UNIVERSITAT POLITÈCNICA
DE CATALUNYA
BARCELONATECH

Advancing the state of the art of directive-based programming for GPUs: runtime and compilation

Kazuaki Matsumura

ADVANCING THE STATE OF THE ART OF DIRECTIVE-BASED PROGRAMMING FOR GPUS: RUNTIME AND COMPILATION

KAZUAKI MATSUMURA

Dissertation submitted in partial fulfillment of the requirements
for obtaining the academic degree

Doctor of Philosophy in Computer Architecture

Universitat Politècnica de Catalunya

February 2024

SUPERVISOR:
Dr. Antonio Jose Peña Monferrer

MENTOR:
Dr. Simon Garcia de Gonzalo

AFFILIATION:
Departament d'Arquitectura de Computadors

SUPPORT:
Barcelona Supercomputing Center

LOCATION:
Barcelona

ABSTRACT

The rapid development in computing technology has paved the way for directive-based programming models towards a principal role in maintaining software portability of performance-critical applications. Efforts on such models involve a least engineering cost for enabling computational acceleration on multiple architectures, while programmers are only required to add meta information upon sequential code. Optimizations for obtaining the best possible efficiency, however, are often challenging. The insertions of directives by the programmer can lead to side-effects that limit the available compiler optimization possible, which could result in performance degradation. This is exacerbated when targeting asynchronous execution or multi-GPU systems, as pragmas do not automatically adapt to such mechanisms, and require expensive and time-consuming code adjustment by programmers. Moreover, directive-based programming models such as OpenACC and OpenMP often prevent programmers from making additional optimizations to take advantage of the advanced architectural features of GPUs because the actual generated computation is hidden from the application developer.

This dissertation explores new possibilities for optimizing directive-based code from both runtime and compilation perspectives. First, we introduce a runtime framework for OpenACC to facilitate dynamic analysis and compilation. Especially, our framework realizes automatic asynchronous execution and multi-GPU use based on the status of kernel execution and data availability while taking advantage of an on-the-fly mechanism for compilation and program optimization. We add a versatile code-translation method for multi-device utilization by which manually-optimized applications can be distributed automatically while keeping original code structure and parallelism. Second, we implement a novel flexible optimization technique that operates by inserting a code emulator phase to the tail-end of the compilation pipeline. Our tool emulates the generated code using symbolic analysis by substituting dynamic information and thus allowing for further low-level code optimizations to be applied. We implement our tool to support both CUDA and OpenACC directives as the frontend of the compilation pipeline, thus enabling low-level GPU optimizations for OpenACC that were not previously possible. Third, we propose the use of a modern optimization technique, equality saturation, to optimize sequential code utilized in directive-based programming for GPUs. Our approach realizes less computation, less memory access, and high memory throughput simultaneously. Our fully-automated framework constructs single-assignment forms from inputs to be entirely rewritten while keeping dependencies and extracts optimal cases. Overall, we cover runtime techniques and optimization methods based on dynamic information, low-level operations, and user-level opportunities.

We evaluate our proposals on the state-of-the-art GPUs and provide detailed analysis for each technique. For multi-GPU use, we show in some cases nearly linear scaling on the part of kernel execution with the NVIDIA V100 GPUs. While adaptively using multi-GPUs, the resulting performance improvements amortize the latency of GPU-to-GPU communications. Regarding low-level optimization, we demonstrate the capabili-

ties of our tool by automating warp-level shuffle instructions that are difficult to use by even advanced GPU programmers. While evaluating our tool with a benchmark suite and complex application code, we provide a detailed study to assess the benefits of shuffle instructions across four generations of GPU architectures. Lastly, with sequential code optimization, we demonstrate a significant performance improvement on several compilers through practical benchmarks. Then, we highlight the advantages of computational reordering and emphasize the significance of memory-access order for modern GPUs. All the implementations of our frameworks are publicly available and contribute new methods of acceleration to directive-based programming.

CONTENTS

1 INTRODUCTION 1
- 1.1 Motivation 1
- 1.2 Problem Statement 2
- 1.3 Proposal and Contributions 3
- 1.4 Thesis Outline 5

2 BACKGROUND 6
- 2.1 GPU Architecture 6
- 2.2 CUDA Programming 8
- 2.3 Directive-Based Programming 9
- 2.4 NVIDIA PTX 11
- 2.5 Shuffle Operations 12

3 RUNTIME EFFORTS 13
- 3.1 JACC 13
- 3.2 Basic Extension 14
  - 3.2.1 Automated Asynchronous Execution 14
  - 3.2.2 Kernel Specialization 17
- 3.3 Multi-GPU Utilization with Predicates 17
  - 3.3.1 Predicate-Based Filtering 17
  - 3.3.2 Division of Multidimensional Arrays 18
  - 3.3.3 Adaptive Utilization 20
  - 3.3.4 Implementation 21
- 3.4 Experimental Methodology 23
  - 3.4.1 Hardware and Software 23
  - 3.4.2 Benchmarks 23
- 3.5 Results 25
  - 3.5.1 Basic Extension 25
  - 3.5.2 GCC Custom Allocation 26
  - 3.5.3 Multi-GPU Utilization 26
- 3.6 Related Work 33

4 LOW-LEVEL CODE OPTIMIZATION 35
- 4.1 PTXASW 35
- 4.2 Symbolic Emulator 37
  - 4.2.1 Instruction Encoding 37
  - 4.2.2 Execution Branching 38
  - 4.2.3 Memory Analysis 38
- 4.3 Shuffle Synthesis 38
  - 4.3.1 Detection 40
  - 4.3.2 Code Generation 41
- 4.4 Experimental Methodology 42
- 4.5 Evaluation 44

4.6 Analysis 47
4.6.1 Kepler 47
4.6.2 Maxwell 47
4.6.3 Pascal 49
4.6.4 Volta 49
4.6.5 Application Example 50
4.7 Related Work 50
5 SOURCE-CODE OPTIMIZATION 53
5.1 ACC Saturator 53
5.2 Program Representation 55
5.2.1 E-Graph Creation 55
5.2.2 Code Selection 56
5.3 Optimization with Saturation 56
5.3.1 Rewriting Rules 56
5.3.2 Cost Model 57
5.4 Code Generation 57
5.4.1 Temporary-Variable Insertion 58
5.4.2 Bulk Load 59
5.5 Experimental Methodology 59
5.6 Evaluation 62
5.7 Related Work 68
6 SUMMARY AND CONCLUSION 70
6.1 Summary 70
6.2 Conclusion 71
6.3 Future Work 73
6.4 Publications 73
6.4.1 Referred Conferences 73
6.4.2 Referred Presentations 73
6.4.3 Software 73
Acknowledgements 75
Appendix 76
BIBLIOGRAPHY 79

LIST OF FIGURES

Figure 2.1 Performance changes of recent GPUs 6
Figure 2.2 NVIDIA Volta GV100 GPU 7
Figure 2.3 NVIDIA Volta GV100 Streaming Multiprocessor (SM) 8
Figure 3.1 Execution flow of automated asynchronous execution 15
Figure 3.2 Compilation flow with on-the-fly kernel specialization 16
Figure 3.3 Example of predicate-based filtering in C code 18
Figure 3.4 Original code and corresponding predicated code 19
Figure 3.5 Execution flow of predicate-based filtering 21
Figure 3.6 Actual case of JACC code generation in Fortran 22
Figure 3.7 Async and kernel optimization on NVIDIA Tesla V100 SXM2 26
Figure 3.8 Performance scaling of predicate-based filtering using NPB with NVHPC/GCC 27
Figure 3.9 Performance scaling of predicate-based filtering using the Fortran mini-apps with NVHPC 28
Figure 3.10 Scaling with the number of GPUs for the stencil kernel in the Himeno benchmark 31
Figure 3.11 Comparison between predicate-based filtering with/without the adaptive algorithm and MACC using NPB-CG 32
Figure 4.1 Overview of PTXASW 36
Figure 4.2 Performance comparison between the original code and the version implementing automated shuffle on four GPUs 44
Figure 4.3 Speedup by PTXASW compared to Original 45
Figure 4.4 Occupancy of SMs by PTXASW 46
Figure 4.5 Stall breakdown in the order of Original/NO LOAD/NO CORNER/PTXASW 48
Figure 5.1 Overview of ACC Saturator 54
Figure 5.2 NPB's speedup results on NVIDIA A100-PCIE-40GB for each variation compared to original 60
Figure 5.3 Breakdown of NPB-BT 62
Figure 5.4 Speedup results of the SPEC ACCEL benchmark suite on NVIDIA A100-PCIE-40GB 64
Figure 5.5 NPB's speedup results on NVIDIA A100-SXM4-80GB 66
Figure 5.6 Speedup results of the SPEC ACCEL benchmark suite on NVIDIA A100-SXM4-80GB 67

LIST OF TABLES

Table 2.1 Latencies (clock cycles) 12
Table 3.1 Benchmark description 24
Table 3.2 Performance details with NVHPC in the use of four GPUs 29
Table 4.1 The KernelGen benchmark suite 43
Table 5.1 ACC Saturator's rewriting rules 57
Table 5.2 NAS Parallel Benchmarks 60
Table 5.3 The SPEC ACCEL benchmark suite 61
Table 5.4 Top-10 kernel breakdown of NPB-BT 63

LISTINGS

Listing 2.1 Addition kernel in CUDA 9
Listing 2.2 Matrix multiplication kernel in C and OpenACC 9
Listing 2.3 Multi-device use in OpenACC 10
Listing 2.4 Addition kernel in PTX 11
Listing 2.5 The use of `shfl.sync` in PTX 12
Listing 3.1 Accelerator programming in OpenACC 13
Listing 3.2 Converted code to JACC 13
Listing 3.3 Generated code for kernel launch 14
Listing 3.4 Kernel code from NPB-BT 20
Listing 4.1 Addition of bitvectors 37
Listing 4.2 Addition instruction 37
Listing 4.3 Jacobi kernel in Fortran and OpenACC 39
Listing 4.4 Global-memory trace of Jacobi kernel through the symbolic emulation in order 39
Listing 4.5 Shuffle synthesis on Jacobi kernel 41
Listing 5.1 One of kernels in NPB-BT's z_solve.c 58
Listing 5.2 Generated Code of ACC Saturator 58
Listing A.1 Pseudocode for the algorithm of PTXASW 76

ACRONYMS

ASICs	Application-Specific Integrated Circuits
CFD	Computational Fluid Dynamics
CPUs	Central Processing Units
CSE	Common Subexpression Elimination
FFI	Foreign Function Interface
FMA	Fused Multiply-Adds
GPUs	Graphical Processing Units
HBM2	High Bandwidth Memory 2
HPC	High Performance Computing
ILP	Instruction Level Parallelism
NPB	NAS Parallel Benchmarks
SIMD	Single Instruction Multiple Data
SMs	Streaming Multiprocessors
SMT	Satisfiability Modulo Theories
SSA	Static Single Assignment
UM	Unified Memory

1

INTRODUCTION

Beginning this thesis, Chapter 1 engages in discourse about supercomputing, accelerators, and directive-based programming. Then, we argue the current issues that they face in modern systems. Our proposal and contributions are given to indicate the direction of our study. The thesis outline follows up to the end of the chapter.

1.1 MOTIVATION

Effectively utilizing the vast amount of computational performance available in modern supercomputers remains a challenge to this day. Hardware, middleware, and parallel algorithms should be carefully orchestrated so that ideal efficiency may be obtained for solving large real-world problems in High Performance Computing (HPC). Compiler technologies are developed with highly-automated program optimizations that use domain-specific knowledge and target architecture specialization to solve a part of this puzzle. With the end of Moore's Law [1] approaching, the focus on supercomputing technology is shifting toward even more specialized accelerators, which in turn increases their complexity. This trend further signifies the importance of compiler technology to relieve programmers from the burden of understanding the complex architecture of modern accelerators to be able to efficiently optimize their applications.

The adoption of newer technologies brings newer challenges. Currently, supercomputers employ various kinds of accelerators in the range from Single Instruction Multiple Data (SIMD) units available on Central Processing Units (CPUs) to discrete devices such as Graphical Processing Units (GPUs), Application-Specific Integrated Circuits (ASICs), and Intel Xeon Phis [2]. Those heterogeneous systems tend to force a certain amount of engineering efforts on application developers to establish an adequate distribution of program execution and attain performance for practical use. Notably, GPUs are the most widely adopted accelerator technology, and they work by speeding up applications upon its highly parallelized yet cooperative architecture. To take advantage of GPUs, programmers must be proficient in writing complex and time-consuming GPU code in low-level languages, which is typically the most profitable approach but involves complex data dependencies. As a result, adapting vendor-specific languages like CUDA [3] and OpenCL [4] requires extensive knowledge of the application and creates separate code entities just for computation offloading, which increases programming costs.

To overcome the complexity of low-level GPU code development, current GPU accelerators support programming with abstract models. OpenACC [5] and OpenMP [6] have emerged as popular models that alleviate the difficulty of accelerator use by providing

code directives to existing languages. These directives allow users to specify compute-intensive parts of the original code, which compilers may automatically translate into accelerator code. At runtime, offloading is realized without any user intervention.

Currently, major compilers including NVIDIA's NVHPC compiler [7], GCC [8], Intel Compilers [9], and Clang [10] support GPU programming with directives. While some divergences exist between language models, they share common interests and similar objectives for high-performance computing with existing software resources. Several workshops are dedicated to the development of those programming models [11, 12], and researchers in various fields continuously publish conference papers to take advantage of this method of computational acceleration [13]. As many research projects have already made significant efforts on those models, directive-based code already became legacy with multiple working applications.

1.2 PROBLEM STATEMENT

Directive-based code typically maintains a general programming style and remains unspecialized until compilation. It is the responsibility of compilers to handle the computation, the redundancy of memory accesses, and the order of those while considering the nature of accelerators. However, code rewriting while preserving semantics may be challenging. The insertion of directives by the programmers often results in compilation side-effects that lead to less program-characteristics exposure for compilation [14, 15, 16]; thus, programmers aiming at better efficiency are forced to reshape their code merely for adjusting to the environment such as compilers, software stacks and heterogeneous architectures. Moreover, to follow the program modification, additional runtime parameters are often introduced for each program segment. Therefore, managing rewritten code is far from clear regardless of the complexity of transformation. Specifically, the parallelism among kernels is rarely addressed and the multi-device utilization is basically dismissed due to the little usability of data dependency information.

The gap between high-level languages and state-of-the-art accelerators demands compiler techniques to elaborate efficient execution from sequential programs and auxiliary information. Current compilers, however, only consider lower-level codes that code generators also emit for CPU execution without specialization. In particular, user-specified directives fix loop structures, so little work has been done on optimizing code under directives other than relying on compiler techniques used for sequential programs [17]. Moreover, the optimization applied for each metric in order limits opportunities to utilize source-code information for improved performance. The compilers need more complex approaches to improve the performance of directive-based code as those programming models lack the ability to take advantage of low-level architecture-specific optimizations.

Despite the standardized interface of directive-based code, automatic program optimization typically involves the compiler implementation in depth [18, 19]. It is unlikely that code generation techniques are shared with other compilers or frameworks. Even a very simple peephole optimization requires a reworking of a compiler whose design is often monolithic and forces repetitive implementation to perform similar analyses due to the lack of cooperation. Hence, the scope of applications, target architecture, or algorithms is restricted before finishing the development of optimizations without knowing actual im-

provements to be gained. Such compiler-dependent tasks do not easily succeed in catching up with the rapid change of computational environments. Agile and autonomous code generation assuming only one language for both the input and the output is necessary to extend performance opportunities in various compilers and applications while preserving and utilizing high-level source-code information.

1.3 PROPOSAL AND CONTRIBUTIONS

To explore new optimization opportunities for directive-based programming, our study delves in three parts of compiler systems: runtime, backend, and frontend. In the first part of our study (Chapter 3), we create an OpenACC runtime framework which enables the dynamic extension of OpenACC programs by serving as a transparent layer between the program and the compiler. Our framework **JACC** provides an environment for dynamic analysis, rescheduling and distribution of execution along with on-the-fly kernel specialization by wrapping up existing OpenACC compilers. Although other directive-based programming work develops dedicated runtime systems for specific optimization [20, 21, 22], our framework integrates the compilation and runtime phases so as to utilize both aspects for additional efforts especially aiming at exploiting parallelism. Additionally, we address multi-GPU work distribution, while considering the high memory latency of GPUs. To accomplish this, we add a novel code-translation technique named *predicated-based filtering* to automate multi-device use. We never split loop ranges nor introduce fine dependency analysis, but divide data ranges to be updated on each device. This idea allows to distribute highly-tuned code neither changing code structure nor parallelism. Our contributions in the first part are as follows:

1. We create JACC, an OpenACC framework which facilitates various dynamic features including runtime data analysis and compilation. JACC enables kernel-level parallelization through an asynchronous mechanism.

2. We propose and describe a new multi-GPU kernel distribution method by leveraging JACC for complex applications. Our proposed technique is successful at multi-GPU execution on applications where previous work fails to offload to multiple devices.

3. We propose and describe an adaptive algorithm that automatically determines the adequate kernels for multi-GPU execution. Our adaptive algorithm considers the bandwidth of the peer-to-peer communication between GPUs when selecting kernels for multi-GPU execution.

4. We evaluate all our contributions by using two different OpenACC compilers on a NVIDIA V100 multi-GPU system. We show that using our proposed methods and techniques available through JACC, we are able to achieve nearly linear scaling when excluding the latency of communication. Additionally, we are able to successfully improve the performance of a variety of manually-tuned NAS Parallel Benchmarks.

The second part of our study (Chapter 4) expands the backend for additional optimization, which is further based on target architecture regardless of programming models. In this part, we describe a backend environment to extend the code of the NVIDIA GPU

assembly PTX and enable, for the first time in the literature, *automatic* shuffle synthesis to *explore the opportunity* of this operation. Our environment, **PTXASW** (Wrapper of PTX optimizing ASsembler), addresses the entire computational flow of PTX, leveraging a symbolic emulator that can symbolically extract memory-access patterns. Following the emulating results, PTXASW detects the global-memory loads that are possible to be covered by the shuffle operation. Around those loads, additional instructions are implanted, while supporting corner cases and circumventing overheads. We conduct the shuffle synthesis on OpenACC, a directive-based programming model having no user exposure to warp-level instructions. Our implementation functions as a plugin of the compilation tool yielding moderate overhead. Applying our technique, we find various opportunities to enable shuffle over the original code of an OpenACC benchmark suite. The performance improvement achieved is up to 132% with no user intervention on the NVIDIA Maxwell GPU. Additionally, based on the results of the experiments using several generations of GPUs, we analyze the latency caused for the shuffle operations to provide guidelines for shuffle usage on each GPU architecture. In summary, the contributions of the second part are:

1. We create a symbolic emulator to analyze and optimize GPU computing code, coupled with a Satisfiability Modulo Theories (SMT) solver for the comparison of symbolic expressions, induction variable recognition for loops, and various optimizations to reduce overheads.

2. Through symbolic analysis, we *automatically* find the possible cases to utilize the shuffle operation, which previously required in-depth domain knowledge to be applied. Then, we synthesize those to the applications, while avoiding expensive computation.

3. Using a directive-based programming model (OpenACC), we generate various shuffle codes on several generations of GPUs and show the cases that attain performance improvement *with no manual effort*.

4. We show the latency breakdown of the optimization on each GPU architecture and provide general guidelines for the use of shuffle operations.

The third part of our study (Chapter 5) steps forward to automate the modification of directive-based source code. We propose the use of a modern optimization technique, *equality saturation* [23], to fine-tune directive-based GPU code. Our tool, **ACC Saturator**, achieves less computation, fewer memory accesses, and high throughput at the same time, while easily integrating into the compilation of both OpenACC and OpenMP compilers. ACC Saturator performs kernel optimization by passing programs through an e-graph, a graph structure for equality saturation. Unlike other optimization methods, our approach neither transforms the abstract syntax trees nor it changes directives. Despite this, we attain significant performance improvements of up to 2.23x with the NVHPC compiler and 5.08x with GCC. We present a detailed performance analysis that highlights the benefits of each optimization on the state-of-the-art GPU architecture, NVIDIA Tesla A100. Our contributions are four-fold:

1. A fully automated OpenACC/OpenMP framework for equality saturation is proposed.
2. Static single-assignment form is combined with e-graph to optimize directive-based code.
3. We demonstrate that our approach provides significant performance opportunities for GPUs.
4. We provide a detailed analysis of kernel performance and the effectiveness of optimization techniques using the latest NVIDIA GPU architecture for HPC.

As each framework operates on one of the layers of compiler infrastructures, it allows each other optimizations to pursue a thorough performance update. All our tools are publicly available through the BSC Accelerators and Communications for HPC Group's software download page[1].

1.4 THESIS OUTLINE

The remaining chapters of this thesis are structured as follows:

- **Background** (Chapter 2): This chapter provides background knowledge on GPUs, directive-based programming, and code optimization.
- **Runtime Efforts** (Chapter 3): In this chapter, we first propose a runtime framework for OpenACC, and upon that, illustrate an extension for automated asynchronous execution and dynamic kernel specialization. Moreover, we add a code-translation method for multi-GPU utilization and distribute manually-tuned benchmarks that hardly allow precise data-dependency analysis.
- **Low-Level Code Optimization** (Chapter 4): This chapter introduces a backend environment to extend the NVIDIA GPU assembly code. There, we automate shuffle instructions without assuming programming models; thus, both OpenACC and CUDA codes can be specialized for target architecture. Applying our tool to a benchmark suite, we assess the benefit of the shuffle instructions across four generations of GPUs.
- **Source-Code Optimization** (Chapter 5): In this chapter, we explain our approach to represent general sequential code for equality saturation and optimizations and then describe our novel code generation technique from e-graphs towards high-throughput GPU execution. Our evaluation modifies the source code of practical benchmarks, shows performance benefits, and provides in-depth analysis on each technique.
- **Summary and Conclusion** (Chapter 6): Finally, this chapter summarizes our work and concludes our contribution to directive-based programming models.

1 https://www.bsc.es/discover-bsc/organisation/scientific-structure/accelerators-and-communications-hpc/team-software

2

BACKGROUND

GPU computing is the primary target whose efficiency our study tries to improve. Chapter 2 focuses on the fundamental knowledge of GPUs and programming by CUDA, directives, and PTX. We suggest several methods of optimization in relation to performance.

2.1 GPU ARCHITECTURE

Graphics Processing Units (GPUs) originated as graphic processors supporting massive parallelism. In contrast to CPUs, which are used for general computations and system management, GPUs are usually dedicated to those parts of applications featuring a high degree of parallelism. Since GPUs can hide the latency of memory requests by overlapping it with execution, the peak bandwidth of GPUs is significantly larger than that of CPUs. Furthermore, the hierarchical memory system including multi-level caches, shared memory and registers, allows exploiting spatial and temporal locality.

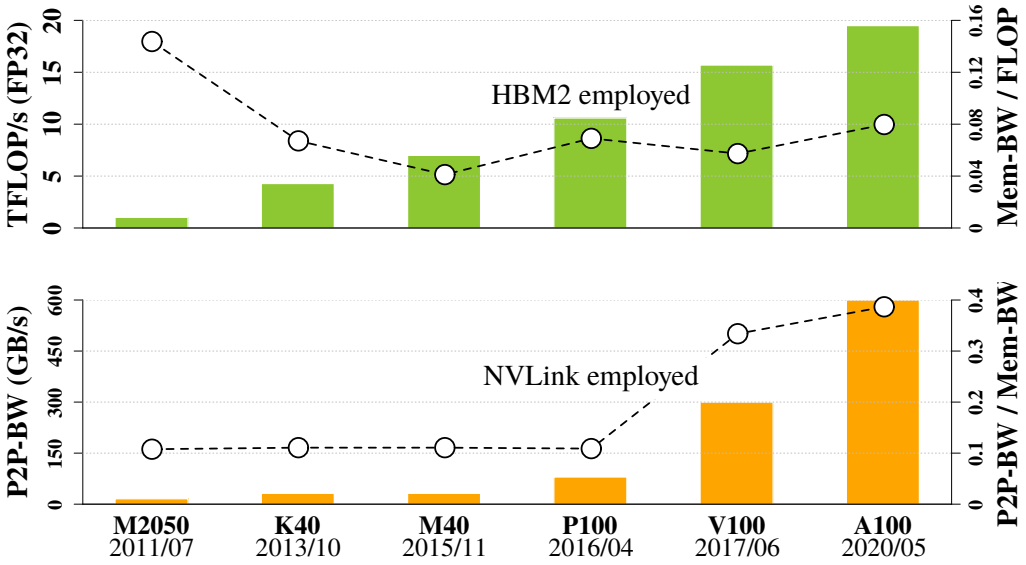


Figure 2.1: Performance changes of recent GPUs. The bars show peak performance in TFLOP/s (top) and peak GPU-to-GPU bidirectional bandwidth in GB/s. The ratio to the memory bandwidth is shown by the line plot.

We show the recent performance changes of NVIDIA GPUs [24, 25, 26, 27, 28, 29] in Fig. 2.1. Until P100, which began employing the second generation of High Bandwidth

Memory 2 (HBM2), the performance gap between memory bandwidth and computational throughput has been growing larger generation after generation. Therefore, memory accesses tend to be the performance bottleneck of applications. Currently, the NVLink interconnect [30] is enhancing peer-to-peer communication among multiple GPUs. The NVLink bandwidth allows not only compute-intensive but memory-intensive applications to utilize several devices. Notably, Unified Memory (UM) realizes data accesses that automatically solve data coherence in the entire system while performing peer-to-peer communications. However, frequent page faults cause heavy performance degradation [31]; hence, users are basically required to avoid data sharing or perform explicit memory copies among GPUs.

Figure 2.2: NVIDIA Volta GV100 GPU; taken from Fig. 4 in [28]

Fig. 2.2 shows the architecture of NVIDIA Volta GV100 GPU. One GPU has multiple *Streaming Multiprocessors* (SMs) which share one global memory. GPU programs, which are ordinarily created by the CUDA programming model as will be described later, divide the execution into *thread-blocks*. Each thread-block consists of the GPU's smallest unit of parallelism called *threads*. A thread has its own registers, and threads in one thread-block hold their own shared memory. NVIDIA Volta GV100 features 84 SMs and each SM contains 32 FP64 cores along with multiple LOAD/STORE units [28].

Fig. 2.3 shows the SM architecture. Each SM executes distributed thread-blocks in groups of 32 threads, called *warps*. Using inner parallel processing units, the SM takes advantage of Instruction Level Parallelism (ILP) as well as parallelism among warps and thread-blocks. With no speculative execution, the warp scheduler can hide the latency of load/store by another warp's execution. The execution of thread-blocks in one SM can also be overlapped as long as the summation of thread-blocks' resource requirement does not exceed the resource limitation (the number of thread-blocks, threads and registers,

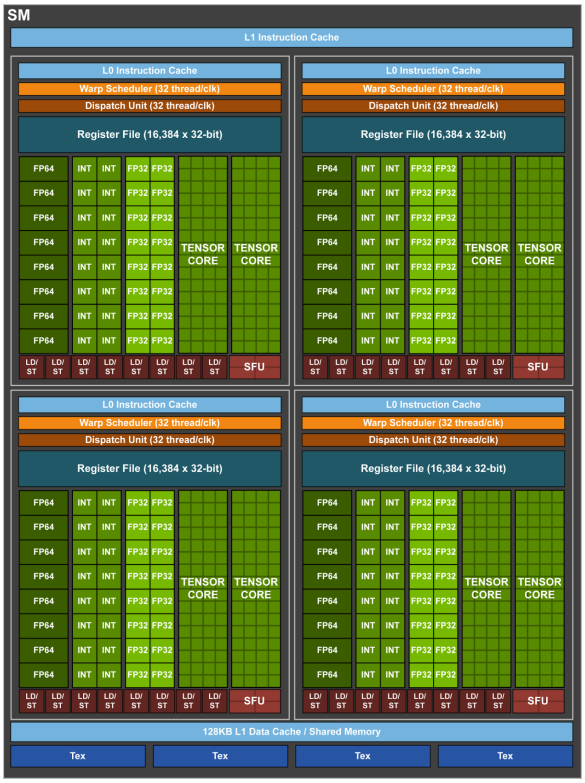


Figure 2.3: NVIDIA Volta GV100 Streaming Multiprocessor (SM); taken from Fig. 5 in [28]

the size of shared memory) of the SM. Although GPUs provide high memory bandwidth, the global memory access latency is also higher than that of CPUs [32, 33, 34]. Therefore, optimal throughput may be attained by covering memory requests with computational execution and hiding the latency of data movement. Additionally, the order of memory accesses is essential due to the memory hierarchy, because the distance among accesses by neighboring threads often results in limited bandwidth [35]. While the memory-access latency increases with the accesses to higher levels of the memory hierarchy and the concept of locality is highly respected for performance, the locality optimization brings both additional synchronization and resource use to programs. Warp-level primitives (Section 2.5), available since the NVIDIA Kepler generation of GPUs, allow for the communication among threads within the same warp [36], avoiding accesses to either shared or global memory.

2.2 CUDA PROGRAMMING

Each GPU thread shares the same program code, known as GPU *kernels*, customarily written in CUDA [3], an extended form of the C++ language for NVIDIA GPUs. Listing 2.1 shows an addition kernel in CUDA. CUDA decides the behavior of compute kernels in accordance with IDs of thread and thread-block, while accepting kernel arguments, the number of threads, and the number of thread-blocks as variables. One thread has its own registers, threads in one thread-block hold their own shared memory, and users are responsible for setting their configurations to obtain correct results. The on-chip resource

```
__global__ void add(float *c, float *a, float *b, int *f) {
    int i = threadIdx.x + blockIdx.x * blockDim.x;
    if (f[i]) c[i] = a[i] + b[i];
}
```

Listing 2.1: Addition kernel in CUDA

use decides processor occupancy, which limits final efficiency together with ILP and synchronization.

Since the host code is executed on a CPU, GPU execution can be asynchronous to CPU execution and multiple kernels can be launched on the same GPU simultaneously. Global-memory accesses are done by loading or storing through device pointers passed as kernel arguments. To keep data locally, shared memory and local variables are used. Specified local buffers are directly mapped to shared memory in a SM. Thread-private variables are mapped to registers as long as space allows. The number of registers can be restricted by the compile-time option `--maxrregcount`. However, when the CUDA compiler cannot schedule register allocation under the restriction, register spilling occurs and degrades the application's performance.

For cooperative computation among threads, CUDA provides a thread-block-level synchronization primitive `__syncthreads()` so that each thread can synchronize other threads belonging to the same thread-block. On the recent NVIDIA GPUs, the user can also use warp-level synchronization primitives `__syncwarp()`.

2.3 DIRECTIVE-BASED PROGRAMMING

Decomposing sequential execution into parallel entities is challenging. Routine calls often create data dependencies across multiple files, hindering the extraction of compute-intensive code. Even with extracted code, loops often possess intricate access patterns on different structures that challenge automatic parallelization for efficiency. Directive-based programming models, such as OpenACC [5] and OpenMP [6], alleviate the burden of introducing additional code for hardware acceleration. Both OpenACC and OpenMP provide code directives on the *de facto* standard languages for scientific applications (C/C++/Fortran) to specify offloading parts as *compute kernels* with the declaration of explicit parallelism, enabling seamless automatic accelerator use.

```
#pragma acc kernels loop independent
//#pragma omp target teams distribute
for (int i = 0; i < cy; i++) {
#pragma acc loop independent gang(16) vector(256)
//#pragma omp parallel for simd
  for (int j = 0; j < cx; j++) {
    double tmp = 0.f;
    for (int l = 0; l < ax; l++)
      tmp += a[i][l] * b[l][j];
    r[i][j] = alpha * tmp + beta * c[i][j];
}}
```

Listing 2.2: Matrix multiplication kernel in C and OpenACC (OpenMP equivalent commented)

Listing 2.2 provides an example of OpenACC code that utilizes code directives to specify the parallelism of loops. The **kernels** directive guards a sequence of kernels, while the **loop** directive explicitly sets the parallelism of a loop. Users may also specify a single kernel region by using the `parallel` directive (not shown in the listing). OpenACC's parallelism consists of three levels of abstraction: gang, worker and vector (in coarse to fine order). For instance, on the NVHPC compiler [7], both the top `i` and the middle `j` loops feature gang parallelism with the degree of the trip count `cy` and 16, respectively, and these are distributed over thread-blocks on GPUs. The middle `j` loop exposes additional vector parallelism and launches 256 threads to execute the inner statements. The blue comments show an equivalent form in OpenMP, which does not allow the reuse of parallelism across nested loops. Loop parallelism and data transfers may be implicit, and the compiler automatically solves data dependencies and sets optimal parameters. The NVHPC compiler generates embarrassingly parallel code for GPUs from directive code. On the other hand, GCC [8] utilizes a principal-agent model for both OpenACC and OpenMP, and Clang [10] follows the same approach for OpenMP.

The components of OpenACC are made of kernels and routines. An OpenACC kernel is the unit of program execution on accelerators to be launched with specified parallelism. The environment for kernel execution, such as device, copied data from/to the host, and synchronous behaviors, can be controlled by OpenACC routines. OpenACC directives are provided for specifying code segments as kernels or defining data on devices along with several options. Although data-related directives can be replaced by routine calls, OpenACC kernels have to be embedded on original source files with directives to be converted to device-specific code at compile time. Therefore, additional code segments have to be put in place along with additional variables in order to be calculated from various dynamic parameters at runtime.

Listing 2.3 shows an example of multi-device utilization in OpenACC with asynchronous execution to each other devices. Additional code segments surround the kernel, calling an OpenACC routine to switch devices (Line 2 of Listing 2.3) and setting variables to divide the loop execution (Lines 3-4 of the same figure). In real applications, many other factors such as runtime information and device-to-device communication are concerned; hence, in-situ kernel declarations bring additional complexities to directive-based software development. Also, OpenACC kernels are statically declared; thus, for a complex dynamic application the programmer is required to prepare adjusted kernels before compilation, regardless of whether runtime information is available.

```
1 for (int d = 0; d < NUM_DEVICES; d++) {
2   acc_set_device_num(d, 0);
3   int length = N/NUM_DEVICES;
4   int init   = length * d; int until  = length * (d + 1);
5   #pragma acc data copyout(x[init:length]) present(y) async(d)
6   #pragma acc parallel loop async(d)
7   for (int i=init; i<until; i++)
8     x[i] = y[i] * y[i];
9 }
```

Listing 2.3: Multi-device use in OpenACC

User-level code implemented manually in CUDA or OpenACC is brought to execution on GPUs through NVIDIA PTX [37], a virtual machine and ISA for general-purpose parallel thread execution. PTX programs feature the syntax and sequential execution flow of assembly language. Thread-specific variables are replicated to be run over SMs in parallel using the same program but different parameters. Since the actual machine code (SASS) cannot be modified from official tools [38], PTX is the closest documented and standard GPU code layer that may be modified.

PTX code consists of kernel and function declarations. Those have parameters and instruction statements along with variable declarations, labels, and predicates. Listing 2.4 provides the CUDA-generated PTX kernel from Listing 2.1. Variable declarations from several data spaces and types correspond to the usage of on-chip resources, especially registers. Accepting options and types (e.g. `.eq`, `.s32`), PTX instructions leverage defined registers and compute results, while some of these enable access to other resources (e.g., `ld.global.u32`). Predicates (`@%p1`) limit the execution of the instructions stated under them, which may lead to branching based on the thread-specific values, such as thread and thread-block IDs (`%tid.x`, `%ctaid.x`). Labels (e.g., `$LABEL_EXIT`) are branch targets and allow backward jumps that may create loops.

```
1  .visible .entry add(.param .u64 c, .param .u64 a,
2                      .param .u64 b, .param .u64 f) {
3    /* Variable Declarations */
4    .reg .pred %p<2>; .reg .f32 %f<4>;
5    .reg .b32 %r<6>;  .reg .b64 %rd<15>;
6    /* PTX Statements */
7    ld.param.u64 %rd1, [c];    ld.param.u64 %rd2, [a];
8    ld.param.u64 %rd3, [b];    ld.param.u64 %rd4, [f];
9    cvta.to.global.u64 %rd5, %rd4;
10   mov.u32 %r2, %ntid.x;      mov.u32 %r3, %ctaid.x;
11   mov.u32 %r4, %tid.x;       mad.lo.s32 %r1, %r3, %r2, %r4;
12   mul.wide.s32 %rd6, %r1, 4; add.s64 %rd7, %rd5, %rd6;
13   // if (!f[i]) goto $LABEL_EXIT;
14   ld.global.u32 %r5, [%rd7]; setp.eq.s32 %p1, %r5, 0;
15   @%p1 bra $LABEL_EXIT;
16   // %f3 = a[i] + b[i]
17   cvta.to.global.u64 %rd8, %rd2; add.s64 %rd10, %rd8, %rd6;
18   cvta.to.global.u64 %rd11,%rd3; add.s64 %rd12, %rd11,%rd6;
19   ld.global.f32 %f1, [%rd12]; ld.global.f32 %f2, [%rd10];
20   add.f32 %f3, %f2, %f1;
21   // c[i] = %f3
22   cvta.to.global.u64 %rd13,%rd1; add.s64 %rd14, %rd13,%rd6;
23   st.global.f32 [%rd14], %f3;
24   $LABEL_EXIT: ret;
25 }
```

Listing 2.4: Addition kernel in PTX (simplified)

In GPU architectures prior to NVIDIA Kepler, each sequential execution of a given thread was allowed to transfer data to another thread only through non-local memories, accompanied by a block-level or grid-level synchronization barrier. Modern GPU architectures now support additional data sharing within warps. Intra-warp communication is performed via shuffle operations. Listing 2.5 shows the `shfl.sync` instruction in PTX, in which data is shifted unidirectionally (`.up`, `.down`) across the threads of the warp, swapped in a butterfly way (`.bfly`), or exchanged by precise indexing (`.idx`).

In the unidirectional shuffle, the delta part, which has no source lane from the same warp, will be unchanged and obtain a false value in the resultant predicate (`%p1`); only the active threads (`%mask`) of the same control flow participate in the same shuffle. Inactive threads or threads from divergent flows produce neither valid results nor predicates to destination lanes. Each operation is accompanied by the warp-level synchronization, some of which are optimized away during compilation. While shuffle instructions allow for sub-warp granularity, our work focuses on the unidirectional instruction with 32 threads using 32-bit data, as applying sub-warp granularity to applications tends to feature corner cases and suffers from exception handling for intricate patterns.

```
1 activemask.b32 %mask;
2 // val[warp_id] = %src; %dst = val[warp_id-%i]
3 shfl.sync.up.b32   %dst1|%p1, %src, %i,  0, %mask;
4 // val[warp_id] = %src; %dst = val[warp_id+%i]
5 shfl.sync.down.b32 %dst2|%p2, %src, %i, 31, %mask;
6 // val[warp_id] = %src; %dst = val[warp_id^%i]
7 shfl.sync.bfly.b32 %dst3|%p3, %src, %i, 31, %mask;
8 // val[warp_id] = %src; %dst = val[%i]
9 shfl.sync.idx.b32  %dst4|%p4, %src, %i, 31, %mask;
```

Listing 2.5: The use of `shfl.sync` in PTX

Table 2.1 shows the latencies (clock cycles) of shared memory (no-conflict) and L1 cache as reported by [39], besides that of shuffle, from a microbenchmark based on [40]. In the table, Kepler is NVIDIA Tesla K80, Maxwell is M60, Pascal is P100 and Volta is V100, while Tesla K40c/TITAN X are used for the shuffle of Kepler/Maxwell. This table reveals that shuffle brings benefits over shared memory as a communication mechanism when data movement is not redundantly performed, so storing and synchronization are avoidable. In particular, latencies of L1 cache on Maxwell/Pascal are higher compared to Kepler/Volta, which integrate shared memory with L1 cache. Those allow the shuffle to be utilized as a register cache for performance improvement, but the engineering efforts in order to modify the fundamental parts of parallel computation are considerably high.

Table 2.1: Latencies (clock cycles) reported in [39, 40]

Name	Shuffle (up)	Shared-Memory Read	L1 Hit
Kepler	24	26	35
Maxwell	33	23	82
Pascal	33	24	82
Volta	22	19	28

3

RUNTIME EFFORTS

Our work introduces dynamic analysis and compilation to OpenACC directive-based programming, allowing further efforts on optimization at runtime. All the components of OpenACC, here, are provided as runtime routines leveraging existing compilers. By transforming directives into a sequence of routine calls, OpenACC compilers can enable on-the-fly features such as kernel specialization and load-balancing.

3.1 JACC

```
1 #pragma acc data copyout(x[0:N]) present(y)
2 #pragma acc parallel loop
3 for(int i=0; i<N; i++) x[i] = y[i] * y[i];
```

Listing 3.1: Accelerator programming in OpenACC

```
1  /* Entry of #pragma acc data */
2  jacc_create(x, N * sizeof(float));
3
4  /* #pragma acc parallel loop */
5  jacc_kernel_push(
6    "#pragma acc parallel present(x, y)\n"
7    "#pragma acc loop\n"
8    "for(int i=0; i<N ; i ++) /* ... */",
9     /* args */, /* flags */);
10
11 /* Exit of #pragma acc data */
12 jacc_copyout(x, N * sizeof(float));
```

Listing 3.2: Converted code to JACC (arguments omitted)

We build JACC, a just-in-time compilation system for OpenACC, in which input directives are replaced with runtime routines. JACC hides every OpenACC feature behind a runtime library to cushion dependency to specific compilers. Once a kernel is compiled for the first time, its device code is cached to be reused for subsequent launches. Even though JACC is developed upon existing compilers, it allows calling of CUDA routines and kernels through its library. Listing 3.2 shows the converted code of Listing 3.1, which shows an OpenACC code written in C that updates array x with the multiplication of array y, to call runtime routines. First, combined directives (e.g. **`parallel loop`** of Line 2 of Listing 3.1) are decomposed into three basic directives of **`parallel`**, **`loop`** and **`data`**.

```
1  void kernel0(float *x,
2               float *y, size_t N) {
3    #pragma acc parallel present(x, y)
4    #pragma acc loop
5    for(int i=0; i<N; i++) x[i] = y[i] * y[i];
6  }
```

Listing 3.3: Generated code for kernel launch (formatted)

Then, for each directive, JACC inserts corresponding routines that are implemented in its library, shown in Listing 3.2 (Lines 2, 5 and 12).

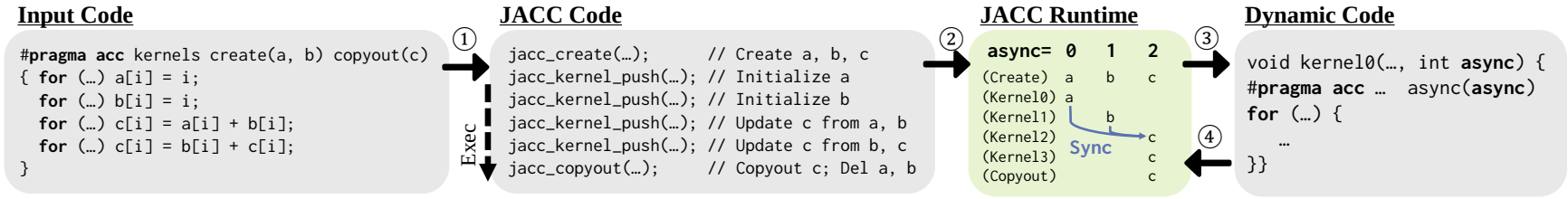
During program execution, JACC data-related routines that wrap OpenACC routines (Lines 2 and 12 of Listing 3.2) assume the roles of the original directives. The routine `jacc_kernel_push` launches kernel execution while accepting source code in a string with arguments that hold runtime information (Lines 5-9 of the same figure). It should be noted that the `loop` directive is used for marking parallelism; therefore, the directive is kept in kernel strings. When the routine finds no compiled kernel for a given source code or needs to update existing kernels, function code (Listing 3.3) is generated to emit device code by a specified compiler and to have additional arguments for code extension. After linked dynamically, this function is called through a Foreign Function Interface (FFI) for passing arbitrary arguments. JACC's library for each routine is extended to collect runtime information and support dynamic features.

3.2 BASIC EXTENSION

Whereas the OpenACC APIs are added explicitly to a program as directive clauses with a specific intent [41, 20], the intent and effect of base-language modifications to a program are implicit. There exist a large body of work that studies the implications of base-language modification on OpenACC compilation [42, 17, 43, 44]. JACC works as a runtime solution for dynamic optimization for both the OpenACC APIs and base-language program modification. Because of JACC's ability to handle both types of optimizations, it can automatically overlap kernel execution, and thus achieve inter-kernel parallelism. Also, additional on-the-fly kernel specialization using runtime information extracted from profiling results for better resource use and intra-kernel parallelism are possible.

3.2.1 *Automated Asynchronous Execution*

Since JACC automatically provides a function interface for each OpenACC kernel, additional runtime information needed for extended execution shown in lines 2-5 of Listing 2.3, can be passed through as arguments to the JACC function interface without the need to generate redundant code snippets. Instead of compiler-generated code, which is hard to manage as global program information, JACC's runtime calculates the required arguments on its own runtime environment and then proceeds to call the kernel functions using them. The benefit of this approach is that information across multiple execution instances can be easily shared for further optimizations. Moreover, JACC provides the ability to update kernels dynamically with additional runtime information after program




```
1  typedef enum { NONE = 0, ARRAY = 1, STATIC = 2, PRESENT = 4, REDUCTED = 8, DIST = 16, WRITTEN = 32, READ = 64
2  } Jacc_Attr;
3
4  typedef struct Jacc_Arg {
5    const char *type; /* "int" */  const char *symbol; /* "a" */  void *variable_address; /* &a */
6    void *addr; /* a */  size_t size; /* sizeof(a) */  int attr; /* STATIC | ARRAY | READ | PRESENT */
7    struct Jacc_Arg *next;
8  } Jacc_Arg;
9
10 /* jacc_create(a, 10000); */
11 void jacc_create(void *a, size_t len);
12 /* jacc_kernel_push("#pragma parallel loop\n for (..) a[i] = i;", (Jacc_Arg) { "int", ... }); */
13 void jacc_kernel_push(const char *code, Jacc_Arg *arg);
14 /* jacc_copyout(a, 10000); */
15 void jacc_copyout(void *a, size_t len);
```

Figure 3.1: Execution flow of automated asynchronous execution. First, the JACC code is generated from the input ①. Following the execution, the runtime manages data declarations and checks data dependency among asynchronous queues based on data ranges of actual kernel arguments ②. Dynamic code is created from kernel code to be executed on a destined queue based on the kernel argument **async** ③ and synchronization is performed with the host if necessary ④. The code below provides the prototype of each routine and the definition of each data type with actual use in comments. The description of JACC's routines is provided in Section 3.1.

compilation is invoked, thus, providing a straightforward mechanism for runtime extensions of original kernel declaration.

For asynchronous execution, we automatically overlap kernel launches and data operations with each other as well as host execution. Each JACC routine has the ability to track array references, and if data dependencies are encountered across two or more routines, JACC will schedule them in the same asynchronous queue. When multiple queues are concerned, synchronous operations are performed only among those queues that require them, while skipping redundant synchronization on already solved dependencies. JACC achieves this by maintaining timestamps of data accesses and the most recent synchronization among queues. If there is no data dependency to prior execution, the least recently used queue is selected. We guarantee original code semantics by obligatory synchronization that is performed immediately after kernel execution that deals with array writes to undefined data regions and explicit variable updates such as reduction. Data ranges linked to given pointers are tracked through JACC's runtime routines, and managed in a red-black tree as OpenACC compilers do to accept any address of declared data [45].

Fig. 3.1 shows JACC's automatic asynchronous code optimization and execution flow. During execution by the JACC runtime (Step 2 of Fig. 3.1), synchronization between queues is performed. For example, before `Kernel2` execution is allowed, a synchronization call is performed to wait for the updated arrays `a` and `b`. However, for `Kernel3`, the dependency on `b` is already solved by the previous synchronization, thus the execution does not wait for other queues. For the overlapping execution, during the dynamic code generation (Step 3 of Fig. 3.1), the OpenACC clause `async` is set for each kernel declaration, and asynchronous execution queues are selected.

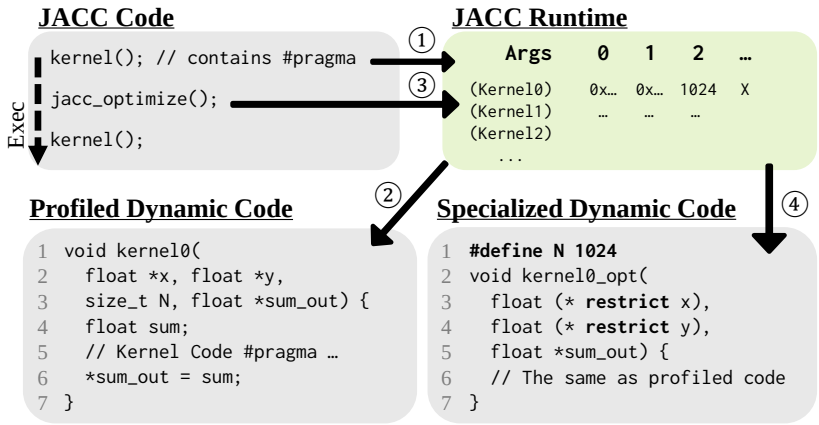


Figure 3.2: Compilation flow with on-the-fly kernel specialization. To collect runtime information about kernel arguments, mainly addresses of pointers and values of variables, profiling execution is conducted ① with original kernel declarations ②. With a user-invoked compilation event ③, specialized code is generated based on the profile results and thereafter used for succeeding kernel execution ④. The `sum` variable in the profiled code is exported through the `sum_out` pointer, thus it is kept for specialized code as a dynamic variable (notated as "X" in the argument log of the JACC runtime). `0x..` indicates an address value obtained from an expression such as `x` and `y` (The `addr` member of `Jacc_Arg` in Fig. 3.1).

3.2.2 *Kernel Specialization*

We attempt to refine resource use by attaching runtime information to function code so as to enable aggressive optimization in kernel compilation. The compilation flow is shown in Fig. 3.2. For specialization, profile execution is conducted before the optimization; then, at an additional compilation event (`jacc_optimize` of JACC code in Fig. 3.2), parameters being invariable during the execution are substituted with constants (Line 1 of specialized code in Fig. 3.2) to lower on-chip resource use. Besides that, pointer references that never conflict with others are declared with the `restrict` keyword to ensure intra-kernel parallelism (Lines 3-4 of specialized code in Fig. 3.2). Even though user-invoked events can be substituted automatically by existing just-in-time compilation techniques [46], we explicitly invoke the desired compilation optimizations to measure the potential performance benefits.

3.3 MULTI-GPU UTILIZATION WITH PREDICATES

We further improve utilization of intra-kernel parallelism by enabling multi-GPU execution with JACC. Whereas previous studies have persistently focused on loop splitting over plural GPUs [21, 47], this work divides data regions that each GPU updates to support real applications that usually entangle memory accesses among loop iterations.

3.3.1 *Predicate-Based Filtering*

Our technique, named *predicate-based filtering*, limits memory accesses depending on data regions that the GPU writes to, assuming that redundant computational code and parameters do not degrade performance due to low computational latency and high memory latency on GPUs. First, we introduce data ranges for each updated array so that array writes can be filtered based on the assigned range. For instance, in C code, array write `a[i]*=2` is rewritten to `(a_lb <= i && a_ub >= i) ? a[i]*=2 : a[i]`, where `a_ub` and `a_lb` indicate the upper and lower bound of array `a`, that are specified depending on the GPU. In Fortran, since there is no nested assignment, we use IF statement for filtering, with subsequent ELSE statement which contains an assignment of the same expression (`a(i)=a(i)`) that is later optimized away but facilitates compiler analysis. Additionally, we develop data-flow analysis for the innermost parallel region in each kernel to detect data dependencies between arrays. Then, we filter them to restrict accesses while solving dependencies as shown in Fig. 3.3. This analysis converts both array and variable references into the Static Single Assignment (SSA) form, and iteratively finds dependencies among array accesses.

Updated data are sent to all other GPUs after each kernel execution to establish data coherency. Device-memory allocations and host-to-GPU communications are replicated on all the GPUs and the primary GPU is used for GPU-to-host transfers. To guarantee the result of our analysis, we check kernel arguments so as to duplicate computation and disable communications on data that are referred through more than two pointers which at least one of them is read and one is written (i.e. aliased pointers, which are usually avoided in OpenACC programs for loop independence). When several pointers share the

```
1  a[i]=x; b[i]=a[i]; x=c[j]; a[k]=x; b[k]=a[k];
```

```
1  /* a[i]=x */
2  ((a_lb<=i && a_ub>=i)||
3   (b_lb<=i && b_ub>=i)) ? a[i]=x:a[i];
4  /* b[i]=a[i] */
5  ((b_lb<=i && b_ub>=i)) ? b[i]=a[i]:b[i];
6  /* x=c[j] */
7  x=((a_lb<=k && a_ub>=k)||
8     (b_lb<=k && b_ub>=k)) ? c[j]:0;
9  /* a[k]=x */
10 ((a_lb<=k && a_ub>=k)||
11  (b_lb<=k && b_ub>=k)) ? a[k]=x:a[k];
12 /* b[k]=a[k] */
13 ((b_lb<=k && b_ub>=k)) ? b[k]=a[k]:b[k];
```

Figure 3.3: Example of predicate-based filtering in C code. Original (top) and filtered code (bottom). References to array `a` have predicates for updating array `b` and itself (Lines 2-3 and 10-11), the references to array `b` have for itself (Lines 5 and 13), and the reference to array `c` has for array `a` and `b` (Lines 7-8).

same array to update, we merge their access ranges to follow the widest. The necessary computation for array-write indexing is always duplicated. Regarding reduction or variable writes that are explicitly exported to host, we filter the computation based on the range of the outermost parallel iterator.

Fig. 3.4 shows the actual cases of the transformation to predicated code. Indirect accesses are supported just by filtering dependencies which include loop ranges and array indices (Lines 5-8 and 13-14 in Fig. 3.4 (b)). The OpenACC atomic operation is converted to be predicated around the operation (Lines 28-30 in Fig. 3.4 (b)). We filter nested structures at the shallowest array accesses of the member reference since otherwise their ranges could be vector values that require additional array accesses during kernel execution (Lines 45-49 in Fig. 3.4 (b)). The calculation of array-write indices is always outside the filters to be used in predicates (Lines 54-55 in Fig. 3.4 (b)). The detail of multidimensional access is described in Section 3.3.2. Those are common patterns in OpenACC programs, while precise data-dependency analysis such as polyhedral computation is hardly possible to treat the parallelism due to the complexity of the analysis and the limitation to affine loops [48, 49].

3.3.2 *Division of Multidimensional Arrays*

While being applicable to all OpenACC kernels as far as array writes are concerned, our filtering technique needs to duplicate execution on each GPU when references between split ranges (such as all-to-all dependencies in Listing 3.4) are found inside the kernel. We alleviate this restriction by leveraging dimensional information.

If multidimensional arrays are linearly split regardless of the dimensional characteristic, the data dependency could be dispersed to the entire sections of array accesses. For example, the write to `lhsY` in Listing 3.4 (Lines 7-14) would be filtered for succeeding reads;

(a) **Original Code**

```
// Indirect Access (NPB-CG)
#pragma acc parallel/
  num_gangs(end) num_workers(4)
  vector_length(32)
{
  #pragma acc loop gang
  for (j = 0; j < end; j++) {
    tmp1 = rowstr[j];
    tmp2 = rowstr[j+1];
    sum = 0.0;
    #pragma acc loop worker/
         vector reduction(+:sum)
    for (k = tmp1; k < tmp2; k++) {
      tmp3 = colidx[k];
      sum = sum + a[k]*p[tmp3];
    }
    q[j] = sum;
  }
}
```

Basic Idea:
- Split Array-Writes Evenly
- Filter Access to Dependencies
- All-to-All Communication

```
// Atomic Operation ------------
#pragma acc parallel loop
for (int i = 0; i < N; i++) {
  #pragma acc atomic
  h[a[i]] += 1
}
```

Filter Around the Operation

```
// Multidimentional Access -----
#pragma acc parallel
for (int k = 0; k < N; i++) {
  a[x][k*k][y] = a[x-1][k][y];
}
```

Filter on Parallel Dimensions

```
// Nested Structures -----------
#pragma acc kernels
for (int i = 0; i < M; i++) {
  for (int j = 0; j < N; j++) {
    a.x[j].y[i].p += 1;
    b[j].z[i]     += 1;
  }
}
```

Filter with
the Shallowest Arrays

```
// Write-Index Calculation -----
#pragma acc parallel
for (int i = 0; i < N; i++) {
  int i2 = index[i];
  int i3 = out1[i2] + 1;
  out2[i3]++;
}
```

Always Calculate Write-Indices

(b) **Predicated Code**

```
#pragma acc parallel num_gangs (end)/
      num_workers(4) vector_length(32)
#pragma acc loop gang
for(j = 0; j < end; j++) {
  tmp1 = ((r_lb <= j) && (r_ub >= j)) ?
                       rowstr[j]     : 0;
  tmp2 = ((r_lb <= j) && (r_ub >= j)) ?
                       rowstr[j + 1] : 0;
  sum = 0.0;
  #pragma acc loop worker vector/
                   reduction(+:sum)
  for (k = tmp1; k < tmp2; k++) {
    tmp3 = ((r_lb <= j) && (r_ub >= j)) ?
                           colidx[k] : 0;
    sum = sum +
     (((r_lb <= j) && (r_ub >= j)) ?
                              a[k] : 0) *
     (((r_lb <= j) && (r_ub >= j)) ?
                              p[tmp3]:0);
  }
  ((r_lb <= j) && (r_ub >= j)) ?
                    (r[j] = sum) : r[j];
}
```

```
#pragma acc parallel
#pragma acc loop
for (int i = 0; i < N; i++) {
  if ((h_lb <= a[i]) && (h_ub >= a[i]))
    #pragma acc atomic
    h[a[i]] += 1;
}
```

```
#pragma acc parallel
for (int k = 0; k < N; i++) {
  ((a_lb <= k*k) && (a_ub >= k*k)) ?
   (a[x][k*k][y] = a[x-1][k][y]) :
   a[x][k*k][y];
}
```

```
#pragma acc parallel
#pragma acc loop
for (int i = 0; i < M; i++) {
  #pragma acc loop
  for (int j = 0; j < N; j++) {
    ((a_x_lb <= j) && (a_x_ub >= j)) ?
       (a.x[j].y[i].p += 1) :
       a.x[j].y[i].p;
    ((b_lb <= j) && (b_ub >= j)) ?
       (b[j].z[i]     += 1) : b[j].z[i];
}}
```

```
#pragma acc parallel
for (int i = 0; i < N; i++) {
  int i2 = index[i];
  int i3 = out1[i2] + 1;
  ((out2_lb <= i3) && (out_ub >= i3)) ?
        out2[i3]++ : out2[i3];
}
```

Figure 3.4: Original code (left) and corresponding predicated code (right)

```
1  #pragma acc parallel loop gang
2  for (i = 1; i <= gp02; i++) {
3   #pragma acc loop worker vector
4   for (k = 1; k <= gp22; k++) {
5    for(m = 0; m < 5; m++)
6     for(n = 0; n < 5; n++) {
7      lhsY[n][m][BB][jsize][i][k] =
8        hsY[n][m][BB][jsize][i][k]
9        - lhsY[n][0][AA][jsize][i][k]
10       * lhsY[0][m][CC][jsize-1][i][k]
11    /* - lhsY[n][1..3][AA][jsize][i][k]
12       * lhsY[1..3][m][CC][jsize-1][i][k] */
13       - lhsY[n][4][AA][jsize][i][k]
14       * lhsY[4][m][CC][jsize-1][i][k];
15     }}}
```

Listing 3.4: Kernel code from NPB-BT. Two inner loops are unrolled in actual code. Linear splits cause all-to-all dependencies among statements.

thus, all the computations would be duplicated on each GPU. Here, we utilize parallel iterators (such as `i` and `k` in Listing 3.4) to locate *parallel dimensions*, where arrays can be split without duplicated computation. Based on the number of iterators each dimension contains, we select the parallel dimension for each updated array to have the most parallel iterators while containing the least sequential iterators (such as `m` and `n`). When there are several candidates, we choose the leftmost dimension in the C language and the rightmost dimension in Fortran to gain better performance with suitable accesses to the memory layout (row-major and column-major order, respectively).

Each kernel execution is performed while equally dividing parallel dimensions among GPUs and accompanied by the GPU-to-GPU communication through `cudaMemcpy2DAsync`. Each array is concurrently transferred regarding other data and other GPUs. We synchronize GPUs at the beginning and ending of the communication.

3.3.3 *Adaptive Utilization*

In order to avoid lower performance due to data distribution overheads, we enable multi-GPU execution for each kernel in an adaptive way, while otherwise duplicating computation on all GPUs and performing no GPU-to-GPU communication.

First, we start the execution on the mode of duplication. After an initial warm-up run, we profile the average ratio of array-write size (WriteSize) to execution time ($\mathrm{time}_{\mathrm{Kernel}}$) as $\mathrm{eff}_{\mathrm{dup}}$, until we observe five executions that satisfy the requirement to have the peak performance be better than duplication:

$$\mathrm{time}_{\mathrm{Kernel}} \;>\; \mathrm{time}_{\mathrm{Kernel}}/n_{\mathrm{GPUs}} + \mathrm{WriteSize}/\mathrm{peak}_{\mathrm{P2P}} \tag{3.1}$$

Here, $\mathrm{peak}_{\mathrm{P2P}}$ is the unidirectional bandwidth of one GPU-to-GPU connection (e.g. 25GB/s in NVIDIA DGX-1) and n_{GPUs} is the number of GPUs used.

After switching to multi-GPU execution, we disable it when either one of the two following conditions is satisfied at least five times and the average difference of the left value and the smaller right value goes above zero in equations (3.2-3.3).

1. The total execution time including the communication time ($\mathrm{time}_{\mathrm{Comm}}$) becomes longer than the kernel execution time multiplied by n_{GPUs}:

$$\mathrm{time}_{\mathrm{Kernel}} + \mathrm{time}_{\mathrm{Comm}} \quad > \quad \mathrm{time}_{\mathrm{Kernel}} \times n_{\mathrm{GPUs}} \tag{3.2}$$

2. The total execution time surpasses the profiled execution time of duplication:

$$\mathrm{time}_{\mathrm{Kernel}} + \mathrm{time}_{\mathrm{Comm}} \quad > \quad \mathrm{eff}_{\mathrm{dup}} \times \mathrm{WriteSize} \tag{3.3}$$

The first condition excludes the case that the GPU-to-GPU communication has larger latencies than expected. The second prevents performance degradation caused by kernels that are unsuccessfully parallelized.

3.3.4 *Implementation*

We integrate predicated-based filtering into JACC, which translator is implemented as an XcodeML [50] converter. The execution flow of predicate-based filtering is shown in Fig. 3.5. From OpenACC code in C or Fortran, our implementation generates JACC code, in which kernel code is embedded as strings. Although the kernel code can be translated at runtime, we apply predicated-based filtering beforehand for our experiments; thus, the embedded kernel code already has the predicates. JACC's runtime code generator is utilized for setting array ranges and constructing multi-GPU reduction code based on the arguments of runtime routines. Runtime overheads of JIT dynamic compilation are well

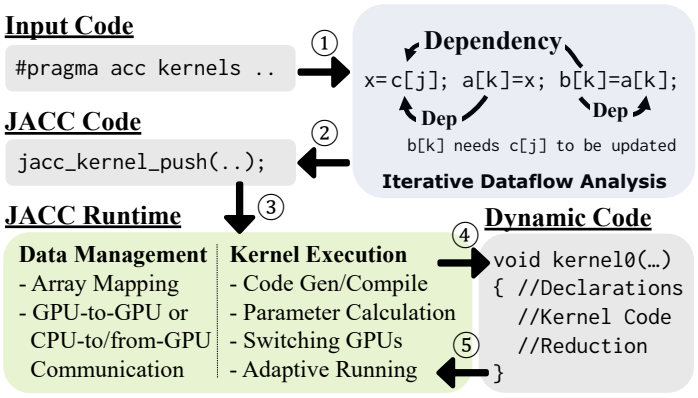


Figure 3.5: Execution flow of predicate-based filtering. Having the results of iterative dataflow analysis ①, JACC code is generated from the input ②. On the execution ③, dynamic code is created from kernel code, extended declarations and reduction code in accordance with runtime information ④. Once the dynamic code is compiled with a specified compiler, the kernel execution is conducted with dynamic parameters while switching GPUs and reflecting results to the host and each other devices ⑤. In the example of the figure of iterative dataflow analysis, updating `b[k]` needs the updated value of `a[k]` as well as `c[j]` to update `a[k]`. Both the read from `c[j]` and the write to `a[k]` are predicated with the range of write access to `a[k]` and `b[k]`.

known, and itself a target of research [46]; however, it is outside the scope of this work and left as a possible future area for optimization. Thus, we evaluate the performance without dynamic compilation overheads (Section 3.2.2), which is on average about 2 seconds per kernel for the initial compilation.

For simultaneous execution of multiple GPUs, we use OpenMP rather than OpenACC's asynchronous mechanism that holds some non-negligible latencies [51, 52]. OpenMP's pragmas are put only inside JACC's library. Whereas GCC does not allow the mix of OpenACC and OpenMP, our separated-compilation strategy realizes a combinatory use for both NVHPC and GCC. The OpenMP use is not inherent here and our techniques introduced in this work are general enough to support other compilers and other programming models.

Fig. 3.6 shows actual JACC code in Fortran. The kernel arguments are built with static and runtime parameters to create dynamic code correctly and the kernel code is set with line information for debugging purposes (Lines 3-8 and Line 14 in JACC code of Fig. 3.6, respectively). For predicate-based filtering, JACC's runtime automatically launches the dynamic code on each GPU and performs GPU-to-GPU communications based on those passed arguments.

Input Code `CloverLeaf calc_dt_kernel.f90`

```
1  !$ACC KERNELS
2  ! …
3  !$ACC LOOP INDEPENDENT REDUCTION(min:dt_min_val) GANG(128)
4    DO k=y_min,y_max
5  !$ACC LOOP INDEPENDENT REDUCTION(min:dt_min_val)
6      DO j=x_min,x_max
7        IF(dt_min(j,k).LT.dt_min_val) dt_min_val=dt_min(j,k)
8      ENDDO
9    ENDDO
10 !$ACC END KERNELS
```

JACC Code

```
1  BLOCK
2    INTEGER(KIND=8) :: arg = 0
3    CALL jacc_arg_build(&
4      &"real\0", "dt_min\0“,            &! Type, Symbol
5      & c_loc(dt_min), sizeof(dt_min),  &! Address, Size
6      & …, &! Analysis Info (e.g. WRITE, PRESENT, REDUCTED)
7      & lbound(dt_min), ubound(dt_min), &! Boundary
8      & arg)
9    CALL jack_arg_build(…)
10   …
11   CALL jacc_kernel_push(&
12   && ! Kernel Code
13   "  !$ACC PARALLEL PRESENT(dt_min) NUM_GANGS(128)\n &
14   & # 139 "calc_dt_kernel.f90"\n &
15   & … !$ACC END PARALLEL\0“, arg)
16 END BLOCK
```

Figure 3.6: Actual case of JACC code generation in Fortran from the input (top). The arguments and kernel code are partially shown in JACC code (bottom).

3.4.1 *Hardware and Software*

We measure the performance changes of our proposed techniques using the NVIDIA Tesla V100 SXM2 GPUs (16GB Memory) on NVIDIA DGX-1. DGX-1 contains eight GPUs, where each four GPUs are interconnected with NVLink in an all-to-all fashion. Additionally, each GPU has one NVLink connection to one of the other four GPUs, while the remaining GPUs and CPUs are connected with PCI Express Gen3 x16. We use only the tightly coupled four GPUs to perform our experiments where each link offers a unidirectional bandwidth of 25GB/s, while $GPU_0 \leftrightarrow GPU_3$ and $GPU_1 \leftrightarrow GPU_2$ are dually linked. Tesla V100 has a peak single-precision performance of 15.7 TFLOP/s and a peak memory throughput of 900GB/s. DGX-1 uses dual 20-core Intel Xeon processors.

For the compilation, we use the NVHPC compiler 20.9 and GCC 10.2.0 with CUDA 11.0. We compile OpenACC kernels and our runtime library using the following sets of compiler arguments: `"-O2 -acc -mp -ta=tesla:cc70 -Mcuda"` for NVHPC and `"-O2 -f(openacc|openmp) -foffload=nvptx-none -foffload=-lm -fno-strict-aliasing"` for GCC. Currently, the Fortran translation is tested only for predicate-based filtering with NVHPC. The experiments with GCC are conducted while omitting the `worker` parallelism so as not to exceed the maximum number of threads allowed in GCC per thread-block.

3.4.2 *Benchmarks*

For the evaluation, we use a manually-tuned OpenACC version of NAS Parallel Benchmarks (NPB) written in C [53] and three Fortran mini-apps: CloverLeaf [54], CCS-QCD [55] and the Himeno benchmark [56]. Each benchmark of NPB, briefly described in Table 3.1, is executed with the largest problem size for the target GPUs (Class C) except EP, where we choose Class D for longer execution. Moreover, to prolong the execution time of CG and MG, we multiply the number of iterations by 50. With regard to CloverLeaf, we select the same input as the SPEC ACCEL benchmark suite [57], while maximizing the GPU memory utilization of the other two mini-apps.

EP is the only application to be compute-bound for arithmetic operations of random-number generation involving fewer array accesses, while other benchmarks become memory bound on state-of-the-art accelerators. There is no data dependency among parallel threads in EP.

BT/LU/SP deal with three-dimensional Computational Fluid Dynamics (CFD) with different solvers. BT updates multidimensional arrays from 3D to 6D, especially 4D to 6D have the leftmost 1D to 3D dimensions for loop-independent indices as shown in Fig. 3.4 (Lines 7-14), respectively. LU/SP use 3D to 4D arrays in a similar fashion to BT. LU has relatively quicker execution for each kernel compared to BT/SP and the latency of BT mainly consists of the execution of several time-consuming kernels.

FT conducts 3D fast Fourier transform (FFT) incurring all-to-all accesses, and MG is the benchmark of multigrid computation which requires long and short communications. CG calculates a minimum eigenvalue of a sparse symmetric positive matrix with the conjugate gradient method, causing irregular accesses to an updated 1D array.

Table 3.1: Benchmark description

Name	Description	Dependency	Problem Size	Memory	Num Kernels
BT	CFD with Block Tri-Diagonal Solver	Halo (3D)	Class C: 162x162x162 (FP64), iter=200	4.915 GB	46
CG	Minimum-Eigenvalue Calculation	Irregular	Class C: size=150000 (FP64), iter=3750	0.747 GB	16
EP	Random-Number Generation in Parallel	None	Class D: size=137438953472 (FP64)	2.305 GB	4
FT	Discrete 3D Fast Fourier Transform	All-to-All	Class C: 512x512x512 (FP64), iter=20	9.317 GB	12
LU	CFD with Lower-Upper Gauss-Seidel Solver	Halo (3D)	Class C: 162x162x162 (FP64), iter=250	1.471 GB	59
MG	Multigrid Discrete Poisson Equation	Long & Short	Class C: 512x512x512 (FP64), iter=1000	6.114 GB	16
SP	CFD with Scalar Penta-Diagonal Solver	Halo (3D)	Class C: 162x162x162 (FP64), iter=400	1.700 GB	65
CloverLeaf	2D Euler Equations Solver	Halo (2D)	3840x1920 (FP64), step=1800	1.749 GB	114
CCS-QCD	Lattice QCD Simulation	Halo (3D)	32x32x32x128 (FP64), iter=1000 for BiCGStab	15.255 GB	27
Himeno	19-point Jacobian Stencil Computation	Halo (3D)	Size XL:1024x512x512 (FP32), iter=1000	14.409 GB	2

CloverLeaf is a hydrodynamics mini-application consisting of 100+ kernels, which sufficiently demonstrates the workflow of real-world applications. Based on Euler's method, 2D stencil grids are updated by each kernel having minimum control logics and halo accesses through double buffers for avoiding dependencies among loop iterations.

CCS-QCD simulates lattice quantum chromodynamics (QCD) with a linear-equation solver for a large sparse matrix in 3D. The execution is mainly composed of the biconjugate gradient stabilized method (BiCGStab) along with neighboring transfers and all-to-one reduction. The reduction accumulates slight errors upon each execution, which affect the total number of iterations when the computational order is changed due to multi-GPUs. Therefore, we fix the number of iterations for BiCGStab to 1,000.

Himeno iteratively updates a 19-point stencil grid according to Jacobi's method. The code structure is far simpler than the other two mini-apps. The kernels are constructed from three nested loops where each iteration corresponds to the grid's dimension and no dependence exists among them. Optimally, 3D halo accesses solve inter-kernel dependencies, and computational errors are reduced after kernel execution.

We report performance after an initial warm-up run that causes runtime compilation and profiling for adaptive utilization. The benchmark-reported data is quoted for the result of NPB and the total execution times for the Fortran mini-applications.

3.5 RESULTS

3.5.1 *Basic Extension*

Fig. 3.7 shows the performance changes with JACC's basic extension for both the NVHPC compiler and GCC using NPB. The `JIT w/ (NVHPC/GCC)` bars indicate the performance of converted code without any optimization. Here, only one asynchronous queue is used with `+Async`, whereas 16 queues are used with `+Overlap`. Along with that, the `+Var Opt` execution adds kernel optimization with constant parameters transformation discussed in Section 3.2.2. Furthermore, `+Restrict` adds `restrict` to pointers. Since GCC produces incorrect results with the original code of CG/LU/MG, they are omitted from results.

First, performance degradation is observed for converted code compared to original code in the case of BT/LU with NVHPC, where generated code fails to leverage static array sizes for some optimization at compile time because static arrays are separately declared. However, improvements are observed in the case of MG with NVHPC and EP with GCC. Otherwise, original performances are mostly kept. On average, asynchronous execution with single queue achieves better performance by 3.43% with NVHPC and 22.08% with GCC, respectively. However, the time-consuming kernels in each benchmark prevent overlapping execution; thus, `+Overlap` does not improve the performance from `+Async`. With `+Restrict`, we achieve better performance up to 23.39% in the case of BT/LU with NVHPC and 5.59% less performance in EP with GCC. The performance difference between NVHPC and GCC is primarily caused by the latency of memory allocation; NVHPC owns memory pools for device memory, while GCC does not.

The `+Var Opt` version has no performance change in most cases and rather worsen efficiencies in the case of EP/SP with GCC. Further exploration showed that some cases of `+Var Opt` suffer from limited arithmetic unit utilization caused by ineffective threads

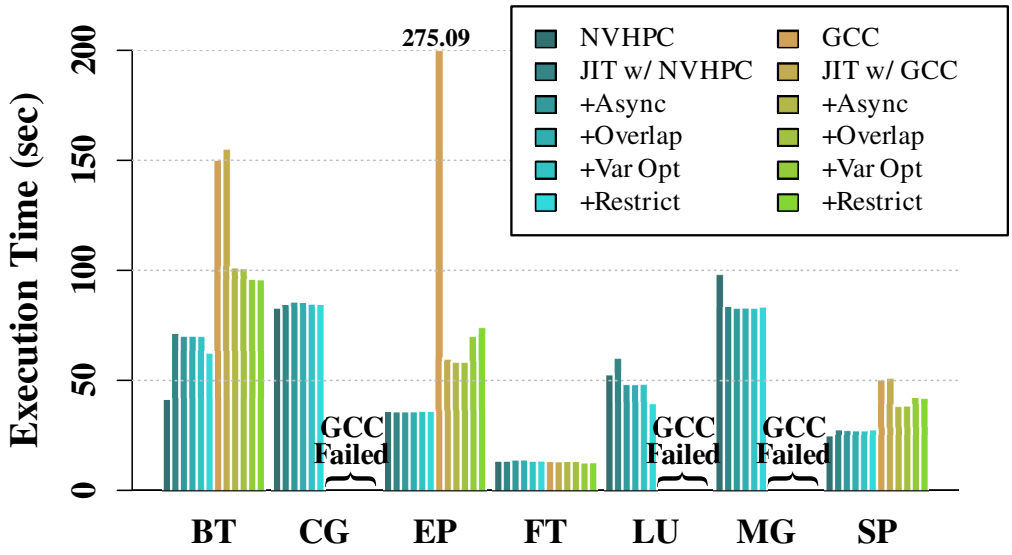


Figure 3.7: Async and kernel optimization on NVIDIA Tesla V100 SXM2

which are created due to reduced register use. On the other hand, performance improvements of `+Restrict` are achieved by parallelized memory accesses which require additional registers.

3.5.2 *GCC Custom Allocation*

Since the original version of GCC suffers the performance degradation by GPU-memory allocation, we integrate memory pools into GCC's runtime library libgomp for our multi-GPU experiments to show explicitly the performance improvements by kernel parallelization. We prepare two pools: one is for user-invoked memory allocation such as through pragmas and runtime routines. The other is for runtime-managed allocation of variables and stacks, which naturally tends to be much smaller than the former. We manage those pools to keep unused memory segments and reuse them for new allocation by selecting the smallest but capable segment on the device.

With the memory-pool integration, GCC's efficiency becomes competitive to NVHPC while having -7.83% ~ 5.05% better throughputs for the plain JACC code except the case of EP, where the kernel execution poses a 38.31% overhead due to GCC's device-code efficiency, as shown in Fig. 3.8.

3.5.3 *Multi-GPU Utilization*

3.5.3.1 *Total Improvements & Kernel Speedups*

We show the overall performance and kernel speedups with predicate-based filtering in Fig. 3.8 and Fig. 3.9. When compared to single-GPU execution, the total execution time with our proposed technique is better in five among 10 evaluated benchmarks, from 4.05% up to 43.43% when enabling four GPUs. Especially, when only the kernel

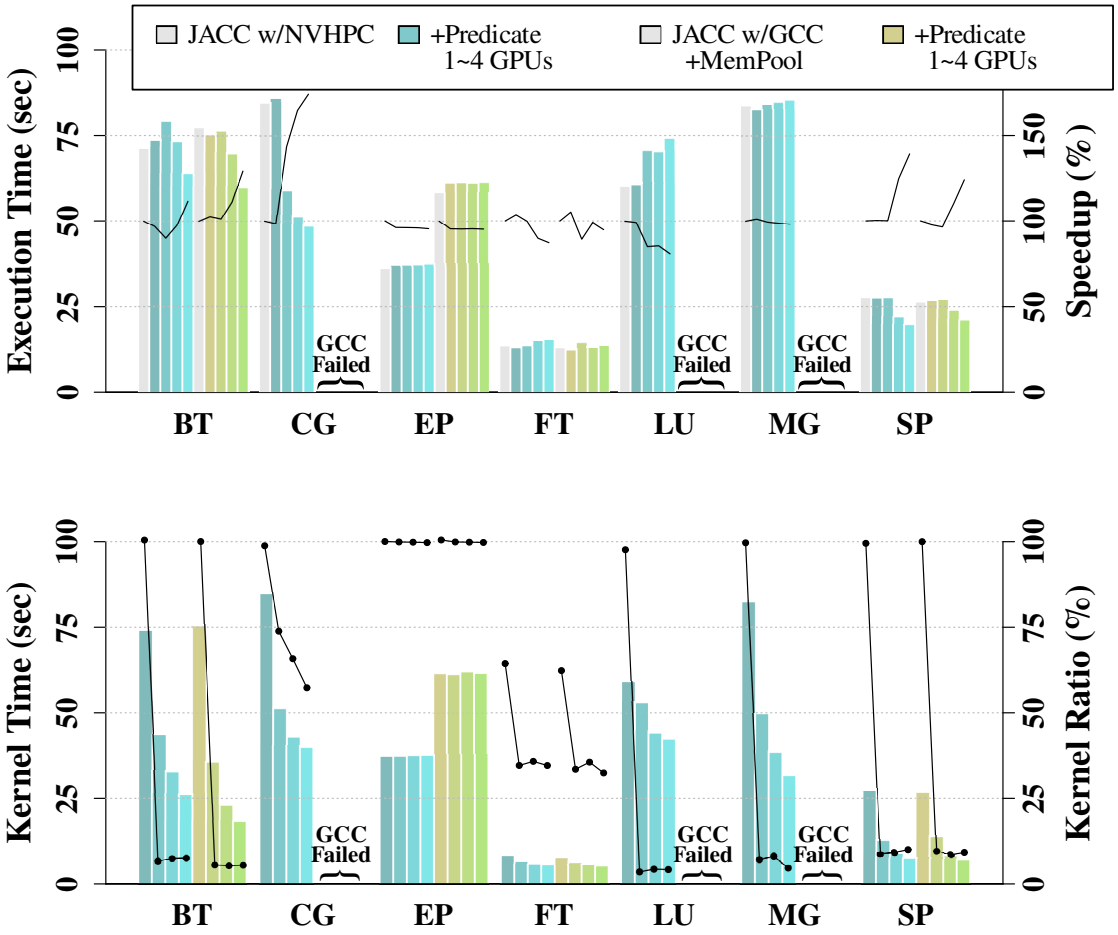


Figure 3.8: Performance scaling of predicate-based filtering using NPB with NVHPC/GCC. The top figure provides the execution times with bars and the speedups with lines compared to the plain JACC code. The bottom figure shows the kernel time when we enable multi-GPU execution with no adaptive algorithm; the kernel ratio to the total time (the total kernel execution time divided by the application execution time) is given by the line.

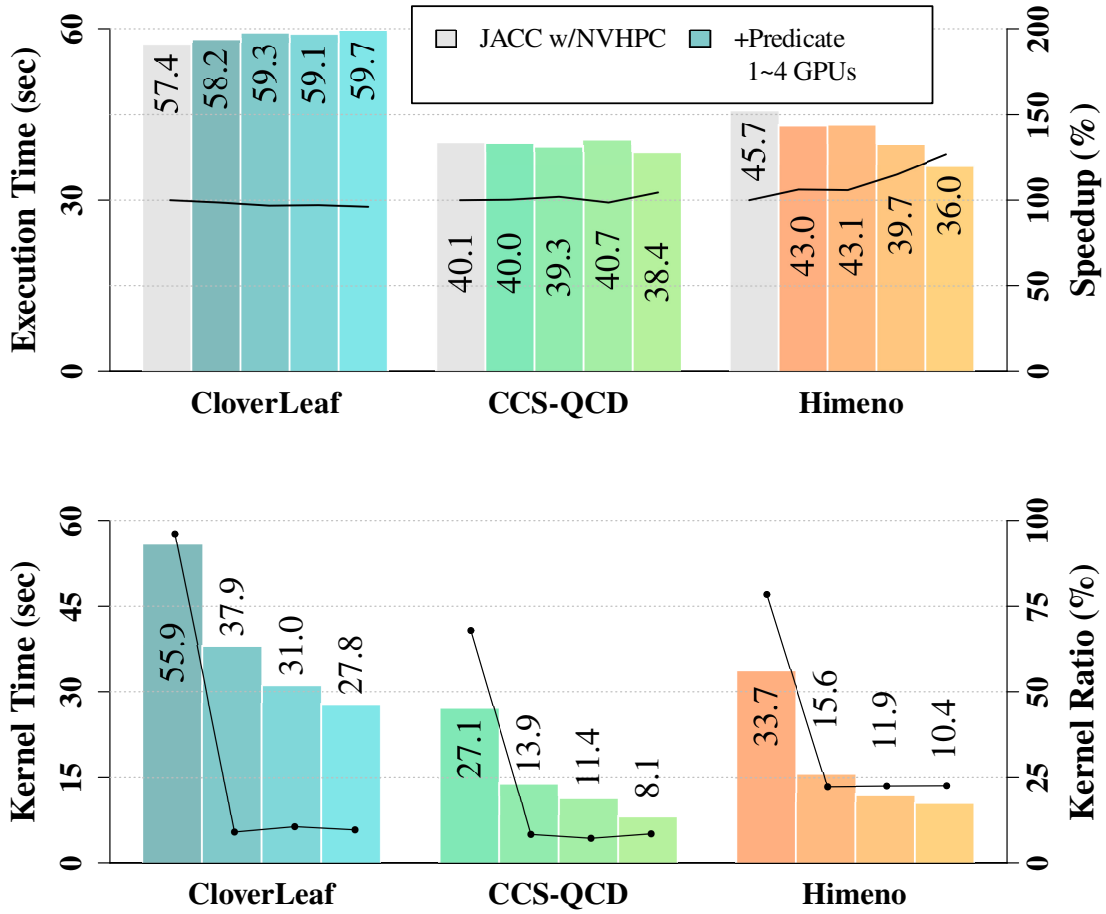


Figure 3.9: Performance scaling of predicate-based filtering using the Fortran mini-apps with NVHPC. The top figure provides the execution times with bars and the speedups with lines compared to the plain JACC code. The bottom figure shows the kernel time when we enable multi-GPU execution with no adaptive algorithm; the kernel ratio to the total time (the total kernel execution time divided by the application execution time) is given by the line.

Table 3.2: Performance details with NVHPC in the use of four GPUs. The result of duplicated execution on all the GPUs is used for the Kernel Dup column only; Other columns use the results of adaptive execution. The Kernel Adapted column shows the kernel execution time for multi-GPU adapted kernels with the average. The **bold** values indicate performance improvements from the duplicated execution.

Name	Num Kernels	Kernel Dup [ms]	Num Adapted	Comm + Kernel [ms]	Kernel Total (Average) [ms]	Kernel Adapted (Average) [ms]	Comm (Average) [ms]	Average WriteSize	GPU-to-GPU Bandwidth
BT	46	88,715	3	**63,856**	46,639 (0.44)	14,940 (24.75)	17,217 (28.52)	684.10 MB	23.99 GB/s
CG	16	75,178	7	**44,137**	38,720 (0.08)	34,365 (0.12)	5,417 (0.02)	0.10 MB	5.40 GB/s
EP	4	37,485	3	37,787	37,710 (24.55)	37,710 (24.55)	77 (0.05)	0.03 MB	0.54 GB/s
FT	12	8,472	4	8,757	6,096 (47.29)	4,983 (62.37)	2,661 (33.31)	806.92 MB	24.23 GB/s
LU	59	76,325	7	**71,614**	64,342 (0.09)	3,886 (0.03)	7,272 (0.06)	0.39 MB	6.28 GB/s
MG	16	83,586	3	83,654	83,654 (0.47)	5,604 (0.35)	0 (0.00)	0.00 MB	0.00 GB/s
SP	65	27,809	3	**19,609**	16,648 (0.64)	1,969 (1.64)	2,961 (2.47)	58.24 MB	23.60 GB/s
CloverLeaf	114	55,017	3	55,272	54,079 (0.22)	10,980 (4.27)	1,193 (0.46)	5.46 MB	11.76 GB/s
CCS-QCD	27	26,517	11	**24,641**	13,031 (0.97)	5,811 (1.91)	11,610 (3.82)	91.67 MB	24.02 GB/s
Himeno	2	33,726	1	**23,149**	11,894 (5.93)	8,318 (8.30)	11,255 (11.23)	271.47 MB	24.18 GB/s

execution and GPU-to-GPU transfers are concerned, six benchmarks (BT/CG/LU/SP/CCS-QCD/Himeno) improve the execution time by 23.9% on average as shown in Table 3.2. Other benchmarks still remain unchanged with some slight degradation up to 3.36% while having several kernels enabled for multi-GPU execution. The noticeable slowdowns we observe in the total execution time of LU/MG are caused due to other factors necessary for multi-GPUs such as memory allocation and synchronization. As an opposite fashion to `+Var Opt`, the predicate-appended code mostly holds the performance of the plain JACC code with single-GPU use.

Profiling the kernel speedups with no adaptive execution showed that predicate-based filtering parallelizes many kernels. Using the memory-intensive benchmarks BT/SP, NVHPC achieves 2.83x and 3.59x improvements on four GPUs and GCC does 4.13x and 3.85x, respectively. For LU, however, the shorter than 1ms running time of each kernel execution limits acceleration to 1.40x, involving overheads for duplicating program structures on all the GPUs. EP does not have any improvement due to its compute-bound nature. Comparing adaptive and non-adaptive execution, CG has almost the same improvement, whereas other benchmarks are prevented from full parallelization due to the high communication-kernel ratios. For example, in Table 3.2, around 20% of the execution of CloverLeaf is distributed over multi-GPUs, but those kernels are not well enhanced, while the remaining execution is duplicated because of the excessive communication latencies, hence, resulting in no speedup.

3.5.3.2 *Data-Size Scaling*

Fig. 3.10 shows the performance scaling with Himeno using different program sizes. Since we equally split array ranges for each GPU, the transfer size per GPU-to-GPU connection becomes smaller and the proportion of communication decreases when the number of GPUs is increased. From two to four-GPU use, we see different scaling of total GPU-to-GPU transfers: 1.70x speedup with size M, 1.95x with L and 1.99x with XL. In regard to kernel performance scaling from single to four GPUs, we achieve 1.53x, 2.19x and 3.64x improvements for size M, L and XL, respectively. For size M, multi-GPU execution suffers the overheads of both kernel and communication. Better scaling can be obtained with longer kernel execution and larger transfers as in the case of BT, which is successfully parallelized with communications of a six-dimensional array decomposed per GPU in 75 segments of 8MB size, having original kernel execution longer than 10ms.

Our technique further reduces the GPU-to-GPU communication latency as more GPUs are used. As future architectures move to having many accelerators with all-to-all interconnects, applications could benefit further from predicate-based filtering.

3.5.3.3 *Comparison*

Related work MACC [21] successfully parallelizes only two of those applications over multi-GPUs based on code-level access-range analysis: CG and Himeno, whose performance bottlenecks have non-overlapping linear writes for each loop iteration. Other multi-GPU work based on memory coherence mechanisms [47, 58, 59] is also unable to support the remaining benchmarks without user effort.

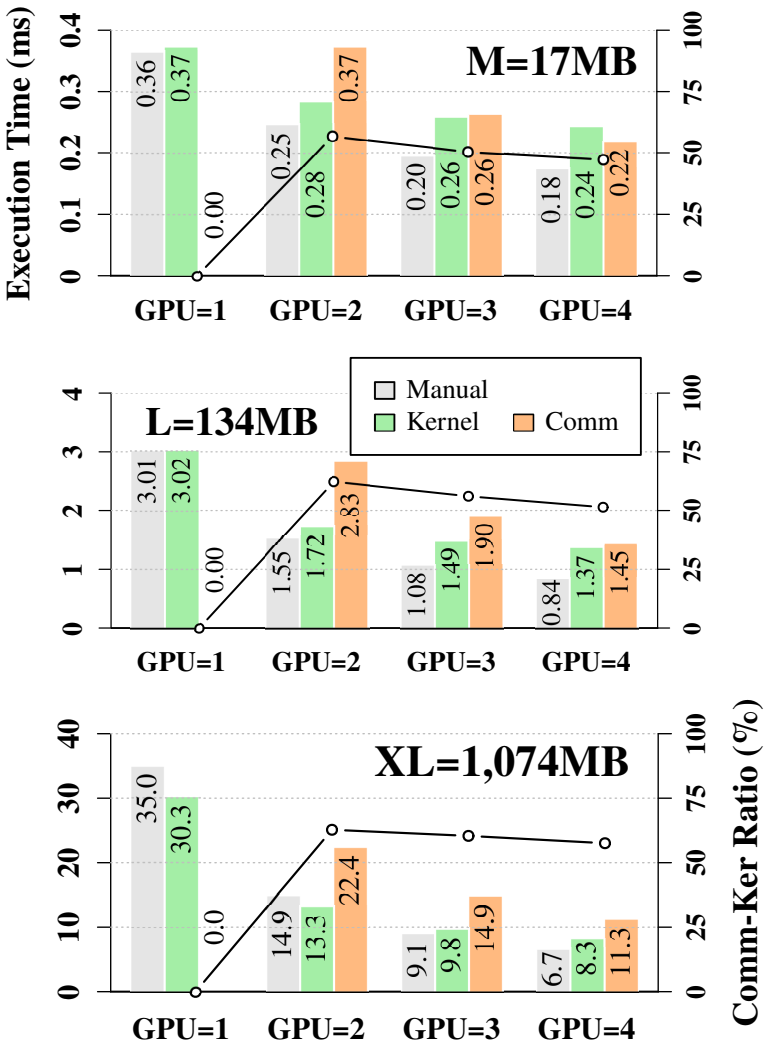


Figure 3.10: Scaling with the number of GPUs for the stencil kernel in the Himeno benchmark. The grey bar shows the average execution time of a manually-tuned loop-splitting version which includes the communication time and the kernel time. The kernel and communication latencies by our technique are shown by the green and orange bars. The line plots the communication-to-kernel latency ratio. Since the kernel requires only 3D halo accesses, the optimized transfer finishes within 30µs, 50µs, and 110µs for the size M, L, and XL, respectively, regardless of the number of GPUs used. The displayed data size equals the total grid proportions that all the GPUs update.

We compare our technique to loop splitting. Fig. 3.10 includes the scaling of the manual code which uses the same algorithm as MACC. We notice that the loop-splitting code has better scaling for kernel execution: from single to four GPUs, it achieves 2.08x, 3.57x and 5.26x improvements for size M, L and XL, respectively. Moreover, the optimized communication for the stencil application significantly reduces the latencies. Those domain-specific approaches can be automated as long as compiler analysis allows; thus, we consider integrating them into our work for future refinement.

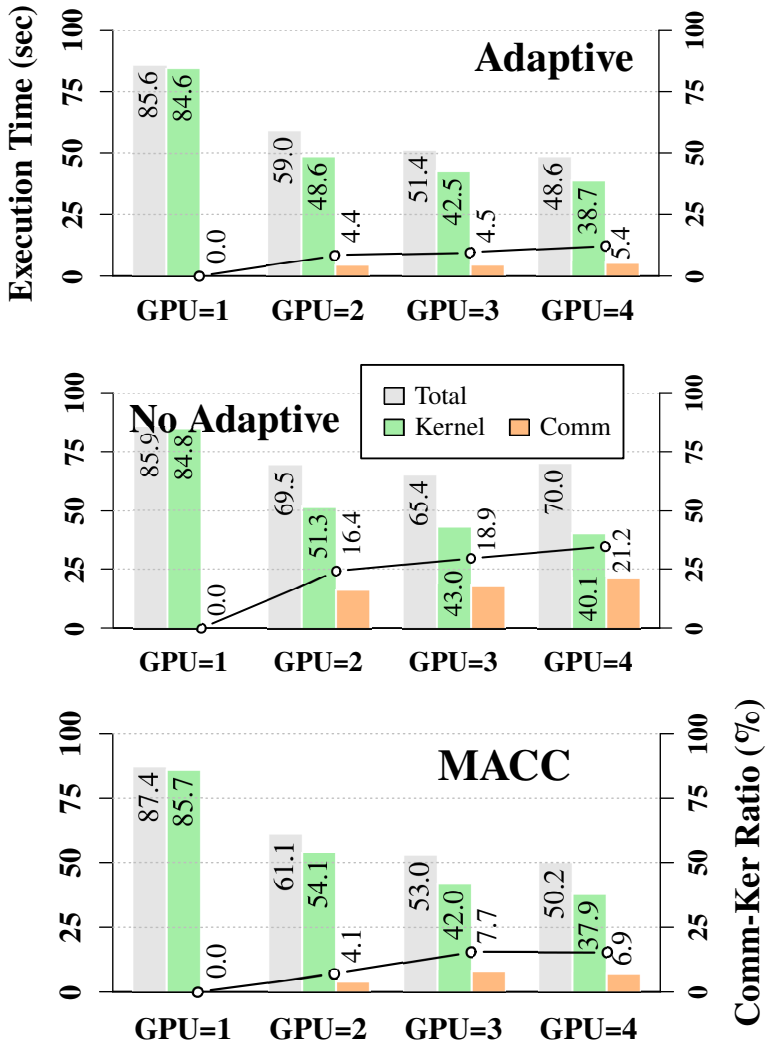


Figure 3.11: Comparison between predicate-based filtering with/without the adaptive algorithm and MACC using NPB-CG. The total execution, kernel execution and GPU-to-GPU transfer times are shown by the grey, green and orange bars, respectively. The latency ratio of communication to kernel is depicted by the line.

Fig. 3.11 shows the comparison between our technique and MACC using CG. MACC automates loop splitting of all the kernels in CG but employs no adaptive algorithm. From two to four-GPU utilization, our technique achieves better efficiency up to 3.50% by disabling multi-GPU execution for lower-latency kernels. Besides that, the adaptive execution has smaller kernel latencies than non-adaptive execution due to the same reason, with 44.09% better total efficiency. We note that predicate-based filtering successfully parallelizes a sparse-matrix vector multiplication (SpMV), where the memory accesses are data dependent as seen in Fig. 3.4, by sending out updated segments to each other GPUs.

Several studies conduct optimization upon the source code of directive-based programming models. In [43], Tian et al. perform scalar replacement on OpenACC code, that substitutes redundant array accesses with scalar references until the compiler reports that all available registers are utilized or all reused references are replaced. Barua et al. [17] optimally unroll OpenACC loops while estimating memory throughputs based on ILP. OptACC [20] finds a better OpenACC parallelism with either grid or direct search. Hoshino et al. [42] propose OpenACC directives for array transformation. JACC eases the implementation of those extensional work in a portable fashion to the user's environment while utilizing dynamic information.

There is some work addressing automated multi-GPU utilization with OpenACC. MACC [21] provides dynamic access-range analysis to distribute execution with GPU-to-GPU communication. Although MACC achieves better performance than a UM system, its analysis is only applicable to affine loops. Komada et al.'s compiler [47] keeps data coherence by tracking array writes in a similar way to UM but incurring additional array writes for it and performing data transfers after each kernel execution. Both previous works divide loop execution equally for each GPU and principally do not allow any intersection of array updates among devices, thus, those cannot support many applications that our work parallelize.

Distributed-memory systems including multi-GPUs are also discussed regardless of programming models. Loop models such as polyhedral have been widely employed to detect data dependency among compute nodes [48, 60, 49, 61]. However, the input is typically restricted to affine or almost-affine loops and those works fix workloads on each node before computing dependency, which involves intricate communication patterns or imposes loop transformations beforehand. Some libraries and frameworks are dedicated to multi-GPU execution through an abstraction which entails GPU-to-GPU communication [62, 63, 64, 65]. Software-level memory managements that maintain data coherency are also capable of accommodating program distribution with little user intervention, but manual efforts are required to adequately partition updated arrays while not overlapping them or otherwise introducing overheads [66, 67, 59, 58]. The Myrias parallel do model had a computational mechanism similar to our algorithm for scheduling tasks [68]. Our predicate-based filtering provides a new way to parallelize many kernels based on source-code level transformation and dynamic information.

Dynamic compilation brings additional opportunities for performance improvement to the runtime system. NVIDIA's jitifiy [69] is a library that simplifies the use of CUDA Runtime Compilation (NVRTC). KernelGen [70] is a Fortran/C compiler that automates GPU code generation with polyhedral loop analysis of LLVM IR. Those works present dynamic features such as runtime alias analysis and parameter tuning alongside kernel specialization. On the other hand, JACC wraps OpenACC compilers and holds C/Fortran code for optimization.

A few projects aim to assist code generation with directive-based programming. Juggler [22] compiles OpenMP task code into a unified GPU program in order to alleviate the latency of global synchronizations. It requires profiling execution to inspect dynamic information and restricts data transfers to be outside task regions. In Juggler, worker

thread-blocks retrieve tasks from queues while managing the execution with a dependence matrix. Andión et al. [71] perform program analysis to coalesce data accesses while maximizing register and shared-memory use of directives. DawnCC [72] automates annotating OpenMP/OpenACC directives based on static analysis of LLVM IR while coalescing data movements. CLACC [73] is an OpenACC compiler that converts input into OpenMP. CCAMP [41] is an interoperable framework for OpenACC and OpenMP, equipped with device-specific optimization. Our dynamic approach complements other work to maximize their scope.

4

LOW-LEVEL CODE OPTIMIZATION

Rather than peephole optimizations attempted in the previous chapter having performance fluctuate, we need different computational mechanisms or orders to achieve faster execution. The following two chapters are devoted to fundamental changes in directive-based programs. Chapter 4 introduces a new method to enable optimizations at the backend of the code generator. After the overview of our tool, the detailed algorithms of symbolic emulation and code generation are given. We demonstrate the use case of a GPU-specific mechanism, shuffle, by experiments on several generations of GPUs.

4.1 PTXASW

To expand the backend of GPU code generation, we introduce a tool named PTXASW that can substitute the original PTX assembler, which accepts input code from arbitrary sources. We do not rely on specific information of any certain language or any certain generation of GPU architecture.

Fig. 4.1 provides a high-level overview of PTXASW's execution flow. PTXASW primarily aims at shuffle synthesis on PTX code. The input is produced by user-level code compilers, while directive-based programming models (OpenACC/OpenMP) do not expose control over warp-level operations, and CUDA prevents code extension due to its code complexities. Once PTXASW inserts shuffles, the resultant code is assembled to GPU binary by the original PTX assembler. We note that, as seen in Section 2.5, shuffle operations are the mechanism to communicate across threads. PTXASW allows user programs to perform shuffles automatically to obtain the results of memory access found in neighboring threads.

PTXASW emulates the PTX execution based on the input. Since runtime information is not provided, we employ symbolic evaluation for each operation. First, ❶ register declarations are processed to be mapped in a symbolic register environment (described in Section 4.2.1). Second, ❷A for each statement of PTX instructions, a corresponding operation is performed to update registers (Section 4.2.1). While continuing the execution, ❷B PTXASW gathers branch conditions for avoiding unrealizable paths (Section 4.2.2) and creates memory traces (Section 4.2.3). When the entire emulation is finished, ❸ we discover shuffle opportunities from memory traces (Section 4.3.1). Finally, ❹ we insert shuffle operations to the input code (Section 4.3.2); then, the generated code is consumed by the original PTX assembler.

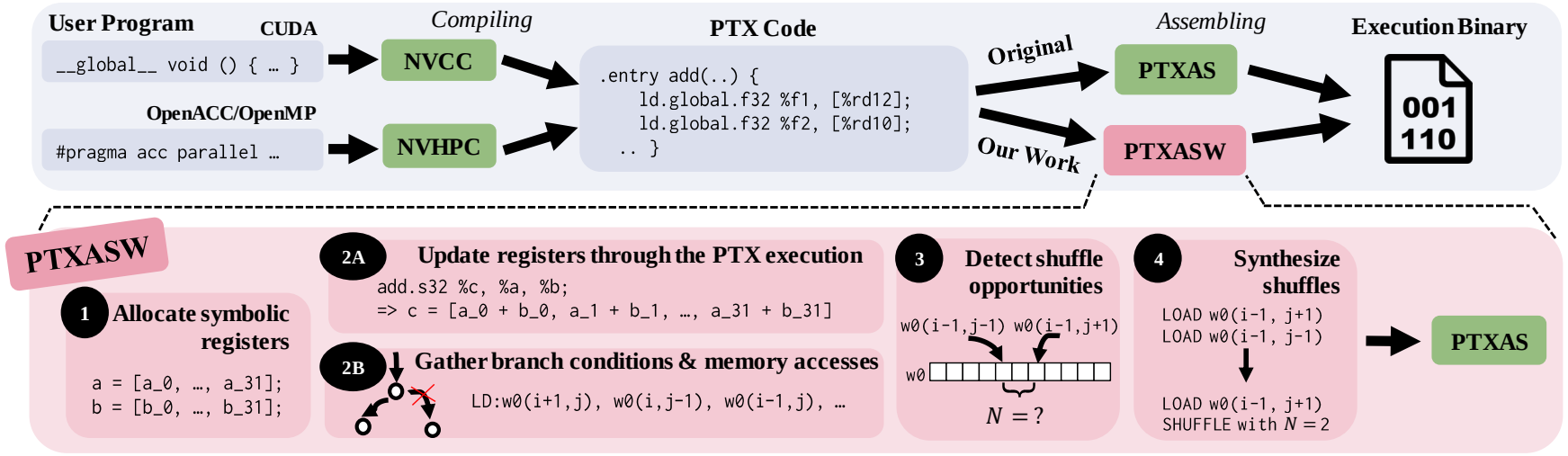


Figure 4.1: Overview of PTXASW. NVCC is the compiler of CUDA, which compiles CUDA kernels such as Listing 2.1 into PTX. NVHPC compiles OpenACC/OpenMP code such as Listing 2.2 generating PTX code. An example of PTX code is provided in Listing 2.4. PTXAS is the PTX assembler and outputs binary files containing SASS code that GPUs recognize. PTXASW rewrites PTX code and creates optimized binary files using PTXAS.

Analysis of high-level code has posed questions about its applicability to abstract program structures or other user-level languages. While high-level code analysis may process intact code information, enormous engineering efforts are required just for specific forms within one language [74, 75]. Therefore, virtual machines are utilized for providing a cushion between real architectures and user codes. In particular, analysis and optimization of the virtual-machine code tend to be reusable without the restriction of input types [76, 77, 78].

Our work uses PTX as the virtual machine layer and performs general analysis through code emulation. We introduce symbolic emulation to encapsulate the runtime information in symbol expressions and compute concolic (concrete + symbolic) values for each register. Although a number of previous works have been conducted on symbolic emulation for the purpose of software testing [79], our work (PTXASW) especially aims at code optimization of memory access on GPUs, since it is often regarded as one of the bottlenecks of GPU computing [80]. Those computed values are utilized for code generation as described in Section 4.3.

4.2.1 *Instruction Encoding*

Since the subsequent PTX assembler, while generating SASS code, will eliminate redundant operations and resources, we may abundantly use registers while not causing register pressure by unnecessary data movement outside of the static single assignment form (SSA). First, PTXASW recognizes variable declarations and prepares a symbolic bitvector of the corresponding size for each register. Since arithmetic calculation and bitwise operations are supported on the combination of concrete and symbolic bitvectors, we encode each PTX instruction as the computation over vectors. For example, addition for 16-bit vectors is encoded as in the pseudocode of Listing 4.1.

```
1  a = [a_0, a_1, .., a_15]; //a_N is a 1-bit element
2  b = [b_0, b_1, .., b_15];
3  c = a + b
4    = [a_0 + b_0, a_1 + b_1, .., a_15 + b_15]; // Carries must be considered
```

Listing 4.1: Addition of bitvectors (The addition of carries is omitted)

With the `add` instruction corresponding to the above calculation, we detect the instruction type and source registers (`%a`, `%b`) and compute the result as Listing 4.2.

```
1  add.u16 %c, %a, %b; // dst: %c; src: %a, %b
```

Listing 4.2: Addition instruction

Then, having the binding with the name of the destination register (`%c`), we keep the computed value in the register environment. PTXASW defines each instruction to update the destination registers according to the instruction options and types, and those registers may be fully concrete with the movement or computation from constant values. Also, to support floating-point instructions, we insert the conversion by uninterpreted functions at loading and storing bitvectors to and from floating-point data. Regarding casting operands among integer types and binary types, truncating or extending is performed based on the PTX specification. The computational instructions under predicates

issue conditional values in registers. Since registers are not used before initialization, these always have evaluated values, except for special registers, such as thread IDs and uninterpreted functions of loops and memory loads, which are described in following sections.

4.2.2 *Execution Branching*

Branching is caused by jumping to labels under binary predicates that are computed by preceding instructions. Since inputs and several parameters are unknown at compilation time, unsolvable values of predicates are often observed leading to undetermined execution flows where computation is boundless. Thus, we abstract the repeated instructions in the same execution flow. At the entry point to the iterative code block, we modify each iterator of the block to have uninterpreted functions with unique identities and perform operations only once upon those uninterpreted functions. Since those uninterpreted functions produce incomparable values, we clip the initial values out and add them to registers containing uninterpreted functions at the block entry, for better accuracy in the case of incremental iterators to be found by induction variable recognition [81, 82].

We continue each branching while duplicating the register environment for succeeding flows. All the flows finish at re-entry to iterative blocks or at the end of instructions, completing their own results. The symbolic expressions in predicates used at the prior divergence are recorded as assumptions while updating those predicates, to have constant booleans in the register environment, based on whether it is assumed as true. Conflicting values in assumptions are removed according to an SMT solver (Z3 [83]) when new expressions are added. If the destination of a new branch can be determined providing assumptions to the solver, unrealizable paths are pruned for faster emulation. Also, we skip redundant code-block entry bringing the same register environment as other execution flows by memoization, to force new results at each entry.

4.2.3 *Memory Analysis*

We collect memory loads forwardly through the emulation and express them by uninterpreted functions accepting addresses and returning data of corresponding sizes. The trace of memory loads is intervened by memory stores, and both loads and assumptions are invalidated by stores that possibly overwrite them, using the same mechanism for conflicting assumptions mentioned in Section 4.2.2.

Listing 4.3 shows a Jacobian kernel implemented in Fortran for GPUs using OpenACC. Its memory trace is obtained as in Listing 4.4 by PTXASW emulating the PTX code generated by NVHPC compiler 22.3. The address of each load is symbolically calculated as register values, thus containing uninterpreted functions and special registers. In the case of divergence, branched flows maintain such traces while sharing the common parts of the original flow.

4.3 SHUFFLE SYNTHESIS

Mapping programs over thread-level parallelism, while pursuing the performance of modern complex architectures and ensuring correctness, is a far–from–easy task. Most likely,

```
!$acc kernels loop independent gang(65535) present(w0(1:nx,1:ny), w1(1:nx,1:ny))
do j = 2, ny-1
  !$acc loop independent vector(512)
  do i = 2, nx-1
    w1(i,j) = c0*w0(i,j) +&
              c1*(w0(i-1,j)+w0(i,j-1)+w0(i+1,j)+w0(i,j+1)) +&
              c2*(w0(i-1,j-1)+w0(i-1,j+1)+w0(i+1,j-1)+w0(i+1,j+1))
  enddo
enddo
```

Listing 4.3: Jacobi kernel in Fortran and OpenACC

```
LD: 0xc + (load(param2) + ((((0x1 + %ctaid.x) * load(param6) // w0(i-1, j+1)
                + ((%tid.x + %ctaid.y << 0x9) + (- load(param5)))) + loop(0, 14)) + loop(0, 53)) << 0x2)
LD: 0xc + (load(param2) + (((load(param6) * (0x3 + %ctaid.x) // w0(i+1, j+1)
                + ((%tid.x + %ctaid.y << 0x9) + (- load(param5)))) + loop(0, 13)) + loop(0, 52)) << 0x2)
LD: 0x4 + (load(param2) + ((((0x1 + %ctaid.x) * load(param6) // w0(i-1, j-1)
                + ((%tid.x + %ctaid.y << 0x9) + (- load(param5)))) + loop(0, 14)) + loop(0, 53)) << 0x2)
/* LD: w0(i+1, j-1), w0(i  , j+1), w0(i+1, j  ), w0(i  , j-1), w0(i-1, j  ), w0(i  , j  ) */
ST: 0x8 + (load(param3) + (((%tid.x + %ctaid.y << 0x9)       // w1(i  , j  )
           + loop(0, 57)) + ((- load(param5)) + load(param6) * ((0x2 + %ctaid.x) + loop(0, 21)))) << 0x2)
```

Listing 4.4: Global-memory trace of Jacobi kernel through the symbolic emulation in order. Sign extensions are omitted. Numerical numbers, shown in hexadecimal, are originally in bitvectors. **load**/**loop** are uninterpreted functions for parameter loads having addresses and loop iterators having unique identities, respectively.

existing GPU programs are already optimized in terms of resource use and scheduling, which does not smoothly allow for further optimization, especially at the low-level code. The shuffle operation performs at its best when the communication is fully utilized [84], but such cases are not common in compiler-optimized code or even manually-tuned code in HPC. The big trouble is corner cases. Not only halo, but fractional threads emerged from rounding up dynamic input sizes, demand exceptional cases to be operated on GPUs. While the generality and applicability of GPU shuffle instructions for all types of applications or computational patterns are yet unknown, the level of difficulty in manually applying shuffle instructions in different cases adds further hardness to the already complex task of understanding the true nature of the performance of shuffle operations.

Hence, we implement automatic shuffle synthesis through PTXASW to drive the lower-latency operations seen in Section 2.5, while supporting corner cases and covering global-memory loads with warp-level communication. PTXASW is accordingly extended to seek shuffle candidates among loads, and embed shuffle instructions into code while alleviating register pressure.

4.3.1 *Detection*

Warps are comprised of neighboring threads. We do not consider adjacent threads in non-leading dimensions, since those tend to generate non-sequential access patterns. Upon finding a global-memory load, PTXAS compares its load address to those of previous loads found through the same execution flow and not invalidated by any store. If for all threads in a warp the load is overlapped with existing loads, those instructions are recorded as possible shuffle sources. To utilize a load with an address represented as $A(\texttt{\%tid.x})$ for another having the address $B(\texttt{\%tid.x})$, there must exist an integer N such that $A(\texttt{\%tid.x} + N) = B(\texttt{\%tid.x})$ and $-31 \leqslant N \leqslant 31$ to be within the same warp, which consists of 32 threads. For example, when $N = 0$, the load can be fully utilized in the same thread. When $N = 1$, we can adapt the `shfl.sync.down` instruction to convey existing register values to next threads while issuing the original load for the edge case ($\texttt{\%warp\_id} = 31$). In the case of the memory trace in Listing 4.4, the load accesses of `w0(i-1, j+1)` and `w0(i-1, j-1)` are uniformly aligned with the close addresses to each other, so we can search the variable N, which satisfies the above condition, by supplying N along with those addresses to the solver and find $N = -2$.

We make sure that each shuffle candidate has the same N as a shuffle delta in all the execution flows. This delta must be constant regardless of runtime parameters. Since the steps of loop iterators in PTX code could be any size (e.g. NVHPC Compiler uses the thread-block size), shuffles are detected only in straight-line flows, whereas live variable analysis is employed to exclude the case in which source values possibly reflect a different iteration from the destination. For faster analysis, we construct control-flow graphs before shuffle detection, while pruning unrelated instructions to memory operations and branches, and at the use of the SMT solver, uninterpreted functions are converted to unique variables.

4.3.2 *Code Generation*

Warp divergence may be caused by various reasons, including the dynamic nature of the program execution, which is inconvenient to optimization, where the uniformity of threads matters for collaboration. Not only inactive threads, but an insufficient number of threads to constitute complete warps, raises corner cases in which original computation should be retained. Our shuffle synthesis handles both situations by adding dynamic checkers for uniformity.

Listing 4.5 presents an example of the synthesis by PTXASW. Once all the emulation is finished, the results are collected and filtered to satisfy all the above-mentioned conditions. Then, PTXASW selects the possible shuffle for each load with the smallest shuffle delta (N) and allows only the least corner cases. At the code generation, each source load instruction is extended to be accompanied by the `mov` instruction to prepare the source register (`%source`). The destination load is covered with the shuffle operation and a corner-case checker. First, we check if the thread has no source from the same warp (`%out_of_range`). Second, the incompleteness of the warp (`%incomplete`) is confirmed with a warp-level querying instruction. In any case, the shuffle operation is performed at the position of the original load, shifting the value of the source register with the distance of the extracted shuffle delta. Finally, only the threads participating in an incomplete warp or assuming no source lane execute the original load under the predicate (`%pred`). When $N < 0$, the `shfl` instruction takes the `.up` option and when $N > 0$, the `.down` option is selected. If $N = 0$, just the `mov` instruction is inserted instead of all the synthesized code. In actual code, the calculation of `%warp_id` is shared among shuffles and set at the beginning of the execution to reduce the computational latency.

To preserve the original program characteristics, such as the register use, uniformity, and ILP, following ways of generation are avoided. We can produce the correct results even if `shfl` is predicated by `%incomplete`, but it often imperils the basic efficiency with an additional branch, which limits ILP. On the other hand, our code introduces only one predicate to each shuffle and does not leave any new branch in the resultant SASS code. Also, we do not use a select instruction for merging the results between shuffles

```
1  ld.global.nc.f32 %f4, [%rd31+12];// w0(i-1, j+1)
2  /* ... */
3  ld.global.nc.f32 %f7, [%rd31+4]; // w0(i-1, j-1)
4  ------------------------------------------------
5  ld.global.nc.f32 %f4, [%rd31+12];                PTXASW
6  mov.f32 %source, %f4; /* ... */
7  mov.u32 %wid, %tid.x; rem.u32 %wid, %wid, 32;
8  activemask.b32 %m; setp.ne.s32 %incomplete, %m, -1;
9  setp.lt.u32 %out_of_range, %wid, 2;
10 or.pred %pred, %incomplete, %out_of_range;
11 shfl.sync.up.b32 %f7, %source, 2, 0, %mask;
12 @%pred ld.global.nc.f32 %f7, [%rd31+4];
```

Listing 4.5: Shuffle synthesis on Jacobi kernel. Upper is original and lower is synthesized code; variable declarations are omitted and the naming is simplified.

and corner cases, because it would aggravate register pressure. The output predicate by shuffle poses execution dependency and provides the invalid status of inactive threads; thus, it is ignored. Moreover, we only create shuffles from direct global-memory loads and do not implement shuffles over shuffled elements for better ILP.

4.4 EXPERIMENTAL METHODOLOGY

We build PTXASW using Rosette [85], a symbolic-evaluation system upon the Racket language. PTXASW is equipped with a PTX parser and runs the emulation of the parsed code while expressing runtime parameters as symbolic bitvectors provided by Rosette. Our shuffle synthesis is caused at code generation, which prints the assembler-readable code. We evaluate our shuffle mechanism with the NVHPC compiler [7] by hooking the assembler invocation and overwriting the PTX code before it is assembled. The NVHPC compiler accepts the directive-based programming models OpenACC and OpenMP to generate GPU code, which have no control over warp-level instructions. The emulation is also tested for GCC with OpenACC/OpenMP code and LLVM with OpenMP code, but these use a master-worker model to distribute computation across thread-blocks [86] and do not directly refer to the thread ID in each thread, so mainly ineffective results are obtained. Our synthesis is not limited to global-memory loads and works on shared memory (such as Halide [87]), but the performance is not improved due to the similar latency of shared-memory loads and shuffles. The NVHPC compiler utilizes the same style to translate both OpenACC and OpenMP codes written in C/C++/Fortran to PTX, hence supporting any combinations.

For the evaluation, we use the KernelGen benchmark suite for OpenACC [88], shown in Table 4.1. Each benchmark applies the operator indicated in the benchmark name, to single or multiple arrays and updates different arrays. The benchmarks **gameoflife**, **gaussblur**, **jacobi**, **matmul**, **matvec** and **whispering** are two-dimensional, whereas others are three-dimensional, both having a parallel loop for each dimension, in which other loops might exist inside—except **matvec**, which features only one parallel loop. The thread-level parallelism is assigned to the innermost parallel loop and the thread-block level parallelism to the outermost. We show the total time of running the shuffle-synthesized kernel ten times on Kepler (NVIDIA Tesla K40c with Intel i7-5930K CPU), Maxwell (TITAN X with Intel i7-5930K), Pascal (Tesla P100 PCIE with Intel Xeon E5-2640 v3), and Volta (Tesla V100 SXM2 with IBM POWER9 8335-GTH). We use NVHPC compiler 22.3 with CUDA 11.6 at compilation, but due to environmental restrictions, run the programs using CUDA driver 11.4/11.4/10.0/10.2 for Kepler/Maxwell/Pascal/Volta, respectively. The compiler options in NVHPC are `"-O3 -acc -ta=nvidia:cc(35|50|60|70),cuda11.6,loadcache:L1"`. To fully utilize computation, 2D benchmarks select 32768x32768 as their dynamic problem sizes and 3D compute 512x1024x1024 grids, except **uxx1**, which leverages 512x512x1024 datasets and **whispering**, where more buffers are allocated, computing over 8192x16384 data elements. To assess a performance breakdown, we prepare two other versions of PTXASW: NO LOAD and NO CORNER. The former eliminates loads that are covered by shuffles, whereas the latter only executes shuffles instead of original loads, without the support of corner cases.

Table 4.1: The KernelGen benchmark suite. Lang indicates the programming language used (C or Fortran). Shuffle/Load shows the number of shuffles generated among the total number of global-memory loads. Delta is the average shuffle delta. Analysis is the execution time of PTXASW on Intel Core i7-5930K.

Name	Lang	Shuffle/Load	Delta	Analysis
divergence	C	1 / 6	2.00	4.281s
gameoflife	C	6 / 9	1.50	3.470s
gaussblur	C	20 / 25	2.50	7.938s
gradient	C	1 / 6	2.00	4.668s
jacobi	F	6 / 9	1.50	4.119s
lapgsrb	C	12 / 25	1.83	14.296s
laplacian	C	2 / 7	1.50	4.816s
matmul	F	0 / 8	-	13.971s
matvec	C	0 / 7	-	4.929s
sincos	F	0 / 2	-	1m41.424s
tricubic	C	48 / 67	2.00	1m39.476s
tricubic2	C	48 / 67	2.00	1m41.855s
uxx1	C	3 / 17	2.00	7.466s
vecadd	C	0 / 2	-	3.281s
wave13pt	C	4 / 14	2.50	6.967s
whispering	C	6 / 19	0.83	6.288s

To alleviate the overheads of PTXASW, we run both the comparison between load addresses and iterator analysis of loops in parallel, as well as the emulation of each kernel. Also, we set the timeout for one solver execution to one minute. As Table 4.1 shows, most benchmarks cause moderate latency. In **sincos**, the mathematical functions sin and cos leave multiple nested loops in the analyzed code; hence, the execution spends much time on iterator analysis. In **tricubic**, triple-nested loops of size 4x4x4 are unrolled manually, while in **tricubic2** these are sequential loops to be unfolded by the compiler. Therefore, we see 67 loads on both, which also require substantial analysis time. While our current implementation uses the solver to detect inductive variables in a similar way to the shuffle detection, it may be refined by classical methods [81, 82]. Appendix provides the pseudocode of PTXASW.

The shuffle synthesis fails on four benchmarks. In **matmul** and **matvec**, the innermost sequential loop contains loads, but these do not have neighboring accesses along the dimension of the thread ID. The benchmarks **sincos** and **vecadd** do not have several loads sharing the same input array. We note that PTXASW does not attempt to replace data-dependent memory access in order to alleviate runtime overheads. Yet, as seen in Table 4.1, standard applications provide shuffle opportunities for PTXASW with straightforward memory access.

Fig. 4.2 shows the execution time of benchmarks on each GPU with original code and PTXASW-generated code. The performance improvement on Kepler/Maxwell/Pascal/-Volta is confirmed with 7/6/9/4 benchmarks showing up to 16.9%/132.3%/9.1%/14.7% performance improvement, respectively. We see performance degradation with Volta in the case where more than ten shuffles are generated. Other GPUs mostly gain better performance with such cases. With increased shuffle deltas, more corner cases are expected. Volta shows optimal efficiency when $N \leqslant 1.5$, while other GPUs benefit from the case of $N = 2.5$. For example, Maxwell attains the best performance with **gaussblur**

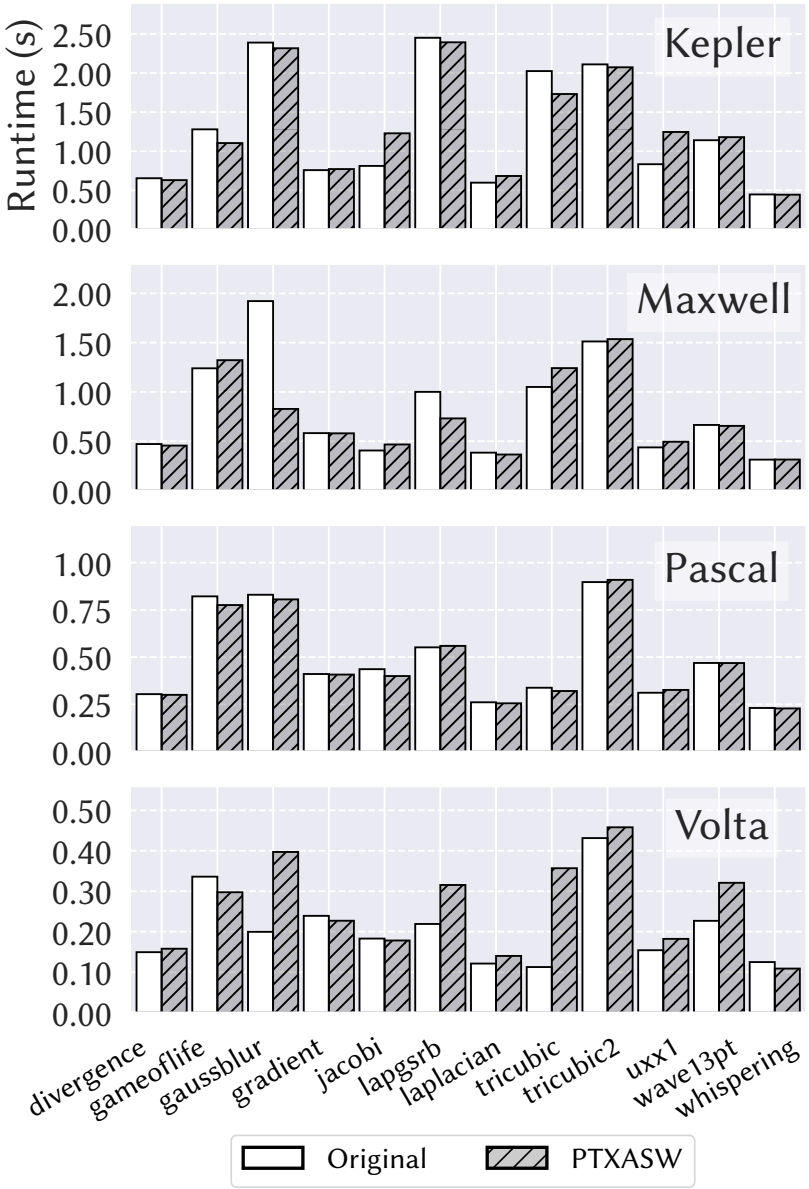


Figure 4.2: Performance comparison between the original code and the version implementing automated shuffle on four GPUs.

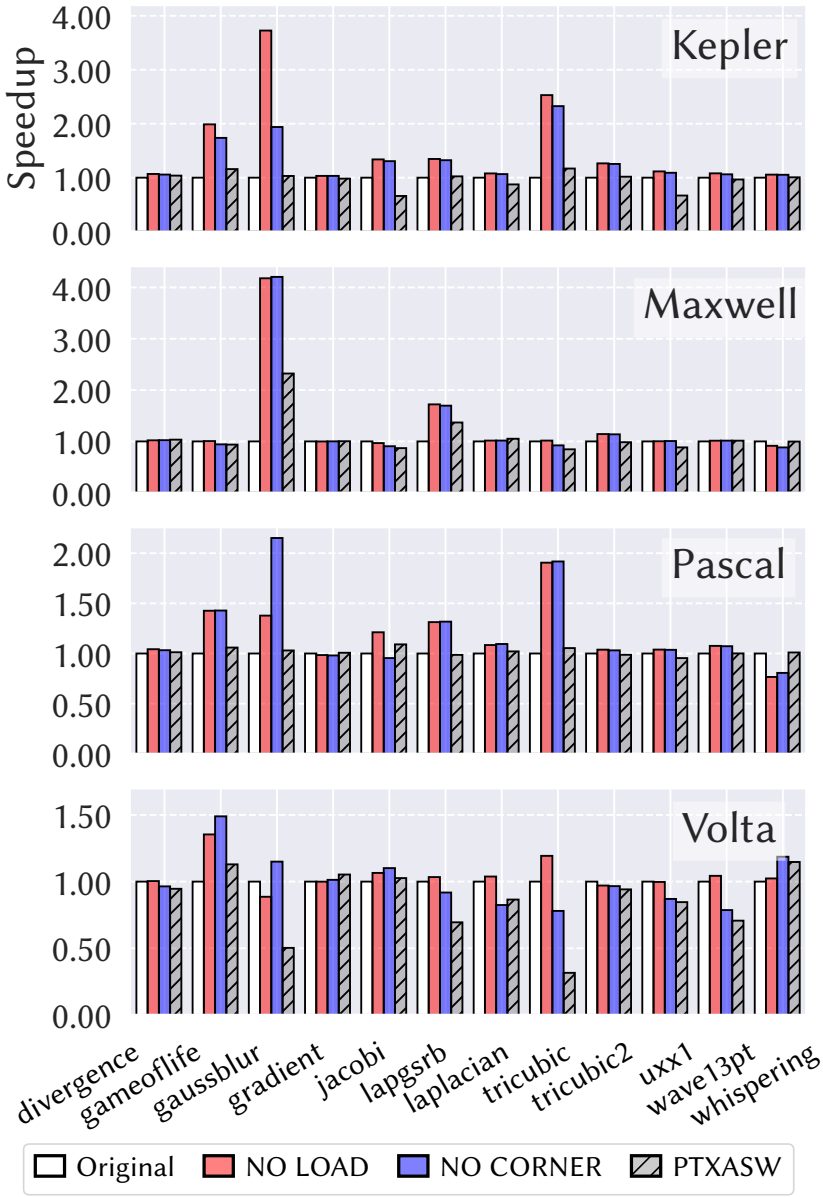


Figure 4.3: Speedup compared to Original. NO LOAD/NO CORNER produce invalid results

(N = 2.5), although Volta's performance drops by half for the same case. The average improvement across all GPU generations is -3.3%/10.9%/1.8%/-15.2% for Kepler/Maxwell/Pascal/Volta, respectively.

The speedup graphs for PTXASW, along with the NO LOAD and NO CORNER versions discussed in Section 4.4, are provided in Fig. 4.3. NO LOAD eliminates the need for memory loads and attains better performance with memory-bound applications such as **gaussblur** (except on Volta). Fig. 4.4 presents the SM occupancy of each benchmark. Since there is no resource change other than the register use from the original execution, the occupancy rate is directly affected by the number of registers. With NO LOAD, the performance of **gaussblur** on Volta and **whispering** on Maxwell/Pascal degenerate. Since the number of active registers is fewer, compared to the original, more thread blocks may

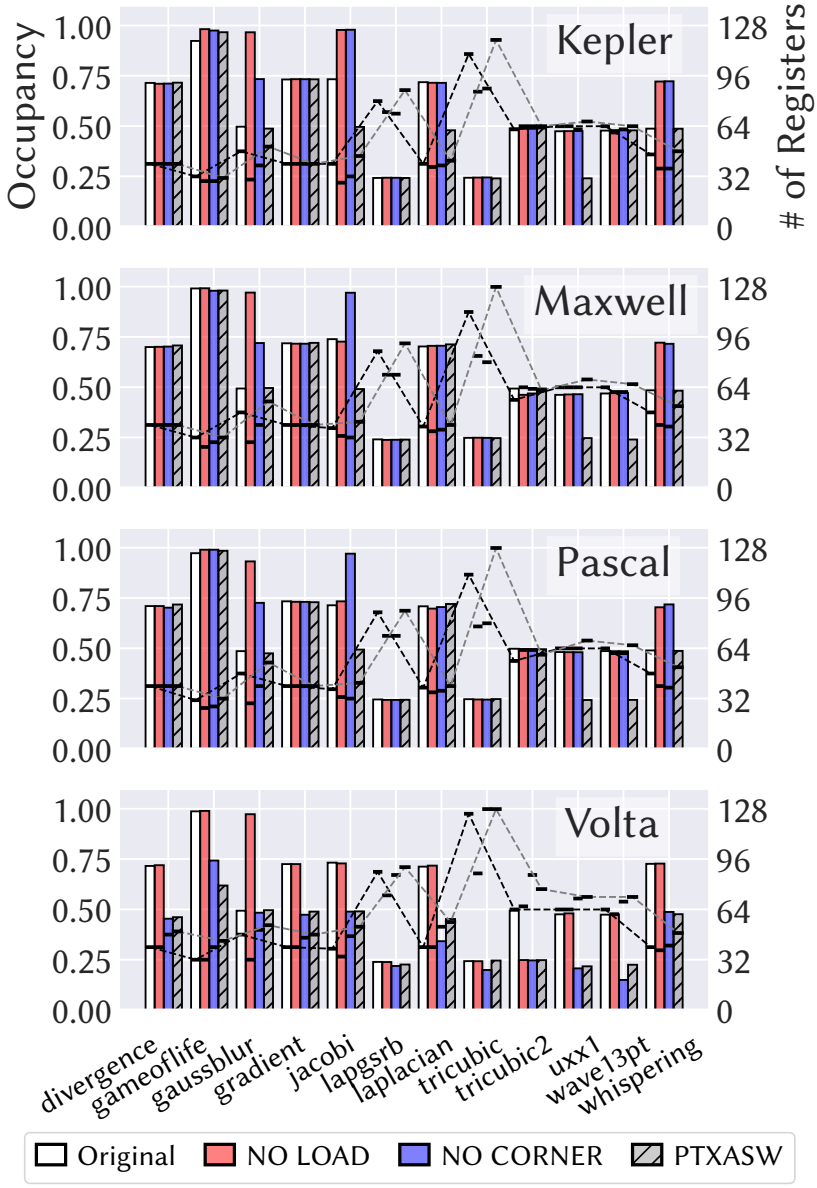


Figure 4.4: Occupancy of SMs. Line plots show the number of registers used per thread

be scheduled on the SMs, causing demand conflicts on the registers and internal units. NO CORNER replaces the original loads and attains competitive performance compared with NO LOAD on Kepler/Maxwell/Pascal. On Volta, the occupancy drop by NO CORNER is explicit, which causes performance improvements over NO LOAD at **gameoflife**. With **gausslur** on Volta, NO CORNER maintains similar occupancy compared to the original, yet achieves 15.0% better efficiency. The benchmark **gaussblur** on Pascal/Volta decreases occupancy from NO LOAD to NO CORNER, which generates 56.2%/29.7% better throughputs. Basically, NO CORNER imposes one shuffle latency at the location of original memory loads, which is observed as performance degradation in **tricubic**. Accordingly, PTXASW increases the execution time of these, compared to NO CORNER, while some benchmarks such as **jacobi** on Pascal and **gradient** on Volta are refined due

to the lower occupancy. On Kepler and Maxwell, which have high computational latency compared to the other two GPUs, the occupancy drop harms the efficiency. The benchmark **jacobi** experiences poor performance with PTXASW, where the number of registers is increased from 40 to 45 on Kepler and from 38 to 42 on Maxwell, compared to the original.

Overall, the performance improvement by PTXASW is found when NO LOAD and NO CORNER have sufficiently better performance compared to the original and when the occupancy typically rises on Kepler/Maxwell and drops on Pascal/Volta. The average number of additional registers with NO LOAD/NO CORNER/PTXASW compared to the original is -6.4/-5.2/2.7 on Kepler, -6.6/-5.9/4.2 on Maxwell, -7.0/-5.9/3.8 on Pascal, and -6.4/6.8/9.2 on Volta.

4.6 ANALYSIS

This section provides the performance detail of our shuffle synthesis on each GPU. Fig. 4.5 shows the ratio of stall reasons sampled by the profiler for all the benchmarks. Those characteristics of computation appear as the results of the program modification (e.g. register use, shuffle delta) and the architecture difference (e.g. computational efficiency, cache latency).

4.6.1 *Kepler*

The Kepler GPU has long stalls on computational operations with each benchmark. The average execution dependency is 24.7% and pipeline busyness is 7.5% with the original. When we look at the memory-bound benchmarks such as **gameoflife**, **gaussblur**, and **tricubic**, NO LOAD significantly reduces memory-related stalls. Especially, **tricubic** has 56.0 percentage points below memory throttles from the original to NO LOAD, yielding 2.53x performance. From NO LOAD to NO CORNER, the execution dependency increases by 4.0 percentage points and the pipeline busyness decreases by 1.6 percentage points on average. The performance degradation at NO CORNER with the memory-bound benchmarks is observed with the latency of the pipelines and the wait for the SM scheduler. PTXASW suffers from memory throttling and additional computation for the corner cases, which limit the improvement up to 16.9%.

The memory throttling and the additional computation bottlenecks suffered by PTXASW may be hidden if the shuffle operations reduce the original computation and communication into just one transfer among threads, functioning as a warp-level cache. Otherwise, there is a need to face a trade-off between the redundancy of operations and the efficiency on the architecture. On Kepler, both heavy computation and memory requests are imposed by the corner case. Therefore, in the general use of shuffles, the uniformity of calculation is crucial and it requires domain-specific knowledge.

4.6.2 *Maxwell*

There are two obvious compute-bound benchmarks: **gameoflife** and **tricubic**. For these, no improvement is perceived with NO LOAD, and there are no particular changes in occu-

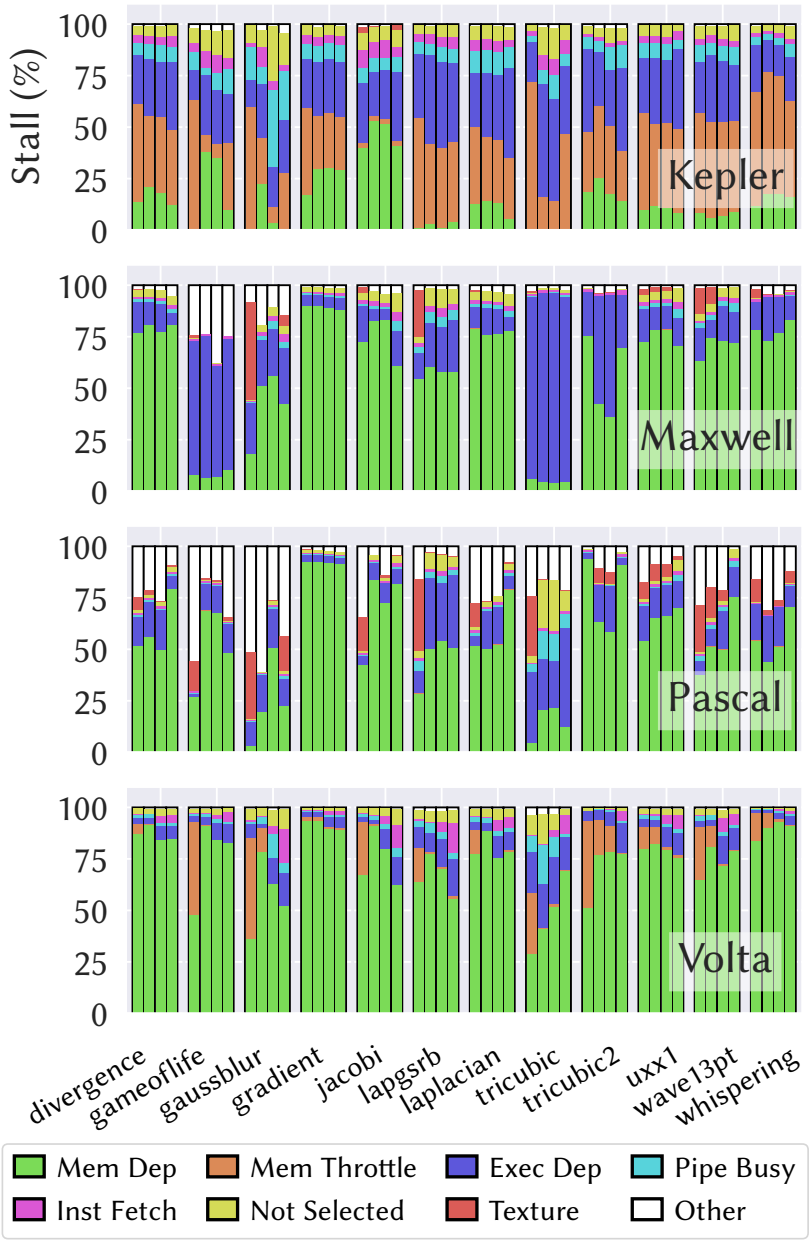


Figure 4.5: Stall breakdown in the order of Original/NO LOAD/NO CORNER/PTXASW from left to right for each benchmark. `Mem Dep`: waiting for the completion of memory access; `Mem Throttle`: no available memory unit; `Exec Dep`: waiting for the results of the preceding computation; `Pipe Busy`: no available computational unit; `Inst Fetch`: latency of fetching operations; `Not Selected`: the warp is not scheduled; `Texture`: utilization of the texture memory; `Other`: reasons unspecified by the profiler.

pancy or stalls throughout the four different versions. In summary, **gameoflife** experiences -0.1%/5.7%/6.2% lower performance and **tricubic** shows -1.6%/7.7%/15.4% lower performance with NO LOAD/NO CORNER/PTXASW, respectively, compared to the original version. In other cases, memory dependency is dominant. However, the merit of

NO LOAD is limited to **gaussblur** and **lapgsrb**, which experience large texture-memory latency of read-only cache loads, successfully replaced with shuffles by PTXASW. The texture stall was reduced from 47.5% to 5.3% in **gaussblur** and from 23.0% to 0.1% in **lapgsrb** from the original to PTXASW, attaining 132.2% and 36.9% higher throughput. Other benchmarks do not feature stalls that allow for clear performance improvement by NO LOAD. As it can be observed in Fig. 4.5, the memory dependency stalls are maintained for most benchmarks, except for **tricubic2**, which shows 32.9 percentage points lower memory dependency and only 14.3% overall improvement with NO LOAD. Those values are mostly absorbed by the corner cases.

On the Maxwell GPU, only the texture stalls are improvable for efficiency in the tested cases. Since we observe a moderate overhead of the corner cases, our synthesis tool may enhance the overall performance. The memory-dependency stalls work as a good indicator of the memory utilization. If, in addition, a high execution dependency would exist, it would provide the warp-level shuffle optimization the opportunity to be beneficial to speed up the computation.

4.6.3 *Pascal*

Even more than in Maxwell, texture stalls are found in most benchmarks and those produce higher throughput with NO LOAD. Especially, **gameoflife** and **tricubic**, the compute-bound kernels on Maxwell, become memory intensive on Pascal and the performance increases by 5.9% and 5.4% with PTXASW. The unspecific latency ("Other") fills many parts of computation on Pascal. Further investigation shows that this mainly consists of the latency from register bank conflicts and the instructions after branching. With the optimization adding a predicate to check the activeness of the warp (`@!incomplete`) before the shuffle and generating a uniform branch, the ratio of this latency improves from 34.4% to 8.6% with PTXASW at **gameoflife**, obtaining 150.8% efficiency compared to the original. However, as mentioned in Section 4.3.2, it decreases the average relative execution time to 0.88x slowdown.

Since the latency of the L1 cache is higher than that of one shuffle operation, the computation may be hidden by data transfers. Once the memory-dependency stall ratio increases due to replacing the texture stalls, Pascal may maintain the efficiency with the corner cases, resulting in speed-up in nine benchmarks. For shuffle instructions to be beneficial, the execution should be less divergent and careful register allocation is recommended to maximize the thread utilization.

4.6.4 *Volta*

On Volta, most benchmarks become memory-bound and memory-intensive applications become sensitive to memory throttles. Nevertheless, the speed-up by NO LOAD is limited to up to 1.35x (**gameoflife**), due to the highly efficient cache mechanism. As argued in Section 4.5, some of the benchmarks attain higher performance with NO CORNER than in the case of NO LOAD for the lower occupancy. Other than that, we observe performance degradation due to increased execution dependency for **lapgsrb** and **tricubic** with NO CORNER. Those further reduce the efficiency with PTXASW while featuring stalls for

instruction fetching. Also, the memory dependency of **tricubic** develops a large latency for memory accesses with PTXASW even though the corner cases experience fewer loads. This leads to unstable speed-ups between 0.315x and 1.15x.

The calculation through shuffles is expected to be effective depending on the utilization of communication, and the nonentity of warp divergence. Especially, as Volta shows minimal latency at each operation, the penalty of non-aligned computation becomes apparent and must be avoided by the algorithm.

4.6.5 *Application Example*

We also apply PTXASW for the compilation of CUDA benchmarks extracted from applications. We select three benchmarks that appeared as complex 3D stencil operations in [89]: **hypterm**, **rhs4th3fort**, and **derivative**, to run on the Pascal GPU. **hypterm** is a routine from a compressible Navier-Stokes mini-app [90]. **rhs4th3fort** and **derivative** are stencils from geodynamics seismic wave SW4 application code [91]. Each thread in the benchmarks accesses 152/179/166 elements over 13/7/10 arrays, respectively. In order to satisfy the requirement of our shuffle generation algorithm, We modify the execution parameters to execute at least 32 threads along the leading thread-block dimension and use the float data type. Since we saw in the prior section the overhead of long-distance shuffles, which generate many corner cases, we limited the shuffle synthesis to be $|N| \leqslant 1$ and found shuffles only with $|N| = 1$.

hypterms contains three kernels that work along different dimensions. In the kernel for the leading dimension, 12 shuffles are generated over 48 loads, producing 0.48% improvement. **rhs4th3fort** and **derivative** feature a single kernel each. **rhs4th3fort** experiences 2.49% higher throughput by PTXASW while placing 44 shuffles among 179 loads. For **derivative**, having 52 shuffles from 166 loads, PTXASW attains 3.79% speed-up compared to the original execution.

4.7 RELATED WORK

Ever since warp-shuffle instructions were introduced during the Kepler generation of GPUs, these have been the subject of various lines of research. Early work described their manual use for specific computational patterns such as reduction operations [92] and matrix transposition [93]. Other research described the use of warp-shuffle instructions in the context of domain-specific optimizations such as employing them as a register cache for stencil operations [94], or to replace memory access for Finite Binary Field applications [94].

Research on the automatic generation of warp-shuffle instructions has been explored. Swizzle Inventor [84] helps programmers implement swizzle optimizations that map a high-level "program sketch" to low-level resources such as shuffle operations. The authors meticulously design the abstraction of shuffles, synthesize actual code roughly based on algorithms found in previous literature, and attain enhanced performance while reducing the amounts of computation. Tangram, a high-level kernel synthesis framework, has also shown the ability to automatically generate warp-level primitives [74]. Unlike PTXASW, both of the above-mentioned efforts leverage domain-specific information to map com-

putational patterns such as stencil, matrix transposition, and reductions to shuffle operations. Thus, Swizzle Inventor and Tangram are not able to benefit from low-level shuffle instructions for general cases that fall outside their scope. In addition, both require the programmer to first write a high-level program that is compatible with their tool.

Other systematic efforts with shuffle operations have been established as a way for optimization. Software systolic arrays [80] improve data locality by assigning global data to each thread. These compute results by propagating and accumulating partial results through warp-level shuffles across overlapped blocks rather than using shared memory. Wang et al. [95] introduce shuffles to OpenMP programs and experience strictly higher throughputs compared to the use of shared memory. While those models may reduce the number of memory accesses, the applicability is restricted to summation and a deep understanding of applications is required for the implementation.

Recent code-generation techniques allow for obtaining optimal SIMD code generation. Cowan *et al.* [96] generate program sketches for execution on ARM processors, by synthesizing additional instructions, as well as input/output registers, to implement the shortest possible SIMD code of reduction. Unlike PTXASW, which uses an SMT solver to find the optimal shuffle deltas, this work runs a comprehensive search of multiple possible code versions; thus, the search space is exponential to the number of instructions. VanHattum et al. [97] attain faster execution on digital signal processors while employing *equality saturation* [23], a modern way of optimization that generates possible code as much as possible from a basic program according to the rules of term rewriting. They derive shuffles along with vector I/O and computation from sequential C code. Their intermediate code contains instructions in one nested expression and the shuffle operation only works for memory loads that appear as arguments of the same vector operation. Therefore, the code rewriting for shuffles assumes a top-down style where outer expressions have to be vectorized first, in order to vectorize inner expressions containing shuffled loads. While their technique may provide a powerful method to the implementation of libraries, irregular patterns such as corner cases usually found in HPC applications are out of scope. Partial control-flow linearization [98] applies SIMD operations to irregular data-parallel loops by masking inactive lanes out while holding non-divergent branching. Vector folding [99] changes the data layout of multi-dimensional stencils for SIMD computation. The ideas used for SIMD may be beneficial to GPUs due to the architectural resemblance between SIMD and SIMT.

Reductions in OpenACC/OpenMP are often mapped to warp-level shuffles for acceleration [100, 101]. The shared-memory clause introduced to OpenACC shows poor performance [102], as we experienced in our tested benchmarks, and thus requires a substantial effort to gain efficiency.

Several symbolic analyzers [103, 104] have been proposed for GPU computing to check memory access, synchronization, data races, correctness, and performance. Analysis and optimization of low-level GPU code are explored for GPU-specific problems such as locality, divergence, and register use [105, 106, 107]. GPU simulators [108, 109] may provide characteristic details of performance in relation to the design of GPU architectures. Unofficial SASS assemblers [38, 110] are provided for multiple GPUs to reveal the depth of GPU programming.

To our knowledge, PTXASW is the first tool that enables automatic generation of shuffle instructions for all cases, without additional domain information, and is immediately compatible with all PTX generating languages and tools, without any further programmer intervention to application codes.

5

SOURCE-CODE OPTIMIZATION

The previous chapter worked on an extension of the GPU code assembler to allow fundamental changes in the execution system of directive-based programs. For more aggressive optimizations, Chapter 5 deals with the frontend. We first provide the overview of our tool and then describe the details of our source-code optimization algorithms, such as equality saturation and bulk load. Using state-of-the-art GPUs, we show significant performance improvements.

5.1 ACC SATURATOR

Compiler techniques have been critical for the last half-century in utilizing the best resources available for program execution on the target architecture [111, 112]. Various features of hardware, including registers, cache systems, pipelines, and parallelization, rely on the compiler's efforts for performance, enabling target-specific optimization under resource constraints. However, compilers tend to miss a holistic view of performance opportunities and require additional discrete efforts for each discrete optimization mechanism [89]. *Equality saturation*, a state-of-the-art technique for compilers [23], defines a set of rewriting rules over the accumulation of equal expressions of target code. The rewriting continues until the expression gets *saturated*, finding no other forms, or hits a time or size limitation, attaining optimal codes based on a cost model that considers the entire computation.

A dedicated graph structure called *e-graph* accepts the accumulation of equal expressions, while sharing redundant code blocks over expressions. The graph structure consists of *e-classes*, groups of equal *e-nodes*. One e-node may point to e-classes as its subexpressions, and the accumulation works by extending or merging e-classes in accordance with rewriting rules, while preserving the relationship between parental e-nodes and child e-classes [113, 114]. To extract optimal solutions, each e-class selects one contained e-node with a minimum cost, while optionally counting common expressions only once and completing Common Subexpression Elimination (CSE).

ACC Saturator is the implementation of our proposal of equality saturation for directive-based code. We provide a convenient command-line tool that wraps normal C-compiler invocation and replaces the original inputs with saturated codes.

Fig. 5.1 provides an overview of our work. ACC Saturator optimizes the sequential parts of parallel loops by packing and unpacking e-graphs for extracted expressions while preserving original code structures. Array references, branching, loops, function calls, and member references are supported under a dataflow abstraction. Once e-graphs have the

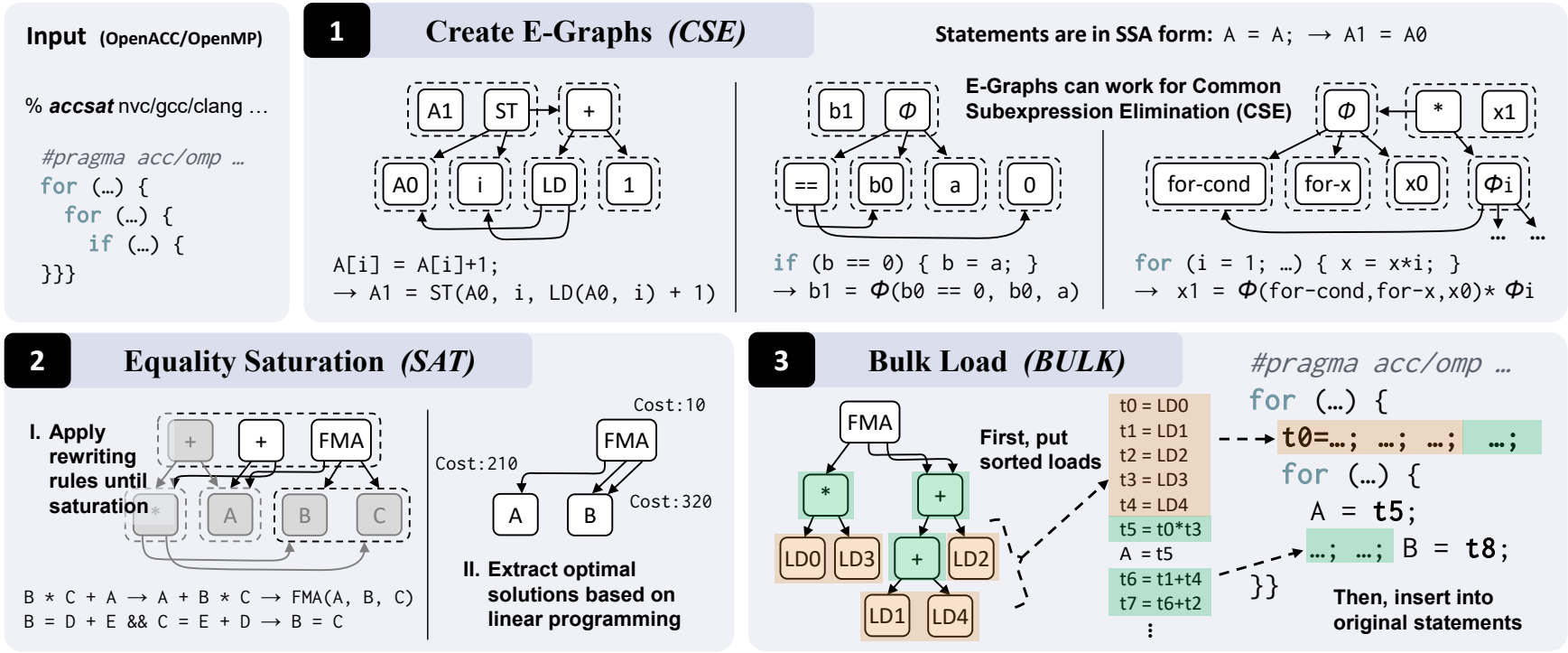


Figure 5.1: Overview of ACC Saturator

inputs, we run equality saturation with an arbitrary set of rewriting rules and succeedingly select optimal solutions featuring minimal total costs according to our model. With new expressions, we update user code while arranging the order of computation and continue to the compiler invocation, since generated code is compatible with NVHPC, GCC, and Clang.

The e-graph operation by ACC Saturator consists of three phases. First, 1 we build a static single-assignment form (SSA) from the input code to create initial e-graphs (Section 5.2). E-graphs accept our SSA representation holding dependencies, conditions, and iterations over C-style operations. Second, 2 we define rewriting rules and a cost model for performance improvement (Section 5.3). Our tool runs equality saturation and extracts optimal solutions under time and size limitations. Last, 3 we generate output code (Section 5.4). As a result, user kernels are reduced to minimum computation with a new order.

5.2 PROGRAM REPRESENTATION

During compiler optimizations, the program semantics and behavior are preserved while improving its performance. Maintaining the order of data accesses is especially important to ensure reproducibility of results. To express clear dependencies among statements, program analysis often relies on *static single-assignment form (SSA)*, which allows only one definition per variable [115]. Early work on equality saturation [23] used an SSA-based graph structure to represent and rewrite an entire program. For directive-based code, however, user-specified parallelism significantly affects performance [116], so compilers are limited to respect users' decisions.

Our proposed methodology optimizes OpenACC/OpenMP code while preserving a connection between e-graphs and original code. We track optimal expressions in e-graphs using SSA variables and insert optimized code into programs with the same structures and directives as input. The rest of this section covers the conversion process from directive-based code to e-graphs and explains the solution finding process based on SSA. By doing so, we are able to improve the performance of parallel code without changing its semantics or behavior.

5.2.1 *E-Graph Creation*

Both OpenACC and OpenMP contain sequential parts within the innermost parallel loops that may be executed independently across multiple threads and thus optimized for more efficient execution or reduced computation. To attain this, we create an e-graph for each innermost parallel loop as follows: First, we introduce conditional ϕ nodes [117] to represent control structures such as `if` and `for`, while merging data flows. Second, we assign an ID to each variable/array assignment or ϕ. Third, we update each variable/array load to refer to the latest ID along its data flow. Last, for each assignment or ϕ, we assign both the ID and the expression to the same e-class.

Examples of e-graphs for store/load, `if`, and `for` operations are illustrated in 1 at Fig. 5.1, with some IDs omitted for simplicity. The e-nodes are depicted as white blocks, and the e-classes as dotted boxes, while the black arrows represent parent-child relationships among them. As shown in each example, multiple references to the same variables

are now directed toward common e-classes, thereby reducing redundancy. For each updated variable, ϕ is created within `if` and `for` structures with conditions that may be concrete or abstract. Our tool supports C-style operations, such as function calls, pointers, and member references, within the same SSA framework. To optimize the code, ACC Saturator leverages our e-graph representation with rewriting rules and cost models, as detailed in Section 5.3.

5.2.2 *Code Selection*

To select optimal codes in the e-graph, we aim to find an equal expression that corresponds to a set of all assignments. We extract the lowest-cost expression that contains all the e-classes of assignments based on a sequence of IDs. The total cost is calculated as the sum of the cost of each e-class, with common e-classes being counted only once. To attain this, we use linear programming techniques [118].

Each assignment is updated based on its ID and the extracted expression. To enable reuse across assignments, the values of common expressions are stored in temporary variables. ACC Saturator allows for customizing the generation order of these temporary variables (Section 5.4).

5.3 Optimization with Saturation

In this section, we present our approach to rewriting OpenACC/OpenMP code using equality saturation. Our strategy involves a carefully crafted set of rewriting rules and a cost model that aims to minimize both computation and memory access.

5.3.1 *Rewriting Rules*

Reordering computations allows programs to explore different optimization possibilities. Rather than selecting a single alternative, compilers aim to choose the most profitable candidate according to their objective. Equality saturation defers this decision and instead accumulates equal expressions using multiple rewriting rules. With this approach, ACC Saturator can derive efficient operations and facilitate the reuse of expressions simultaneously, as shown in **2** at Fig. 5.1.

We apply two sets of rewriting rules to our equality saturation process. The first set introduces Fused Multiply-Adds (FMA) operations, which can improve code generation and resource utilization on GPUs [89]. Table 5.1 lists the minimum set of rules we use. When we encounter expressions that match the FMA pattern, we add the corresponding operations to the e-classes of matched expressions. The second set of rules benefits from the commutative and associative properties of the plus and multiply operators to reorder computation. This can enable common subexpression elimination and produce new FMA operations. We also incorporate constant folding of arithmetic operations with integer and floating-point numbers. Generating FMA operations is a simple expansion to the expression A + B * C. We note that, as Rawat *et al.* [89] argue, explicitly creating FMA forms often benefits code generation as practical compilers go through many optimization phases sequentially, losing other optimization opportunities.

While ACC Saturator can rewrite subtraction, division, memory access order, conditional expressions, and iterations, these rules can increase the size of e-graphs and lead to slow extraction of optimal solutions in real-time. Therefore, we restrict the tool to only use the set of rules mentioned earlier for efficient performance. Our rewriting rules are legal with both C and Fortran, except for floating-point accuracy. ACC Saturator can easily turn off the rules that could change floating-point results.

Table 5.1: ACC Saturator's rewriting rules

Name	Pattern		Result
FMA1	`A + B * C`	→	`FMA(A, B, C)`
FMA2	`A - B * C`	→	`FMA(A, -B, C)`
FMA3	`B * C - A`	→	`FMA(-A, B, C)`
COMM-ADD	`A + B`	→	`B + A`
COMM-MUL	`A * B`	→	`B * A`
ASSOC-ADD1	`A + (B + C)`	→	`(A + B) + C`
ASSOC-ADD2	`(A + B) + C`	→	`A + (B + C)`
ASSOC-MUL1	`A * (B * C)`	→	`(A * B) * C`
ASSOC-MUL2	`(A * B) * C`	→	`A * (B * C)`

5.3.2 *Cost Model*

GPUs are complex systems that require careful analysis to accurately predict their efficiency [109, 119]. While applications running on GPUs often face memory-bound limitations [120], it is not just the number of memory accesses that affects overall bandwidth. Factors such as on-chip resource utilization, processor occupancy, thread/grid-level parallelism across multiple memory layers, and instruction-level parallelism (ILP) all play a role, and improving one metric often comes at the expense of another. In ACC Saturator, our focus is on reducing memory access and computation by utilizing registers for common expressions, while maintaining ILP as described in Section 5.4.

Our cost model is simple: constant numbers pose no cost, each input variable or ϕ counts as 1, all computational operations except division and modular arithmetic count as 10, and each memory access, division, modular arithmetic, or function call counts as 100. The assigned costs are based on empirical testing and aim to reflect the relative cost of different operations. We acknowledge that future research could refine the cost values.

5.4 CODE GENERATION

To implement the extracted solutions from e-graphs in the input code, we introduce temporary variables. During the generation step, ACC Saturator leverages a novel technique called *bulk load* that prioritizes high memory pressure by reordering computations.

```
#pragma acc parallel loop gang num_gangs(ksize-1)\
                  num_workers(4) vector_length(32)
for (k = 1; k <= ksize-1; k++) {
#pragma acc loop worker
  for (i = 1; i <= gp02; i++) {
#pragma acc loop vector
    for (j = 1; j <= gp12; j++) {
      temp1 = dt * tz1; temp2 = dt * tz2;
      lhsZ[0][0][AA][k][i][j] =
   - temp2 * fjacZ[0][0][k-1][i][j]
   - temp1 * njacZ[0][0][k-1][i][j] - temp1 * dz1;
      // ... Similar 74 statements continue
}}}
```

Listing 5.1: One of kernels in NPB-BT's z_solve.c

```
#pragma acc parallel loop gang num_gangs(ksize-1)\
                  num_workers(4) vector_length(32)
for (k = 1; k <= ksize-1; k++) {
#pragma acc loop worker
  for (i = 1; i <= gp02; i++) {
#pragma acc loop vector
    for (j = 1; j <= gp12; j++) {
      double _v277, _v274, _v3 /* ... */;
      _v277 = njacZ[0][0][k][i][j];
      _v274 = njacZ[0][1][k][i][j];
      // ... Addr calculation + 123 loads continue
      temp1 = _v3;
     {double _v25; _v25 = dt * tz2; temp2 = _v25;
     {double _v435, _v434, _v283, _v433, _v436;
      _v283 = (- _v25);  _v433 = _v3 * _v432;
      _v434 = (- _v433);
      _v435 = _v434 + (_v283 * _v431);
      _v436 = _v435 - (dz1 * _v3);
      lhsZ[0][0][0][k][i][j] = _v436;
     {/* ... 74 stores */}}}
}}}
```

Listing 5.2: Generated Code of ACC Saturator (formatted)

5.4.1 *Temporary-Variable Insertion*

3 at Fig. 5.1 depicts the extracted expression in a directed graph. For each selected e-node, ACC Saturator generates a temporary variable to store the computational result, placing it immediately before the corresponding use. In cases where multiple statements reference an e-node, we select the innermost scope to declare a variable for those statements. Every assignment modifies the right-hand expression to include the variable of the corresponding e-node.

The compilers of directive-based code can optimize the redundant use of registers. Our code-generation style reduces duplicate computation and leverages optimal instructions, such as FMA, while preserving ILP.

5.4.2 *Bulk Load*

Since GPUs suffer high memory-access latency, reducing only memory accesses or computation may not lead to improved performance. ACC Saturator follows a different approach by reordering statements to increase memory pressure first and then minimizing memory operations for the remaining execution.

We address the high memory-access latency on GPUs by utilizing our proposed technique, bulk load. This technique relocates every memory load to the first place where its dependencies are resolved. When multiple loads share one location, we sort these based on their static indices. We prioritize increasing memory pressure at the beginning of the execution and then avoiding memory operations for the rest of the execution. To illustrate the effectiveness of this technique, we compare the performance of a time-consuming kernel in the OpenACC version of NAS Parallel Benchmarks' BT (NPB-BT) before and after optimization by ACC Saturator. Listings 5.1 and 5.2 show the original and optimized code, respectively. Despite featuring the same directives and code structure, the optimized code performs all the loads before the first assignment (`temp1`), and each subsequent store refers to local variables, leading to minimum number of operations while utilizing FMA.

5.5 EXPERIMENTAL METHODOLOGY

We implement ACC Saturator in Racket [121] using XcodeML [50] to parse and generate OpenACC/OpenMP source codes in C. We use the egg library [113] to perform equality saturation. ACC Saturator integrates with NVHPC [7] and GCC [8] for OpenACC/OpenMP compilation, and with Clang [10] for OpenMP compilation. For evaluation, we use NVHPC 22.9 with options `"-O3 -gpu=fastmath -Msafeptr -(acc|mp)=gpu"`, GCC 12.2.0 with `"-O3 -ffast-math -f(openacc|openmp)"` and Clang 15.0.3 with `"-O3 -ffast-math -fopenmp"`. Our experiments run on an NVIDIA A100-PCIE-40GB GPU with an Intel Xeon Silver 4114 CPU.

We evaluate the kernel-execution performance of ACC Saturator on two benchmark suites: NAS Parallel Benchmarks in OpenACC/C (NPB) [122] and the SPEC ACCEL benchmark suite in both OpenACC/C and OpenMP/C (SPEC) [57]. Tables 5.2 and 5.3 provide the detail of NPB and SPEC, respectively. To ensure adequate memory usage, we select CLASS C as the problem sizes of all NPB benchmarks (the largest size within standard test problems), SPEC uses the referential sizes (Ref) except for GCC's OpenACC cases of **ep**, **sp**, and **bt**, for which we select the testing sizes (Test) due to high execution latency. NPB's **BT**, **CG**, **EP**, and **SP** feature the same computation as SPEC's **bt**, **cg**, **ep**, and **csp**, but the implementation of NPB is based on OpenACC's parallel directive, while that of SPEC's OpenACC benchmarks is on the kernels directive. We report the best kernel performance of three executions. To avoid producing incorrect results, we remove the user-specific parallelism in NPB's **CG** for GCC, disable the device-side reduction of GCC's OpenACC in NPB's **LU** and **MG**, and omit the degree specification of the worker parallelism from GCC's NPB cases, since it surpasses GCC's thread limit.

On average, ACC Saturator spends 91.8 ms (σ = 253.3; 1.4 ~ 1885.0 ms) to construct SSA and generate code for each kernel. Equality saturation is executed within a limit of 10,000 e-nodes, 10 seconds of saturation time, 10 rewriting iterations, and a 30-second extraction

time limit. The results of the benchmarks show that each kernel requires an average of 0.63 sec (σ = 3.37; 0.00 ~ 31.2 sec) for equality saturation.

Table 5.2: NAS Parallel Benchmarks [122]

Name	Compute	Access	Num. Kernels	Original Time NVHPC	GCC
BT	CFD	Halo (3D)	46	14.85s	28.04s
CG	Eigenvalue	Irregular	16	1.27s	26.17s
EP	Random Num	Parallel	4	2.65s	3.35s
FT	FFT	All-to-All	12	3.06s	3.10s
LU	CFD	Halo (3D)	59	15.36s	24.86s
MG	Poisson Eq	Long & Short	16	0.79s	0.79s
SP	CFD	Halo (3D)	65	10.00s	12.00s

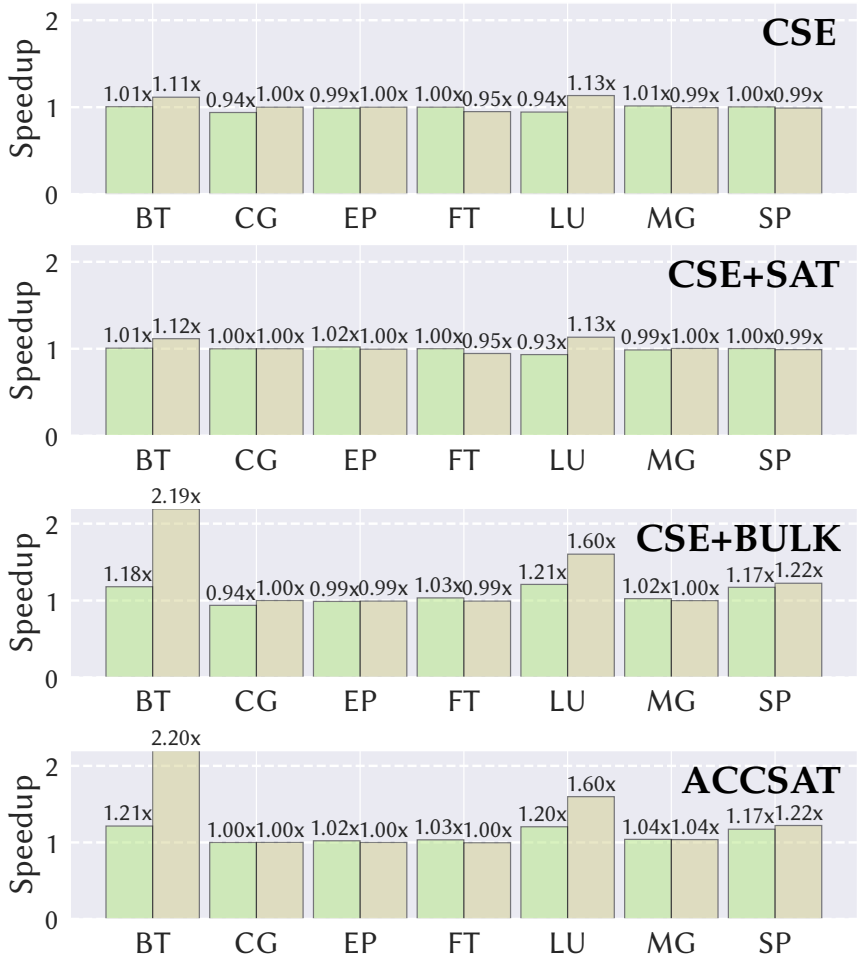


Figure 5.2: NPB's speedup results on NVIDIA A100-PCIE-40GB for each variation compared to original. ■ NVHPC, ■ GCC.

Table 5.3: The SPEC ACCEL benchmark suite [57]

Name	Compute	Access	Num. kernels	Size	Original Time (ACC)		Original Time (OMP)		
					NVHPC	GCC	NVHPC	GCC	Clang
ostencil	Jacobi	Halo (3D)	1	Ref	3.87s	10.28s	7.75s	107.54s	34.60s
olbm	CFD	Halo (3D)	3	Ref	7.11s	13.32s	7.11s	13.47s	5.91s
omriq	MRI	Structure-of-arrays	2	Ref	16.02s	16.18s	5.99s	18.54s	11.87s
ep	Random Num	Parallel	5	Ref / Test (CLASS D / W)	45.33s	69.91s	62.42s	90.35s	71.32s
cg	Eigenvalue	Irregular	16	Ref (> CLASS C)	4.28s	662.58s	5.06s	19.03s	18.42s
csp	CFD	Halo (3D)	68	Ref / Test (CLASS C / S)	7.71s	27.26s	111.79s	589.87s	105.75s
bt	CFD	Halo (3D)	50	Ref / Test (CLASS B / W)	3.24s	130.43s	555.44s	60.45s	562.83s

Fig. 5.2 presents the speedup results of NPB using four generated code versions. **CSE** is a version that eliminates redundant loads without performing equality saturation or bulk load. **CSE+SAT** and **CSE+BULK** are **CSE** with equality saturation and bulk load, respectively. **ACCSAT** is a default generated code of ACC Saturator, which includes both equality saturation and bulk load. With **ACCSAT**, NVHPC attains up to 1.21x improvement, while GCC attains up to 2.20x speedup. The **CSE** version maintains the performance of both compilers, varying the execution efficiency by 0.98x with NVHPC and by 1.03x with GCC on average. **CSE+SAT** provides NVHPC with an average speedup of 0.86% and improves GCC's performance by only 0.01%. However, **CSE+BULK** significantly accelerates memory-intensive applications such as **BT**, **LU**, and **SP**, while most other benchmarks maintain their performance. **ACCSAT** does not degrade the original performance and attains 2.00% and 0.66% better throughputs than **CSE+BULK** on NVHPC and GCC, respectively. In total, **ACCSAT** attains average speedups of 1.10x on NVHPC and 1.29x on GCC.

Table 5.4 breaks down the top 10 kernels in NPB's **BT**, and Fig. 5.3 shows the speedup of each kernel in each version. The speedups of **CSE+BULK** and **ACCSAT** are similar to those of **CSE** and **CSE+SAT**, respectively. **ACCSAT** attains up to 2.23x and 5.08x improvements on NVHPC and GCC, respectively, resulting in up to 2.08x and 3.19x memory bandwidth. The top three kernels on NVHPC suffer performance degradation from **CSE+BULK** to **ACCSAT**, because **ACCSAT** spills more registers to global memory, facilitating the reuse of computation. The next three kernels execute 8.3% fewer instructions

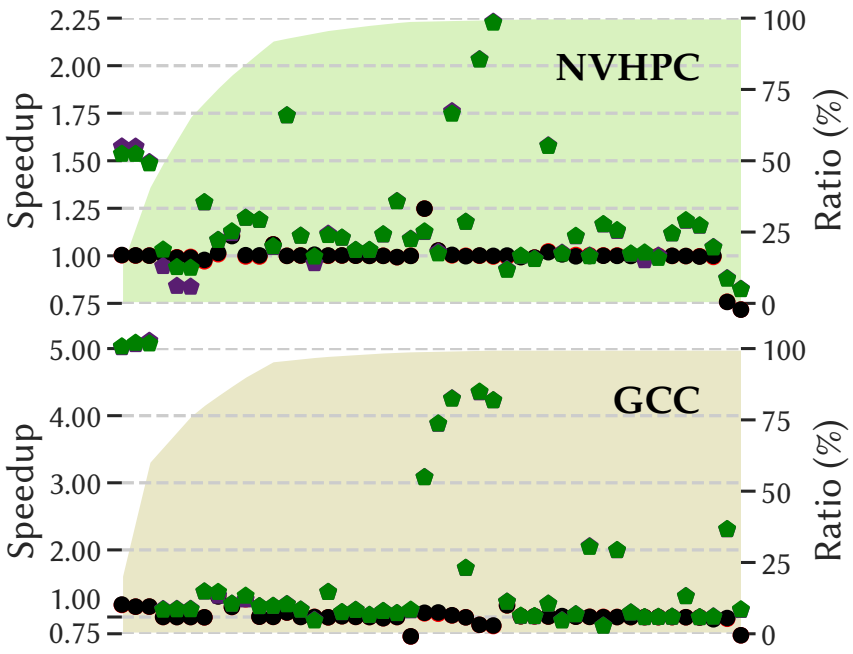


Figure 5.3: Breakdown of NPB-BT; The background color depicts the cumulative ratio of the execution time along the speedup points for each kernel. ● **CSE**, ● **CSE+SAT**, ⬟ **CSE+BULK**, ⬟ **ACCSAT** (CSE+SAT+BULK).

Table 5.4: Top-10 kernel breakdown of NPB-BT

%	NVHPC (14.85 sec)					+ CSE (14.77 sec)					+ CSE+SAT (14.75 sec)					+ CSE+BULK (12.59 sec)					+ ACCSAT (12.23 sec)				
13.6%	10.08	78.0	34.50%	152	**0.19**	10.05	100%	+0.68%	**-2**	+0.00	10.04	**99%**	+0.22%	+10	+0.00	**6.40**	106%	**+38.00%**	+103	-0.06	6.57	104%	+37.26%	+103	-0.06
13.6%	10.07	78.0	34.58%	152	**0.19**	10.06	100%	-0.74%	**-2**	+0.00	10.05	**99%**	+0.12%	+10	+0.00	**6.40**	106%	**+38.02%**	+103	-0.06	6.56	104%	+37.26%	+103	-0.06
13.5%	9.98	78.0	35.86%	152	**0.12**	9.96	100%	-0.09%	**-2**	+0.00	9.97	**99%**	-0.07%	+10	+0.00	**6.69**	106%	**+32.68%**	+103	+0.00	6.72	104%	+32.33%	+103	+0.00
8.8%	6.48	116.3	54.06%	**176**	**0.12**	6.42	100%	+0.02%	+0	+0.00	6.43	100%	+0.01%	+2	+0.00	6.85	108%	-2.02%	+78	+0.00	**6.28**	**99%**	**+2.02%**	+79	+0.00
8.1%	**6.01**	116.3	**57.11%**	**176**	**0.12**	6.05	100%	-0.14%	+2	+0.00	6.06	100%	-0.40%	+2	+0.00	7.14	108%	-6.69%	+78	+0.00	6.39	**99%**	-0.01%	+79	+0.00
8.0%	**5.92**	116.3	57.32%	**176**	**0.12**	5.96	100%	-0.25%	+2	+0.00	5.98	100%	-0.57%	+2	+0.00	7.08	108%	-5.92%	+78	+0.00	6.33	**99%**	**+0.11%**	+79	+0.00
5.1%	0.023	0.1	39.62%	**47**	**0.62**	0.024	103%	-1.04%	+0	+0.00	0.024	103%	-0.03%	+0	+0.00	0.018	**99%**	+5.53%	+39	-0.31	**0.018**	99%	**+6.86%**	+39	-0.31
5.0%	3.66	35.6	52.23%	96	**0.25**	3.63	**97%**	+1.37%	-2	+0.00	3.61	99%	+1.28%	**-4**	+0.00	**3.38**	97%	+2.74%	+0	+0.00	3.38	99%	**+3.43%**	+12	+0.00
4.6%	3.39	34.9	52.50%	**90**	**0.25**	3.06	**96%**	-0.70%	+23	+0.00	3.07	98%	-0.77%	+27	+0.00	3.03	96%	**+0.61%**	+23	+0.00	**3.00**	98%	+0.21%	+27	+0.00
4.0%	0.018	0.1	44.22%	**47**	**0.62**	0.019	103%	+0.54%	+0	+0.00	0.018	103%	+0.01%	+0	+0.00	0.016	**99%**	+2.83%	+39	-0.31	**0.016**	99%	**+4.37%**	+39	-0.31

%	GCC (28.04 sec)					+ CSE (25.16 sec)					+ CSE+SAT (25.13 sec)					+ CSE+BULK (12.79 sec)					+ ACCSAT (12.74 sec)				
20.1%	28.01	226.2	21.66%	**134**	**0.19**	24.29	89%	+2.40%	+0	+0.00	24.26	89%	+2.43%	+0	+0.00	5.53	**50%**	**+41.55%**	+121	-0.06	**5.51**	50%	+41.09%	+121	-0.06
20.1%	28.06	226.2	19.81%	**130**	**0.19**	23.80	89%	+2.98%	+0	+0.00	23.67	89%	+3.00%	+0	+0.00	5.59	**50%**	+43.07%	+125	-0.06	**5.58**	50%	**+43.41%**	+125	-0.06
20.0%	27.93	226.2	20.95%	**134**	**0.19**	24.20	89%	+2.44%	+2	+0.00	24.16	89%	+2.42%	+0	+0.00	**5.46**	50%	**+42.48%**	+121	-0.06	5.50	**50%**	+42.16%	+121	-0.06
5.4%	7.54	97.5	56.38%	**98**	**0.25**	7.54	100%	+0.06%	+0	+0.00	7.55	**99%**	+0.49%	+0	+0.00	**6.74**	101%	**+12.70%**	+157	-0.12	6.78	100%	+12.17%	+157	-0.12
5.4%	7.53	97.5	56.42%	**98**	**0.25**	7.53	100%	-0.04%	+0	+0.00	7.54	**99%**	-0.02%	+0	+0.00	**6.74**	100%	**+12.35%**	+157	-0.12	6.77	100%	+12.29%	+157	-0.12
5.4%	7.54	97.5	56.77%	98	0.25	7.54	100%	-0.76%	+0	+0.00	7.54	**99%**	-0.47%	+0	+0.00	**6.76**	100%	+11.93%	+157	-0.12	6.82	100%	**+12.10%**	+157	-0.12
3.9%	0.034	0.3	**43.50%**	**56**	**0.56**	0.034	100%	-0.35%	+0	+0.00	0.034	100%	-0.44%	+0	+0.00	0.025	**92%**	-9.16%	+48	-0.31	**0.025**	92%	-8.73%	+48	-0.31
3.4%	4.71	37.8	48.48%	**96**	**0.31**	3.58	87%	-1.79%	+0	+0.00	3.60	86%	-3.22%	+0	+0.00	3.58	87%	-2.30%	+0	+0.00	**3.43**	**85%**	**+0.24%**	+26	-0.06
3.3%	4.64	39.9	41.44%	**96**	**0.31**	4.03	87%	**+8.93%**	+8	-0.06	4.03	87%	+8.43%	+10	-0.06	**3.88**	87%	+8.11%	+12	-0.06	3.88	**87%**	+7.09%	+10	-0.06
3.2%	4.50	39.1	**49.96%**	**96**	**0.31**	3.56	88%	-3.55%	+0	+0.00	3.54	**86%**	-4.09%	+0	+0.00	3.58	89%	-4.06%	+0	+0.00	**3.43**	87%	-1.04%	+14	-0.06

indicates the average execution time per launch (ms), the number of executed instructions (x10^6), the memory utilization, the number of registers per thread, and the SM occupancy. In the columns of optimization, the last four use relative numbers comparing to the original. The most reduced/utilized numbers are shown bold.

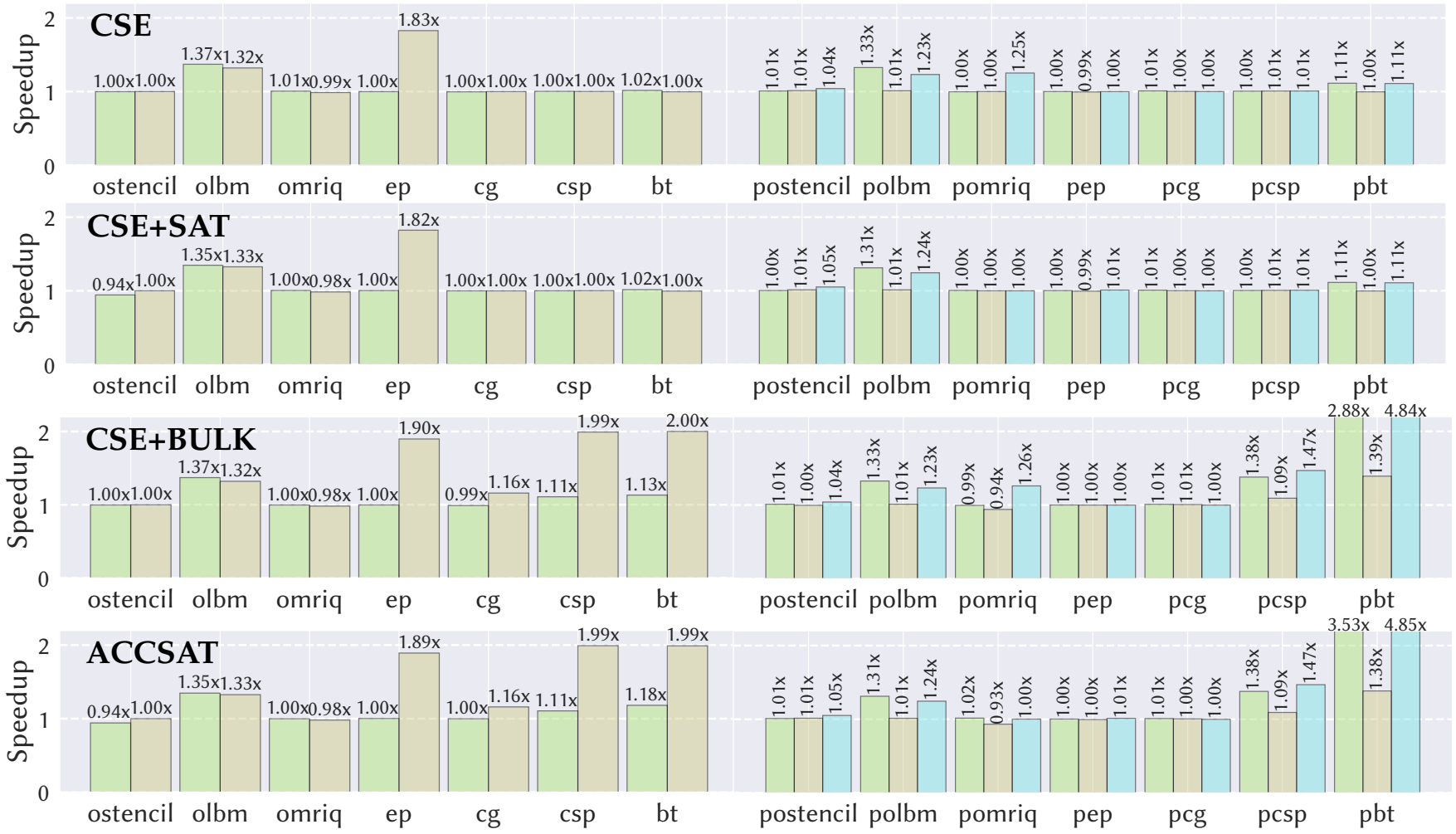


Figure 5.4: Speedup results of the SPEC ACCEL benchmark suite on NVIDIA A100-PCIE-40GB. NVHPC, GCC, Clang.

and achieve around a 1.11x speedup compared to **CSE+BULK**, since our optimized code clarifies dependencies and reduces both computation and stores. We gain a total speedup of 1.22x when equality saturation is disabled when registers spill.

On NVHPC, **CSE** increases the latency of **CG**, resulting in less SM occupancy due to increased register use. However, **CSE+SAT** and **ACCSAT** alleviate this register pressure by utilizing FMA. Equality saturation enables **EP** to become faster by executing 10.53% more FMA operations and 1.67% fewer total operations than **CSE**. Although other parts of **MG** suffer from register pressure, one part of the benchmark obtains a 1.14x speedup from **CSE+BULK** to **ACCSAT** by increasing the L1 cache hit ratio. Furthermore, **FT**, **LU**, and **SP** benchmarks attain improved memory throughputs and refined total performance with bulk load.

Fig. 5.4 depicts the speedups of SPEC benchmarks for both OpenACC and OpenMP versions. Benchmarks with names starting with "**p-**" indicate the OpenMP versions. On average, with OpenACC, **CSE**, **CSE+SAT**, **CSE+BULK**, and **ACCSAT** attain 1.06x, 1.04x, 1.08x, and 1.08x speedups, respectively on NVHPC, while with OpenMP, these attain 1.07x, 1.06x, 1.37x, and 1.47x speedups, respectively. OpenACC's **ostencil** sees a 16.7% reduction in SM occupancy due to equality saturation, leading to decreased performance. **CSE** reduces memory loads by around 50% for **olbm** and **polbm**, resulting in improved performance. Bulk load significantly boosts the performance of memory-intensive benchmarks such as **csp**, **bt**, **pcsp**, and **pbt**, just as in NPB. **ACCSAT** attains 3.62x speedup in one part of **pbt**, which executes only one thread-block over nested loops, by eliminating 77.2% memory loads and 50.9% stores. Our optimization decreases operations and reorders memory accesses, being effective for both parallel and sequential iterations.

GCC attains average speedups of 1.16x, 1.16x, 1.48x, and 1.48x for OpenACC using **CSE**, **CSE+SAT**, **CSE+BULK**, and **ACCSAT**, respectively, and 1.00x, 1.00x, 1.06x, and 1.06x for OpenMP. The original versions of OpenMP result in high register pressure, which limits the benefits of bulk load. Conversely, the initial versions of OpenACC use fewer registers while setting inadequate parallelism, likely due to the immature support of OpenACC's kernels directive. **CSE** reduces memory loads by 54.8% in **olbm**, yielding a 1.32x speedup. For **ep**, **CSE** minimizes operations, and bulk load enhances overall memory throughput. Bulk load also benefits **cg**, **csp**, **bt**, **pscp**, and **pbt** by mitigating global-memory latency. However, **pomriq**'s SM occupancy decreases with bulk load and equality saturation, leading to reduced efficiency.

Clang attains average speedups of 1.06x, 1.06x, 1.69x, and 1.66x for OpenMP using **CSE**, **CSE+SAT**, **CSE+BULK**, and **ACCSAT**, respectively. **CSE+SAT** refines the performance of **postencil**, **polbm**, and **pep** compared to **CSE**. **ACCSAT** further improves **pbt** over **CSE+BULK**, while reducing **pomriq**'s ILP due to decreased register usage. **CSE+BULK** attains a maximum speedup of 4.84x through optimized memory accesses.

For comparison, Fig. 5.5 provides the speedups of NPB on an NVIDIA A100-SXM4-80GB, which features 1.31x higher memory bandwidth than the A100-PCIE-40GB. The original performance is improved by 5.79% with NVHPC and 4.65% with GCC on average, while most benchmarks preserve the performance changes on the other GPU. We confirm that **BT** gains further 1.25x and 2.31x speedups by **ACCSAT** on NVHPC and GCC, respectively. Using the same GPU, **CSE+BULK** improves the execution by 1.21x and 2.30x, and **ACCSAT** improves both computation and memory throughputs in **BT**

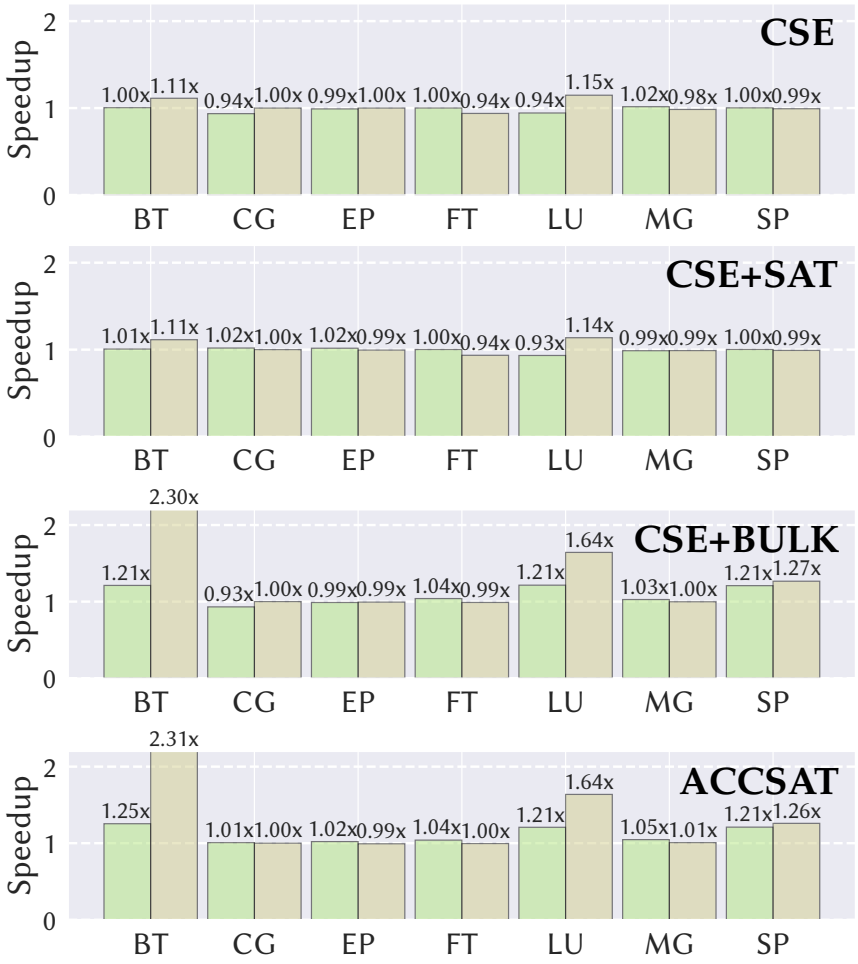


Figure 5.5: NPB's speedup results on NVIDIA A100-SXM4-80GB. NVHPC, GCC.

more for faster memory systems as the latency of computation becomes distinct. With **ACCSAT**, NVHPC and GCC attain average speedups of 1.11x and 1.31x, respectively. Our technique mitigates the memory latency that results from GCC's inadequate thread utilization for the kernels directive, resulting in superior performance gains compared to NVHPC.

Fig. 5.6 shows the speedups of SPEC benchmarks on NVIDIA A100-SXM4-80GB. With OpenACC, NVHPC increases the original performance by 7.42% and GCC increases by 3.11% using the GPU on average, while the original performance with OpenMP is increased by 3.33% on NVHPC, -13.3% on GCC, and 1.04% on Clang. Especially, **pcg** on GCC suffers from the latency of memory barriers, which decreases the original performance by 59.3% and degenerates the execution of optimized codes. **CSE+SAT** causes a lower L1 cache hit ratio for **pep** on NVHPC. Overall, **ACCSAT** obtains average speedups of 1.09x on NVHPC and 1.47x on GCC with OpenACC and average speedups of 1.45x on NVHPC, 1.02x on GCC, and 1.66x on Clang with OpenMP.

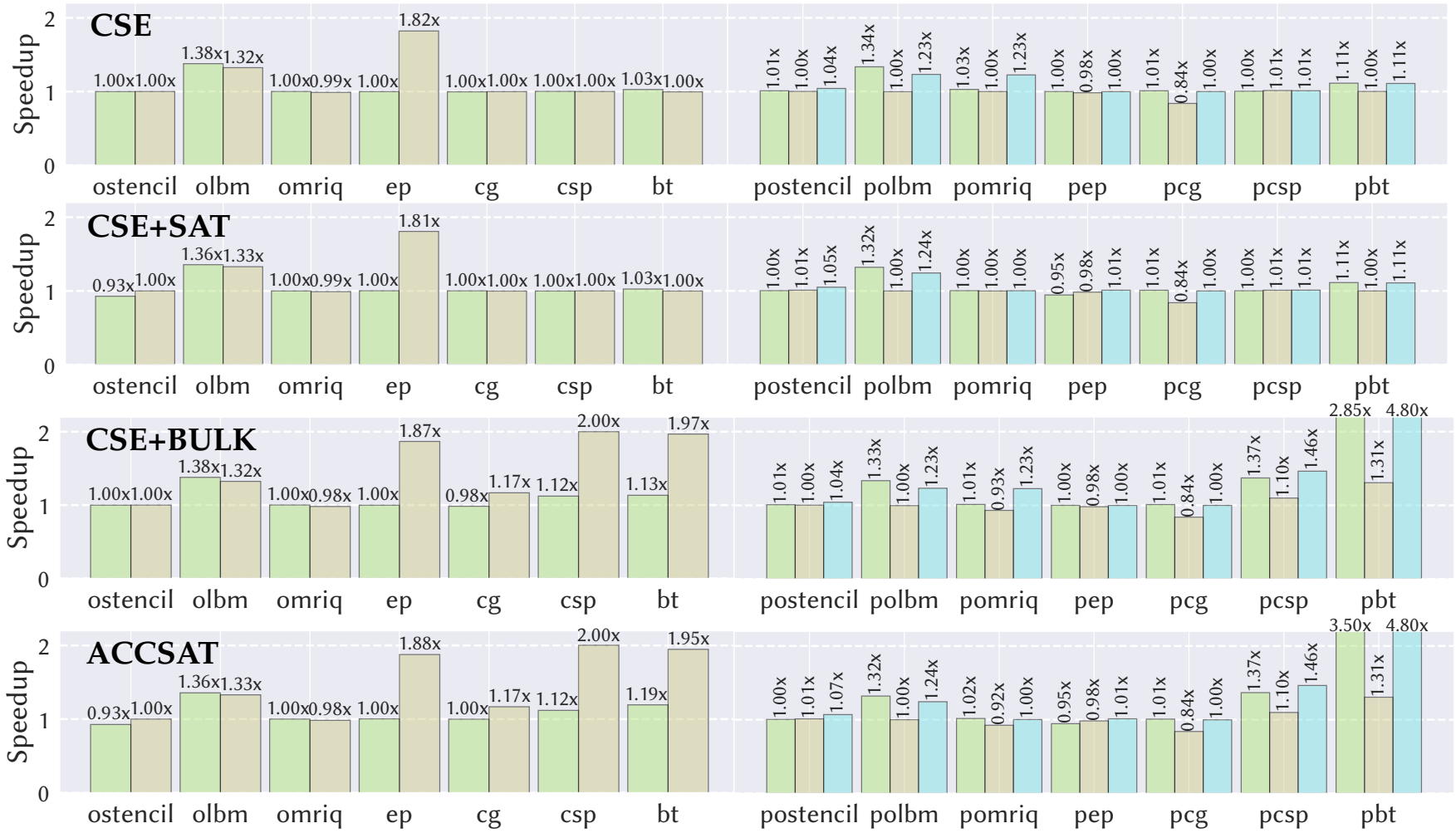


Figure 5.6: Speedup results of the SPEC ACCEL benchmark suite on NVIDIA A100-SXM4-80GB. NVHPC, GCC, Clang.

Maximizing architectural utilization is a significant challenge in HPC, requiring application code optimization. Compilers play a crucial role for enhancing performance by applying various code-generation techniques, enabling both generic and architecture-specific optimizations. Numerous programming models and state-of-the-art techniques, especially for GPUs, have been introduced to adapt codes to parallel architectures [87, 3, 123, 124, 125]. Directive-based programming models [5, 6] extend sequential languages, allowing complex scientific applications to offload loop iterations to accelerators while maintaining their structures. However, such compilers often rely on the basis of sequential code generation, thus limiting optimization opportunities to general computation [17].

Several projects have explored domain-specific or architecture-specific approaches for directive-based code. The CLAW DSL [126] provides directives for grid-based algorithms, enabling target-specific optimizations such as spatial blocking while supporting OpenACC/OpenMP code generation. Our work JACC is an OpenACC runtime framework offering just-in-time kernel compilation with dynamic constant expansion. OptACC [116] performs runtime parameter searches to optimize OpenACC parallelism. CCAMP [127] interchanges OpenACC and OpenMP, optimizing parallelization for each combination of models and architecture. Barua *et al.* [17] develop an automated OpenACC-kernel optimizer for maximizing ILP through loop unfolding. SAFARA [128] fully utilizes register resources to facilitate array reference reuse in OpenACC kernels.

Our ACC Saturator differs in three aspects: (1) automation of optimization through rewriting rules and a cost model-based optimal code selection, (2) integration of bulk load optimization technique for significant GPU memory throughput improvement, and (3) preservation of original code structures while being applicable to both OpenACC and OpenMP without requiring domain-specific information.

Since the introduction of the equality-saturation library egg [113], numerous studies have leveraged it, particularly in the context of GPU computing, for accelerating deep learning applications [129, 130, 131, 132]. These works employ rewriting rules for arithmetic expressions, abstract operations, or tensor graphs to optimize convolutions, sparse tensor algebra, or whole tensor operations. Diospyros [97] synthesizes efficient DSP operations from C code using equality saturation, while Gowda *et al.* [133] implement a symbolic algebra system with egg for automatic parallelism assignment.

Although the initial work of equality saturation was demonstrated as a Java bytecode optimizer [23], recent works using egg focus on program synthesis and code optimization without control statements. Our work is the first to bridge the gap between user code and equality saturation by automatically extracting computation and constructing SSA information for data dependencies. This approach enables novel equality-saturation optimizations for directive-based programming, as detailed in this chapter, without requiring further programmer intervention.

Several works propose innovative code optimization approaches. Ben-Nun *et al.* [134, 135] develop data-centric flow graphs to focus on data-oriented optimizations, such as dead memory elimination. Their representation supports collective operations and macroscopic parallelization, enabling optimizations irrespective of calculation. The MLIR framework [136] utilizes multi-level IRs for cooperative domain-specific optimizations. Gins-

bach *et al.* [137] define code patterns for performance opportunities, replacing matched code with library or DSL implementations. Additionally, machine learning is gaining popularity for automatic compiler-optimization tuning [138, 139, 140]. Our tool employs e-graph operations to identify optimal solutions without source code analysis or abstract syntax tree transformations.

Numerous optimization techniques have been developed for GPU computing [141]. Rawat *et al.* [89] reorder stencil computation using heuristic algorithms to alleviate register pressure and increase SM occupancy. Software systolic arrays [80] assign global data to GPU threads, computing results by propagating and accumulating partial results through warp-level shuffles across overlapped blocks, improving data locality without shared memory usage. Hong *et al.* [142] utilize lightweight kernel emulation to provide auto-tuners with performance bottleneck feedback. In contrast, ACC Saturator adopts a straightforward rewriting mechanism, enabling its application to complex programs. Moreover, our method is complementary to other code optimizations.

ACC Saturator is a more practical compiler optimization framework than previous work [23] because it maintains code structures, is capable of restoring the original state, and is applicable to standard languages such as C and Fortran. ACC Saturator allows equality saturation over directive-based code, upholding the importance of structure information which previous work disregarded. Reusing the code style that the input provides, ACC Saturator can support the user's intended target architecture by rewriting rules that lead to a new order of computation. Each rule is simple, but by combining many rules, we discover better solutions and attain speedups over the state-of-the-art industry compilers.

6

SUMMARY AND CONCLUSION

Closing this thesis, Chapter 6 summarizes our study and concludes the work.

6.1 SUMMARY

Computation is necessary for understanding humankind's situations. With the best knowledge of science and technology, the amount of calculations required for societies has grown enormously every day, so computers play principal roles in maintaining human activities and assisting their wills. Computer programming was originally indivisible from computer design. But, nowadays, multiple layers are put between the code and hardware to pursue easy integration of new ideas. Thus, programming styles are respected and create a trade-off between time and efficiency before the arrival of newer architectures.

Accelerator programming is based on the idea that offloading time-consuming tasks to a specialized device is sufficient to gain target performance improvements. Therefore, compilers do not take the entire programming execution into account for the offload and only consider part of applications. The vendor-specific programming models such as CUDA and OpenCL cover computational information by introducing different coding styles, which force additional engineering efforts. In contrast, directive-based programming models are harmonious extensions of standard languages. The directives are mere annotations, and the compilers are liable for generating the device code with the runtime that manages the total execution.

The compilation of directive-based code typically follows the CPU code generation, since offloaded kernels are found among CPU codes. Specializing the device code is supplemental to existing compiler work. The runtime and the device code generator are the places that can enhance accelerator use. Yet, limited insights available on execution environments prevent applications from achieving performance comparable to manually-written device code. While facing difficulties with its implementation to arrange compiler groundwork, substantial work has been proposed for the optimization of directive-based programming models. However, there is much work left to be done in the runtime, the backend, and the frontend.

In this thesis, we have presented JACC, an OpenACC framework that facilitates runtime extension. Organizing data mappings and kernel arguments as runtime information, JACC creates dynamic code from original kernels and compiles it with a specified compiler in order to support on-the-fly code extension automatically. Due to the memory-bound nature of GPUs, we proposed predicate-based filtering, a novel code-translation technique of multi-GPU utilization, for distributing highly-tuned applications without ad-

ditional user effort. JACC employs an adaptive algorithm for switching distribution based on the overhead of GPU-to-GPU communication. Having many kernels parallelized on a multi-GPU environment, we showed the performance improvements of several tested benchmarks where precise data-dependency analysis is always restrained.

This thesis also introduced symbolic emulation to compiling GPU code in order to discover hidden opportunities for optimization. We employed several languages, enabling OpenACC directives such as in C and Fortran, for the frontend to generate GPU assembly code. Then, our tool emulated the code upon symbols that substituted dynamic information. While pruning control flows to reduce the emulation time, we automatically found possible warp-level shuffles that may be synthesized to assembly code to bypass global-memory accesses. We applied this technique to a benchmark suite and complex application code showing results that improved multiple benchmarks on several generations of GPUs. We also provided the latency analysis across multiple GPUs to identify the use case of shuffles.

Lastly, this thesis presented ACC Saturator, an equality-saturation framework for directive-based programming models that optimizes code using rewriting rules and a cost model. It is the first framework to bridge the gap between user code and equality saturation optimizations while preserving original code structures and data dependencies. We introduced a novel technique, bulk load, which we enabled through our framework, to generate code with intentional high memory pressure. We demonstrated the effectiveness of ACC Saturator on various practical benchmarks using multiple compilers for both OpenACC and OpenMP on a state-of-the-art GPU architecture. Our analysis highlighted the significance of memory-access order and computational reordering, which ACC Saturator enables, for achieving significant performance improvements and increased memory throughput.

6.2 CONCLUSION

Exploring new directions for optimizing directive-based code, we found multiple methods to improve high-performance computing. Most of our techniques are automated because directive-based programming should be intended to accelerate legacy code bases with little effort. We designed each tool as open-source software available online to process one level of source code. By avoiding the use of compiler infrastructures, those tools became agile in terms of development and utilization. Our experimental results were shown using different generations of GPUs, different environments, and different compilers. Overall, we have implemented and explored various methods in the course of the study that had been never done with directive-based programming or GPUs. The following is a summary of the results we achieved through the course our study took:

- Our OpenACC framework opened a way to manipulate the runtime system of directive-based programming by wrapping up existing compilers, managing data movement, and extending both the compilation and the execution.
- The just-in-time compilation feature allowed kernel specialization based on the runtime information. We also attempted dynamic loop unrolling and kernel fusion. However, the performance of generated kernels fluctuated because of both reduced

and increased register uses. We reckoned the necessity of program optimization that changes execution mechanisms or computational orders while not updating code structures.

- The data-management feature automated asynchronous execution by solving data dependency among kernel launches. The latency of memory allocation was the only improved metric because the time-consuming kernels prevented overlapping execution. Therefore, we needed to gain more intra-kernel parallelism. Also, we made the performance of GCC comparable to NVHPC by adding a memory pool.

- We integrated predicate-based filtering, which exploits further intra-kernel parallelism by duplicating the same kernels over multi-GPU. With this algorithm, we never split loop ranges but computed only the part that updates assigned data segments according to the runtime information. On an NVLINK system, our framework successfully distributed applications whose data dependency never allows loop parallelization. Also, we tried to adopt Unified Memory to avoid unnecessary communications among GPUs. However, at the moment of our experiment, switching the Unified Memory mode caused memory thrashing. Thus, we let GPUs transfer the updated memory data to each other and obtained performance improvements.

- In order to introduce other execution mechanisms, we extended the GPU code assembler and enabled an intermediate emulator that collects register data in symbolic expressions. The assembly code became adaptable to the environment based on the emulation results and the set of rules that our tool implemented. We attempted peephole optimizations such as rewriting volatile operations and obtained speedups by higher memory throughputs.

- To promote more GPU-specific execution in directive-based programming, our assembler tool automated the generation of shuffle instructions. The shuffle instructions successfully replaced redundant global-memory loads while achieving performance improvement of up to 132%. By experimenting with various types of execution, we illustrated the use case of shuffles.

- The lessons on the importance of user-specified directives were seen through previous experiments. Modifying parallelism or loop structures highly affects the utilization of computational resources. The insertion of directives makes legacy code efficient, but at least we should respect such user specifications. Thus, our source-code optimizer was designed to reorder computational statements of directive-based programming models while maintaining the original code structures, resource use, and directives.

- We implemented equality saturation to synthesize the code with minimum costs. By simplifying the cost calculation, we could exhaustively search optimal codes from a colossal amount of candidates.

- We integrated a memory access optimization named bulk load. Our experimental results with practical benchmarks showed performance improvements by reordering and concentrating global memory loads, expecting further speedups on newer GPUs

as they broaden the gap between memory bandwidth and memory latency. On the whole, the synthesized code could discover hidden opportunities for speedups.

6.3 FUTURE WORK

We might want to know what computers are really doing. Given the complexity of modern hardware, it is necessary to profile performance characteristics automatically. We also expect the automation of hardware design and programming in the near future. Consequently, the aspect of optimization should change. Existing codebases are the heritage that benefits us to know the world better. Future computers would not compile original source code, but instead learn the code to import its idea based on the nature of architecture. Hence, automating program construction can be the direction of future work. We can challenge approximate computing while approaching both theoretical performance limits and scientific knowledge with some corrections. Directives for program approximation should become one step toward automation.

6.4 PUBLICATIONS

6.4.1 *Referred Conferences*

- **Kazuaki Matsumura**, Simon Garcia De Gonzalo, Antonio J. Peña. JACC: An OpenACC runtime framework with kernel-level and multi-GPU parallelization. International Conference on High Performance Computing, Data, and Analytics (*HiPC*), 2021.
- **Kazuaki Matsumura**, Simon Garcia De Gonzalo, Antonio J. Peña. A symbolic emulator for shuffle synthesis on the NVIDIA PTX code. International Conference on Compiler Construction (*CC*), 2023.
- **Kazuaki Matsumura**, Simon Garcia De Gonzalo, Antonio J. Peña. ACC Saturator: Automatic kernel optimization for directive-based GPU code. Preprint: https://arxiv.org/abs/2306.13002 (Under submission to an international conference)

6.4.2 *Referred Presentations*

- **Kazuaki Matsumura**, Simon Garcia De Gonzalo, Antonio J. Peña. Wrapping up existing OpenACC compilers for runtime extension. Poster at ISC High Performance 2021.
- **Kazuaki Matsumura**, Simon Garcia De Gonzalo, Antonio J. Peña. JACC: Automatically retargeting OpenACC kernels for multi-GPUs. Talk at NVIDIA GTC, 2022.

6.4.3 *Software*

- **JACC**: https://github.com/epeec/JACC

- **PTXASW**: https://github.com/khaki3/ptxas-wrapper
- **ACC Saturator**: https://github.com/khaki3/acc-saturator
  (To be published after the paper acceptance.)

ACKNOWLEDGEMENTS

Our work has been funded by the EPEEC project from the European Union's Horizon 2020 research and innovation program under grant agreement No. 801051, the Ministerio de Ciencia e Innovación—Agencia Estatal de Investigación (PID2019-107255GB-C21/AEI/10.13039/501100011033), and Departament de Recerca i Universitats from the Generalitat de Catalunya as the Research Group AccMem (Code: 2021 SGR 00807). We partially carried out our experiments on the ACME cluster that is owned by CIEMAT and funded by the Spanish Ministry of Economy and Competitiveness project CODEC-OSE (RTI2018-096006-B-I00). We gratefully acknowledge the support of the NVIDIA Solutions Lab who provided us the remote access to their GPU environment. We would like to acknowledge the NVIDIA AI Technology Center (NVAITC) Europe for their valuable help.

APPENDIX

Listing A.1 shows, in a Python-like pseudocode, our algorithm used for implementing PTXASW. The function `emulate` performs symbolic emulation and returns memory traces. `extract_shuffles` finds shuffle opportunities whose correctness is guaranteed by comparison among all the branches. `replace_loads` inserts shuffles and register-save code. `solve`/`verify` call an SMT solver internally.

Listing A.1: Pseudocode for the algorithm of PTXASW

```
def ptxasw(input_string):
  kernels = parse_ptx(input_string)             # String to Object

  # Optimize each kernel
  updated_kernels = [optimize(k) for k in kernels]

  output_string = to_string(update_kernels)     # Object to String

  run_original_ptxas(output_string)                     # Assemble

def optimize(kernel):
  memory_traces = emulate(kernel)         # Run symbolic emulation

  shuffle_pairs = extract_shuffles(memory_traces)  # Find shuffles

  return replace_loads(kernel, shuffle_pairs)  # Shuffle synthesis

# memory_traces contains relative positions between loads for each
# branch:   [[(hash(dst_load), hash(src_load), distance) ...] ...]
def extract_shuffles(memory_traces):
  # Get a unique list of dst_load
  all_dst = unique([b[0] for a in memory_traces for b in a])

  pairs = []

  for d in all_dst:
    # Find src_load that has the same distance in all the branches
    const_src = find_consistent_src(d, memory_traces)

    if len(const_src) > 0:
      # Select the src_load that has the shortest distance
      (src, dist) = find_closest_src(d, const_src, memory_traces)

      pairs.append((d, src, dist))

  return pairs

def replace_loads(kernel, shuffle_pairs):
  updated_kernel = []
```

Listing A.1 (Cont.): Pseudocode for the algorithm of PTXASW

```
40
41    for instruction in kernel:
42      r = extract_destination_operand(instruction)
43
44      # Find the instruction in shuffle_pairs
45      d = [t for t in shuffle_pairs if t[0] == hash(instruction)]
46      s = [t for t in shuffle_pairs if t[1] == hash(instruction)]
47
48      if len(d) == 1:
49        (_, src, dist) = d[0]
50        # Add the shuffle code of Listing 6
51        updated_kernel.append(
52          create_shuffle_code(r, src, dist, instruction))
53      else:
54        # Add the original instruction
55        updated_kernel.append(instruction)
56
57      if len(s) >= 1:
58        # Hold '%source' of Listing 6 ("mov %hash(instruction), %r")
59        updated_kernel.append(
60          create_register_save_code(hash(instruction), r))
61
62    return updated_kernel
63
64  def emulate(kernel):
65    # Get instructions which involve in I/O addr and the control
66    kernel = eliminate_unnecessary_instructions(kernel)
67
68    env = make_register_environment(kernel) # [(reg, value) ...]
69
70    return run(env, kernel, [], [], [])
71
72  def run(env, kernel, memory_trace, instruction_trace, bra_conds):
73    if len(kernel) == 0: # End of the execution
74      return [memory_trace]
75
76    inst = kernel[0]
77
78    # Finish once the execution enters the same code block again
79    if hash(inst) in instruction_trace:
80      return [memory_trace]
81
82    instruction_trace = instruction_trace + [hash(inst)]
83
84    if inst.is_loop_header:
85      # [(reg, value) ...] -> [(reg, wrap(value)) ...] (Sec 4.2)
86      env = update_iterators(env, inst)
87
88    if inst.is_branch:
89      # Get a symbolic value of a register for the branch condition
90      cond = dict(env)[inst.cond_reg]
91
```

Listing A.1 (Cont.): Pseudocode for the algorithm of PTXASW

```
92      # Determine the destination based on previous conds (Sec 4.2)
93      if verify(assume=bra_conds, equation=(cond == True)):
94        # (1) Jump
95        return run(env, inst.branch_destination, memory_trace+[],
96                 instruction_trace, bra_conds + [(cond == True)])
97      elif verify(assume=bra_conds, equation=(cond == False)):
98        # (2) Ignore branching
99        return run(env, kernel[1:], memory_trace+[],
100                instruction_trace, bra_conds + [(cond == False)])
101     else:   # Cause divergence
102       return (run(...) + run(...)) # (1) + (2)
103
104   env = execute_instruction(env, inst) # Sec 4.1
105   addr = calc_addr(inst, env) # Get a symbolc value of I/O addr
106
107   if inst.is_store:
108     # Leave the I/O address in the trace (Sec 4.3)
109     memory_trace += [(hash(inst), hash(inst), STORE_CONST, addr)]
110   elif inst.is_load:
111     memory_trace += [(hash(inst), hash(inst), LOAD_CONST, addr)]
112
113     for (src, src_addr) in extract_valid_loads(memory_trace):
114       # Replace %tid.x with %tid+N (Sec 5.1)
115       symbol_N = gensym(); sa = replace_tid(src_addr, symbol_N)
116
117       # Find N that satisfies addr(%tid)==src_addr(%tid+N)
118       n = solve(assume=(symbol_N<=31 and symbol_N>=-31),
119                 equation=(addr==sa), find=symbol_N)
120
121       if n and verify(assume=(symbol_N==n), equation=(addr==sa)):
122         memory_trace += [(hash(inst), src, n)]
123
124   return run(env, kernel[1:],
125              memory_trace+[], instruction_trace, bra_conds)
```

BIBLIOGRAPHY

[1] G. E. Moore. 1998. Cramming more components onto integrated circuits. *Proceedings of the IEEE*, 86, 1, 82–85. ISSN: 0018-9219. DOI: 10.1109/JPROC.1998.658762.

[2] TOP500.org. 2022. November 2022 | TOP500. (November 2022). https://www.top500.org/lists/top500/2022/11/.

[3] NVIDIA Corporation. 2022. Programming guide :: CUDA toolkit documentation. (March 2022). https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html.

[4] The Khronos Group Inc. 2023. OpenCL overview - The Khronos Group Inc. (January 2023). https://www.khronos.org/api/opencl.

[5] The OpenACC Organization. 2011. OpenACC. (2011). https://www.openacc.org/.

[6] The OpenMP ARB. 1997. OpenMP. (1997). https://www.openmp.org/.

[7] NVIDIA Corporation. 2022. High performance computing (HPC) SDK | NVIDIA. (April 2022). https://developer.nvidia.com/hpc-sdk.

[8] Free Software Foundation, Inc. 2023. GCC, the GNU compiler collection - GNU Project. (June 2023). https://gcc.gnu.org/.

[9] Intel Corporation. 2023. Compile cross-architecture: Intel® oneAPI DPC++/C++ Compiler. (June 2023). https://www.intel.com/content/www/us/en/developer/tools/oneapi/dpc-compiler.html.

[10] The LLVM Project. 2023. Clang C language family frontend for LLVM. (June 2023). https://clang.llvm.org/.

[11] WACCPD 2023. 2023. WACCPD 2023 – Tenth workshop on accelerator programming and directives. (June 2023). https://waccpd.org/.

[12] IWOMP. 2023. IWOMP 2022 - The 18th int'l. workshop on OpenMP. (June 2023). https://www.iwomp.org/.

[13] John Russell. 2017. OpenACC shines in global climate/weather codes - HPCwire. (November 2017). https://www.hpcwire.com/2017/11/14/openacc-shines-global-climateweather-codes/.

[14] Tetsuya Hoshino, Naoya Maruyama, Satoshi Matsuoka, and Ryoji Takaki. 2013. CUDA vs OpenACC: Performance case studies with kernel benchmarks and a memory-bound CFD application. In *Proceedings of the 13th IEEE/ACM International Symposium on Cluster, Cloud, and Grid Computing* (CCGRID '13). IEEE Press, Delft, Netherlands, 136–143. ISBN: 9780768549965. DOI: 10.1109/CCGrid.2013.12. https://doi.org/10.1109/CCGrid.2013.12.

[15] Suejb Memeti, Lu Li, Sabri Pllana, Joanna Kołodziej, and Christoph Kessler. 2017. Benchmarking OpenCL, OpenACC, OpenMP, and CUDA: Programming productivity, performance, and energy consumption. In *Proceedings of the 2017 Workshop on Adaptive Resource Management and Scheduling for Cloud Computing* (ARMS-CC '17). Association for Computing Machinery, Washington, DC, USA, 1–6. ISBN: 978145035-1164. DOI: 10.1145/3110355.3110356. https://doi.org/10.1145/3110355.3110356.

[16] Mikhail Khalilov and Alexey Timoveev. 2021. Performance analysis of CUDA, OpenACC and OpenMP programming models on TESLA V100 GPU. In *Journal of Physics: Conference Series* number 1. Volume 1740. IOP Publishing, 012056. DOI: 10.1088/1742-6596/1740/1/012056. https://doi.org/10.1088/1742-6596/1740/1/012056.

[17] Prithayan Barua, Jun Shirako, and Vivek Sarkar. 2018. Cost-driven thread coarsening for GPU kernels. In *Proceedings of the 27th International Conference on Parallel Architectures and Compilation Techniques* (PACT). Association for Computing Machinery, Limassol, Cyprus. ISBN: 9781450359863. DOI: 10.1145/3243176.3243196. https://doi.org/10.1145/3243176.3243196.

[18] Güray Özen, Simone Atzeni, Michael Wolfe, Annemarie Southwell, and Gary Klimowicz. 2018. OpenMP GPU offload in Flang and LLVM. In *2018 IEEE/ACM 5th Workshop on the LLVM Compiler Infrastructure in HPC (LLVM-HPC)*, 1–9. DOI: 10.1109/LLVM-HPC.2018.8639434.

[19] Shilei Tian, Joseph Huber, John Tramm, Barbara Chapman, and Johannes Doerfert. 2022. Just-in-time compilation and link-time optimization for OpenMP target offloading. In *OpenMP in a Modern World: From Multi-device Support to Meta Programming*. Michael Klemm, Bronis R. de Supinski, Jannis Klinkenberg, and Brandon Neth, (Eds.) Springer International Publishing, Cham, 145–158. ISBN: 978-3-031-15922-0.

[20] Calvin Montgomery, Jeffrey L. Overbey, and Xuechao Li. 2015. Autotuning OpenACC work distribution via direct search. In *Proceedings of the 2015 XSEDE Conference: Scientific Advancements Enabled by Enhanced Cyberinfrastructure* (XSEDE). Association for Computing Machinery, St. Louis, Missouri. ISBN: 9781450337205. DOI: 10.1145/2792745.2792783. https://doi.org/10.1145/2792745.2792783.

[21] Kazuaki Matsumura, Mitsuhisa Sato, Taisuke Boku, Artur Podobas, and Satoshi Matsuoka. 2018. MACC: An OpenACC transpiler for automatic multi-GPU use. In *Supercomputing Frontiers*. Rio Yokota and Weigang Wu, (Eds.) Springer International Publishing, Cham, 109–127. DOI: 10.1007/978-3-319-69953-0_7.

[22] Mehmet E. Belviranli, Seyong Lee, Jeffrey S. Vetter, and Laxmi N. Bhuyan. 2018. Juggler: A dependence-aware task-based execution framework for GPUs. In *Proceedings of the 23rd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming* (PPoPP). Association for Computing Machinery, Vienna, Austria, 54–67. ISBN: 9781450349826. DOI: 10.1145/3178487.3178492. https://doi.org/10.1145/3178487.3178492.

[23] Ross Tate, Michael Stepp, Zachary Tatlock, and Sorin Lerner. 2009. Equality saturation: A new approach to optimization. In *Proceedings of the 36th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (POPL '09). Association for Computing Machinery, Savannah, GA, USA, 264–276. ISBN: 9781605-583792. DOI: 10.1145/1480881.1480915. https://doi.org/10.1145/1480881.1480915.

[24] NVIDIA Corporation. 2010. NVIDIA Tesla | datasheet | apr10. (April 2010). https://www.nvidia.com/docs/IO/43395/NV_DS_Tesla_M2050_M2070_Apr10_LowRes.pdf.

[25] NVIDIA Corporation. 2014. NVIDIA Tesla GPU accelerators. (October 2014). https://www.nvidia.com/content/dam/en-zz/Solutions/Data-Center/tesla-product-literature/TeslaK80-datasheet.pdf.

[26] NVIDIA Corporation. 2015. Accelerating hyperscale data center applications with Tesla GPUs. (November 2015). https://devblogs.nvidia.com/accelerating-hyperscale-datacenter-applications-tesla-gpus/.

[27] NVIDIA Corporation. 2016. NVIDIA Tesla P100. (May 2016). https://images.nvidia.com/content/pdf/tesla/whitepaper/pascal-architecture-whitepaper.pdf.

[28] NVIDIA Corporation. 2017. NVIDIA Tesla V100 GPU architecture. (August 2017). https://images.nvidia.com/content/volta-architecture/pdf/volta-architecture-whitepaper.pdf.

[29] NVIDIA Corporation. 2020. NVIDIA A100 tensor core GPU architecture. (May 2020). https://www.nvidia.com/content/dam/en-zz/Solutions/Data-Center/nvidia-ampere-architecture-whitepaper.pdf.

[30] NVIDIA Corporation. 2020. NVLink & NVSwitch: Advanced multi-GPU systems | NVIDIA. (2020). https://www.nvidia.com/en-us/data-center/nvlink/.

[31] Lingda Li and Barbara Chapman. 2019. Compiler assisted hybrid implicit and explicit GPU memory management under unified address space. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis* (SC '19). Association for Computing Machinery, Denver, Colorado. ISBN: 9781450362290. DOI: 10.1145/3295500.3356141. https://doi.org/10.1145/3295500.3356141.

[32] Zhe Jia, Marco Maggioni, Benjamin Staiger, and Daniele Paolo Scarpazza. 2018. Dissecting the NVIDIA Volta GPU architecture via microbenchmarking. *CoRR*, abs/1804.06826. arXiv: 1804.06826. http://arxiv.org/abs/1804.06826.

[33] Zhe Jia, Marco Maggioni, Jeffrey Smith, and Daniele Paolo Scarpazza. 2019. Dissecting the NVIDIA Turing T4 GPU via microbenchmarking. (2019). DOI: 10.48550/ARXIV.1903.07486. https://arxiv.org/abs/1903.07486.

[34] Markus Velten, Robert Schöne, Thomas Ilsche, and Daniel Hackenberg. 2022. Memory performance of AMD EPYC Rome and Intel Cascade Lake SP server processors. In *Proceedings of the 2022 ACM/SPEC on International Conference on Performance Engineering* (ICPE '22). Association for Computing Machinery, Beijing, China, 165–175. ISBN: 9781450391436. DOI: 10.1145/3489525.3511689. https://doi.org/10.1145/3489525.3511689.

[35] Hamid Reza Zohouri and Satoshi Matsuoka. 2019. The memory controller wall: Benchmarking the Intel FPGA SDK for OpenCL memory interface. In *2019 IEEE/ACM International Workshop on Heterogeneous High-performance Reconfigurable Computing (H2RC)*, 11–18. DOI: 10.1109/H2RC49586.2019.00007.

[36] NVIDIA Corporation. 2022. Kepler tuning guide :: CUDA toolkit documentation. (March 2022). https://docs.nvidia.com/cuda/archive/11.4.4/kepler-tuning-guide/index.html.

[37] NVIDIA Corporation. 2022. PTX ISA :: CUDA toolkit documentation. (March 2022). https://docs.nvidia.com/cuda/parallel-thread-execution/index.html.

[38] Da Yan, Wei Wang, and Xiaowen Chu. 2020. Optimizing batched winograd convolution on GPUs. In *Proceedings of the 25th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. Association for Computing Machinery, New York, NY, USA, 32–44. ISBN: 9781450368186. https://doi.org/10.1145/3332466.3374520.

[39] Zhe Jia, Marco Maggioni, Benjamin Staiger, and Daniele P. Scarpazza. 2018. Dissecting the NVIDIA Volta GPU architecture via microbenchmarking. (2018). DOI: 10.48550/ARXIV.1804.06826. https://arxiv.org/abs/1804.06826.

[40] Henry Wong, Misel-Myrto Papadopoulou, Maryam Sadooghi-Alvandi, and Andreas Moshovos. 2010. Demystifying GPU microarchitecture through microbenchmarking. In *2010 IEEE International Symposium on Performance Analysis of Systems Software (ISPASS)*, 235–246. DOI: 10.1109/ISPASS.2010.5452013.

[41] Jacob Lambert, Seyong Lee, Jeffrey S. Vetter, and Allen D. Malony. 2020. CCAMP: An integrated translation and optimization framework for OpenACC and OpenMP. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis* (SC). IEEE Press, Atlanta, Georgia. ISBN: 9781728199986.

[42] T. Hoshino, N. Maruyama, and S. Matsuoka. 2016. A directive-based data layout abstraction for performance portability of OpenACC applications. In *2016 IEEE 18th International Conference on High Performance Computing and Communications; IEEE 14th International Conference on Smart City; IEEE 2nd International Conference on Data Science and Systems (HPCC/SmartCity/DSS)*, 1147–1154. DOI: 10.1109/HPCC-SmartCity-DSS.2016.0161.

[43] X. Tian, D. Khaldi, D. Eachempati, R. Xu, and B. Chapman. 2016. Optimizing GPU register usage: Extensions to OpenACC and compiler optimizations. In *2016 45th International Conference on Parallel Processing (ICPP)*, 572–581. DOI: 10.1109/ICPP.2016.72.

[44] A. Lashgar, E. Atoofian, and A. Baniasadi. 2018. Loop perforation in OpenACC. In *2018 IEEE Intl Conf on Parallel Distributed Processing with Applications, Ubiquitous Computing Communications, Big Data Cloud Computing, Social Computing Networking, Sustainable Computing Communications (ISPA/IUCC/BDCloud/SocialCom/SustainCom)*, 163–170. DOI: 10.1109/BDCloud.2018.00036.

[45] Michael Wolfe, Seyong Lee, Jungwon Kim, Xiaonan Tian, Rengan Xu, Barbara Chapman, and Sunita Chandrasekaran. 2018. The OpenACC data model: Preliminary study on its major challenges and implementations. *Parallel Computing*, 78, 15–27. DOI: 10.1016/j.parco.2018.07.003.

[46] Prasad A. Kulkarni. 2011. JIT compilation policy for modern machines. In *Proceedings of the 2011 ACM International Conference on Object Oriented Programming Systems Languages and Applications* (OOPSLA '11). Association for Computing Machinery, Portland, Oregon, USA, 773–788. ISBN: 9781450309400. DOI: 10.1145/2048066.2048126. https://doi.org/10.1145/2048066.2048126.

[47] Toshiya Komoda, Shinobu Miwa, Hiroshi Nakamura, and Naoya Maruyama. 2013. Integrating multi-GPU execution in an OpenACC compiler. In *2013 42nd International Conference on Parallel Processing (ICPP)*. IEEE, 260–269. DOI: 10.1109/ICPP.2013.35.

[48] M. Classen and M. Griebl. 2006. Automatic code generation for distributed memory architectures in the polytope model. In *Proceedings 20th IEEE International Parallel Distributed Processing Symposium (IPDPS)*, 7 pp.–. DOI: 10.1109/IPDPS.2006.1639500.

[49] Thejas Ramashekar and Uday Bondhugula. 2013. Automatic data allocation and buffer management for multi-GPU machines. *ACM Trans. Archit. Code Optim.*, 10, 4, (December 2013). ISSN: 1544-3566. DOI: 10.1145/2544100. https://doi.org/10.1145/2544100.

[50] Omni Compiler Project (RIKEN CCS). 2009. XcodeML. (2009). https://omni-compiler.org/xcodeml.html.

[51] S. Ghosh, T. Liao, H. Calandra, and B. M. Chapman. 2012. Experiences with OpenMP, PGI, HMPP and OpenACC directives on ISO/TTI kernels. In *2012 SC Companion: High Performance Computing, Networking Storage and Analysis*, 691–700. DOI: 10.1109/SC.Companion.2012.95.

[52] Jonas Hahnfeld, Christian Terboven, James Price, Hans Joachim Pflug, and Matthias S. Müller. 2018. Evaluation of asynchronous offloading capabilities of accelerator programming models for multiple devices. In *Fourth Workshop on Accelerator Programming Using Directives (WACCP)*. Sunita Chandrasekaran and Guido Juckeland, (Eds.) Springer International Publishing, Cham, 160–182. DOI: 10.1007/978-3-319-74896-2_9.

[53] Rengan Xu, Xiaonan Tian, Sunita Chandrasekaran, Yonghong Yan, and Barbara Chapman. 2014. NAS parallel benchmarks for GPGPUs using a directive-based programming model. In *International Workshop on Languages and Compilers for Parallel Computing*. Springer, 67–81. DOI: 10.1007/978-3-319-17473-0_5.

[54] J. A. Herdman, W. P. Gaudin, S. McIntosh-Smith, M. Boulton, D. A. Beckingsale, A. C. Mallinson, and S. A. Jarvis. 2012. Accelerating hydrocodes with OpenACC, OpenCL and CUDA. In *2012 SC Companion: High Performance Computing, Networking Storage and Analysis*, 465–471. DOI: 10.1109/SC.Companion.2012.66.

[55] RIKEN CCS. 2014. Fiber. (2014). http://fiber-miniapp.github.io/.

[56] RIKEN CCS. 2001. Himeno benchmark. (2001). https://i.riken.jp/en/supercom/documents/himenobmt/.

[57] Standard Performance Evaluation Corporation. 2014. SPEC ACCEL®. (2014). https://www.spec.org/accel/.

[58] T. Diop, S. Gurfinkel, J. Anderson, and N. E. Jerger. 2013. DistCL: A framework for the distributed execution of OpenCL kernels. In *2013 IEEE 21st International Symposium on Modelling, Analysis and Simulation of Computer and Telecommunication Systems*, 556–566. DOI: 10.1109/MASCOTS.2013.77.

[59] Jungwon Kim, Sangmin Seo, Jun Lee, Jeongho Nah, Gangwon Jo, and Jaejin Lee. 2012. SnuCL: An OpenCL framework for heterogeneous CPU/GPU clusters. In *Proceedings of the 26th ACM International Conference on Supercomputing* (ICS). Association for Computing Machinery, San Servolo Island, Venice, Italy, 341–352. ISBN: 9781450313162. DOI: 10.1145/2304576.2304623. https://doi.org/10.1145/2304576.2304623.

[60] U. Bondhugula. 2013. Compiling affine loop nests for distributed-memory parallel architectures. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis* (SC). Association for Computing Machinery, Denver, Colorado. ISBN: 9781450323789. DOI: 10.1145/2503210.2503289. https://doi.org/10.1145/2503210.2503289.

[61] Alexander Matz, Johannes Doerfert, and Holger Fröning. 2020. Automated partitioning of data-parallel kernels using polyhedral compilation. In *49th International Conference on Parallel Processing Workshops* (ICPP Workshops). Association for Computing Machinery, Edmonton, AB, Canada. ISBN: 9781450388689. DOI: 10.1145/3409390.3409403. https://doi.org/10.1145/3409390.3409403.

[62] Sebastian Schaetz and Martin Uecker. 2012. A multi-GPU programming library for real-time applications. In *International Conference on Algorithms and Architectures for Parallel Processing*. Springer, 114–128. DOI: 10.1007/978-3-642-33078-0_9.

[63] Tal Ben-Nun, Michael Sutton, Sreepathi Pai, and Keshav Pingali. 2017. Groute: An asynchronous multi-GPU programming model for irregular computations. In *Proceedings of the 22nd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming* (PPoPP). Association for Computing Machinery, Austin, Texas, USA, 235–248. ISBN: 9781450344937. DOI: 10.1145/3018743.3018756. https://doi.org/10.1145/3018743.3018756.

[64] Tal Ben-Nun, Ely Levy, Amnon Barak, and Eri Rubin. 2015. Memory access patterns: The missing piece of the multi-GPU puzzle. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis* (SC). Association for Computing Machinery, Austin, Texas. ISBN: 9781450337236. DOI: 10.1145/2807591.2807611. https://doi.org/10.1145/2807591.2807611.

[65] S. Potluri, A. Goswami, D. Rossetti, C. J. Newburn, M. G. Venkata, and N. Imam. 2017. GPU-centric communication on NVIDIA GPU clusters with InfiniBand: A case study with OpenSHMEM. In *2017 IEEE 24th International Conference on High Performance Computing (HiPC)*, 253–262. DOI: 10.1109/HiPC.2017.00037.

[66] Prasanna Pandit and R. Govindarajan. 2014. Fluidic kernels: Cooperative execution of OpenCL programs on multiple heterogeneous devices. In *Proceedings of Annual IEEE/ACM International Symposium on Code Generation and Optimization* (CGO). Association for Computing Machinery, Orlando, FL, USA, 273–283. ISBN: 978145032-6704. DOI: 10.1145/2581122.2544163. https://doi.org/10.1145/2581122.2544163.

[67] Janghaeng Lee, Mehrzad Samadi, Yongjun Park, and Scott Mahlke. 2015. SKMD: Single kernel on multiple devices for transparent CPU-GPU collaboration. *ACM Trans. Comput. Syst.*, 33, 3, (August 2015). ISSN: 0734-2071. DOI: 10.1145/2798725. https://doi.org/10.1145/2798725.

[68] Monica Beltrametti, Kenneth Bobey, and John R. Zorbas. 1988. The control mechanism for the myrias parallel computer system. *SIGARCH Comput. Archit. News*, 16, 4, 21–30. ISSN: 0163-5964. DOI: 10.1145/54331.54333. https://doi.org/10.1145/54331.54333.

[69] NVIDIA Corporation. 2017. GitHub - NVIDIA/jitify. (2017). https://github.com/NVIDIA/jitify.

[70] D. Mikushin, N. Likhogrud, E. Z. Zhang, and C. Bergström. 2014. KernelGen – The design and implementation of a next generation compiler platform for accelerating numerical models on GPUs. In *2014 IEEE International Parallel Distributed Processing Symposium Workshops*, 1011–1020. DOI: 10.1109/IPDPSW.2014.115.

[71] José M. Andión, Manuel Arenaz, François Bodin, Gabriel Rodríguez, and Juan Touriño. 2016. Locality-aware automatic parallelization for GPGPU with OpenHMPP directives. *Int. J. Parallel Program.*, 44, 3, (June 2016), 620–643. ISSN: 0885-7458. DOI: 10.1007/s10766-015-0362-9. https://doi.org/10.1007/s10766-015-0362-9.

[72] Gleison Mendonça, Breno Guimarães, Péricles Alves, Márcio Pereira, Guido Araújo, and Fernando Magno Quintão Pereira. 2017. DawnCC: Automatic annotation for data parallelism and offloading. *ACM Trans. Archit. Code Optim.*, 14, 2, (May 2017). ISSN: 1544-3566. DOI: 10.1145/3084540. https://doi.org/10.1145/3084540.

[73] J. E. Denny, S. Lee, and J. S. Vetter. 2018. CLACC: Translating OpenACC to OpenMP in Clang. In *2018 IEEE/ACM 5th Workshop on the LLVM Compiler Infrastructure in HPC (LLVM-HPC)*, 18–29. DOI: 10.1109/LLVM-HPC.2018.8639349.

[74] Simon Garcia De Gonzalo, Sitao Huang, Juan Gómez-Luna, Simon Hammond, Onur Mutlu, and Wen-mei Hwu. 2019. Automatic generation of warp-level primitives and atomic instructions for fast and portable parallel reduction on GPUs. In *2019 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, 73–84. DOI: 10.1109/CGO.2019.8661187.

[75] Sven Verdoolaege and Tobias Grosser. 2012. Polyhedral extraction tool. In *Second International Workshop on Polyhedral Compilation Techniques (IMPACT)*. Paris, France. http://impact.gforge.inria.fr/impact2012/.

[76] Tobias Grosser, Armin Groesslinger, and Christian Lengauer. 2012. Polly — performing polyhedral optimizations on a low-level intermediate representation. *Parallel Processing Letters*, 22, 04, 1250010. DOI: 10.1142/S0129626412500107. eprint: https://doi.org/10.1142/S0129626412500107. https://doi.org/10.1142/S0129626412500107.

[77] Philip Ginsbach, Toomas Remmelg, Michel Steuwer, Bruno Bodin, Christophe Dubach, and Michael F. P. O'Boyle. 2018. Automatic matching of legacy code to heterogeneous APIs: An idiomatic approach. In *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems*. Association for Computing Machinery, New York, NY, USA, 139–153. ISBN: 9781450349116. https://doi.org/10.1145/3173162.3173182.

[78] Xiaolong Xie, Yun Liang, Xiuhong Li, Yudong Wu, Guangyu Sun, Tao Wang, and Dongrui Fan. 2015. Enabling coordinated register allocation and thread-level parallelism optimization for GPUs. In *2015 48th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 395–406. DOI: 10.1145/2830772.2830813.

[79] Roberto Baldoni, Emilio Coppa, Daniele Cono D'elia, Camil Demetrescu, and Irene Finocchi. 2018. A survey of symbolic execution techniques. *ACM Comput. Surv.*, 51, 3. ISSN: 0360-0300. DOI: 10.1145/3182657. https://doi.org/10.1145/3182657.

[80] Peng Chen, Mohamed Wahib, Shinichiro Takizawa, Ryousei Takano, and Satoshi Matsuoka. 2019. A versatile software systolic execution model for GPU memory-bound kernels. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis* (SC '19). Association for Computing Machinery, Denver, Colorado. ISBN: 9781450362290. DOI: 10.1145/3295500.3356162. https://doi.org/10.1145/3295500.3356162.

[81] Robert van Engelen. 2001. Efficients symbolic analysis for optimizing compilers. In *Proceedings of the 10th International Conference on Compiler Construction* (CC '01). Springer-Verlag, Berlin, Heidelberg, 118–132. ISBN: 354041861X.

[82] Robert van Engelen. 2000. Symbolic evaluation of chains of recurrences for loop optimization. Technical report TR-000102. Computer Science Dept., Florida State University.

[83] Leonardo de Moura and Nikolaj Bjørner. 2008. Z3: An efficient SMT solver. In *Tools and Algorithms for the Construction and Analysis of Systems*. Springer Berlin Heidelberg, Berlin, Heidelberg, 337–340. ISBN: 978-3-540-78800-3.

[84] Phitchaya Mangpo Phothilimthana, Archibald Samuel Elliott, An Wang, Abhinav Jangda, Bastian Hagedorn, Henrik Barthels, Samuel J. Kaufman, Vinod Grover, Emina Torlak, and Rastislav Bodik. 2019. Swizzle Inventor: Data movement synthesis for GPU kernels. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems* (ASPLOS '19). Association for Computing Machinery, Providence, RI, USA, 65–78. ISBN: 9781450362405. DOI: 10.1145/3297858.3304059. https://doi.org/10.1145/3297858.3304059.

[85] Emina Torlak and Rastislav Bodik. 2013. Growing solver-aided languages with Rosette. In *Proceedings of the 2013 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming & Software* (Onward! 2013). Association for Computing Machinery, Indianapolis, Indiana, USA, 135–152. ISBN: 978145032-4724. DOI: 10.1145/2509578.2509586. https://doi.org/10.1145/2509578.2509586.

[86] Arpith Chacko Jacob et al. 2017. Efficient fork-join on GPUs through warp specialization. In *2017 IEEE 24th International Conference on High Performance Computing (HiPC)*, 358–367. DOI: 10.1109/HiPC.2017.00048.

[87] Jonathan Ragan-Kelley, Connelly Barnes, Andrew Adams, Sylvain Paris, Frédo Durand, and Saman Amarasinghe. 2013. Halide: A language and compiler for optimizing parallelism, locality, and recomputation in image processing pipelines. In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation* (PLDI '13). Association for Computing Machinery, Seattle, Washington, USA, 519–530. ISBN: 9781450320146. DOI: 10.1145/2491956.2462176. https://doi.org/10.1145/2491956.2462176.

[88] Dmitry Mikushin, Nikolay Likhogrud, Eddy Z. Zhang, and Christopher Bergström. 2014. KernelGen – The design and implementation of a next generation compiler platform for accelerating numerical models on GPUs. In *2014 IEEE International Parallel Distributed Processing Symposium Workshops*, 1011–1020. DOI: 10.1109/IPDPSW.2014.115.

[89] Prashant Singh Rawat, Fabrice Rastello, Aravind Sukumaran-Rajam, Louis-Noël Pouchet, Atanas Rountev, and P. Sadayappan. 2018. Register optimizations for stencils on GPUs. In *Proceedings of the 23rd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming* (PPoPP '18). Association for Computing Machinery, Vienna, Austria, 168–182. ISBN: 9781450349826. DOI: 10.1145/3178487.3178500. https://doi.org/10.1145/3178487.3178500.

[90] ExaCT 2013. 2013. ExaCT: Center for exascale simulation of combustion in turbulence: Proxy App Software. (2013). http://www.exactcodesign.org/proxy-app-software/.

[91] SW4 2014. 2014. Seismic wave modelling (SW4) - Computational infrastructure for geodynamics. (2014). https://geodynamics.org/resources/sw4.

[92] Justin Luitjens. 2014. Faster parallel reductions on kepler. (February 2014). https://developer.nvidia.com/blog/faster-parallel-reductions-kepler/.

[93] Bryan Catanzaro, Alexander Keller, and Michael Garland. 2014. A decomposition for in-place matrix transposition. In *Proceedings of the 19th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming* (PPoPP '14). Association for Computing Machinery, Orlando, Florida, USA, 193–206. ISBN: 9781450326568. DOI: 10.1145/2555243.2555253. https://doi.org/10.1145/2555243.2555253.

[94] Eli Ben-Sasson, Matan Hamilis, Mark Silberstein, and Eran Tromer. 2016. Fast multiplication in binary fields on GPUs via register cache. In *Proceedings of the 2016 International Conference on Supercomputing* (ICS '16). Association for Computing Ma-

chinery, Istanbul, Turkey. ISBN: 9781450343619. DOI: 10.1145/2925426.2926259. https://doi.org/10.1145/2925426.2926259.

[95] Anjia Wang, Xinyao Yi, and Yonghong Yan. 2020. Supporting data shuffle between threads in OpenMP. In *OpenMP: Portable Multi-Level Parallelism on Modern Systems*. Springer International Publishing, Cham, 98–112. ISBN: 978-3-030-58144-2.

[96] Meghan Cowan, Thierry Moreau, Tianqi Chen, James Bornholt, and Luis Ceze. 2020. Automatic generation of high-performance quantized machine learning kernels. In *Proceedings of the 18th ACM/IEEE International Symposium on Code Generation and Optimization*. Association for Computing Machinery, New York, NY, USA, 305–316. ISBN: 9781450370479. https://doi.org/10.1145/3368826.3377912.

[97] Alexa VanHattum, Rachit Nigam, Vincent T. Lee, James Bornholt, and Adrian Sampson. 2021. Vectorization for digital signal processors via equality saturation. In (ASPLOS 2021). Association for Computing Machinery, Virtual, USA, 874–886. ISBN: 9781450383172. DOI: 10.1145/3445814.3446707. https://doi.org/10.1145/3445814.3446707.

[98] Simon Moll and Sebastian Hack. 2018. Partial control-flow linearization. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation* (PLDI 2018). Association for Computing Machinery, Philadelphia, PA, USA, 543–556. ISBN: 9781450356985. DOI: 10.1145/3192366.3192413. https://doi.org/10.1145/3192366.3192413.

[99] Charles Yount. 2015. Vector Folding: Improving stencil performance via multi-dimen-sional SIMD-vector representation. In *2015 IEEE 17th International Conference on High Performance Computing and Communications, 2015 IEEE 7th International Symposium on Cyberspace Safety and Security, and 2015 IEEE 12th International Conference on Embedded Software and Systems*, 865–870. DOI: 10.1109/HPCC-CSS-ICESS.2015.27.

[100] Verónica G Vergara Larrea, Wael R Elwasif, Oscar Hernandez, Cesar Philippidis, and Randy Allen. 2017. An in-depth evaluation of GCC's OpenACC implementation on Cray systems.

[101] Joshua Hoke Davis, Christopher Daley, Swaroop Pophale, Thomas Huber, Sunita Chandrasekaran, and Nicholas J. Wright. 2021. Performance assessment of OpenMP compilers targeting NVIDIA V100 GPUs. In *Accelerator Programming Using Directives*. Springer International Publishing, Cham, 25–44. ISBN: 978-3-030-74224-9.

[102] Ahmad Lashgar and Amirali Baniasadi. 2016. OpenACC cache directive: Opportunities and optimizations. In *2016 Third Workshop on Accelerator Programming Using Directives (WACCPD)*, 46–56. DOI: 10.1109/WACCPD.2016.009.

[103] Guodong Li, Peng Li, Geof Sawaya, Ganesh Gopalakrishnan, Indradeep Ghosh, and Sreeranga P. Rajan. 2012. GKLEE: Concolic verification and test generation for gpus. In *Proceedings of the 17th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming* (PPoPP '12). Association for Computing Machinery, New Orleans, Louisiana, USA, 215–224. ISBN: 9781450311601. DOI: 10.1145/2145816.2145844. https://doi.org/10.1145/2145816.2145844.

[104] Peng Li, Guodong Li, and Ganesh Gopalakrishnan. 2014. Practical symbolic race checking of GPU programs. In *SC '14: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, 179–190. DOI: 10.1109/SC.2014.20.

[105] Devashree Tripathy, AmirAli Abdolrashidi, Quan Fan, Daniel Wong, and Manoranjan Satpathy. 2021. LocalityGuru: A PTX analyzer for extracting thread block-level locality in GPGPUs. In *2021 IEEE International Conference on Networking, Architecture and Storage (NAS)*, 1–8. DOI: 10.1109/NAS51552.2021.9605411.

[106] Charitha Saumya, Kirshanthan Sundararajah, and Milind Kulkarni. 2022. DARM: Control-flow melding for SIMT thread divergence reduction. In *2022 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, 1–13. DOI: 10.1109/CGO53902.2022.9741285.

[107] Putt Sakdhnagool, Amit Sabne, and Rudolf Eigenmann. 2019. RegDem: increasing GPU performance via shared memory register spilling. (2019). DOI: 10.48550/ARXIV.1907.02894. https://arxiv.org/abs/1907.02894.

[108] Ali Bakhoda, George L. Yuan, Wilson W. L. Fung, Henry Wong, and Tor M. Aamodt. 2009. Analyzing CUDA workloads using a detailed GPU simulator. In *2009 IEEE International Symposium on Performance Analysis of Systems and Software*, 163–174. DOI: 10.1109/ISPASS.2009.4919648.

[109] Mahmoud Khairy, Zhesheng Shen, Tor M. Aamodt, and Timothy G. Rogers. 2020. Accel-Sim: An extensible simulation framework for validated GPU modeling. In *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*, 473–486. DOI: 10.1109/ISCA45697.2020.00047.

[110] Xiuxia Zhang, Guangming Tan, Shuangbai Xue, Jiajia Li, Keren Zhou, and Mingyu Chen. 2017. Understanding the GPU microarchitecture to achieve bare-metal performance tuning. In (PPoPP '17). Association for Computing Machinery, Austin, Texas, USA, 31–43. ISBN: 9781450344937. DOI: 10.1145/3018743.3018755. https://doi.org/10.1145/3018743.3018755.

[111] Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman. 2006. *Compilers: Principles, Techniques, and Tools (2nd Edition)*. Addison-Wesley Longman Publishing Co., Inc., USA. ISBN: 0321486811.

[112] Michael Joseph Wolfe, Carter Shanklin, and Leda Ortega. 1995. *High performance compilers for parallel computing*. Addison-Wesley Longman Publishing Co., Inc., USA. ISBN: 0805327304.

[113] Max Willsey, Chandrakana Nandi, Yisu Remy Wang, Oliver Flatt, Zachary Tatlock, and Pavel Panchekha. 2021. Egg: Fast and extensible equality saturation. *Proc. ACM Program. Lang.*, 5, POPL, (January 2021). DOI: 10.1145/3434304. https://doi.org/10.1145/3434304.

[114] Leonardo de Moura and Nikolaj Bjørner. 2007. Efficient e-matching for SMT solvers. In *Automated Deduction – CADE-21*. Frank Pfenning, (Ed.) Springer Berlin Heidelberg, Berlin, Heidelberg, 183–198. ISBN: 978-3-540-73595-3.

[115] 1997. *Static single-assignment form. Modern Compiler Implementation in ML*. Cambridge University Press, 427–467. DOI: 10.1017/CBO9780511811449.020.

[116] Calvin Montgomery, Jeffrey L. Overbey, and Xuechao Li. 2015. Autotuning OpenACC work distribution via direct search. In *Proceedings of the 2015 XSEDE Conference: Scientific Advancements Enabled by Enhanced Cyberinfrastructure* (XSEDE). Association for Computing Machinery, St. Louis, Missouri. ISBN: 9781450337205. DOI: 10.1145/2792745.2792783. https://doi.org/10.1145/2792745.2792783.

[117] Paul Havlak. 1994. Construction of thinned gated single-assignment form. In *Languages and Compilers for Parallel Computing*. Utpal Banerjee, David Gelernter, Alex Nicolau, and David Padua, (Eds.) Springer Berlin Heidelberg, Berlin, Heidelberg, 477–499. ISBN: 978-3-540-48308-3.

[118] John Forrest and Robin Lougee-Heimer. 2005. CBC user guide. (2005). DOI: 10.1287/educ.1053.0020. eprint: https://pubsonline.informs.org/doi/pdf/10.1287/educ.1053.0020. https://pubsonline.informs.org/doi/abs/10.1287/educ.1053.0020.

[119] Gene Wu, Joseph L. Greathouse, Alexander Lyashevsky, Nuwan Jayasena, and Derek Chiou. 2015. GPGPU performance and power estimation using machine learning. In *2015 IEEE 21st International Symposium on High Performance Computer Architecture (HPCA)*, 564–576. DOI: 10.1109/HPCA.2015.7056063.

[120] Nandita Vijaykumar, Gennady Pekhimenko, Adwait Jog, Abhishek Bhowmick, Rachata Ausavarungnirun, Chita Das, Mahmut Kandemir, Todd C. Mowry, and Onur Mutlu. 2015. A case for core-assisted bottleneck acceleration in GPUs: Enabling flexible data compression with assist warps. In *Proceedings of the 42nd Annual International Symposium on Computer Architecture* (ISCA '15). Association for Computing Machinery, Portland, Oregon, 41–53. ISBN: 9781450334020. DOI: 10.1145/2749469.2750399. https://doi.org/10.1145/2749469.2750399.

[121] Matthias Felleisen, Robert Bruce Findler, Matthew Flatt, Shriram Krishnamurthi, Eli Barzilay, Jay McCarthy, and Sam Tobin-Hochstadt. 2015. The Racket manifesto. In *1st Summit on Advances in Programming Languages (SNAPL 2015)*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik.

[122] Rengan Xu, Xiaonan Tian, Sunita Chandrasekaran, Yonghong Yan, and Barbara Chapman. 2014. NAS parallel benchmarks for GPGPUs using a directive-based programming model. In *International Workshop on Languages and Compilers for Parallel Computing*. Springer, 67–81. DOI: 10.1007/978-3-319-17473-0_5.

[123] Miko M. Stulajter, Ronald M. Caplan, and Jon A. Linker. 2022. Can Fortran's 'do concurrent' replace directives for accelerated computing? In *Accelerator Programming Using Directives*. Sridutt Bhalachandra, Christopher Daley, and Verónica Melesse Vergara, (Eds.) Springer International Publishing, Cham, 3–21. ISBN: 978-3-030-97759-7.

[124] Adam Paszke et al. 2019. PyTorch: An imperative style, high-performance deep learning library. In *Proceedings of the 33rd International Conference on Neural Information Processing Systems*. Curran Associates Inc., Red Hook, NY, USA.

[125] H. Carter Edwards and Daniel Sunderland. 2012. Kokkos array performance-portable manycore programming model. In *Proceedings of the 2012 International Workshop on Programming Models and Applications for Multicores and Manycores* (PMAM '12). Association for Computing Machinery, New Orleans, Louisiana, 1–10. ISBN: 978145031-2110. DOI: 10.1145/2141702.2141703. https://doi.org/10.1145/2141702.2141703.

[126] Valentin Clement, Sylvaine Ferrachat, Oliver Fuhrer, Xavier Lapillonne, Carlos E. Osuna, Robert Pincus, Jon Rood, and William Sawyer. 2018. The CLAW DSL: Abstractions for performance portable weather and climate models. In *Proceedings of the Platform for Advanced Scientific Computing Conference* (PASC '18). Association for Computing Machinery, Basel, Switzerland. ISBN: 9781450358910. DOI: 10.1145/3218176.3218226. https://doi.org/10.1145/3218176.3218226.

[127] Jacob Lambert, Seyong Lee, Jeffrey S. Vetter, and Allen D. Malony. 2020. CCAMP: An integrated translation and optimization framework for OpenACC and OpenMP. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis* (SC). IEEE Press, Atlanta, Georgia. ISBN: 9781728199986.

[128] Xiaonan Tian, Dounia Khaldi, Deepak Eachempati, Rengan Xu, and Barbara Chapman. 2016. Optimizing GPU register usage: Extensions to OpenACC and compiler optimizations. In (August 2016), 572–581. DOI: 10.1109/ICPP.2016.72.

[129] Yichen Yang, Phitchaya Phothilimthana, Yisu Wang, Max Willsey, Sudip Roy, and Jacques Pienaar. 2021. Equality saturation for tensor graph superoptimization. In *Proceedings of Machine Learning and Systems*. A. Smola, A. Dimakis, and I. Stoica, (Eds.) Volume 3, 255–268. https://proceedings.mlsys.org/paper_files/paper/2021/file/65ded5353c5ee48d0b7d48c591b8f430-Paper.pdf.

[130] Cheng Fu, Hanxian Huang, Bram Wasti, Chris Cummins, Riyadh Baghdadi, Kim Hazelwood, Yuandong Tian, Jishen Zhao, and Hugh Leather. 2023. Q-Gym: An equality saturation framework for dnn inference exploiting weight repetition. In *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques* (PACT '22). Association for Computing Machinery, Chicago, Illinois, 291–303. ISBN: 9781450398688. DOI: 10.1145/3559009.3569673. https://doi.org/10.1145/3559009.3569673.

[131] Yisu Remy Wang, Shana Hutchison, Jonathan Leang, Bill Howe, and Dan Suciu. 2020. SPORES: Sum-product optimization via relational equality saturation for large scale linear algebra. *Proc. VLDB Endow.*, 13, 12, (July 2020), 1919–1932. ISSN: 2150-8097. DOI: 10.14778/3407790.3407799. https://doi.org/10.14778/3407790.3407799.

[132] Gus Henry Smith, Andrew Liu, Steven Lyubomirsky, Scott Davidson, Joseph McMahan, Michael Taylor, Luis Ceze, and Zachary Tatlock. 2021. Pure tensor program rewriting via access patterns (representation pearl). In *Proceedings of the 5th ACM SIGPLAN International Symposium on Machine Programming* (MAPS 2021). Association for Computing Machinery, Virtual, Canada, 21–31. ISBN: 9781450384674. DOI: 10.1145/3460945.3464953. https://doi.org/10.1145/3460945.3464953.

[133] Shashi Gowda, Yingbo Ma, Alessandro Cheli, Maja Gwóźzdź, Viral B. Shah, Alan Edelman, and Christopher Rackauckas. 2022. High-performance symbolic-numerics via multiple dispatch. *ACM Commun. Comput. Algebra*, 55, 3, (January 2022), 92–96. ISSN: 1932-2240. DOI: 10.1145/3511528.3511535. https://doi.org/10.1145/3511528.3511535.

[134] Tal Ben-Nun, Johannes de Fine Licht, Alexandros N. Ziogas, Timo Schneider, and Torsten Hoefler. 2019. Stateful dataflow multigraphs: A data-centric model for performance portability on heterogeneous architectures. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis* (SC '19). Association for Computing Machinery, Denver, Colorado. ISBN: 9781450362290. DOI: 10.1145/3295500.3356173. https://doi.org/10.1145/3295500.3356173.

[135] Tal Ben-Nun, Berke Ates, Alexandru Calotoiu, and Torsten Hoefler. 2023. Bridging control-centric and data-centric optimization. In *Proceedings of the 21st ACM/IEEE International Symposium on Code Generation and Optimization* (CGO 2023), 173–185.

[136] Lorenzo Chelini, Andi Drebes, Oleksandr Zinenko, Albert Cohen, Nicolas Vasilache, Tobias Grosser, and Henk Corporaal. 2021. Progressive raising in multi-level ir. In *2021 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, 15–26. DOI: 10.1109/CGO51591.2021.9370332.

[137] Philip Ginsbach, Toomas Remmelg, Michel Steuwer, Bruno Bodin, Christophe Dubach, and Michael F. P. O'Boyle. 2018. Automatic matching of legacy code to heterogeneous APIs: An idiomatic approach. *SIGPLAN Not.*, 53, 2, (March 2018), 139–153. ISSN: 0362-1340. DOI: 10.1145/3296957.3173182. https://doi.org/10.1145/3296957.3173182.

[138] Sunghyun Park, Salar Latifi, Yongjun Park, Armand Behroozi, Byungsoo Jeon, and Scott Mahlke. 2022. SRTuner: Effective compiler optimization customization by exposing synergistic relations. In *Proceedings of the 20th IEEE/ACM International Symposium on Code Generation and Optimization* (CGO '22). IEEE Press, Virtual Event, Republic of Korea, 118–130. ISBN: 9781665405843. DOI: 10.1109/CGO53902.2022.9741263. https://doi.org/10.1109/CGO53902.2022.9741263.

[139] Mircea Trofin, Yundi Qian, Eugene Brevdo, Zinan Lin, Krzysztof Choromanski, and David Li. 2021. MLGO: A machine learning guided compiler optimizations framework. (2021). arXiv: 2101.04808 [cs.PL].

[140] Ameer Haj-Ali, Nesreen K. Ahmed, Ted Willke, Yakun Sophia Shao, Krste Asanovic, and Ion Stoica. 2020. NeuroVectorizer: End-to-end vectorization with deep reinforcement learning. In *Proceedings of the 18th ACM/IEEE International Symposium on Code Generation and Optimization* (CGO 2020). Association for Computing Machinery, San Diego, CA, USA, 242–255. ISBN: 9781450370479. DOI: 10.1145/3368826.3377928. https://doi.org/10.1145/3368826.3377928.

[141] Pieter Hijma, Stijn Heldens, Alessio Sclocco, Ben van Werkhoven, and Henri E. Bal. 2023. Optimization techniques for GPU programming. *ACM Comput. Surv.*, 55, 11, (March 2023). ISSN: 0360-0300. DOI: 10.1145/3570638. https://doi.org/10.1145/3570638.

[142] Changwan Hong, Aravind Sukumaran-Rajam, Jinsung Kim, Prashant Singh Rawat, Sriram Krishnamoorthy, Louis-Noël Pouchet, Fabrice Rastello, and P. Sadayappan. 2018. GPU code optimization using abstract kernel emulation and sensitivity analysis. *SIGPLAN Not.*, 53, 4, (June 2018), 736–751. ISSN: 0362-1340. DOI: 10.1145/3296979.3192397. https://doi.org/10.1145/3296979.3192397.