 51

VECL vector execution control logic. 54, 66

VL vector length. 4, 36, 68

VMCT vector memory control table. 57

VMIT vector memory instruction table. 57, 70

VPU vector processing unit. 16