Addressing Software-Managed Cache Development Effort in GPGPUs

by

Ahmad Lashgar
B.Sc., Jundi Shapor University of Technology, 2010
M.Sc., University of Tehran, 2012

in the Electrical and Computer Engineering Department

Addressing Software-Managed Cache Development Effort in GPGPUs

by

Ahmad Lashgar
B.Sc., Jundi Shapor University of Technology, 2010
M.Sc., University of Tehran, 2012

Supervisory Committee

Dr. Amirali Baniasadi, Supervisor
(Electrical and Computer Engineering)

Dr. Mihai Sima, Departmental Member
(Electrical and Computer Engineering)

Dr. Alex Thomo, Outside Member
(Computer Science Department)

Supervisory Committee

Dr. Amirali Baniasadi, Supervisor
(Electrical and Computer Engineering)

Dr. Mihai Sima, Departmental Member
(Electrical and Computer Engineering)

Dr. Alex Thomo, Outside Member
(Computer Science Department)

ABSTRACT

GPU Computing promises very high performance per watt for highly-parallelizable workloads. Nowadays, there are various programming models developed to utilize the computational power of GPGPUs. Low-level programming models provide full control over GPU resources and allow programmers to achieve peak performance of the chip. In contrast, high-level programming models hide GPU-specific programming details and allow programmers to mainly express parallelism. Later, the compiler parses the parallelization notes and translates them to low-level programming models. This saves tremendous development effort and improves productivity, often achieved at the cost of sacrificing performance. In this dissertation, we investigate the limitations of high-level programming models in achieving a performance near to low-level models. Specifically, we study the performance and productivity gap between high-level OpenACC and low-level CUDA programming models and aim at reducing the performance gap, while maintaining the productivity advantages. We start this study by developing our in-house OpenACC compiler. Our compiler, called IPMACC, translates OpenACC for C to CUDA and uses the system compile to generate GPU binaries. We develop various micro-benchmarks to understand GPU structure and implement a more efficient OpenACC compiler. By using IPMACC, we evaluate the performance and productivity gap between a wide set of OpenACC

and CUDA kernels. From our findings, we conclude that one of the major reasons behind the big performance gap between OpenACC and CUDA is CUDAs flexibility in exploiting the GPU software-managed cache. Identifying this key benefit in low-level CUDA, we follow three effective paths in utilizing software-managed cache similar to CUDA, but at a lower development effort (e.g. using OpenACC instead). In the first path, we explore the possibility of employing existing OpenACC directives in utilizing software-managed cache. Specifically, the cache directive is devised in OpenACC API standard to allow the use of software-managed cache in GPUs. We introduce an efficient implementation of OpenACC cache directive that performs very close to CUDA. However, we show that the use of the cache directive is limited and the directive may not offer the full-functionality associated with the software-managed cache, as existing in CUDA. In the second path, we build on our observation on the limitations of the cache directive and propose a new OpenACC directive, called the fcw directive, to address the shortcomings of the cache directive, while maintaining OpenACC productivity advantages. We show that the fcw directive overcomes the cache directive limitations and narrows down the performance gap between CUDA and OpenACC significantly. In the third path, we propose fully-automated hardware/-software approach, called TELEPORT, for software-managed cache programming. On the software side, TELEPORT statically analyzes CUDA kernels and identifies opportunities in utilizing the software-managed cache. The required information is passed to the GPU via API calls. Based on this information, on the hardware side, TELEPORT prefetches the data to the software-managed cache at runtime. We show that TELEPORT can improve performance by 32% on average, while lowering the development effort by 2.5X, compared to hand-written CUDA equivalent.

PREFACE

This is the list of Ahmad Lashgar's publications at University of Victoria in chronological order:

[1] Ahmad Lashgar and Amirali Baniasadi, A Case Against Small Data Types on GPGPUs, The 25th IEEE International Conference on Application-specific Systems, Architectures and Processors (ASAP), IBM Research, Zurich, Switzerland, June 18-20, 2014.

[2] Ahmad Lashgar, Alireza Majidi, and Amirali Baniasadi, IPMACC: Translating OpenACC API to OpenCL, In poster session of the 3rd International Workshop on OpenCL (IWOCL), Stanford University, California, USA, May 11-13, 2015.

[3] Ahmad Lashgar, Ebad Salehi, and Amirali Baniasadi, Understanding Outstanding Memory Request Handling Resources in GPGPUs, In proceedings of The Sixth International Symposium on Highly Efficient Accelerators and Reconfigurable Technologies (HEART), Boston MA, USA, June 1-2, 2015.

[4] Ahmad Lashgar, Ebad Salehi, and Amirali Baniasadi, A Case Study in Reverse Engineering GPGPUs: Outstanding Memory Handling Resources, ACM SIGARCH Computer Architecture News - HEART '15, Volume 43 Issue 4.

[5] Ahmad Lashgar and Amirali Baniasadi, Rethinking Prefetching in GPGPUs: Exploiting Unique Opportunities, In proceedings of 17th IEEE International Conference on High Performance Computing and Communications (HPCC), New York, NY, USA, August 24-26, 2015.

[6] Ahmad Lashgar and Amirali Baniasadi, Employing Software-Managed Caches in OpenACC: Opportunities and Benefits, ACM Transactions on Modeling and Performance Evaluation of Computing Systems (ToMPECS), Volume 1 Issue 1, March 2016.

[7] Ahmad Lashgar and Amirali Baniasadi. OpenACC cache Directive: Opportunities and Optimizations, In proceedings of Third Workshop on Accelerator Programming Using Directives (WACCPD 2016), (in conjunction with SC 2016), Salt Lake City, Utah, USA, November 14, 2016.

[8] Ahmad Lashgar and Amirali Baniasadi, Efficient Implementation of OpenACC cache Directive on NVIDIA GPUs, To appear in the International Journal of High Performance Computing and Networking (IJHPCN), Special Issue on High-level Programming Approaches for Accelerators.

In [1], [5], [6], [7], and [8], Ahmad Lashgar conducted the research, analyzed the results, and prepared the draft of the manuscripts under the guidance of Dr. Amirali Baniasadi.

In [2], Ahmad Lashgar developed IPMACC framework, conducted the research, analyzed the results, and prepared the draft of the manuscript under the guidance of Dr. Amirali Baniasadi. Alireza Majidi collected the benchmarks.

In [3] and [4], Ahmad Lashgar conducted the research, developed the micro-benchmarks, collected the results, and prepared the draft of the manuscript under the guidance of Dr. Amirali Baniasadi. Ebad Salehi collaborated in analyzing the results.

Contents

List of Tables

List of Figures

ACKNOWLEDGEMENTS

First and foremost, I like to express my gratitude to my supervisor Dr. Amirali Baniasadi. His wisdom and vision paved the way through my years and his advices and supports maintained me on the track. Above all, Dr. Baniasadi is an exceptional human being and taught me very much.

I would also like to thank Dr. Nikitas Domopoulos. During our group meetings, he provided invaluable comments on this work. He also made a huge effort in making this dissertation stronger. Dr. Dimopoulos is an outstanding teacher and professional person and I learned very much from him.

I would also like to thank my supervisory committee members and external examiner: Dr. Mihai Sima, Dr. Alex Thomo, Dr. Brian Wyvill, and Dr. Xipeng Shen. Their valuable comments on this work improved the quality of this dissertation significantly.

I would also like to thank staff of Electrical and Computer Engineering Department: Moneca Bracken, Janice Closson, Kevin Jones, Brent Sirna, Amy Issel, and Ashleigh Burns. They are very supportive and friendly people and they were always there when I was reaching out to ask for help.

I would also like to thank Dr. Ehsan Atoofian from Lakehead University. I had an opportunity to collaborate with him while he was on sabbatical at UVic. I admire his dedication to work and I thank him for generously sharing his knowledge and experience with me.

I would also like to thank my dear friend Ali Shafiee from University of Utah for sharing his insight. He is a smart researcher and having his comments on this work was a privilege.

I would also like to thank my colleagues Parwant Ghuman and Mattew Gara at 3vGeomatics for providing me equipments and opportunity to collaborate on three industry-funded research projects.

I would also like to thank my family for their support and unconditional love. Although they have been physically away, they gently prepared a favourable environment for me to focus on my study. My mother provided a profound love and moral support. My father was the greatest inspiration and made me stronger every day by sharing his experiences. My brother took a very good care of my parents while I was away studying in Canada and encouraged me to focus on my research.

I would also like to thank my best friend, my partner, and the love of my life

Arghavan. She motivated me every day and encouraged me to never settle for less and made me believe I deserve more. Her endless love and tremendous support have been a substantial aid in my low moments.

I would also like to thank my colleagues at UVic for their support: Ali Jooya, Babak Keshavarz Hedayati, Saman Khoshbakht, Zhe Wei, Alexandros Dimopoulos, Mohammad Alkhamis, Mohammed Albulayli, Mostafa Rahimpour, Dr. Alireza Akhgar, and Dr. Behnam Rahimi. They shared their knowledge with me and provided valuable comments on my research projects.

I would also like to thank my colleague and friend Ebad Salehi. He is an amazing friend and we shared many happy moments.

I would also like to thank Tibor Szabo and Lorlina Palencia for their true friendship. In my early days in Victoria, they spend so much time and effort to allow me adapt and move to the new town gracefully. They are considerate, welcoming, supportive, and incredible people.

Finally, I am grateful to God for all the blessings that I was destined for.

After climbing a great hill, one only finds that there are many more hills to climb.
Nelson Mandela

DEDICATION

To my lovely mother and devoted father

Chapter 1

Introduction

1.1 Motivation

For several decades microprocessor performance growth relied mainly on optimizing performance of single CPU core by employing better designs (provided by architectural innovations) and faster and more transistors (provided by manufacturing technology innovations) on a chip. As the technology hit the thermal wall, where all transistors may not run at their maximum switching frequency, academia and industrial experts continue to seek alternative solutions. One major trend is to redesign software and hardware infrastructures to ideally run applications efficiently on many slow-cores platforms rather than a single fast core. Since all types of applications may not run efficiently on many slow-cores (primarily because the application may follow a serial algorithm), industry shifted toward designing heterogeneous systems to provide both fast and slow platforms. In an heterogeneous system, hardware accelerators come along the conventional CPUs to accelerate a portion of application. Designers of heterogeneous systems leveraged both software and hardware to maximize advantages from heterogeneous computing. On the software side, the programmer is required to identify code regions that map well on the target accelerator. Then she explicitly offloads these workloads from the CPU to run them on the accelerator. On the hardware side, the CPU controls the operations of the accelerator and acts as an interface (or host) to the accelerator. CPU and accelerator may or may not be on the same chip. They may also share the same physical memory or have separated memory spaces.

Among the most commonly used accelerators today are GPUs. GPUs were ini-

tially designed as a fixed-function processor but eventually evolved into a general-purpose parallel processor (often referred to as GPGPUs). There are three reasons why GPUs dominated other competitive accelerators (e.g. IBM Cell) at the time they emerged: computation capability, programmability, and affordability. Firstly, typical peak single precision FLOPs of GPUs were 10X larger than CPUs. Secondly, developers were able to program GPUs in C. Thirdly, there was no need to acquire an auxiliary hardware since almost every desktop computer had a GPU. Over the past ten years, many applications have been developed for GPUs, evolving GPUs into an efficient accelerator for both high-performance [93] and low-power [92] supercomputers.

Although GPUs promise very high performance and energy efficiency, delivering efficient implementation of an application comes at the cost of significant development effort in low-level GPU programming models like CUDA or OpenCL. The optimization space of GPU applications can become cumbersomely large in these low-level models, even for well-known problems like matrix multiplication [82]. OpenACC is a high-level programming model which is introduced to offer performance versus development effort tradeoff. The key goal of developing in OpenACC is to simplify the accelerator's programming model and rely on compiler innovations to optimize the code for the target accelerator at compile-time. Today OpenACC compilers are rapidly evolving to implement the latest OpenACC version, integrate more optimization passes for OpenACC kernels, and perform closer to hand-written CUDA equivalent. Our goal in this dissertation is to achieve a performance very close to CUDA, while developing applications in high-level programming models like OpenACC.

One of the key optimizations in CUDA is to use software-managed cache (or SMC in short). SMC can be exploited in various ways to improve the kernel's memory efficiency [59, 89, 95]. By using the software-managed cache, compared to hardware-managed cache, the programmer can assure the data will not be evicted by other cache requests. Also parallel threads can fetch the data tile collaboratively to improve memory-level parallelism. Typically, SMC accesses have 7.3X higher bandwidth [96] and 16.7X lower delay [95] compared to DRAM accesses. Moreover fetching the data from the cache is 32X more energy-efficient than DRAM [10]. SMC has very high impact on GPU performance and energy efficiency as it reduces the number of expensive off-chip data movements [85]. When strong temporal and spacial locality exists, exploiting SMC is critical to GPU performance (as the size of the hardware cache on the GPU is very small and insufficient to capture the localities.). The major

obstacle in exploiting SMC is the development effort. SMC is introduced as a separate memory space to the programmer. The programmer is required to explicitly fetch the data from global memory to the cache space, map addresses from global to cache space, and write dirty data back to global memory. Utilizing SMC in CUDA involves a major change in the code and the resulting code can be complicated to debug and verify.

OpenACC offers the *cache* directive to allow OpenACC applications to exploit a GPU's SMC with minimal development effort. To be able to investigate the effectiveness of the *cache* directive, we first developed our in-house OpenACC compiler framework, referred to as IPMACC [44]. IPMACC supports OpenACC version 1.0 and implements *kernels*, *data*, and *loop* directives. IPMACC translates OpenACC applications to CUDA source and uses NVIDIA *nvcc* to compile CUDA source and generate GPU binaries.

We used our in-house OpenACC framework to investigate the compiler aspect of implementing the *cache* directive [41, 42]. We studied various implementations and optimization opportunities. We started with presenting the lack of efficiency and effectiveness under a straightforward implementation. We showed the mapping of parallel loop iterations to CUDA threads can be configured to share the cache among several loop iterations. This, in respect, improves cache utilization and accelerator occupancy, yielding a significant speedup. We also designed microbenchmarks [45, 43, 38, 36] in CUDA to deeply understand GPU memory hierarchy and implement the *cache* directive efficiently. Applying various optimizations, we showed our implementation of the *cache* directive performs close to the hand-written CUDA version.

Although the *cache* directive is more productive than CUDA in exploiting SMC, it does not offer the full functionalities of SMC as exist in CUDA. SMC in CUDA is primarily used for i) caching read-only data for the lifetime of a thread block (preventing conflict misses), ii) avoiding irregular access to global memory (regular collective fetch from global memory, irregular private retrieve from SMC), and iii) inter-thread fast on-chip communication (rather than slow global synchronizations) [21]. The *cache* directive supports i and ii, but falls short in supporting the third use of SMC. We propose a new directive, referred to as the *fcw* directive [40], to offer the inter-thread communication functionality of SMC in OpenACC. This directive serves as a compiler hint to fetch a data chunk into SMC, replace global memory accesses with SMC accesses, allow concurrent accelerator threads to communicate through

SMC, and write the SMC back to global memory. We introduced a communication model along with the *fcw* directive that allows communication among iterations of a parallel work-sharing loop. We showed that the *fcw* directive can offer functionalities that are missing in the OpenACC *cache* directive. Compared to CUDA, we showed that the *fcw* directive saves significant development effort while delivers a performance close to the hand-written CUDA version.

To lower the SMC development effort even further, we introduced a novel hardware software mechanism, referred to as *TELEPORT*. TELEPORT offloads the SMC development effort from the programmer to the compiler, while not sacrificing performance. Under TELEPORT, the compiler analyzes CUDA kernels to statically identify the data tiles assigned to each thread block [39]. Later, and during runtime, hardware loads the designated tiles into SMC in advance for each thread block. When both TELEPORT and hand-written CUDA versions implement the same algorithm, TELEPORT not only delivers the same performance, but also supersedes CUDA versions via unique hardware optimizations in improving DRAM row locality. TELEPORT is limited by the compile-time limitations and may only be used for read-only data. This means TELEPORT does not fully replace SMC programming. However, when TELEPORT is applicable, it is a fully-automated pass and does not incur extra development effort. We investigated TELEPORT under a wide set of benchmarks and concluded that TELEPORT improves performance of handwritten implementations on average by 32% and yet lowers development effort by 2.5X. Our estimations show that the hardware overhead associated with TELEPORT is below 1%.

1.2 Contributions

The contributions of this dissertation are as follows.

- We introduced IPMACC open-source framework that translates OpenACC applications to CUDA and executes the OpenACC applications over CUDA-capable GPUs. We compared IPMACC to Omni OpenACC compiler and provided insight on implementation choices that impact performance. We used IPMACC and compared performance of OpenACC and CUDA implementations of ten different applications. We identified major limitations of OpenACC that impose a large performance gap between OpenACC and CUDA. This is

presented in Chapter 3.

- We developed micro-benchmarks in CUDA to stress outstanding global memory request handling resources in GPUs. Micro-benchmarks can be configured to generate different memory patterns, stressing various aspects of the resources. We ran our micro-benchmarks on two GPGPUs which have different micro-architectures: Fermi and Kepler. We showed that under Fermi architecture the maximum number of outstanding memory accesses is limited by the number of uncoalesced accesses. Under Kepler architecture the maximum number of outstanding memory accesses is limited by the number of memory instructions a warp can execute. This is presented in Section 4.1.

- We developed micro-benchmarks in CUDA to understand the performance of SMC in GPUs. Micro-benchmarks evaluate the performance impact of row-major or column-major accesses, layout, allocation padding, and data type size. We showed that the layout (2D or flattened) has minor impact on performance and small padding in memory allocation can vastly resolve bank conflicts. This is presented in Section .

- We presented the first work that investigates the implementation aspect of the OpenACC *cache* directive on NVIDIA GPUs. We showed that a naïve implementation hardly improves performance. We provided better understanding regarding implementation challenges and listed compile-time opportunities to enhance performance. We also proposed three methods for implementing the *cache* directive on NVIDIA GPUs. One of the implementations emulates hardware cache and the other two cache a range of values. Methods differ in cache utilization and access overhead. Investigating the design space of our proposal under three different benchmarks, we showed that our best implementation delivers performance comparable to that provided by the hand-written CUDA equivalent. This is presented in Chapter 5.

- We discussed the challenges in integrating SMC in OpenACC and limitations of the *cache* directive. To overcome the challenges and limitations, we proposed a new directive, referred to as the *fcw* directive. Along with the *fcw* directive, we also introduced a new communication model, referred to as inter-iteration communication. This allows loop iterations to communicate through the fast on-chip cache, instead of global memory. We proposed an efficient method for

implementing the *fcw* directive on NVIDIA GPUs. We presented an example usage of the *fcw* directive in a simple reduction case study and also evaluated the *fcw* directive under six different benchmarks. We compared *fcw* directive to the highly-optimized CUDA and baseline OpenACC versions. This is presented in Chapter 6.

- We proposed a hardware/software scheme, referred to as TELEPORT, to exploit SMC fully and automatically (without a hint from the programmer nor development effort). Static compiler passes are proposed to analyze CUDA kernels and extract potentials in using SMC. CUDA API calls are proposed to pass SMC hints to the hardware. Hardware prefetcher is proposed to preload SMC at runtime. We evaluated the performance of TELEPORT under five different benchmarks. We also reported advantages of TELEPORT in terms of development effort over CUDA. This is presented in Chapter 7

1.3 Dissertation Organization

The rest of this dissertation is organized as follows. In Chapter 2, we overview background information about GPGPU programming models and hardware design. In Chapter 3, we introduce our OpenACC framework, IPMACC. In Chapter 4, we introduce our GPU micro-benchmarking and present our findings on two different GPUs. In Chapter 5, we describe our methods for implementing the *cache* directive and evaluate these methods. In Chapter 6, we introduce the *fcw* OpenACC directive and investigate effectiveness of this directive. In Chapter 7, we introduce TELEPORT hardware/software mechanism and investigates its benefits and limitations. In Chapter 8, we overview related work. Finally, in Chapter 9 we offer concluding remarks and future work.

Chapter 2

Background

In this chapter, we overview the software and hardware of the GPGPUs. Figure 2.1 presents the hardware and software stack covering commonly-used accelerator-based computing technologies. High-level APIs for programming accelerators include OpenMP and OpenACC directive-based models. Conventional commercial and research compilers translate OpenMP and OpenACC to low-level APIs before generating accelerator binaries. Low-level APIs for programming accelerators include CUDA, OpenCL, and ISPC. CUDA and OpenCL programmers can inline assembly commands using PTX and SPIR, respectively. ISPC programs are translated to CPU vector extensions, namely AVX2 and SSE4. On the hardware, low-level APIs are compatible with different accelerators. While CUDA and PTX are specific to NVIDIA GPUs, OpenCL and SPIR are compatible with wide variety of GPUs, CPUs, co-processors, and FPGAs. CPU vector extensions are supported by most CPUs and Intel Xeon Phi co-processors. The scope of this work is limited to running OpenACC directive-based model over CUDA on NVIDIA GPUs. In the rest of this chapter, we first overview the OpenACC and CUDA programming interfaces and then we overview typical GPGPU micro-architecture. Finally, we overview software-managed cache (SMC) programming in GPUs.

2.1 Programming Interface

CUDA and OpenACC are commonly used for programming GPGPUs. CUDA introduces notations for developing compute kernels and launching the kernels on the GPU. Programming in CUDA is cumbersome because i) two versions of the code should be

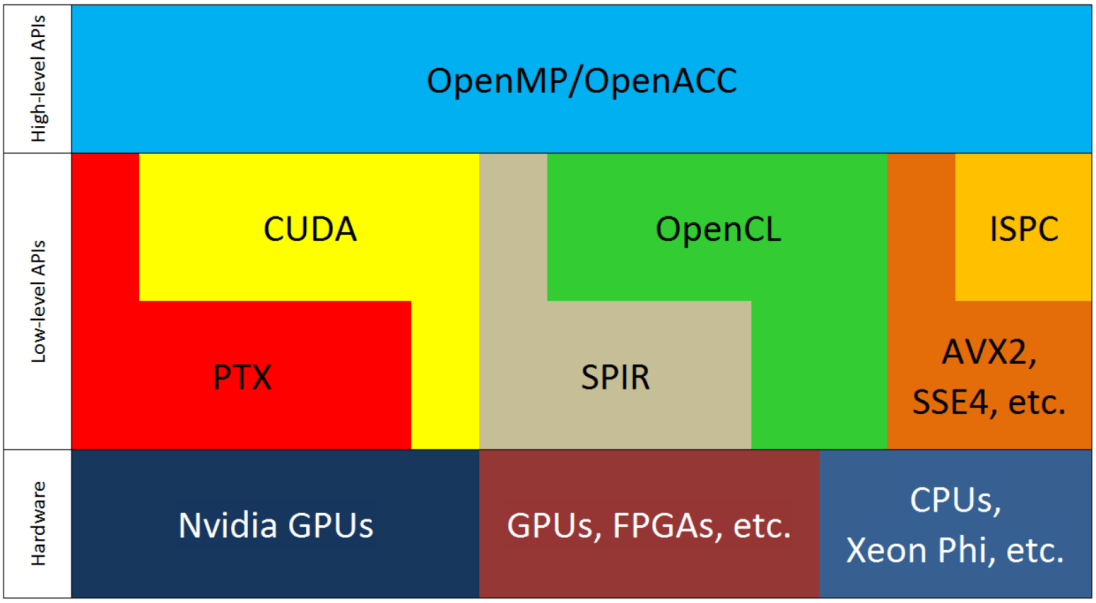


Figure 2.1: Hardware and software stack of accelerator-based computing.

maintained (CPU and GPU versions) and ii) various hardware details are exposed to the programmer (e.g. thread identifier and memory hierarchies). OpenACC is a standard high-level programming model [77] that hides low-level details and reduces the complexity of GPGPU (and generally accelerator) programming. Building on top of the serial CPU version, OpenACC allows the programmer to run the application on the GPU by adding few directives. To run OpenACC on GPUs, one common compilation flow is to translate OpenACC source code to CUDA source code and use the CUDA compiler to generate the GPU binaries. Below we overview CUDA and OpenACC programming models.

2.1.1 CUDA Model

In CUDA [64], an application is composed of host and device code. The host code executes on CPU and the device code executes on system's accelerator, e.g. GPU card. The host controls the operations of the device through procedure calls to CUDA API. CUDA allows programmers to explicitly allocate device memory and transfer data between host and device. The host code launches kernels on the device to harness the computational power. Kernel is executed by certain number of thread blocks where each thread block is composed of certain number of threads (referred to

as thread block size). All threads share common off-chip DRAM memory or global memory. Thread blocks may execute in any order and synchronization among thread block is not feasible. However, threads of the same thread block may synchronize and communicate through a fast on-chip software-managed cache, referred to as *shared memory*[1]. Shared memory is allocated per thread block and is much faster than global memory; e.g. under GTX 280, the latency of global memory and shared memory are 440 and 38 core cycles, respectively [100]. The number of threads per thread block and the number of thread blocks are specified at launch time and remain constant during the kernel execution.

2.1.2 OpenACC Model

OpenACC API introduces a set of compiler directives, library routines, and environment variables to offload a region of code from the CPU and execute it on the system's accelerator [75]. This region is referred to as the kernel or accelerator region.

In essence, OpenACC introduces two types of directives: i) data management and ii) parallelism control. Each directive has a few clauses providing fine-grain control over the behavior of the directive. Data management directives perform data allocation on the accelerator, data transfer between host and accelerator, and passing pointers to the accelerator. This model exposes data transfer to programmers, allowing manual data transfers. The *data* directive applies over the accelerator region, specifying the explicit data transfers. Data is copied to the accelerator before entering the region and copied back from the accelerator after exiting the region. The *data* directive clauses specify the direction of the transfer (host to device or vise versa), host memory pointer, and size of the transfer. Based on this information, OpenACC compiler generates code around the accelerator region to perform the necessary allocation/transfers. The second type of directives, parallelism control, hint the compiler that the iterations of a work-sharing loop may be executed in parallel on the accelerator. The directive might be followed by clauses to control the parallelism granularity, variable sharing or privatization, and variable reduction. OpenACC introduces four terms in loop parallelism: *gang*, *worker*, *vector*, and *thread*. In CUDA terminology, these terms may best map to *kernel*, *thread block*, *warp*, and *thread*, respectively.

[1]We used the terms software-managed cache and shared memory interchangeably.

Listing 2.1: OpenACC and CUDA matrix-matrix multiplications.

```
#pragma acc kernels copyin(a[0:LEN*LEN],b[0:LEN*LEN]) copyout(c[0:LEN*LEN])
#pragma acc loop independent
for(i=0; i<LEN; ++i) {
#pragma acc loop independent
 for(j=0; j<LEN; ++j){
  float sum=0;
  for(l=0; l<LEN; ++l)
   sum += a[i*LEN+l]*b[l*LEN+j];
  c[i*LEN+j]=sum;
 }
}
```

(a) OpenACC.

```
__global__ void matrixMul(int *a, int *b, int *c, int len){
 int i=threadIdx.x+blockIdx.x*blockDim.x;
 int j=threadIdx.y+blockIdx.y*blockDim.y;
 for(int l=0; l<len; ++l)
  sum=a[i*len+l]*b[l*len+j];
 c[i*len+j]=sum;
}
int main(){
 ...
 bytes=LEN*LEN*sizeof(int);
 cudaMalloc(&a_d, bytes); cudaMalloc(&b_d, bytes); cudaMalloc(&c_d, bytes);
 cudaMemcpy(a_d, a, bytes, cudaMemcpyHostToDevice);
 cudaMemcpy(b_d, b, bytes, cudaMemcpyHostToDevice);
 dim3 gridSize(LEN/16,LEN/16), blockSize(16,16);
 matrixMul<<<gridSize,blockSize>>>(a_d,b_d,c_d,LEN);
 cudaMemcpy(c, c_d, bytes, cudaMemcpyDeviceToHost);
 ...
}
```

(b) CUDA.

2.1.3 Matrix-Matrix Multiplication Example

Listing 2.1a and 2.1b illustrate a simple matrix-matrix multiplication in OpenACC and CUDA, respectively. Ignoring the directive lines, Listing 1a shows the baseline serial multiplication of a and b, storing the result in c. Each matrix is *LEN*LEN* in size. The outer loops iterated by i and j induction variables can be performed in parallel.

Listing 2.1a shows how these loops can be parallelized using OpenACC. In this code, the *kernels* directive marks a region intended to be executed on the accelerator. The *loop* directive guides the compiler to consider the loop as a parallel work-sharing loop. Programmers can control the parallelism using kernels and loop directives. As an example of parallelism control, the *independent* clause is used to force the compiler to parallelize the loop. This clause overrides the compiler's auto-vectorization and loop dependency checking. In Listing 2.1a, *copyin* and *copyout* clauses ask the compiler to copy a and b arrays from the host to the accelerator, before the region, and copy out c array from the accelerator to the host, after the region. For each array, the [*start*:n] pair indicates that n elements should be copied from the *start* element of the array. (Notice that the standard does not restrict programmers to unidimensional arrays and the matrices are flattened in this sample to perform faster memory copies.)

Listing 2.1b shows how the parallelization can be exploited in CUDA. __global__ indicates the declaration of kernel code. Parallel threads execute the kernel and operate on different matrix elements, based on their unique indexes (i and j). Inside the host code, device memory is allocated for a, b, and c, keeping the value of the pointers in a_d, b_d, and c_d, respectively. Then, input matrices are copied into device memory. Then, total of *LEN*LEN* light-weight accelerator threads are launched on the device to execute matrixMul kernel. After kernel completion, the resulting matrix c_d is copied back to the host memory.

As presented in Listing 2.1, OpenACC significantly reduces the development effort compared to CUDA. OpenACC hides low-level accelerator-related code from the programmer and provides a unified view over both host and accelerator code.

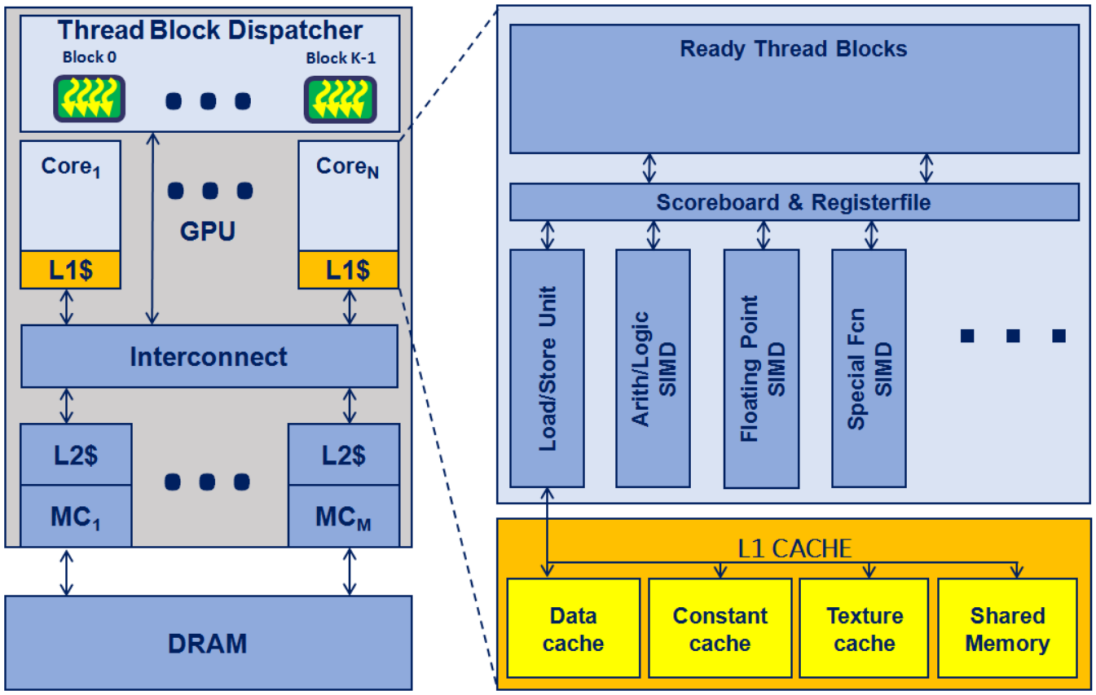


Figure 2.2: Typical GPGPU micro-architecture.

2.2 GPGPU Micro-architecture

For the micro-architecture side of our study, we assume a GPGPU similar to NVIDIA CUDA-capable GPGPUs. In such GPGPU, the chip is composed of one or more GPU cores (also referred to as Streaming Multiprocessor or SM) connected to the off-chip DRAM through memory controllers. This is presented in Figure 2.2. GPU chip is composed of thread block dispatcher, cores, and memory controllers connected through on-chip interconnection network. Thread block dispatcher interfaces to the graphic driver, manages concurrent kernels, and issues tasks (in thread block granularity) to cores. Cores have private fast L1 cache for different memory spaces (data, constant, and texture), shared among concurrent thread blocks of the core. Shared memory is a software-managed cache and is private to each thread block. L1 caches are backed up by the last-level L2 cache. L2 cache is unified (meaning L2 may cache data, constant, and texture memory spaces) and shared among all the cores. L2 cache is divided into several partitions to maximize memory-level parallelism. Each L2 cache partition is logically associated with a memory controller. Memory controllers interface to (and send/receive data to/from) the off-chip DRAM.

Each GPU core is a deep-multithreaded SIMD processor maintaining the context of thousands of threads. Core has different SIMD engines for performing arithmetic, logical, floating-point, and special function operations. SIMD width varies for different operations, ranging from 4-wide to 64-wide in current GPUs. Also multiple SIMD engines of the same kind may be deployed on the core to provide higher throughput for specific operations. GPU core has immense capability in handling concurrent memory requests, suggesting a large memory-level parallelism on the chip. Depending on the GPGPU micro-architecture, a GPU core may support 128 to 1408 concurrent memory requests [45].

Each thread block is executed by one GPU core. The context of the thread block is reserves on the GPU core and is not released until after the thread block completes its execution. GPU core groups threads into coarser scheduling elements called *warp*. Each warp is composed of 32 threads. Threads within a warp are executed in lock-step over the SIMD of the core and share the same control-flow [19]. The context of the thread block includes registers, warps, and shared memory. GPU core may run additional thread blocks as long as the context can be reserved on the GPU core. The GPU core runs under maximum number of threads, or 100% occupancy, if concurrent thread blocks are not limited by the register or shared memory usage. As an example of GPU capabilities, NVIDIA Tesla K20 [72] is composed of 13 GPU cores, where each core supports 64 concurrent warps, 64K registers, and 48 KB of shared memory.

2.3 Software-managed Cache

Conventionally, GPUs have had a small cache per core to buffer input/output of the graphics pipeline [18]. This buffer is critical to the performance of the graphics processor as it bypasses significant amount of global synchronization and DRAM accesses. Later, in the GPU computing era [52], GPGPU programming models introduced a new memory hierarchy, called shared memory in CUDA, to allow programs to take advantage of this buffer. The new memory hierarchy is a software-managed cache (the same hardware component that conventionally is used as buffer in graphics pipeline) and can be shared among collaborating threads (known as thread blocks). This cache can be exploited in various ways to improve kernel's memory efficiency [59, 89, 95]. Typically, software-managed cache accesses have 7.3X higher bandwidth [96] and 16.7X lower delay [95] than DRAM accesses and fetching the data from the cache is 32X more energy-efficient than DRAM [10].

We list three reasons that GPGPU applications might benefit from software-managed cache (SMC):

- **Locality:** SMC can be beneficial when there is a strong spacial locality among concurrent threads, e.g. memory accesses of threads of the thread block fall within a tile of data. In this case, the entire range can be fetched into SMC in advance. This resolves cold cache misses and also maintains the data in SMC for the life time of the thread block (protecting the tile from cache eviction due to cache capacity.).

- **Irregular memory pattern:** Memory pattern of the threads of a warp impacts the memory latency significantly. For example, under regular memory pattern (that threads of the warp access subsequent words), memory accesses are coalesced and one memory transaction is made. However, under irregular memory patterns (that threads of the warp access arbitrary words), load/store unit may stall and serializes the memory accesses in several memory transactions. If the range of irregular memory accesses is known by the programmer, SMC can be used to address this inefficiency. In this case, the memory range is fetched into the SMC first and then memory accesses are mapped to SMC (from the original global memory space).

- **Local communication:** SMC can be used as a communication channel among concurrent threads of the thread block. This communication can be performed very fast (order of tens of cycles). Without using SMC, threads are forced to use global memory for communication which is very slow (order of hundreds of cycles).

Listing 2.2 shows an example of employing software-managed cache in CUDA. This example fetches a range of data from global memory (*a[]*) to software-managed cache (*swcache[]*). Software-managed cache is allocated on Line #4. The space ($4 \times 256 bytes$) is allocated once and then shared among all threads of the thread block (256 threads in this example). Every write to this space will be visible to all threads of the thread block. On Line #5, *a[gid]* is written to index *tid* of the software-managed cache. Since *tid* ranges from 0 to 255 (as executed by all threads of the thread block), *swcache* will be initialized to a subarray from *a[]* in parallel. However, threads run in parallel and may not complete the write at the same time. On Line #6, *__syncthreads()* is used to synchronize the threads and make sure all threads

Listing 2.2: Software-managed cache example in CUDA.

```
01: __global__ void kernel(int *a, int len){
02:  int tid = threadIdx.x;
03:  int gid = threadIdx.x + blockIdx.x*blockDim.x;
04:  __shared__ int swcache[256];
05:  swcache[tid] = a[gid];
06:  __syncthreads();
07:  ...
08: }
09: int main(){
10:  ...
11:  dim3 gridSize(LEN/256), blockSize(256);
12:  kernel<<<gridSize,blockSize>>>(a_d, LEN);
13:  ...
14: }
```

have completed their write operation (notice that *__syncthreads()* only synchronizes the threads of the same thread block, not all threads of the kernel.). Beyond this point, data can be retrieved from software-managed cache explicitly by loads and stores from the software and is shared among all threads of the thread block.

2.4 Terminology

We use CUDA terminology [64] and define the following terms and use them frequently in the remainder of this dissertation. *Parallel work-sharing loop* or simply *parallel loop* refers to a loop which is marked by OpenACC API to be executed on the accelerator. Every parallel loop has a certain number of iterations which are executed in parallel on the accelerator. *Parallel iterations* refer to the iterations of a parallel loop. We assume that each parallel iteration is mapped to one light-weight accelerator thread of CUDA. Therefore, we use the terms parallel iteration and thread interchangeably. We refer to consecutive iterations of a parallel loop as *neighbor iterations* or *consequent iterations*. Accordingly, in the sequence of parallel iterations, every parallel iteration has neighbor iterations. The definition of neighbor iterations discards the size of neighborhood and can be of any range. For example, if a parallel loop iterates from 1 to N by one step, parallel iterations indexed by 8, 9, 10, 11, and 12 are neighbor iterations. As another example, N-3, N-2, N-1, and N are also neighbor iterations.

Chapter 3

IPMACC

In this chapter, we introduce our in-house open-source OpenACC framework, called IPMACC. We developed IPMACC to compile OpenACC for C [77] applications for CUDA-capable accelerators[1]. IPMACC comes with a set of translators to generate the CUDA code which is equivalent to the OpenACC code. After translation to CUDA, IPMACC uses the system compiler to generate the accelerator binary from the CUDA code. Beside the translators, IPMACC also includes a runtime library to support dynamic memory management in OpenACC API.

Below we first overview the structure of IPMACC. Then we explain system and software configurations for the evaluations, followed by the experimental results.

3.1 Framework

IPMACC is a research framework composed of a set of translators translating OpenACC applications to various accelerator languages (e.g. OpenCL or CUDA). In addition to the translators, IPMACC comes with a runtime library to support dynamic memory management operations in OpenACC API. Compared to similar frameworks [81, 90], IPMACC is designed to translate OpenACC directly to a low-level accelerator programming model (e.g. CUDA) and make the source code readable and available to the programmer. In addition, IPMACC is designed to be extensible and allow translation of OpenACC to various programming models. We overview frameworks that are similar to IPMACC in Section 8.

[1] In this chapter, in interest of space, we limit the discussion to CUDA. Very similar discussion is also applicable to OpenCL.

Generating the low-level source code has two advantages. Firstly, this allows taking advantage of the latest innovations in the target compilers for executing OpenACC applications. Secondly, the programmers can have an equivalent version of their serial code on accelerators by simply augmenting the code with OpenACC notation. Later, experienced OpenCL or CUDA programmers can perform further optimizations on top of that, avoiding development from scratch and saving huge amount of development effort. To this end, we did our best to minimize abstraction and generate direct target source code. Currently, IPMACC can translate OpenACC application to two different backends: OpenCL or CUDA. Both translators and runtime library of IPMACC are developed flexible enough to allow easy inclusion of more backends (e.g. ISPC [79]).

Structure. IPMACC framework has a command-line interface for compiling an OpenACC application and generating the destination binary. Compilation starts with validating the OpenACC syntax. Then, the OpenACC kernels and data regions are extracted from the code and these regions are translated to proper target (OpenCL or CUDA). Then, several static passes parse the code to find dimensions of the parallel loops, type and size of the data, user-defined types, user-defined procedure calls, etc. After gathering these information, IPMACC generates the target source code. Finally, the target source code is passed to the system compiler (g++ if the target is OpenCL or nvcc if the target is CUDA) to generate the final object code. The command-line tool accepts all compilation flags that the system compiler understands. Hence, the command-line tool can be used for generating intermediate object codes or final binaries.

Features. IPMACC supports most of OpenACC procedure calls and directives. Currently, all procedure calls except synchronizations are supported. IPMACC supports kernels, loop, data, enter, exit, and cache directives. parallel, device selection, and synchronization clauses are yet to be implemented. IPMACC supports the use of user-defined types and user-defined procedure calls in the kernels region. Nested loops are supported and parallel iterations of each loop nest is mapped to a unique dimension of the CUDA thread block (or OpenCL work-group). IPMACC is an open-source framework and the code is available on github [37].

3.2 Methodology

Benchmarks. We use benchmarks from NVIDIDA GPU Computing SDK [67] and

Rodinia Benchmark Suite [13]. NVIDIA GPU Computing SDK includes a large set of CUDA and OpenCL test cases, each implementing a massively-parallel body of an application in CUDA and OpenCL efficiently. Most test cases also include a serial C/C++ implementation. We developed the OpenACC version of these benchmarks over the serial C/C++ code. Rodinia is a GPGPU benchmark suite composed of a wide set of workloads implemented in C/C++. Originally, each of these benchmarks were implemented in CUDA and OpenCL parallel models. Recently, a third-party [78] added the OpenACC implementation of the benchmarks. We include N-Body simulation from the SDK and the remaining benchmarks from Rodinia.

OpenACC Compilers. We use our in-house framework, IPMACC, for compiling OpenACC applications. The framework and benchmarking suite can be obtained from github [37]. We validated the correctness of our framework by comparing the results of OpenACC benchmarks against the serial version. For the last part of evaluations, we compare performance of IPMACC to Omni OpenACC compiler [90]. Omni compiler executes OpenACC applications over CUDA runtime.

Performance evaluations. We compile the OpenACC version of the benchmarks by our framework and run it over CUDA runtime. We compare these to CUDA implementations available in NVIDIA GPU Computing SDK and Rodinia. In order to evaluate performance, we report the kernel execution, kernel launch, and memory transfer times. We use *nvprof* for measuring these times in CUDA [14]. For kernel execution and memory transfers time, we report the time that *nvprof* reports after kernels/transfers completion. For kernel launch time, we report the time measured by *nvprof* in calling *cudaLaunch*, *cudaSetupArgument*, and *cudaConfigureCall* API procedures. Every reported number is the harmonic mean of 30 independent runs. We use harmonic mean to filter outliers (extremely big values) that appear in measurements. These big numbers may appear if the system is undesirably busy with an unexpected system process/task while we are running a sample. For each kernel, runtime difference among independent runs are very insignificant and we found that 30 samples are large enough to capture the common values.

Platforms. We perform the evaluations under a CUDA-capable accelerator; NVIDIA Tesla K20c. This system uses NVIDIA CUDA 6.0 [67] as the CUDA implementation backend. The other specifications of the system are: CPU: Intel® Xeon® CPU E5-2620, RAM: 16 GB, and operating system: Scientific Linux release 6.5 (Carbon) x86_64. We use GNU GCC 4.4.7 for compiling C files.

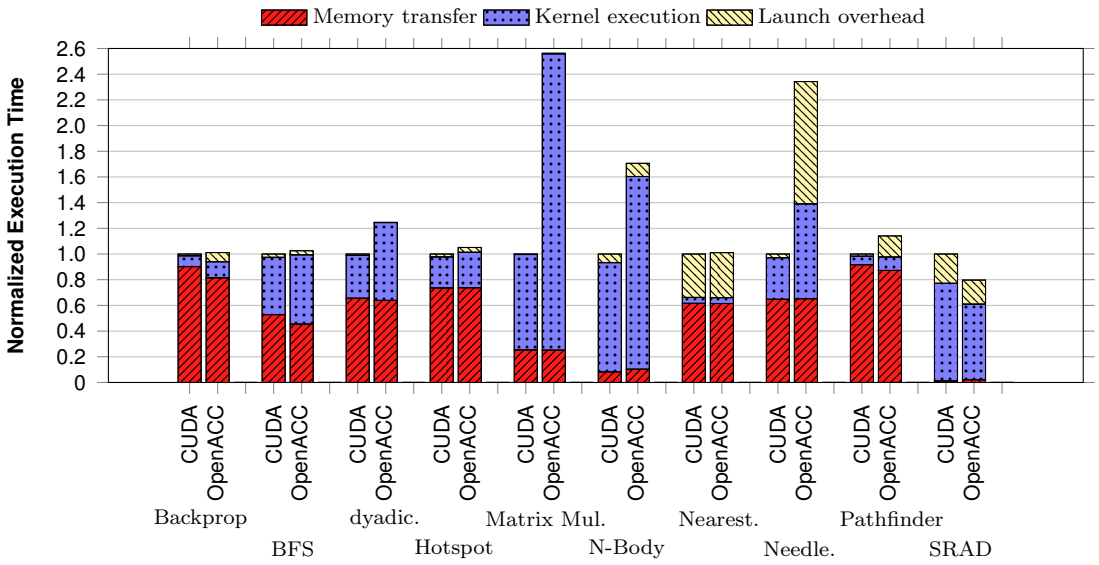


Figure 3.1: Comparing the execution time of OpenACC to highly-optimized CUDA implementations. Each bar shows the duration of time that the application spends on memory transfer, kernel execution, and kernel launch overhead.

3.3 Experimental Results

In this section, we evaluate performance of IPMACC under various aspects. Firstly, we compare a set of OpenACC applications to their highly optimized CUDA version. Our goal is to identify OpenACC's programming limitations resulting in the performance gap between OpenACC and CUDA. We show that CUDA optimizations in using software-managed cache is the main reason causing a huge gap between CUDA and OpenACC. Secondly, we compare the execution time of various OpenACC operations under OpenCL and CUDA backends of IPMACC. Specifically, we report the timing overhead of copyin, copyout, and reduction operations. Finally, we compare performance of IPMACC to a previous open-source compiler, Omni OpenACC compiler.

3.3.1 Standard Benchmarks

Figure 3.1 reports the execution time for OpenACC applications, compared to their CUDA version. The figure reports the breakdown of time spent on the accelerator; kernel launch (launch), kernel execution (kernel), or memory transfer between host and accelerator (memory). Kernel launch time includes the time spent on setting kernel arguments and launching the kernel on the accelerator.

In most cases, CUDA's kernel launch/execution portion is shorter than OpenACC. Also, memory transfer times are comparable on both CUDA and OpenACC. There are exceptions where OpenACC memory transfers are faster (e.g. Backprop) or kernel time of CUDA and OpenACC are equal (e.g. Nearest.). We investigate the differences between CUDA and OpenACC in the following sections.

Below we discuss applications separately providing insight into why CUDA and OpenACC implementations presented in Figure 3.1 have different kernel launch, kernel execution, and memory transfer times.

Back Propagation. Back Propagation (Backprop) is a machine-learning algorithm used to train the weights in a three-layer neural network. In both OpenACC and CUDA versions, there are six back-to-back serial operations where the output of each stage is fed to the immediate next stage as input. Each stage can be performed in parallel on the accelerator. OpenACC and CUDA versions offload the first and last stages to GPU.

OpenACC implementation performs faster memory transfers and slower kernel launch/execution, compared to CUDA. This is explained by the difference between CUDA and OpenACC in implementing the first stage, which is similar to reduction. OpenACC launches multiple kernels to reduce all variables on the accelerator. CUDA, however, performs a two-level reduction; first level on the GPU and the second level on the CPU. This explains why CUDA has lower kernel execution/launch and higher memory transfer time.

BFS. BFS visits all the nodes in the graph and computes the visiting cost of each node. Each node is visited only once. Parallel threads of a kernel visit the nodes belonging to the same graph depth concurrently and the algorithm traverses through the depth iteratively. The operation stops once there is no child to visit.

Compared to the CUDA version, the OpenACC version of BFS spends less time on memory transfers. This can be explained by the fact that the OpenACC version performs data initializations on the GPU. However, the CUDA version initializes the inputs on the host and transfers the inputs to GPU. Compared to the CUDA version, OpenACC spends more time on kernel execution, since it forces a debilitating reduction on a global variable. The global variable is a boolean indicating whether there remained more nodes to visit or not. CUDA avoids global reduction by initializing the variable to FALSE on the host and imposing a control-flow divergent in the kernel to guard the global variable from FALSE writes (allowing TRUE writes only).

Dyadic Convolution. Dyadic Convolution (dyadic.) is an algebra operation

calculating the XOR-convolution of two sequences. The OpenACC implementation parallelizes output calculations, where each thread calculates one output element. Although this implementation is fast to develop, it exhibits a high number of irregular memory accesses. To mitigate irregular memory accesses, the CUDA version uses Fast Walsch-Hadamard Transformation (FWHT) for implementing dyadic convolution (as described in [5]).

As reported in Figure 1, both OpenACC and CUDA versions spend almost the same amount of time on memory transfers. While the CUDA version launches several kernels, OpenACC launches only one kernel. This explains why the CUDA version imposes higher kernel launch overhead. In CUDA the kernels' execution time is 82% faster than OpenACC. This is due to the fact that the CUDA version uses FWHT to mitigate irregular memory accesses. Although OpenACC can implement dyadic convolution using FWHT, the same FWHT algorithm used in CUDA cannot be implemented in OpenACC. CUDA FWHT uses shared memory to share intermediate writes locally between neighbor threads, which is not possible under OpenACC standard.

Hotspot. Hotspot simulates chip characteristics to model the temperature of individual units. At every iteration, the algorithm reads the temperature and power consumption of each unit and calculates new temperatures. Although both OpenACC and CUDA spend the same amount of time on memory transfers, CUDA kernel is faster.

In Hotspot, the temperature of each unit depends on its power consumption and neighbors' temperatures. CUDA kernel exploits this behavior to localize the communication and reduce global memory accesses as follows. In CUDA, threads of the same thread block calculate the temperature of neighbor units. The CUDA version locally updates the new temperature of neighbor units using the threads of the same thread block. This local communication reduces the number of kernel launches used to synchronize the temperature across all thread blocks, explaining why the CUDA version performs faster kernel launches and comes with shorter execution time. In OpenACC, unlike CUDA, the software-managed cache cannot be exploited for local communication. Hence, in OpenACC there are higher number of global synchronizations and kernel launches, which in turn harm performance.

Matrix Multiplication. Matrix Multiplication (Matrix Mul.) performs multiplication of two 1024 by 1024 matrices. Both CUDA and OpenACC implementations use output parallelization, calculating each element of the output matrix in parallel.

CUDA version is different from OpenACC as it processes input matrices tile-by-tile. By processing in tiles, CUDA version fetches the input tiles in few well-coalesced accesses into software-managed cache and shares the tiles among the threads of the same thread block.

While kernel launch and memory transfer times are nearly the same across CUDA and OpenACC, CUDA kernel time is much lower than OpenACC. CUDA version takes advantage of software-managed cache in two ways. First, CUDA version merges the required data of the thread block and fetches them once, minimizing redundant memory accesses across thread of the same thread block. Second, software-managed cache removes cache conflict misses, since the replacement policy is controlled by the programmer. Under OpenACC, although the threads have very high spatial locality, parsing the matrix row-by-row at a time highly pollutes the cache, returning high number of conflict misses. Also having multiple thread blocks per SM exacerbates this effect.

N-Body simulation. N-Body models a system of particles under the influence of gravity force. In each timestep, operations of $O(N^2)$ complexity are performed (for a system of N particles) to calculate forces between all pairs of particles. Inherently, there are many redundant memory reads, since the mass and position information of each particle is fetched by other particles N-1 times to calculate its interaction with other particles.

While both CUDA and OpenACC memory transfers take about the same time, CUDA kernels are much faster. The CUDA version tiles the computations to reduce redundant memory reads [73]. CUDA exploits shared memory to share the particles among all threads of a thread block. In OpenACC, however, the redundant memory accesses are not filtered out by the software-managed cache. As reported, redundant memory accesses can degrade performance significantly.

Nearest Neighbor. Nearest Neighbor (Nearest.) finds the five closest points to a target position. The Euclidean distance between the target position and each of the points is calculated and the top five points with the lowest distance are returned. OpenACC and CUDA versions both calculate Euclidean distances for each point in parallel. OpenACC and CUDA versions spend about the same time on kernel launch, kernel execution, and memory transfer. This is explained by the similarity of parallelization methods applied in both OpenACC and CUDA.

Needleman-Wunsch. Needleman-Wunsch (Needle.) is a sequence alignment algorithm used in bioinformatics. In either CUDA or OpenACC, traverses a 2D

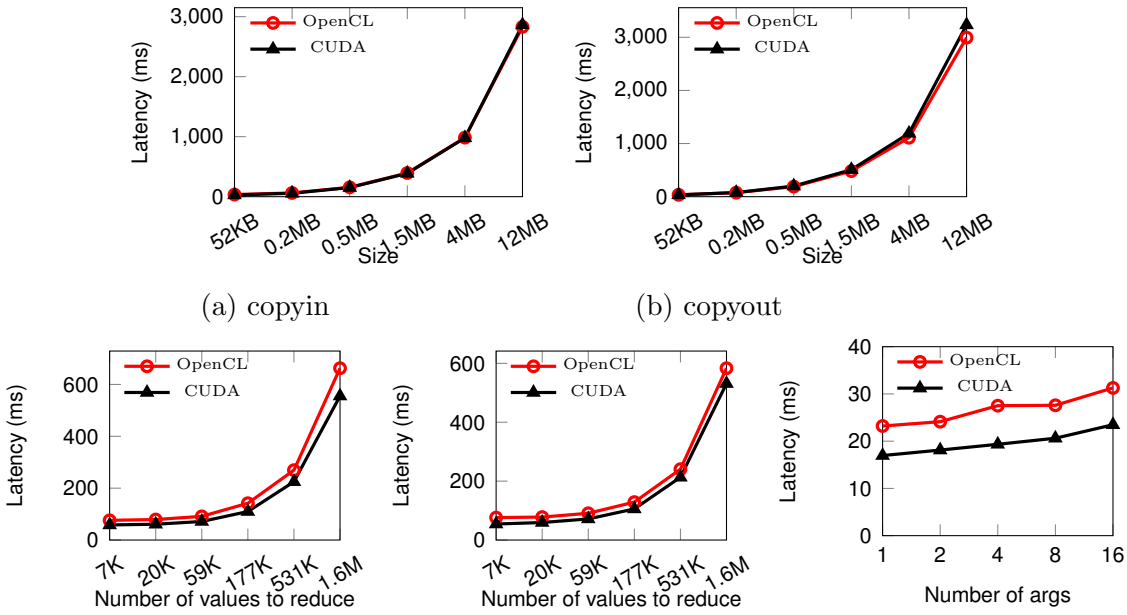
matrix and updates the costs. Upon updating a new cost, four memory locations are read and one location is written.

Although both CUDA and OpenACC versions spend the same amount of time on memory transfers, CUDA kernel launch/executions are much faster than OpenACC kernels. The CUDA version fetches a data chunk of *costs* matrix into shared memory and traverses the matrix at the shared memory bandwidth. This mechanism comes with three advantages: i) filtering redundant global memory accesses by shared memory, ii) minimizing global communication by sharing intermediate results stored in the shared memory, iii) reducing the number of kernel launches and global communications. The fewer number of kernel launches explains why the launch time of CUDA is much less than OpenACC.

Pathfinder. In Pathfinder (Pathfin.) kernel, every working element iteratively finds the minimum of three consequent elements in an array. The CUDA version of Pathfinder performs two optimizations: i) finding the minimum by accessing the data from shared memory, and ii) sharing the updated minimum locally among neighbor threads for certain iterations and then reflecting the changes globally to other threads. Such local communications reduce the number of global synchronizations and kernel launches.

However, OpenACC's API is not flexible enough to allow the programmer exploit the shared memory in a similar way. Therefore neighbor threads in the OpenACC version do not communicate via shared memory. Therefore, each thread fetches the same data multiple times and threads communicate only through global memory. Communication through global memory is implemented through consequent kernel launches. This explains why OpenACC imposes higher kernel launch overhead.

Speckle reducing anisotropic diffusion. Speckle reducing anisotropic diffusion (SRAD) is an image processing benchmark performing noise reduction through partial differential equations iteratively. Compared to CUDA, the kernel time of OpenACC version is lower. Three code blocks construct the computation iterative body of this benchmark: one reduction region and two data parallel computations. Our evaluation shows OpenACC version performs 5% slower than CUDA, upon executing two data parallel computations. However, OpenACC outperforms CUDA in executing the reduction portion. This is explained by the difference in reduction implementations. Our OpenACC framework performs the reduction in two levels: reducing along threads of thread block on GPU and reducing along thread block on CPU. In the CUDA version, however, reduction is performed by multiple serial kernel



(a) copyin (b) copyout

(c) reduction max (d) reduction (+) (e) Kernel launch

Figure 3.2: Comparing the latency of CUDA and OpenCL backends for IPMACC under various OpenACC operations: (a) copyin, (b) copyout, (c) reduction (max), (d) reduction (+), (e) kernel launch.

launches, all on the GPU. The OpenACC version spends less time on executing the kernel as part of the computation is carried on host. Meanwhile, performing two levels of reduction imposes the overhead of copying intermediate data from GPU to CPU. This explains why the OpenACC version spends slightly more time on memory transfers and less time on kernel launch/execution.

3.3.2 OpenACC Benchmarking

Figure 3.2 compares performance of IPMACC backends (OpenCL and CUDA) under various OpenACC operations. These operations include copying data from host to accelerator (*copyin*), copying data from accelerator to host (*copyout*), reducing writes from parallel threads by maximum (*reduction (max)*) and sum (*reduction (+)*) operators, and kernel launch overhead. To perform this experiment, we measure the time for completing one of these operations (e.g. data directive with copyin clause or kernels loop directive with reduction clause). The directive is called within a sequential loop which iterates for 30 times. We report the harmonic mean of these 30 iterations. The OpenACC benchmarking suite that we use here is included in the

IPMACC package [37].

copyin and **copyout**. OpenCL and CUDA backend perform similar under copyin clause. For copyout clause, OpenCL performs slightly faster under larger data sizes.

reduction (max) and **reduction (+)**. IPMACC implements a two-level reduction algorithm; first, reducing the values within the thread block on the accelerator, then, reducing the results of all thread blocks on the host [26]. Under both OpenCL and CUDA, reduction time starts to grow remarkably after 59K values. Reduction (max) performs slower than reduction (+) under both OpenCL and CUDA, since finding the maximum causes branch divergence. Comparing OpenCL and CUDA backends, CUDA backend performs slightly faster under both types of reduction.

kernel launch. We measure the time taking to launch and execute a kernel with different number of arguments; ranging from one to 16 arguments. To make sure that the compiler is not optimizing the code by removing the kernel arguments, within the kernels region we sum the value of all arguments and write the result back to global memory. Under both OpenCL and CUDA, increasing the number of arguments increases the measured latency. Part of this increase comes from the launch and portion of the increase comes from the increase in the number of sum operations in the kernel. It is not possible to separate these two by OpenACC control. Comparing OpenCL and CUDA, OpenCL kernels perform 4% to 8% slower than CUDA.

3.3.3 Compiler Performance

In this Section, we compare our framework work to Omni OpenACC compiler [90]. We compare the performance of the code generated by Omni and IPMACC under three basic workloads; matrix-matrix multiplication, vector-vector add, and reduction. Our goal is to compare loop-to-thread mapping, overhead of generated code, and efficiency of reduction. Since Omni is only able to run the application over CUDA, here we limit the comparison to CUDA back end of IPMACC.

Matrix-matrix Multiplication. This benchmark consists of three nested loops. The most inner loop is a reduction. Two outer loops can be executed independently in parallel. In this comparison, we parallelize the two outer loops. Figure 3.3 compares the performance of this implementation compiled by IPMACC or Omni. As reported, under small problem size, IPMACC performs faster and under larger problem sizes, Omni performs faster. We explain this by the mapping of the two nested loops to CUDA threads and thread blocks. IPMACC maps each loop nest along one dimension

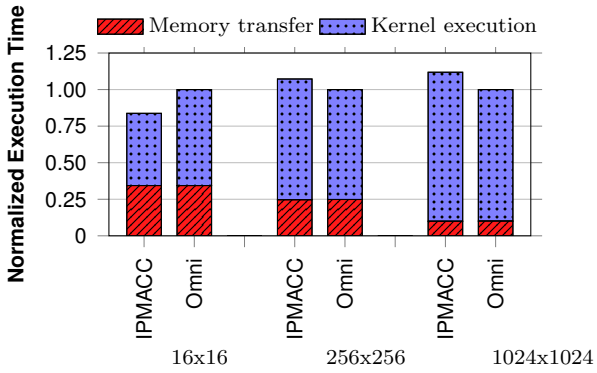


Figure 3.3: Comparing the performance of IPMACC and Omni under matrix-matrix multiplication workload. Each bar shows the duration of time that the application spends on memory transfer and kernel execution. Each bar group reports for particular problem size.

of the grid. This means IPMACC launches as parallel threads as the number of parallel loop iterations. Omni, however, maps all loops to one single-dimensional grid. Omni maps the first loop across the thread blocks and the second loop across the threads of the thread blocks. Since the number of threads per thread block is limited by an small number (e.g. 1024), Omni also limits the number of parallel threads which are launched for executing the second loop. Accordingly, more than one loop iteration (or task) is mapped to each thread. Below we discuss when it is advantageous and when it is not.

Omni's mapping reduces the number of threads and thread blocks. This is advantageous over IPMACC's mapping when there are large number of parallel iterations, as we see under 256 x 256 and 1024 x 1024 problem sizes in Figure 3.3. This is because Omni's mapping reduces the overhead of scheduling large number of threads and thread blocks. On the other hand, assigning more than one task to threads and thread blocks comes with an overhead. This overhead includes control statements for the two loops, assigning more than one job (or one loop iteration) to threads and thread blocks. As we see under 16 x 16 problem size in Figure 3.3, this overhead causes Omni to stay behind IPMACC.

We also found that for different problem sizes, Omni always generates a loop to assign more than one job to each thread block. In our experiments, however, this loop only assigns one job to each thread. Hence, the generation of this loop essentially imposes a control flow overhead and reduces the performance. To clarify and show this overhead, we change the matrix-matrix multiplication implementation and merge

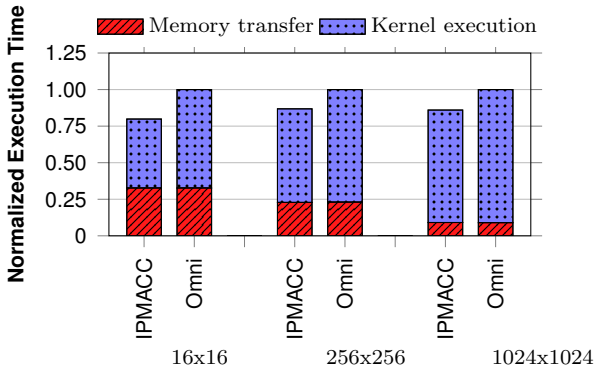


Figure 3.4: Comparing the performance of IPMACC and Omni under a matrix-matrix multiplication where two outer loops are merged and flattened. Each bar shows the duration of time that the application spends on memory transfer and kernel execution. Each bar group reports for particular problem size.

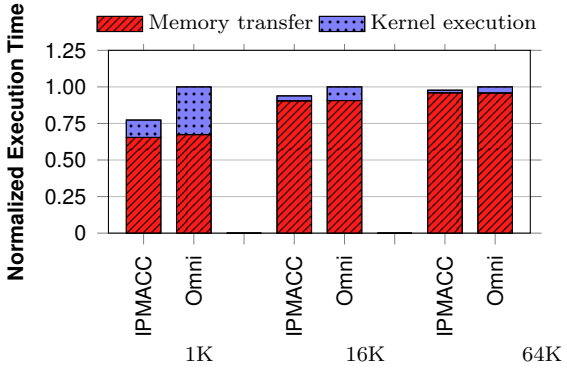


Figure 3.5: Comparing the performance of IPMACC and Omni under vector-vector addition. Each bar shows the duration of time that the application spends on memory transfer and kernel execution. Each bar group reports for particular problem size.

and flatten the two outer loops into single unified loop. Figure 3.4 reports the numbers under the latter implementation. In this case, both compilers generate a code with the same mapping of loops to threads. The difference is an extra loop that Omni generates to assign more than one task to each thread (while always assigning one task to each thread). As reported, in this implementation, IPMACC performs faster than Omni.

Vector-vector Addition. This benchmark adds two input arrays, element by element, and stores the output in another array. Each individual sum can be performed in parallel. Figure 3.5 compares the performance of vector add under IPMACC and Omni. As reported, IPMACC performs faster than Omni. We explain this by the

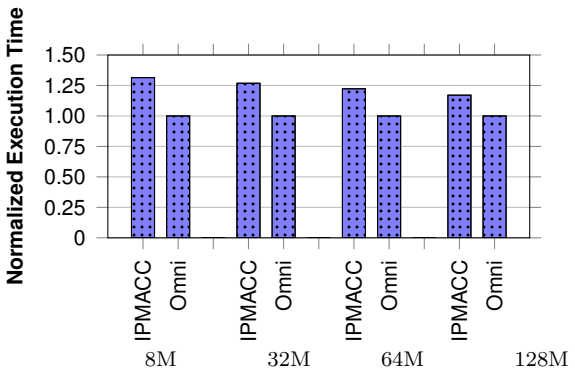


Figure 3.6: Comparing the performance of IPMACC and Omni under reduction clause. Each bar shows the duration of time that the application spends to complete whole reduction. Each bar group reports for particular problem size.

code generated by the two compilers. IPMACC generates a control statement to control number of active threads. Omni, however, generates a loop to control the number of loop iterations assigned to each thread (assigning more than one task to each thread). The generation of this loop is an extra computation on GPU since the runtime profiling indicates Omni launches enough number of threads and assigns one task to each thread.

Reduction. Figure 3.6 compares the performance OpenACC *reduction* clause under IPMACC and Omni. Since our implementation reduces the results of thread blocks on the host, we measure the time taking to complete the whole kernels region (before kernels region starts and after it completes). As reported, Omni implementation of reduction is much faster than IPMACC.

Chapter 4

Micro-benchmarking

In this chapter, we present several micro-benchmarks to improve the understanding of memory hierarchy of the GPUs. Specifically, we microbenchmark global memory and software-managed cache. Our goal is to use the findings here in IPMACC for efficient translation of OpenACC to CUDA.

4.1 Outstanding Memory Request Handling Resources

In this section we use micro-benchmarks to analyze GPGPU capabilities in managing outstanding memory requests. Firstly, we give an overview of two alternative architectural designs for handling outstanding memory requests in GPGPUs (Section 4.1.1). One design is based on the Miss Status/Information Holding Register (MSHR) structure, initially proposed for CPUs [34]. The other design, named Pending Request Table (PRT) is used specifically by many-thread GPUs [74]. Secondly, we propose a micro-benchmark to understand the structure of outstanding memory request handling resources in real GPUs (Section 4.1.2). Finally, we evaluate our micro-benchmark on two real GPUs to understand the design details (Section 4.1.4).

4.1.1 Known Architecture

In this study we focus on NVIDIA Tesla M2070 and Tesla K20 GPGPUs. Tesla M2070 and Tesla K20 are based on Fermi [99] and Kepler architectures [71], respectively. Kepler, the recent architecture, differs from Fermi in various aspects; namely the

number of Streaming Multiprocessors (SM)[1], warps per SM, and SIMD engines per SM. In both architectures, there are 32 threads in a warp. Threads of the warp execute one common instruction at a time, referred to as *warp instruction*. Threads of the warp are executed in lock-step SIMD-like style and are synchronized at every instruction.

Both architectures have two levels of cache hierarchy for data. We assume that the cache hierarchy is not blocking and can handle multiple outstanding memory requests concurrently. This is achieved mainly by using *outstanding memory request handling resources*. Since the L2 cache is backing multiple L1 caches from different SMs, we find it reasonable to assume that the L2 cache is richer than the L1 in terms of outstanding memory request handling resources. Hence, during micro-benchmarking a single SM, the outstanding memory request handling resources at the L1 cache side will be the bottleneck as these resources saturate before the saturation of outstanding memory request handling resources at the L2 cache side.

We assume two possible designs for the organization of outstanding memory request handling resources at the SM or L1 cache side. Below we overview these two designs.

Miss Status Handling Registers

The Miss Status/Information Holding Registers (MSHR) structure is assumed in GPGPU-sim v3.2.2 [7] for modeling outstanding memory request handling resources. In this design, each SM has a certain number of MSHRs. Each MSHR tracks one pending memory request, fetching a memory block equal to the cache line size.

Upon executing a memory instruction for a warp, the Load/Store unit coalesces memory accesses of the threads into memory blocks. For each coalesced access, one MSHR entry is booked and one memory request is issued. Each MSHR entry can also be shared among a certain number of threads, avoiding redundant MSHR allocations for the same address. The number of threads/warps that can share a common MSHR varies from GPU to GPU, impacting the capability of a GPU in dealing with outstanding memory requests. Also the address offset (within the requested memory block) is stored for each of the threads merged in the same MSHR entry.

Once a reply is received from the lower memory level (L2 cache in this case), the requested data and identifier of MSHR are extracted from the reply. For the thread

[1]SM is a GPU core in NVIDIA terminology.

identifiers marked in the corresponding MSHR entry, the Load/Store unit writes the fetched data into the private registers of the threads. The Load/Store unit releases the MSHR entry after completing this process. An operation example is illustrated in [36].

In this design, the number of outstanding memory requests depends on the number of available MSHRs. Accordingly, the SM may stall a warp once there is no free MSHR available. Since each MSHR has limited capability in merging redundant memory requests, it is important that the programmer develops programs that perform well-coalesced accesses to avoid such stalls.

Pending Request Table

The Pending Request Table (PRT) structure is introduced in [74]. PRT has a certain number of entries and each entry tracks one memory instruction per warp (memory instructions include load and store instructions from DRAM.)

Upon executing a memory instruction for a warp, only one PRT entry is reserved (if there is a cache miss). Before booking information in this entry, the Load/Store unit coalesces memory accesses of the threads into memory blocks. For each coalesced access, the Load/Store unit creates one memory request to fetch the data from the lower memory level. In the memory request packet, the Load/Store unit stores the address of the requested block, the identifier of assigned PRT entry, and the mask of threads needing this block. Then, the Load/Store unit sends this memory request to the lower memory level. In the PRT entry, the Load/Store unit stores the number of pending memory requests. It also stores the memory address offset (within the requested memory block) for each thread.

Once a reply is received from the lower memory level, the requested data, the identifier of PRT entry, the address of requested memory block, and the mask of threads are extracted from the reply. Based on the offsets recorded in the corresponding PRT entry, the Load/Store unit serves the memory instruction and stores the fetched data into private registers of the threads marked in the mask. The PRT entry is released upon completion of all memory requests. An example is illustrated in [74].

The number of outstanding memory requests is independent of the number of uncoalesced memory accesses. Since the entire memory accesses of a warp (including coalesced and uncoalesced accesses) are stored in just one PRT entry, the number of outstanding memory instructions is limited by the number of load/store instructions

executed by warps. Essentially, the number of PRT entries specifies the number of warp memory instructions that can be executed concurrently.

4.1.2 Micro-benchmarking Mechanism

We developed a set of micro-benchmarks to measure the maximum number of outstanding memory requests that every SM can handle. We report the information gathered by our micro-benchmark in the form of *Thread-Latency plots*. Thread-Latency plots report the latency of completing certain number of memory requests. Calculating the variance of latencies in this plot, we detect occasional saturations in outstanding memory request handling resources. *Saturation in outstanding memory request handling resources appears as a spike in the variance curve and indicates the maximum number of outstanding memory requests that an SM can handle.* We evaluate various memory access patterns to estimate the capability of an SM in handling pending memory requests. Below we explain our micro-benchmark, Thread-Latency plot, how we derive information from this plot, and how L2 cache might impact our micro-benchmarking. We also list the memory patterns that we use.

Micro-benchmark Skeleton

For generating Thread-Latency plots, we use the source CUDA code skeleton shown in Listing 4.1. The code measures the latency associated with completing a certain number of global memory requests. Within the kernel, we use *clock()* procedure call to measure the latency. The code varies two parameters to control the number of memory requests: i) the number of threads and ii) the number of loads per thread. These two parameters are controlled by *nThreads* and N in the code, respectively. We sweep *nThreads* from 2 to 1024 (maximum thread block size), incrementing steps of 2 threads at a time. We report a separate Thread-Latency plot for each N, swept from 1 to 4. We limit the number of loads per thread below the scoreboard limit. We avoid the configurations that exceed the capability of scoreboard since the performance in these cases is limited by the scoreboard, not the outstanding memory request handling resources [60].

During evaluations, the code launches exactly one thread block (line 21), assuring all threads only stress memory handling resources of a single SM. Within the kernel in the code lines 7 to 14, we measure the latency of completing N memory requests. The addresses of these memory requests are calculated in the lines between 3 to

Listing 4.1: micro-benchmarking code.

```
1  void __global__
2      kernel(int *in, int *out, unsigned long int *time) {
3   unsigned long int addr1=(unsigned long int)in+UNQ(1,gtid);
4    ...
5   unsigned long int addrN=(unsigned long int)in+UNQ(N,gtid);
6
7   __syncthreads();
8   t1=clock();
9   int reg1, ..., regN;
10  asm("ld.global.s32 %0, [%1+0];" : "=r"(reg1) : "l"(addr1) );
11   ...
12  asm("ld.global.s32 %0, [%1+0];" : "=r"(regN) : "l"(addrN) );
13  __syncthreads();
14  t2=clock();
15
16  out[gtid]=reg1+...+regN;
17  time[gtid]=t2-t1;
18 }
19 int main()
20 {
21  kernel<<<1, nThreads>>>(in, out, time);
22 }
```

5. We change the address calculation to simulate different memory patterns. In this example, address calculation assures every memory address points to a unique 128-byte region (indicated by *UNQ* macro). We separate unique addresses by 128 bytes since the cache line size of micro-benchmarked GPUs are 128 bytes [94]. This addressing pattern forces the GPU to issue a unique memory request for each of the addresses. After measuring the time, the time is written back to global memory at line 17. The code writebacks the accumulation of loaded values at line 16, imposing instruction dependencies to prevent compiler optimizations from removing memory instructions. The micro-benchmarking code can be obtained from [38].

Methodology of Deducing Information from Thread-Latency Plot

We present the Thread-Latency plot for reporting the latency measured by Listing 4.1. We derive information from this plot and estimate the maximum number of pending memory requests. Each Thread-Latency plot reports the latency under certain fixed

i) memory pattern and ii) number of loads per threads. The horizontal axis of this plot represents different number of threads (increasing the number of threads means increasing the number of memory instructions). Two vertical axises report the *latency* and *variance of latency*. We report *unbiased sample variance of latency* to make latency variations clearly visible. For every point in Thread-Latency plot, the variance curve reports the variance among the point and the two neighbors (right and left). For every point, the variance is calculated using the following equation:

$$Var(Lat_{thd}) = \frac{\sum_{i=thd-1}^{thd+1} (Lat_i - \mu)^2}{2}, \mu = \frac{\sum_{i=thd-1}^{thd+1} Lat_i}{3}$$

Along the positive direction of the horizontal axis, where memory requests increase, the *variance curve* highlights sudden changes in the *latency curve*, indicating the saturation of memory handling resources in the hardware.

Figure 4.1 shows the Thread-Latency plot for one load per thread, under the memory pattern where each thread sends a unique memory request. In each Thread-Latency plot, two factors can explain the spikes in the variance curve. First, rapid change of latency is caused by serialization in coalescing memory accesses of threads, issuing non-coalesced memory requests, and writing the data back after completion of memory requests. Second, latency variation is caused by the saturation of pending memory requests resources. The second latency variation is larger. Also we observe serialization latency rapid variations appear before the saturation of pending requests resources.

Memory issue and writeback serialization. In Thread-Latency plot, a consistent increase can be seen in the latency when increasing the number of threads from 1 to 32, . This can be explained by the serialization in i) coalescing and issuing memory requests and/or ii) writing the fetched data back to register file for each thread of the warp. We refer to these two serializations as *memory issue serialization* and *writeback serialization*, respectively. As the number of threads strides beyond one warp (higher than 33), the latency does not change significantly. We can infer that memory issue and writeback of different warps can be processed in parallel (this can be achieved by parallel load/store units in an SM). We conclude from Figure 4.1 that up to 96 memory requests can be processed in parallel, as an increase in the latency is observed just after 96 threads.

Pending memory requests. Upon saturation of pending memory request re-

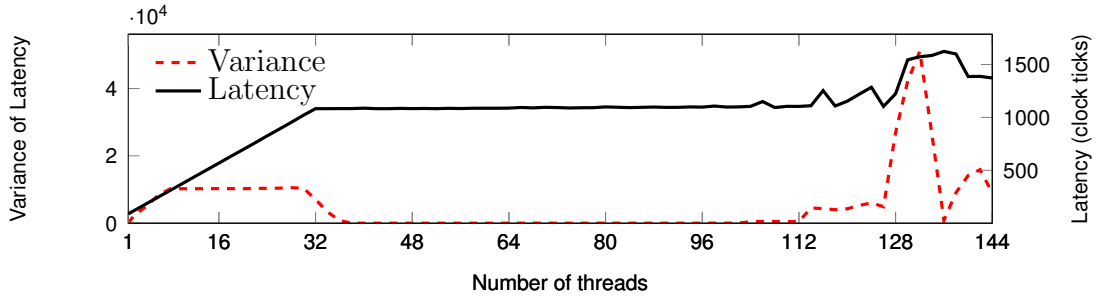


Figure 4.1: Thread-Latency plot under one load per thread and every thread requests one unique 128-byte block.

sources, the memory unit stops issuing further requests and waits for completion of prior memory requests. This can cause a significant spike in the variation curve of Thread-Latency plot. For instance in Figure 4.1 and under one load per thread plot, we can see a significant change in the latency after 128 threads (which equals to 128 memory requests in this plot).

As reported in the variance plot of Figure 4.1, moving from left to right, the first and smaller spike corresponds to memory issue and writeback serialization and the second and larger spike corresponds to the saturation in pending memory requests resources. The second spike can be seen clearly on 128 for one load per thread in Figure 4.1. We conclude from this figure that the maximum number of outstanding memory requests possible to accommodate is 128.

L2 Cache

In NVIDIA Fermi and Kepler GPUs all memory transfers from host to device pass through the L2 cache. This means that once memory transfers are completed, kernels may access the data at the latency of L2 cache (not DRAM). As we rely on studying variations in memory latency, understanding the L2 cache is critical.

To understand the L2 cache, we wrote a kernel which simply copies a chunk of data from one part of the global memory to another. Each thread is responsible for copying a single 8-byte data element. Accordingly, threads fetch the data into the GPU cores and then write it back to another address. We measure the time of completing the kernel under different data sizes. We expected to see L2 cache hits so long the overall data size stays below the L2 cache capacity. In Figure 4.2, the plots labelled as M2070 and K20 show the result of this micro-benchmarking under Tesla M2070 (Fermi) and K20 (Kepler), respectively. To signify the performance impact

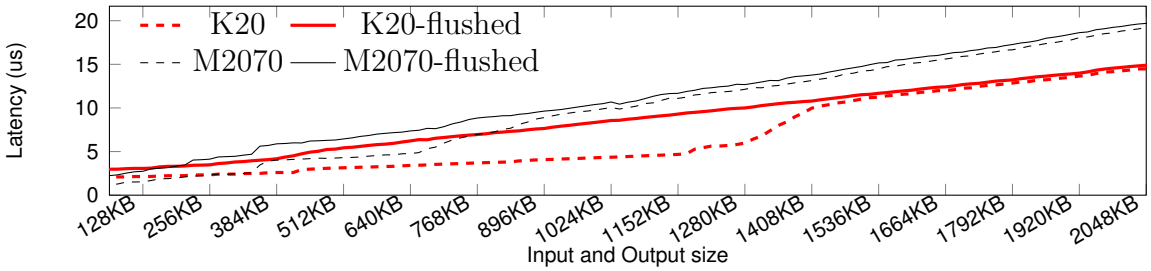


Figure 4.2: Micro-benchmarking L2 cache under Tesla M2070 and K20. Comparing flushed to non-flushed plots clearly shows the saturation of L2 cache after certain data size.

of using the L2 cache, we compare this latency to cases where L2 cache is flushed. We flush the L2 cache by copying a large dummy data (76 MB) from host to device before kernel launch. This essentially clears the L2 cache before the kernel launch that follows. In Figure 4.2, the plots labelled M2070-flushed and K20-flushed represent this. Comparing the flushed and non-flushed plots, we conclude that the performance gap saturates beyond a certain data size. This suggests the saturation of L2 cache after that point. Under M2070 and K20, this occurs nearly at 768 KB and 1536 KB, which are the L2 cache sizes for these GPUs.

We use our findings and make sure that the input data size is below L2 cache size. Consequently the spikes observed our a result of the saturation of outstanding memory accessing resources and not the L2 cache.

Memory Patterns

To understand the parameters of MSHR and PRT tables, we issue memory requests with different patterns. We use the following memory patterns:

- **All-unique:** every individual thread of the warp sends a load request for a unique address (total of 32 unique requests per warp),
- **Two-coalesced:** every two neighbor threads of the warp send a load request for the same address (total of 16 unique requests per warp),
- **Four-coalesced:** every four neighbor threads of the warp send a load request for the same address (total of eight unique requests per warp),
- **Eight-coalesced:** every eight neighbor threads of the warp send a load request for the same address (total of four unique requests per warp),

- **16-coalesced:** every 16 neighbor threads of the warp send a load request for the same address (total of two unique requests per warp),
- **32-coalesced:** every 32 threads of the warp send a load request for the same address (total of one unique request per warp),

These patterns can be implemented in Listing 4.1 simply by modifying the *UNQ* procedure.

4.1.3 Experiment Methodology

We run our evaluations on two systems with different GPGPUs; NVIDIA Tesla M2070 and NVIDIA Tesla K20. We measure the time using *clock()* procedure call of CUDA. Every reported point is a harmonic mean of three different runs. We use CUDA Toolkit 5.5 for compiling the benchmarks.

Our micro-benchmarking shows that every warp can issue a limited number of independent instructions concurrently. This limitation is imposed by the scoreboard structure used in each SM [60]. In the evaluated micro-benchmark, we found that Tesla M2070 can issue up to four concurrent instructions per warp. We also found that Tesla K20 can issue up to three concurrent instructions per warp.

4.1.4 Results

In this section, we use the micro-benchmark presented in Section 4.1.2 to estimate the capability of an SM in managing pending memory requests. We reconfigure and run the code under various configurations; varying memory patterns, number of threads, and number of load instructions per thread. In this section, we follow a two-step approach to understand outstanding memory request handling resources of Tesla M2070 and Tesla K20. At the first step, we specify whether the micro-architecture follows an MSHR-like design (refer to Section 4.1.1) or a PRT-like design (refer to Section 4.1.1). The micro-architecture may have a design similar to MSHR table, if the spikes in Thread-Latency plot of GPU depend on the i) number of concurrent threads and ii) number of loads per thread. The micro-architecture may have a design similar to PRT, if the spikes in Thread-Latency plot of GPU depend on the number of warp instructions. In either case, at the second step, we tune the micro-benchmarks to measure design specifics. For MSHR, we look for the number of MSHRs and mergers fields per MSHR entry. For PRT, we look for the number of PRT entries per SM.

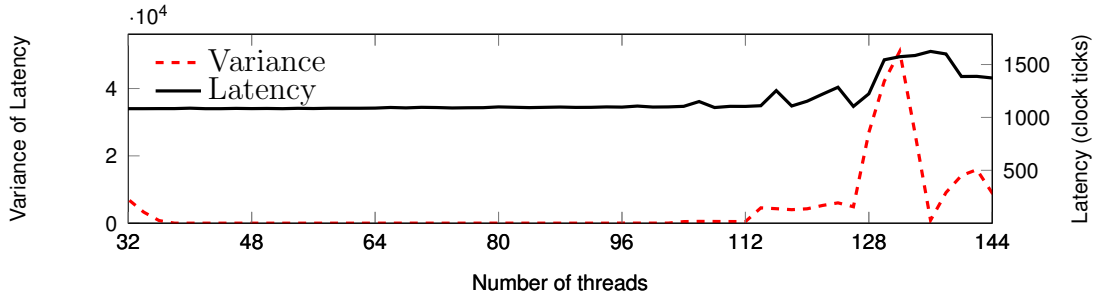


Figure 4.3: Thread-Latency plot under Tesla M2070, one load per thread, and All-unique memory pattern.

Tesla M2070

MSHR or PRT? Figure 4.3 shows a Thread-Latency plot where each thread sends a load request for one unique 128-byte memory block. As reported, a significant variation in latency can be observed upon increasing the number of threads from 128 to 130. From this figure, we infer that SM allows up to 128 unique memory requests. Hence, the outstanding memory handling resources might be an MSHR table with 128 entries or PRT table with four entries ($128/warpsize = 4$). In Figure 4.4 we replicate Figure 4.3, but we change the memory pattern to Two-coalesced. If the structure is PRT we should see the spike at the exact same number of threads as Figure 4.3. However, the spike is moved to 256 threads, suggesting that there should be an extra redundant memory access merging capability in outstanding memory handling resources. Since this capability only exists in MSHR table, the structure may not be PRT. We can conclude that each SM of Tesla M2070 has a structure similar to MSHR table for handling outstanding memory requests.

Parameters of MSHR. Knowing each SM exploits an MSHR table, we look for the number of redundant memory requests that each MSHR entry may carry or merge. Based on our earlier micro-benchmarking, we increase the number of redundant memory requests within the warp to measure the merging capability. We evaluate Two-, Four-, Eight-, 16-, and 32-coalesced memory patterns.

Figure 4.4 shows Thread-Latency plot under *Two-coalesced* memory pattern where each thread executes *one* load instruction. As shown in the figure, there is a huge spike upon moving beyond 256 threads, which equals to 128 unique memory requests. Hence, we can infer that SM allows up to 128 unique memory requests and can also merge up to 128 redundant requests. In other words, SM has enough resources to keep 128 pending memory requests and can merge at least one other redundant request

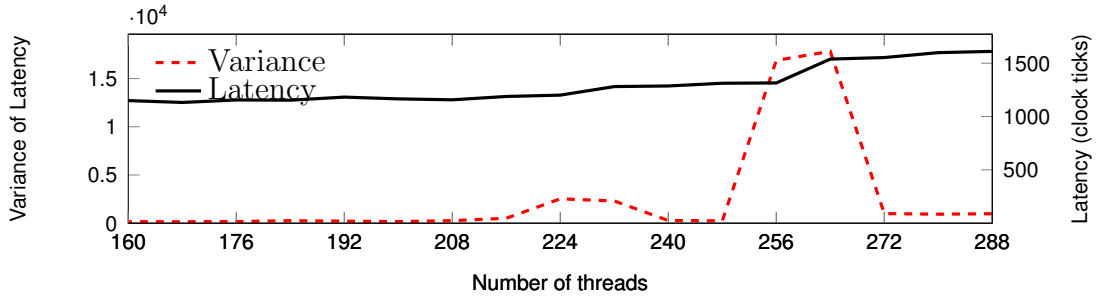


Figure 4.4: Thread-Latency plot under Tesla M2070, one load per thread, and Two-coalesced memory pattern.

with each of those pending requests.

Figure 4.5 reports Thread-Latency plot under *Four-coalesced* memory pattern where each thread executes *two* load instructions. There is a significant variation in the latency when increasing the number of threads beyond 256. This point equals to a total of 512 requests (256×2) from 256 threads, which are later merged into 128 unique memory requests ($512/4 = 128$) according to the memory pattern). We can infer that each SM can keep up to 128 pending memory requests and can merge four other requests with each of the pending requests.

Figure 4.6 reports Thread-Latency plot under *Eight-coalesced* memory patter where each thread executes *four* load instructions. There is a significant spike in variation plot just after 256 threads. This point equals to 1024 requests (256×4) from 256 threads, which are later merged into 128 unique requests ($1024/8 = 256$) according to the memory pattern). We can infer that each SM can keep up to 128 pending memory requests and can merge eight other requests with each of the pending requests.

For *16-coalesced* and *32-coalesced*, we could not observe a significant spike in Thread-Latency plots. Our guess is another resource may have been saturated and the delay acts as a buffer, flattening MSHR saturation spikes.

Finding. According to the observations above, we infer that each SM of Tesla M2070 has a 128-entry MSHR table to handle outstanding memory requests. We can also conclude each MSHR entry can merge up to eight requests.

Tesla K20

MSHR or PRT? Figure 4.8 shows Thread-Latency plot under Tesla K20. In this figure, there are two loads per thread; every thread issues two unique memory requests.

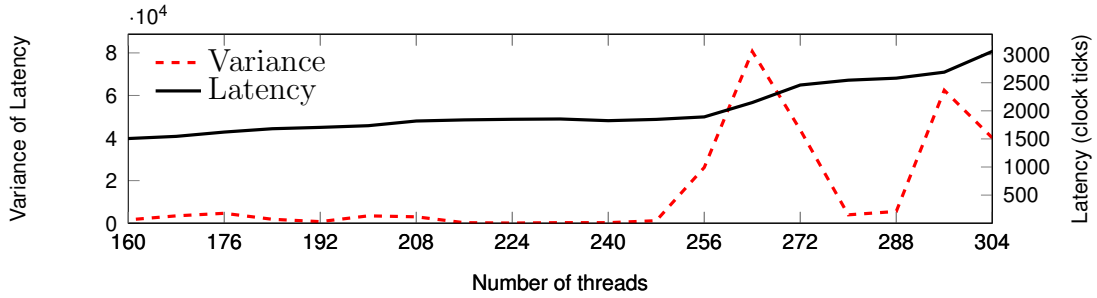


Figure 4.5: Thread-Latency plot under Tesla M2070, two loads per thread, and Four-coalesced memory pattern.

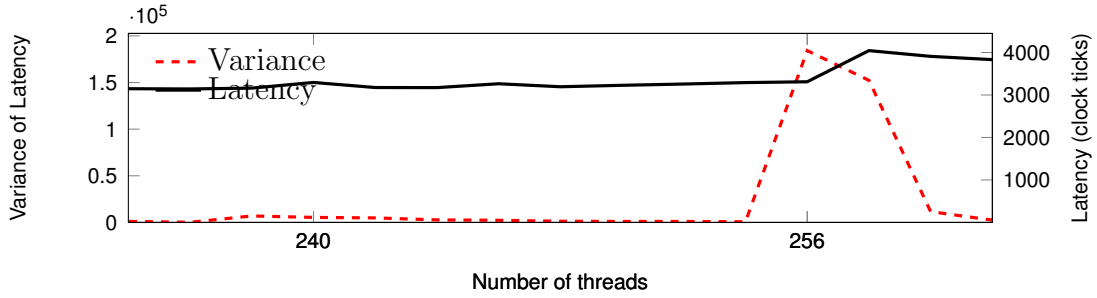


Figure 4.6: Thread-Latency plot under Tesla M2070, four loads per thread, and Eight-coalesced memory pattern.

As can be seen, there is a spike at 680 threads. Hence, the outstanding memory request handling resources can be an MSHR table with $2 \times 680 = 1360$ entries or a PRT table with 44 entries ($2 \times 680/warpsize \approx 44$). In Figure 4.9 we replicate Figure 4.8, but we change the memory pattern to Two-coalesced. If the structure is MSHR we should see the spike at 1360 threads, twice as it is in Figure 4.8. However, the spike is at the exact same number of threads as it is in Figure 4.8, suggesting that there is not an extra redundant memory access merging capability in outstanding memory handling resources. Therefore, the structure may not be MSHR. We conclude that this GPU uses a design similar to PRT.

Finding. Figure 4.7, 4.8, and 4.10 show Thread-Latency plot under Tesla K20 GPGPU. All figures report numbers for All-unique memory pattern; every load instruction has a unique address. Figures differ in the number of load instructions per thread. Figure 4.7, 4.8, and 4.10 report numbers under one, two, and three load instructions per thread. Under one load instruction per thread, we do not observe a significant spike in Thread-Latency. This can be explained by the fact that outstanding memory handling resources are not saturated. Under two load instructions

per thread, we observe a significant spike just after 680 threads. At this point, we have launched 22 warps. Assuming two load instructions per warp, we can also say we issue 44 warp load instructions at this point. Under three load instructions per thread, we observe a significant spike just after 454 threads. At this point, we have launched 15 warps. Assuming three load instructions per warp, we can also say we issue 45 warp load instructions at this point. Comparing our findings in Figures 4.8 and 4.10, we can conclude that each SM can concurrently issue at least 44 warp load instructions. We conclude that there is strong evidence that outstanding memory requests handling resources in Tesla K20 may have a design similar to PRT described in 4.1.1 and the table has at least 44 entries. Since each entry should store 32 requests to support the entire threads of the warp, we can also conclude that each SM can support up to 1408 (44 x 32) outstanding unique memory requests.

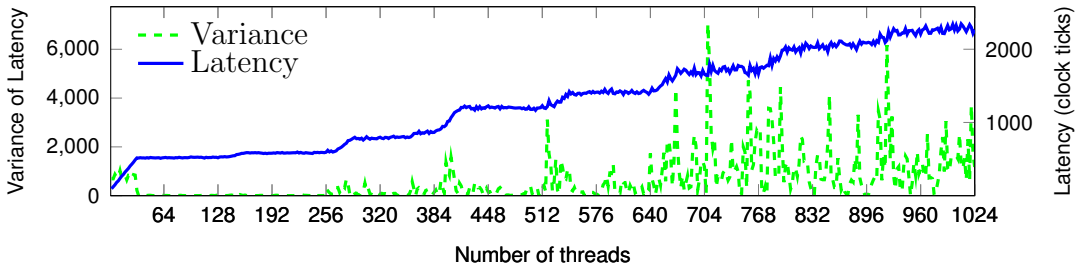


Figure 4.7: Thread-Latency plot under Tesla K20, one load per thread, and All-unique memory pattern.

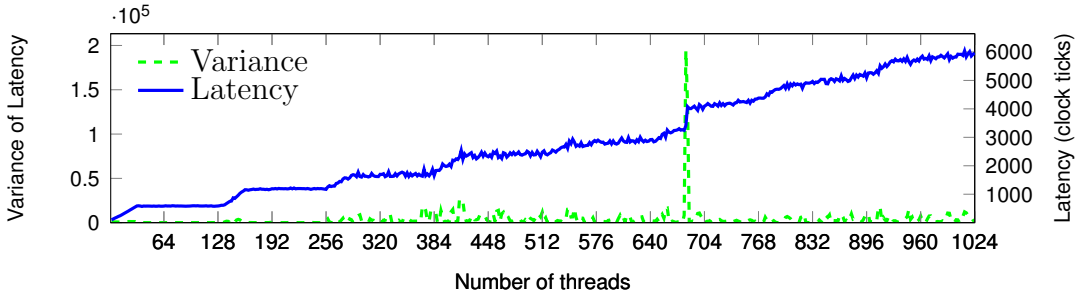


Figure 4.8: Thread-Latency plot under Tesla K20, two loads per thread, and All-unique memory pattern.

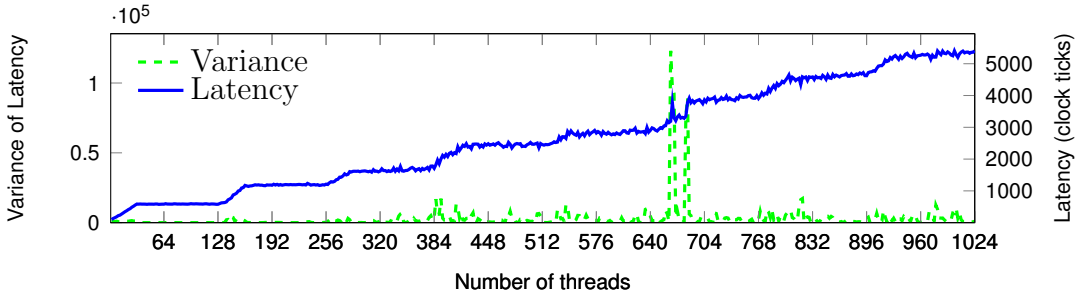


Figure 4.9: Thread-Latency plot under Tesla K20, two loads per thread, and Two-coalesced memory pattern.

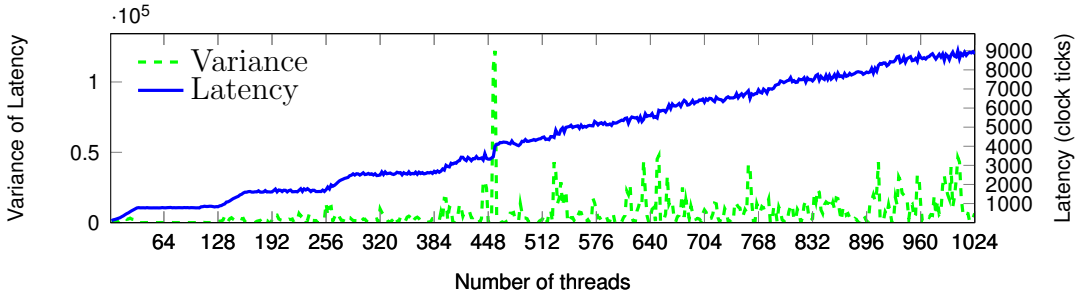


Figure 4.10: Thread-Latency plot under Tesla K20, three loads per thread, and All-unique memory pattern.

4.2 Software-Managed Cache

Software-managed cache in NVIDIA GPUs (also called shared memory) employs multiple banks to deliver high bandwidth. Every generation of NVIDIA GPUs has a certain configuration of shared memory; namely a specific number of banks and the bank line size. A bank conflict occurs once a warp (group of threads executing instructions in lock-step over the SIMD) executes a shared memory instruction and threads of a warp need different rows of the same bank. Bank conflicts cause access serialization if the bank does not have enough read/write ports to deliver data in parallel. We developed a CUDA micro-benchmark to evaluate the impact of several parameters on bank conflict. This test should run separately for every backend supported by the compiler to allow hardware-specific optimizations. Below we first review the micro-benchmark structure, followed by presenting results obtained on NVIDIA Tesla K20c. Finally we summarize the findings.

Micro-benchmark setup

We assume one two-dimensional shared memory array per thread block. We also assume two-dimensional thread blocks. We develop a simple kernel in which every thread reads four locations of shared memory and writes one location. These reads/writes are in a loop iterated several times. The code is shown in Listing 4.2. We report the execution time of this kernel and evaluate the impact of the following parameters in the kernel body:

- **Datatype size (TYPE):** The datatype size of shared memory array is the number of bytes allocated for each element of array. Variations in datatype size impact bank conflict since it determines the layout of array in the shared memory (e.g. one element per bank, two elements per bank, etc.).

- **2D array allocation:** We investigate two alternatives in allocating 2D shared memory: 2D array notation or 1D array notation (flattened notation). 2D array notation is simpler in indexing and code readability. We are also interested to understand whether flattened notation has a different layout in the shared memory from 2D array.

- **Padding (PAD):** When the size of shared memory array is multiple of memory banks, adding a small padding to the array can mitigate the bank conflict. The

Listing 4.2: CUDA micro-benchmark for understanding shared memory.

```
// compiled for different TYPE, ITER, PAD, XY
__global__ void kernel(TYPE *GLB, int size){
    __shared__ int SHD[16+PAD] [16+PAD];
    // mapping config to shared memory
    #ifdef XY
    int row=threadIdx.x, rows=blockDim.x;
    int col=threadIdx.y, cols=blockDim.y;
    #else
    int row=threadIdx.y, rows=blockDim.y;
    int col=threadIdx.x, cols=blockDim.x;
    #endif
    // fetch
    int index=(threadIdx.x+blockIdx.x*blockDim.x)*size+
              (threadIdx.y+blockIdx.y*blockDim.y);
    SHD[row][col]=GLB[index];
    // computation core
    int S = (row==(rows-1))?row:row+1;
    int N = (row==0)     ?0  :row-1;
    int W = (col==(cols-1))?col:col+1;
    int E = (col==0)     ?0  :col-1;
    int k=0; TYPE sum=0;
    for(k=0; k<ITER; k++){
        sum=(SHD[row][col]+ SHD[S][col]+ SHD[N][col]+ SHD[row][E]+
            SHD[row][W])*0.8;
        __syncthreads(); SHD[row][col]=sum; __syncthreads();
    }
    // write-back
    GLB[index]=SHD[row][col];
}
```

padding increases the row pitch, spreading the columns of a row across different banks.

- **Access pattern:** Since bank conflict only occurs among the threads of the same warp, it is important to mitigate bank conflict algorithmically. We evaluate the impact of these algorithmic optimizations by mapping threads of the thread block to different dimensions of the shared memory array. Operating in XY *mapping*, threads along the x dimension of the thread block are mapped to the first dimension of the shared memory array and threads along the y dimension are mapped to the second dimension. YX *mapping* reverses this as threads

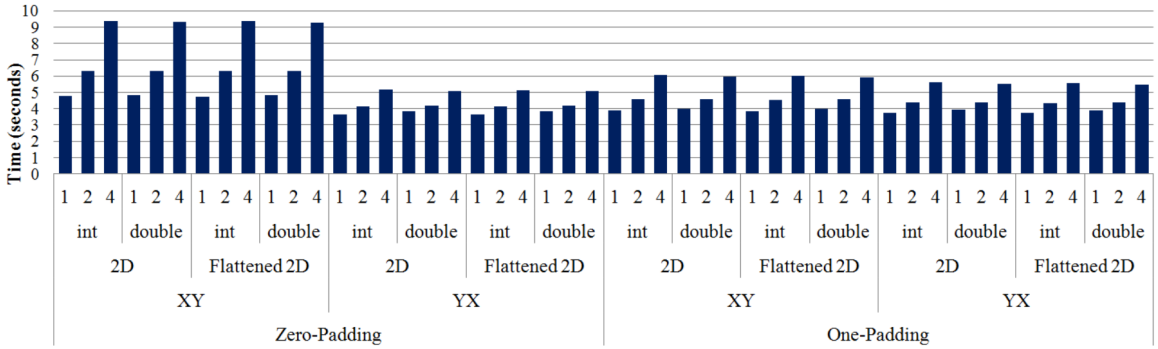


Figure 4.11: Comparing execution time of kernel under various shared memory configurations.

along x and y dimensions are mapped to the second and first dimensions of the array, respectively.

- **Iterations (ITER):** Number of iterations of the loop in the kernel body. This number indicates the ratio of shared memory accesses to global memory accesses.

Results

Figure 4.11 reports the execution time of the kernel in Listing 4.2 under various configurations. Bars report the execution time for three different ITERs (1, 2, and 4), two TYPEs (4-byte integer and 8-byte floating-point), two array allocation schemes (2D and flattened 2D), two shared memory access patterns (XY and YX), and two padding sizes (zero and one).

As shown in the figure, TYPE has modest impact on the execution time. Also the allocation scheme has minor impact on performance. The latter suggests that the layout of 2D array in the shared memory banks is similar to that of the flattened 2D array.

Access pattern, however, impacts performance significantly. In this benchmark, YX mapping delivers a better performance compared to XY. This is explained by how threads are grouped into warps. Warps are occupied first by the threads along the x dimension and then by the threads along y. Therefore threads along x should access consecutive words in order to reduce shared memory bank conflict. This is precisely what YX mapping does.

As shown in the figure, adding a padding to the array can have an impact similar to that of access pattern tunings, lowering the execution time roughly the same amount. Adding a padding to the array can lower the execution time by 57% and 56% under *double* and *int*, respectively. It should be noted that under the cases where the array is padded there is still room for improvement as evidenced by the results. Under one padding, modifying the code algorithmically for reducing bank conflict, as comparing XY to YX shows, can further lower the execution time by 8% (for both *int* and *double*).

Increasing the number of iterations (ITER) increases the importance of the shared memory performance in the overall performance. For larger iterations, the impact of access pattern and padding is more significant. For example, under one iteration, the gap between zero-padding and one-padding is 23%. This gap grows to 37% and 55% under 2 and 4 iterations, respectively.

Summary of findings

We make the following conclusions from the findings presented in this section and use them to optimize our implementations. First, the layout of 2D arrays allocated in the shared memory is found to be the same as flattened 2D arrays. Since no performance advantage is found in using flattened 2D arrays, we use multi-dimensional arrays for caching multi-dimensional subarrays to simplify array indexing code generation. Second, our implementation adjusts mapping of parallel loops to x and y dimensions of the thread blocks with the goal of having threads along x accessing consecutive bytes. We use a heuristic to map the most inner parallel loop to the x dimension of the grid. This is due to the fact that, intuitively, the inner loop has stronger locality and traverses arrays column-wise. Third, adding a small padding can pay off if other compiler optimizations do not allow mapping inner parallel loops over x dimension.

Chapter 5

Efficient Implementation of OpenACC cache Directive on NVIDIA GPUs

OpenACC's programming model presents a simple interface to programmers, offering a trade-off between performance and development effort. OpenACC relies on compiler technologies to generate efficient code and optimize for performance. The *cache* directive is among the challenging to implement directives. The *cache* directive allows the programmer to utilize accelerator's hardware- or software-managed caches by passing hints to the compiler. In this chapter, we investigate the implementation aspect of *cache* directive under NVIDIA-like GPUs and propose optimizations for the CUDA backend. We use CUDA's shared memory as the software-managed cache space. We first show that a straightforward implementation can be very inefficient, and undesirably downgrade performance. We investigate the differences between this implementation and hand-written CUDA alternatives and introduce the following optimizations to bridge the performance gap between the two: i) improving occupancy by sharing the cache among several parallel threads and ii) optimizing cache fetch and write routines via parallelization and minimizing control flow. Investigating three test cases, we show that the best *cache* directive implementation can perform very close to hand-written CUDA equivalent and improve performance up to 2.4X (compared to the baseline OpenACC.)

The rest of this chapter is organized as follows. In Section 5.1 we discuss inefficiencies of a naïve cache implementation. In Section 5.2 we present our proposed

implementations for the *cache* directive. In Section 5.3 we introduce optimizations applicable to the proposed implementations. In Section 5.4 we evaluate performance of the proposed methods. In Section 5.5 we discuss the limitations of our approach. Finally, in Section 5.6 we summarize our findings.

5.1 Motivation

OpenACC API is designed to program various accelerators with possibly different cache/memory hierarchies. Generally, the compiler is responsible for generating an efficient code to take advantage of the hierarchies. Static compiler passes can figure out specific variables or subarrays with an opportunity for caching. However, as static passes are limited, OpenACC API also offers a directive, allowing programmers to hint the compiler. The *cache* directive is provided to facilitate such compiler hints. The directive is not accelerator-specific and is abstracted in a general form. These hints specify the range of data showing strong locality within individual iterations of the outer parallel loop, which might benefit from caching.

The *cache* directive is used within a *parallel* or *kernels* region. The directive associates with a *for* loop (where the locality is formed) and can be used over or in the loop. The line below shows the syntax of the directive in C/C++:

#pragma acc cache(var-list)

var-list passes the list of variables and subarrays. Subarray specifies a particular range from an array with the following syntax:

arr[lower:length]

lower specifies the start index and *length* specifies the number of elements that should also be cached. *lower* is derived from constant and loop invariant symbols. This can also be an offset of the *for* loop induction variable. *length* is constant.

According to OpenACC specification [77], variables and subarrays listed in *var-list* should be fetched into the highest level of the cache for the body of the loop. We refer to the scope of the loop as *cache region*. In the *cache region*, all accesses to the variables and subarrays listed in *var-list* should be served from the cache.

Listing 5.1 shows an example of the *cache* directive. The example is based on one-dimensional stencil algorithm. 1D stencil smooths the values of array iteratively, repeating for certain number of iterations, here K times. In this example, the array length and 1D stencil radius are LEN and one element, respectively. The new value of every element is calculated as the average of three elements; the element and right

and left neighbors. The programmer can provide a hint to the compiler to highlight this spatial locality within each iteration of the parallel loop. On line #7, the *cache* directive hints the compiler that each iteration of the loop requires *three* elements of *a[]*, starting from *i-1*. Provided with this hint, the compiler can potentially cache this data in registers, software-managed cache, or read-only cache (depending on the target). Also depending on the accelerator-specific optimization strategies, the compiler can ignore the hint, which is not the focus of this study.

Listing 5.1: The cache directive example; one-dimensional stencil.

```
1  #pragma acc data copy(a[0:LEN],b[0:LEN])
2  for(n=0; n<K; ++n){
3   #pragma acc parallel loop
4   for(i=1; i<LEN-1; ++i){
5    int lower = i-1, upper = i+1;
6    float sum = 0;
7    #pragma acc cache(a[(i-1):3])
8    for(j=lower; j<=upper; ++j){
9      sum += a[j];
10   }
11   b[i] = sum/(upper-lower+1);
12  }
13  float *tmp=a; a=b; b=tmp;
14 }
```

Figure 5.1 compares the performance of two different *cache* directive implementations (*naïve* and *optimized*) for the code listed in Listing 5.1. These two implementations are compared to the *baseline* (which does not use the *cache* directive). The *naïve* implementation isolates cache space to each parallel iteration of the loop. The *optimized* implementation is equipped with optimizations later introduced in this chapter and exploits the opportunity for sharing cached elements among parallel iterations. Consequently, *optimized* delivers more efficient cache implementation through better occupancy, cache sharing, and initial fetch parallelization. We explain each of these optimizations in the rest of the chapter. This figure emphasizes the importance of optimizing cache implementation.

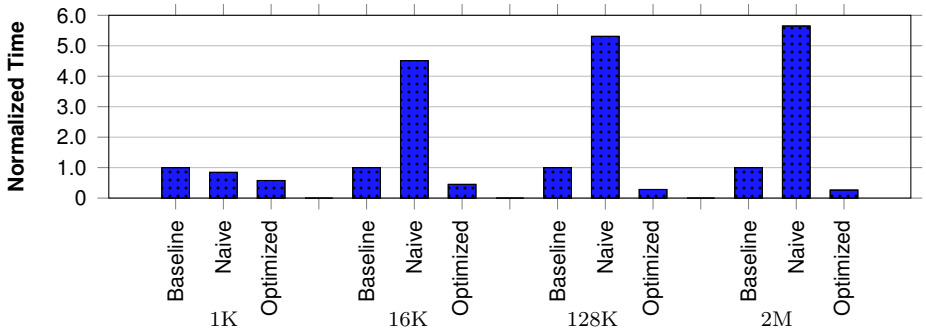


Figure 5.1: Comparing naïve and optimized cache implementations under 1D stencil kernel listed in Listing 5.1 (30-element radius, 1K, 16K, 128K, and 2M elements.)

5.2 Implementations

In this section, we present three *cache* directive implementations for accelerators employing software-managed cache. We discuss methods for the case where the list of variables consists of subarrays (simplified versions of the presented methods are applicable for scalar variables.). For implementing the *cache* directive, the compiler requires two pieces of information: i) the range of the data to be cached and ii) the array accesses (within the cache region) that their array index value falls within the subarray range (we assume pointer aliasing is not the case and pointers are declared as restricted type in the accelerator region, using C's *restrict* keyword.). Using the information provided through the directive, the compiler knows the subarray; data that should be cached. To gather the second piece of information, the compiler must examine the index of every array access in the cache region. If the compiler could statically assure that the index falls within the cache range, the array access might simply be replaced by a cache access in the code. Otherwise, the compiler should generate code to decide to fetch from the cache or global memory on-the-fly. Therefore, depending on the code, the compiler may generate a different control flow. As we show in this chapter, this can be very expensive to calculate in runtime. Starting from OpenACC 2.5 [77], the following restriction has been added to the *cache* directive specification: within the cache region, all references to an array listed in the *cache* directive must refer to the range specified in the *cache* directive. Our first two proposed methods (EHC and RBC) comply with the older OpenACC specification [76, 75] and are suitable for applications written in older OpenACC versions (e.g. v2.0). Our third method takes advantage of the restriction added in OpenACC 2.5 to highly optimize the implementation.

The first method is an emulation of hardware-managed cache through software-managed cache. To this end, data and tag arrays are maintained in the software-managed cache. Operations of hardware cache is emulated using these two arrays. The second and third methods are range-based caching. The second method stores the lower and length specifiers and checks if the value of the index falls within this range. The third method assumes all indexes fall into the fetched range and uses a simple operation to map array indexes to cache locations. Below we elaborate on these methods.

5.2.1 Emulating Hardware Cache (EHC)

Overview. Two arrays are allocated in the software-managed cache; data and tag. Data array stores the elements of the subarray. Tag array stores the indexes of subarray elements that are currently cached. Tag array can be direct-mapped, set-associative, or fully-associative to allow caching the entire or part of the subarray transparently. The decision depends on the subarray size and accelerator capabilities.

Pros and cons. The main advantage of this method is the ability to adapt to the available cache size. If the *cache* directive demands a large space and the accelerator's cache size is small, this method allows storing only a portion of the subarray (other methods might ignore the directive in this case). There are two disadvantages with this method though. First, storing the tag array in the software-managed cache lowers the occupancy of the accelerator and limits concurrent threads. Second, at least two cache accesses (tag plus data) are made for every array access, increasing the read/write delay significantly. In terms of operations, each global memory access is replaced by two cache accesses and few other logical/arithmetic and control operations. This significant overhead impairs the performance advantages as the total latency of the cache hit can exceed the global memory latency (depending on the accelerator's design).

5.2.2 Range-based Conservative (RBC)

Overview. One array and two pointers are allocated in the software-managed cache. The array stores the subarray. Two pointers keep the range of (global) indexes stored in the cache. One of the pointers points to the start index and the other points to the end index (or the offset from the start). Value of these pointers are within the scope of the original array (located in global memory) and subtracting these pointers returns

the size of the cached array. To check if the array index falls within the subarray range or not, the index is checked against the range kept by pointers. Two comparisons evaluate this; index $\geq$ start && index $<$ end. If the condition holds, data is fetched from the cache, otherwise from global memory. Moreover, if the condition holds, the index should be mapped from global memory to cache space. The operation for this mapping is a subtraction (index - start).

Pros and cons. The *cache* directive always points to a stride of data. This method exploits the fact that elements of subarray are a row of consecutive elements from the original array and minimizes the overhead for maintaining the track of the cached data (compared to EHC). The method stores two pointers pointing to the start and end of the stride. The method can be extended to multi-dimensional subarrays by storing a pair of pointers per dimension. The only disadvantage of this method is the performance overhead of the control flow statement generated for checking whether the index falls within the range of stride or not. This control statement might be an expensive operation for multi-dimensional subarrays ($2+1$ logical ops. plus a branch for 1D, $4+3$ logical ops. plus a branch for 2D, etc.).

5.2.3 Range-based Intelligent (RBI)

Overview. This method improves RBC one step further and assumes array indexes always fall within the subarray range. This avoids the costly control flow statements for evaluating whether the data is in the cache or not. The compiler may use this method if the compiler passes are able to find the range of values of the index statically.

Pros and cons. This method has significant performance advantage over RBC as it avoids the costly control statements for checking if the data exists in the cache or not. Assuring that the index always falls within the fetched stride was not a trivial compiler pass in the past. The restrictions added in the latest OpenACC version have addressed this by limiting the subarray references. Accordingly, the latest version of OpenACC (2.5 released in November 2015) adds a restriction to *cache* directive requiring all references to the subarray lie within the region being cached [77]. This essentially means RBI can be used with all applications that follow OpenACC $\geq$ 2.5.

5.2.4 Example

Listing 5.2 and 5.3 show the CUDA implementations of the methods explained above. Three procedures are implemented for each method: i) *__cache_fetch()*, ii) *__cache_read()*,

and iii) *__cache_write()* (as a performance issue, these procedures are declared inline to avoid procedure calls within the accelerator region.). The accelerator code is generated to call *__cache_fetch()* early before the cache region starts. This procedure is responsible for fetching the data into the cache. Within the cache region, the compiler replaces every array read with *__cache_read()* call and array write statement with *__cache_write()* call. For these implementations, we assume a write-through cache (alternative is discussed in Section 5.3.3.).

Listing 5.2: Implementation of Emulating Hardware Cache (EHC) in CUDA.

```
1  __device__ void __cache_fetch(PTRTYPE* g_ptr, PTRTYPE* c_ptr, unsigned*
       ctag_ptr,
2    unsigned st_idx, unsigned en_idx){
3      for(unsigned i=st_idx; i<en_idx; i++){
4          unsigned cache_idx=acc_idx&0x0ff; //direct map
5          c_ptr[cache_idx]=g_ptr[i];     // update data array
6          ctag_ptr[cache_idx]=i;         // update tag array
7      }
8  }
9  __device__ PTRTYPE __cache_read(PTRTYPE* g_ptr, PTRTYPE* c_ptr, unsigned*
       ctag_ptr,
10   unsigned st_idx, unsigned en_idx, unsigned acc_idx){
11     unsigned cache_idx=acc_idx&0x0ff; //direct map
12     if(ctag_ptr[cache_idx]==acc_idx){
13         return c_ptr[cache_idx];       // read from cache
14     }else{
15         c_ptr[cache_idx]=g_ptr[acc_idx]; // read from global memory, update
               data
16         ctag_ptr[cache_idx]=acc_idx;   //   and tag arrays
17         return c_ptr[cache_idx];       // read from cache
18     }
19 }
20 __device__ void __cache_write(PTRTYPE* g_ptr, PTRTYPE* c_ptr, unsigned*
       ctag_ptr,
21   unsigned st_idx, unsigned en_idx, unsigned acc_idx, PTRTYPE value){
22     unsigned cache_idx=acc_idx&0x0ff;    //direct map
23     if(ctag_ptr[cache_idx]!=acc_idx)
24         ctag_ptr[cache_idx]=acc_idx;     // update tag
```

```
25      g_ptr[acc_idx] =c_ptr[cache_idx] =value; // write-through
26  }
```

Listing 5.2 shows the CUDA implementation of EHC where the tag array models a direct-map cache. For this example, we assume a 256-entry cache. In this case, mapping from global memory indexes to cache space is a single logical operation. Listing 5.3 shows the CUDA implementation of RBC. RBI implementation is the same as Listing 5.3, except the control statement in *__cache_read()* and *__cache_write()* is removed as the condition of the control statement is always true in RBI. In this Listing, the mapping is an arithmetic operation; subtracting index from the start pointer. *__cache_fetch()* routine in all implementations has a *for* loop statement. Later in Section 5.3.2, we discuss opportunities to accelerate this loop through parallelization.

5.3 Implementation Optimizations

In this section, we introduce optimizations for implementations introduced in the previous section. Specifically, we present optimizations for cache fetch routine, cache sharing, cache writes, and minimizing index mapping overhead.

5.3.1 Cache Fetch Routine

The cache fetch routine is called before cache region starts. This is done once per parallel instance of the loop which the *cache* directive is associated with (the fetch routine might be called multiple times, if written in a sequential loop.). If the cache region is a large and time consuming code block, this routine's performance may not be the limiting factor. Otherwise, if the cache region is short, the performance of this routine is critical to the overall performance.

Performing our evaluations under NVIDIA GPUs, we found that minimizing control flow statements comes with significant performance advantage. The fetch routine has a *for* loop statement (as presented earlier in Section 5.2.4) which imposes control flow overhead. Loop unrolling can be employed to reduce this overhead, as the length of the loop is a compile-time constant (equal to the length of the subarray). Also the compiler can reduce this overhead further by sharing a single *for* loop among multiple subarray fetches. Compiler heuristics can decide if the loop can be shared among multiple subarrays. For example, the compiler can read the *cache* directive

and group the subarrays having equal length. Subsequently the grouped subarrays can share the same *for* loop, as the number of iterations for fetching the data is the same for all of them.

Listing 5.3: Implementation of Range-based Conservative (RBC) in CUDA.

```
__device__ void __cache_fetch(PTRTYPE* g_ptr, PTRTYPE* c_ptr,
  unsigned st_idx, unsigned en_idx){
    for(unsigned i=st_idx; i<en_idx; i++)
        c_ptr[i-st_idx]=g_ptr[i];
}
__device__ PTRTYPE __cache_read(PTRTYPE* g_ptr, PTRTYPE* c_ptr,
  unsigned st_idx, unsigned en_idx, unsigned acc_idx){
    if(acc_idx>=st_idx && acc_idx<en_idx){
        unsigned cache_idx=acc_idx-st_idx;
        return c_ptr[cache_idx];
    }else
        return g_ptr[acc_idx];
}
__device__ void __cache_write(PTRTYPE* g_ptr, PTRTYPE* c_ptr,
  unsigned st_idx, unsigned en_idx, unsigned acc_idx, PTRTYPE value){
    if(acc_idx>=st_idx && acc_idx<en_idx){
        unsigned cache_idx=acc_idx-st_idx;
        c_ptr[cache_idx]=value;
    }
    g_ptr[acc_idx]=value;
}
```

Another opportunity to optimize the *for* loop is to parallelize the loop. A number of parallel threads, e.g. equal to the size of the thread block, can be employed to fetch the data into the software-managed cache. If the compiler is not using parallel threads for another task, parallel fetch can simply achieve this. However, if parallel threads have already been employed to execute parallel tasks, then the compiler should assure that while threads collaborate for fetching the data, they maintain a separated view of the cache, specially in the case of cache writes. We explain this further in Section 5.3.2.

5.3.2 Cache Sharing

Considering the relative nesting of the *cache* directive in respect to parallel loops, there are two types of parallel loops: *outer parallel loops* and *inner parallel loops*. Iterations of inner parallel loops already share the same data. In this section we introduce a method to find data sharing among the iterations of outer parallel loops. Listing 5.4 clarifies outer and inner loops in an example.

Listing 5.4: Example of inner and outer parallel loops around cache.

```
1  #pragma acc parallel loop
2  for(i=0; i<N; i++){ // OUTER LOOP:
3   // depending on X and Y, the subarray
4   // may or may not be shared among iterations
5   #pragma acc cache(subarray[X:Y])
6   { // beginning of cache region
7    #pragma acc loop
8    for(j=0; j<N; j++){ // INNER LOOP:
9     // the subarray is shared among all iterations
10   }
11  } // end of cache region
12 }
```

The *cache* directive is located within one (or more) *outer* parallel loop(s) and the cache space should be allocated once per parallel instance of outer parallel loop(s). The compiler can optimize cache utilization by unifying the allocations of common data and sharing them among parallel iterations. When it comes to *cache* directive implementations in CUDA, sharing data between parallel iterations is efficiently feasible by mapping parallel iterations (in OpenACC) to threads of the same thread block (in CUDA), sharing data through *CUDA shared memory*.

We have different methods for cache sharing under EHC, RBC, and RBI. Under EHC, cache sharing can be achieved by sharing one single larger data and tag arrays among all iterations. The complexity is in efficiently managing consistency of data and tag arrays, considering parallel accesses to the cache may occur from different iterations. Currently, the only mechanism in CUDA to maintain the consistency is to update data and tag arrays atomically using atomic operations. Since this severely slows down the performance, we found cache sharing unpromising in EHC. Below we discuss cache sharing method under RBC and RBI.

We decompose the cache sharing problem under RBC and RBI to five subproblems: i) extract sharing, ii) find sharing width, iii) renew cache scope, iv) fetch collaboratively, and v) optimize cache size. Below we discuss each problem.

Extract Sharing

The problem is to map outer parallel loops (loops that are marked by the OpenACC loop directive as parallelizable) to thread hierarchies with the constraint of maximizing the subarray overlap among threads of the thread block. Listing 5.5 presents a compiler pass as a solution to this problem. The problem inputs are the *cache* directive (code block where pragma is injected and list of subarrays), outer parallel loops (loop handle, induction variable, and increment step), and the kernel code. The problem output is the mapping of loop iterations to CUDA thread block dimensions.

Listing 5.5: Compiler pass that extracts cache sharing opportunity and suggests a mapping to maximize the overlap among subarrays of consecutive iterations.

```
Inputs:
      cache: the code block id of the cache region
  subarrays: array of subarrays listed in the cache directive
         Ls: array of outer parallel loops, indexed by induction
             variables
        IDs: array of induction variables associated with outer
             parallel loops
       code: the kernel code
Output:
    mapping: structure showing the parallel loops to kernel
        dimensions mapping
Begin
  final_mapping = []
  skipped_subarray = []
  for subarray in subarrays
    unmapped_dimensions = [x, y, z]
    suba_mapping = []
    for dimension in subarray
      lower, length <- get_specifiers(dimension)
      if is_linear(lower, IDs, code, Ls)
        rate, inductionVar, offset <- get_linear_params(lower, IDs,
            code, Ls)
        // map parallel loop iterated by inductionVar to an unmapped
            dimension
        suba_mapping.push(Ls[inductionVar] -> unmapped_dimension.pop())
    if not is_contrary(final_mapping, suba_mapping)
      mapping = merge(final_mapping, suba_mapping)
    else
      skipped_subarrays.push(subarray)
  return final_mapping
End
```

The pass iterates over the subarrays listed in the *cache* directive. For each dimension of the subarray (dealing with multi-dimensional subarrays), *lower* and *length* specifiers are read. If *lower* is a linear function of one single induction variable, consecutive iterations of the loop corresponding to the induction variable are considered for sharing (see examples in Table 5.1). *is_linear()* function returns true if i) *lower* specifier is a linear function of an induction variable and ii) the increment step of the corresponding loop is linear (e.g. i+=1, i-=1, i+=7, etc.). If *lower* is linear, it should be in $rate * inductionVar + offset$ form, where $inductionVar$ is an induction variable and *rate* and *offset* are expressions independent of any induction variable. Forcing the increment step to be linear assures that the neighbor threads cache subsequent elements, forming a sharing range that is densely populated by the data from

Table 5.1: Example of cache sharing when *lower* specifier is a linear function of an induction variable. Assumptions: i is an induction variable of a parallel loop, increment step of the loop iterated by i is *+1*, and thread block size is 3.

subarray	lower	length	ranges mapped to the iterations	shared range
a[i:3]	i	3	T_0 in 0 to 2 T_2 in 1 to 3 T_2 in 2 to 4 etc.	T_0 to T_2 in 0 to 4 etc.
a[2*i+1:3]	2*i+1	3	T_0 in 1 to 3 T_1 in 3 to 5 T_2 in 5 to 7 etc.	T_0 to T_2 in 1 to 7 etc.
a[3*i+4:5]	3*i+4	5	T_0 in 4 to 8 T_1 in 7 to 11 T_2 in 10 to 14 etc.	T_0 to T_2 in 8 to 14 etc.

neighbor threads (consecutive iterations).

get_linear_params() returns *rate*, *inductionVar*, and *offset*. *suba_mapping* is updated to map the parallel loop iterated by *inductionVar* to unmapped thread block dimensions, starting with x dimension. *is_contrary()* returns true if the *suba_mapping* that is found here contrasts with the mapping recorded in *final_mapping*. If this is the case, *subarray* is pushed to *skipped_subarrays*. Cache sharing optimizations will be skipped for the subarrays in *skipped_subarrays*. Otherwise, *final_mapping* is updated to be merged with *suba_mapping*.

Find Sharing Width

Sharing width is referred to the number of iterations (or threads) that share one common cache. Ideally, sharing width is equal to the thread block size. This is the case when the total number of loop iterations is multiple of the thread block size. However, since the total number of loop iterations is a runtime variable mostly, compiler cannot statically assure this number is multiple of thread block size. We propose three different methods to find the sharing width in CUDA; using synchronization, kernel arguments, or fixed.

Synchronization: This method counts the number of threads that have reached the cache region (within the thread block). To count the number of threads, following

device function from CUDA API is used: *__syncthreads_count(bool flag)*. To count the number of threads along x dimension of the thread block, *__syncthreads_count* is called with the argument *threadIdx.y==0 && threadIdx.z==0*. Similarly, for y and z dimensions of the thread block, the function is called with *threadIdx.x==0 && threadIdx.z==0* and *threadIdx.x==0 && threadIdx.y==0* arguments, respectively.

Kernel Arguments: This method exploits the fact that only the last thread blocks across every dimension may have a sharing width different than the thread block size. This width can be pre-calculated and passed to the kernel as an argument, knowing the total number of iterations and the thread block size upon kernel launch. Within the kernel, threads check if they belong to the last thread block of the dimension. If yes, sharing width is set to the value passed as the argument. Otherwise, sharing width is equal to the thread block size. This method has a performance advantage over the first method as it avoids synchronization and reduction.

Fixed: This method simply sets the sharing width equal to the thread block size. This method is only applicable in the case where compiler can statically assure that the total number of loop iterations is multiple of the thread block size.

Renew Cache Scope

From the notation of the *cache* directive, every thread knows the range from *lower* to *lower + length* is cached. For RBC and RBI, *start* and *end* pointers are set to these values. However, when threads of the thread block are sharing the cache, these pointers should be recalculated, since a larger data range is cached in this case. We propose two different methods to recalculate pointers: communicating and private.

Communicating: This method shares pointers among threads of the thread block. To share pointers, these are declared as CUDA *__shared__* variables. To set pointers consistently, one thread is to set *start* and another thread is to set *end*. *start* pointer is set to *lower* by the thread that is demanding subarray's elements located at the lowest address. This is the first thread within the sharing width, if the corresponding loop has increasing increment step (e.g. +=1, +=3, etc.). Otherwise, if the corresponding loop has decreasing increment step (e.g. -=1, -=3, etc.), this thread is the last thread within the sharing width. Similarly, *end* pointer is set to *lower + length* by the thread that is demanding subarray's elements located at the highest address. This is the last thread within the sharing width, if the corresponding loop has increasing increment step. Otherwise, if the corresponding loop has decreasing

increment step, this thread is the first thread within the sharing width.

Private: This method allocates *start* and *end* pointers privately for each thread. Following equations are used to recalculate *start* and *end* pointers privately:

$$\begin{aligned} start &= lower - rate \times threadID \\ end &= start + (length - 1) + rate \times (sharingWidth - 1) \end{aligned} \tag{5.1}$$

where *lower*, *rate*, *threadID*, *length*, and *sharingWidth* parameters are explained below. *lower* and *length* are specifiers of the subarray passed to the *cache* directive. *rate* is obtained from *lower* by using *get_linear_params()* function in Listing 5.5. *sharingWidth* is the number of active threads in the cache region obtained by the methods proposed for finding sharing width. *threadID* is the thread ID within the thread block, ranging from 0 to *sharingWidth - 1* in the cache region. Equations above are applicable to the case where *lower* is a function of an induction variable of a loop with an increasing increment step. Under decreasing increment step, following equations are used:

$$\begin{aligned} start &= lower + rate \times threadID - rate \times (sharingWidth - 1) \\ end &= lower + rate \times threadID + (length - 1) \end{aligned} \tag{5.2}$$

Fetch Collaboratively

If cache sharing is applicable, threads of the thread block share one common data in shared memory. Since the common data is composed of words located at consecutive addresses, threads of the thread block can be used to efficiently fetch the data using few well-coalesced accesses in parallel. To perform this optimization, only *__cache_fetch()* routine in Listing 5.3 needs to be modified. The *for* loop statement should be modified to:

for(unsigned i=threadIdx.x+st_idx; i<en_idx; i+=blockDim.x)

This is for the case where the subarray is one-dimensional and the parallel loop is mapped to x dimension of the thread block. For multidimensional subarrays, this loop is replicated but modified to reflect correct mapping of parallel loops to thread block dimensions.

Optimize Cache Size

When cache is not shared, each thread demands *length* elements from shared memory. While sharing the cache among threads of the thread block, it might seems $length * sharingWidth$ elements from shared memory are required. This is correct as long as subarrays of consecutive loop iterations are located back to back in the memory. Otherwise, if there is an overlap or gap among subarrays, this number overestimates or underestimates the exact size. We use the following formula to optimize the cache size:

$length + rate * (sharingWidth - 1)$

where *sharingWidth* is the number of active threads in the cache region obtained by the methods proposed for finding sharing width. *length* is a specifier of subarray passed in to the *cache* directive. *rate* is obtained from *lower* by using *get_linear_params()* function in Listing 5.5.

5.3.3 Cache Write Policy

Writing to the subarray in the cache region invokes the write routine. We assume two alternative policies for cache write: write-back and write-through. Write-back buffers cache writes and writes final changes back to DRAM at the end of the cache region. Write-through writes every intermediate write to both cache and global memory. Write-back tends to perform better under dense and regular write patterns whereas write-through performs better under sparse irregular write patterns. We compare performance of these two implementations in Section 5.4.2.

If the compiler implements write-back cache, an additional routine should be invoked at the end of the cache region to write the dirty content of the cache to global memory. For tracking the dirty lines, the compiler can decide to i) keep track of the dirty lines through a mask or ii) assume all the lines are dirty. Although keeping track of dirty lines can reduce the total amount of write operations, the compiler can instead use the brute-force write-back on the GPUs for two reasons. First, tracking dirty lines demands extra space from the software-manage cache to store the dirty mask. This, in turn, lowers the occupancy of GPU. Second, the write-back routine can include extra control flow statements to filter out dirty lines. These control flow statements can harm performance (e.g. limiting ILP and loop unrolling). For the two reasons mentioned above, intuitively we believe the brute-force write-back cache delivers higher performance and we only implement this mechanism in this work.

5.3.4 Index Mapping

As we discussed in Section 5.2, mapping global memory indexes to shared memory indexes involves a few arithmetic operations. To mitigate this overhead, the compiler can allocate a register to store the output of operations for the life time of the cache region, if the value of index is not changing in the cache region. The compiler can also reuse this register for other array accesses, if the array indexes have the same value. This optimization saves register usage and mitigates index mapping overhead.

5.4 Experimental Results

In this section, we first study the performance of methods introduced in Section 5.2, under three test cases. This is followed by investigating performance of different cache write policies. Finally, we evaluate performance portability of our implementation.

We use the IPMACC compiler [44] to compile OpenACC applications and implement the *cache* directive. The IPMACC framework translates OpenACC to CUDA and uses NVIDIA *nvcc* compiler to generate GPU binaries. We run evaluations under NVIDIA Tesla K20c GPU. The execution time of the kernel is measured by nvprof [14]. Every number is harmonic mean of 30 independent samples.

5.4.1 Test Cases

Here we investigate the *cache* directive under three different benchmarks; matrix-matrix multiplication (GEMM), N-Body simulation, and Jacobi iterative method. For each benchmark, we compare the performance of four implementations (we found EHC implementation very slow and hence we avoid further discussion on this.):

1. OpenACC without *cache* directive,
2. OpenACC plus *cache* directive implemented using RBC,
3. OpenACC plus *cache* directive implemented using RBI, and
4. hand-written CUDA version.

All cache-based implementations are optimized with the parallel cache fetch and cache sharing optimizations discussed in Section 5.3. Under RBC and RBI, we use

Table 5.2: Development effort of the benchmarks under OpenACC, OpenACC plus cache, and CUDA implementations.

	OpenACC	OpenACC+cache	CUDA
GEMM	84	94	116
N-Body	81	84	108
Jacobi	145	152	189

kernel arguments as the default method for finding sharing width and we use *private* as the default method for renewing cache scope.

Below we first compare development efforts of these four implementations. Next we compare performance of these implementations. Then we investigate how these implementations utilize GPU resources, e.g. register file and software-managed cache. Finally, we investigate the impact of alternative optimizations on the speedup.

Development Effort

We wrote all versions of GEMM and Jacobi. For N-Body Simulation, we used the CUDA version available in GPU Computing SDK [68] and modified the serial version available there to obtain OpenACC versions. We did our best to hand-optimize using the techniques that we are aware of. Table 5.2 compares the development effort of GEMM, N-Body, and Jacobi under OpenACC, OpenACC plus cache, and CUDA implementations. Development effort is measured in terms of the number of statements, including declaration, control, loop, return, and assignment statements. As reported, OpenACC plus cache can be obtained by modifying 3 to 10 lines of the baseline OpenACC version.

Performance

GEMM: Cache-based OpenACC implementations iteratively fetch 16×16 tiles of two input matrices into the software-managed cache using the *cache* directive and keep the intermediate results (sum of products) in registers. The CUDA version also implements the same algorithm using *shared memory* notation. Figure 5.2 compares the performance of these implementations under various square matrix sizes, compared to the baseline OpenACC (without cache). A similar trend can be observed under different input sizes. RBI outperforms OpenACC by nearly 2.4X and performs very close to CUDA. RBC, RBI, and CUDA reduce the global memory traffic

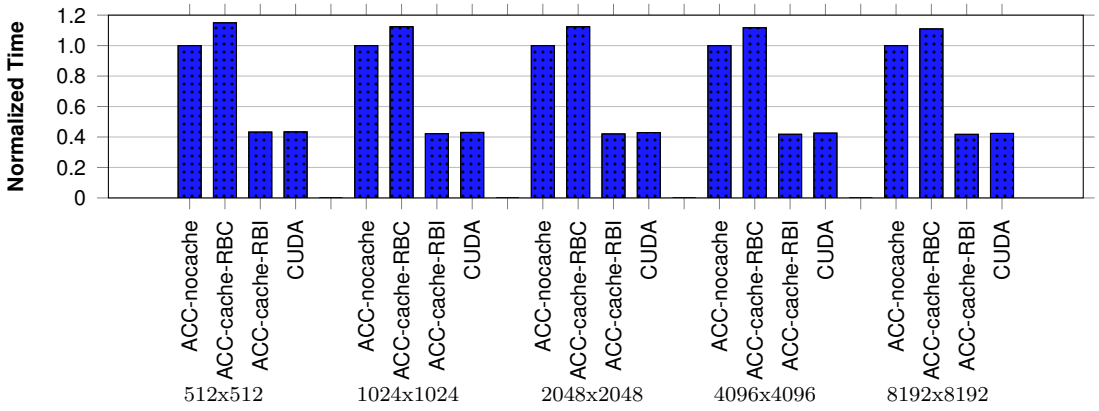


Figure 5.2: Comparing performance of four GEMM implementations under different matrix sizes. For each bar group, bars from left to right represent OpenACC without cache directive, OpenACC with cache directive implemented using RBC, OpenACC with cache directive implemented using RBI, and CUDA.

significantly, compared to OpenACC. By fetching the tiles of input matrices into software-managed cache, these implementations maximize memory access coalescing. Also these implementations exploit the locality among neighbor threads to minimize redundant memory fetches. Using *nvprof* [14], we found that RBI reduces the number of global memory loads by 12X (under 1024x1024 matrices), compared to OpenACC (the very same improvement is observed under RBC and CUDA too). Using RBC, the compiler generates a code to check the memory addresses dynamically and to find out if the address falls within the subarray range or not. If the address falls within the subarray range, the data is fetched from the cache. Otherwise, the data is fetched from the global memory. Under RBI, however, the compiler static passes assure that dynamic memory accesses always fall in the subarray range (if violated, the program can generate incorrect output). Therefore, dynamic checking for the address range is avoided. This explains why RBI always performs faster than RBC. As shown in Figure 5.2, RBC is 2.67X slower than RBI. This gap is caused by RBC's extra logical and control flow instructions per memory access, negating the gain achieved from using the software-managed cache. For the 2D subarray of this benchmark, these extra instructions are one branch, four comparisons, and three ANDs. We discuss this issue further in Section 5.5.

N-Body simulation: Figure 5.3 compares four implementations of N-Body simulation under different problem sizes. To improve performance using software-managed cache, interaction between masses are computed tile-by-tile. Bodies are grouped into

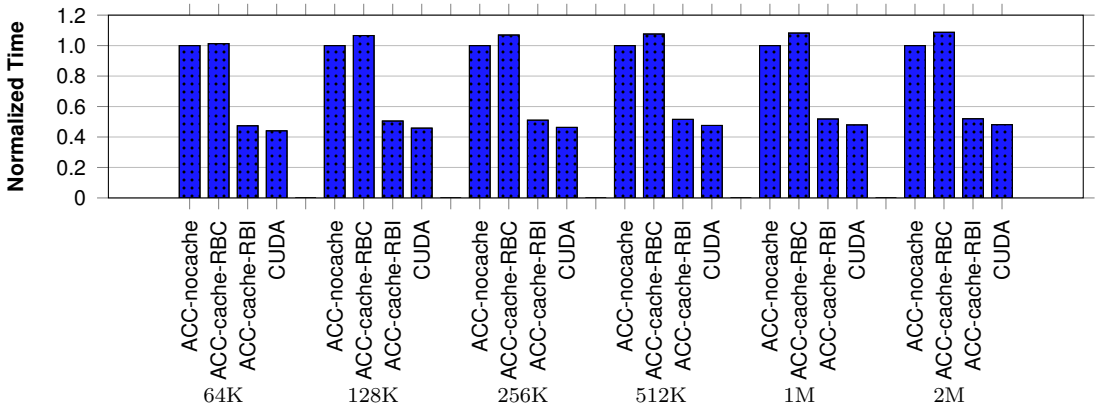


Figure 5.3: Comparing performance of four N-Body simulation implementations under different number of bodies.

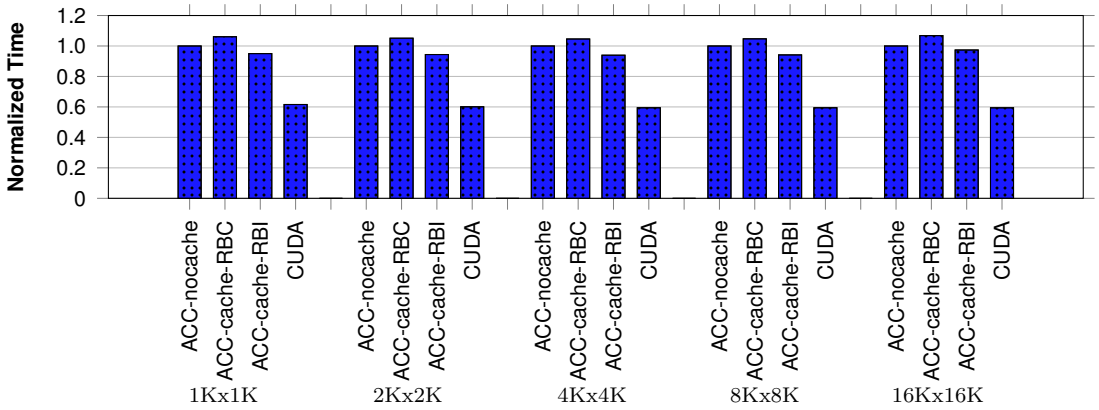


Figure 5.4: Comparing performance of four Jacobi iterative method implementations under different matrix sizes.

tiles and fetched into software-managed cache one tile at a time. This lowers redundant global memory instructions and DRAM accesses. RBI outperforms baseline OpenACC by 92%-111%. While RBI performs very close to CUDA, there is still a gap between them (8-10%). This gap is mainly the result of efficient implementation of the fetch routine in the CUDA version. RBC is unable to improve performance of the baseline OpenACC. This is explained by the overhead for accessing software-managed cache; i.e., assuring the address falls within the range of data existing in the shared memory.

Jacobi iterative method: Figure 5.4 compares four implementations of Jacobi iterative method under different problem sizes. Each thread in Jacobi reads nine neighbor elements (3-by-3 tile) and updates the value of the center element. Consider-

Table 5.3: Comparing occupancy of OpenACC without cache, OpenACC plus cache (RBC and RBI), and CUDA.

	GEMM	N-Body	Jacobi
OpenACC-nocache	100% (24, 0)	100% (32, 0)	100% (16, 0)
OpenACC-cache-RBC	75% (33, 4KB)	75% (30, 8KB)	100% (21, 1.2KB)
OpenACC-cache-RBI	100% (30, 4KB)	75% (30, 8KB)	100% (18, 1.2KB)
CUDA	100% (30, 4KB)	100% (32, 4KB)	100% (11, 1.2KB)

ing a two-dimensional matrix, calculations used by neighbor elements share significant amount of input data (four to six elements.) Fetching this data into software-managed cache and sharing data among threads is one way to optimize baseline OpenACC. We employ this in RBC, RBI, and CUDA implementations. Although our analysis shows RBC lowers global memory accesses, RBC harms overall performance when compared to the baseline. This is explained by the overhead (control flow and logical operations) of assuring addresses fall within the range of the data fetched into the shared memory. RBI removes this overhead and improves performance of baseline OpenACC by 3-6%. Despited this we observe a huge gap between RBI and CUDA. CUDA launches thread blocks equal in size to the size of the data being used by the thread block. RBI, however, launches thread blocks equal in size to the size of the computations being performed by the thread block. This results in the CUDA version using slightly larger thread block size than RBI. Here threads at the boarder of thread block are only used for fetching the data. This reduces irregular control flow in the fetch routine. We found that this can be effectively implemented in OpenACC to reduce the gap between RBI and CUDA. However, we did not investigate it further due to the high development effort required (close to CUDA equivalent), which is not desirable for high-level OpenACC.

Occupancy

Table 5.3 reports CUDA Occupancy of different implementations of GEMM, N-Body, and Jacobi test cases discussed above. The table reports occupancy in percentage and, within the parentheses, the first number reports registers used per thread and the second number report the size of *shared memory* used per thread block. All implementations have the same thread block size: 256 under N-Body and 16 by 16 under GEMM and Jacobi. Occupancy is 100% in most cases, meaning that GPU is able to run up to 2048 threads per GPU core (or Streaming Multiprocessor). There

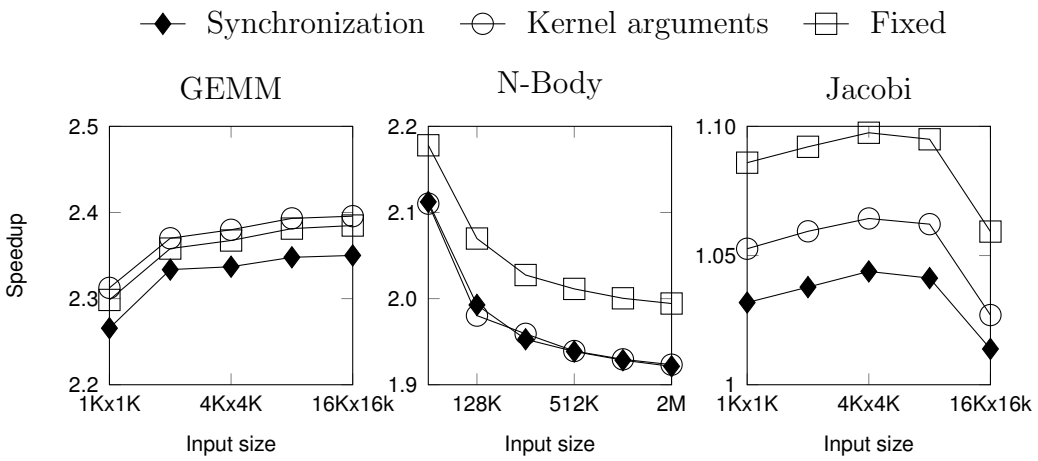


Figure 5.5: Comparing speedup from different finding sharing width methods. Numbers are normalized to the baseline OpenACC without using the cache directive.

are three cases where the occupancy is below 100%. RBC implementation of GEMM uses extra registers and that explains why occupancy drops below 100%. The size of cache after cache sharing is overestimated under RBC and RBI implementations of N-Body. This has lowered down the occupancy to 75%.

Implementation Alternatives

In Section 5.4.1, we reported performance of RBI and RBC under *kernel arguments* method of finding sharing width and *private* method of renewing cache scope. In this section we investigate performance of RBI under alternative methods for finding sharing width and renewing cache scope (very similar discussion applies to RBC as well.)

Find Sharing Width: We compare speedup from three alternative methods for finding sharing width (*kernel arguments*, *synchronization*, and *fixed*), under three test cases introduced earlier (GEMM, N-Body, and Jacobi). *Fixed* method simply sets the sharing width to the thread block size. *Kernel arguments* method uses a control-flow statement per dimension and sets the sharing width either to the thread block size or a pre-calculated number (obtained from kernel arguments). *Synchronization* method performs one reduction per dimension of subarray to find sharing width. As reported in Figure 5.5, *fixed* method performs fastest. Although *fixed* method is the fastest, it is not generally applicable. This is because compiler may not be able to statically guarantee that the total number of loop iterations is multiple of thread

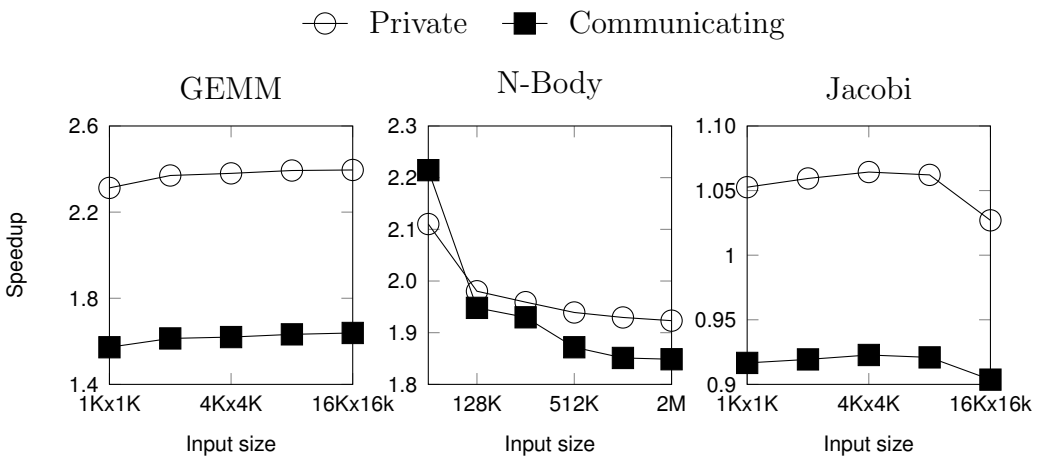


Figure 5.6: Comparing speedup from different renewing cache scope methods. Numbers are normalized to the baseline OpenACC without using the cache directive.

block size. If this is the case, *kernel arguments* method can be used instead of *fixed* method. We found that the performance gap between *fixed* and *kernel arguments* is 3-5%. *Synchronization* method performs slowest under all test cases as reported in Figure 5.5 and performs up to 3% slower than *kernel arguments*. Reductions slow down performance of *synchronization* significantly for multi-dimensional subarrays. This is the case in GEMM and Jacobi that use two-dimensional subarrays. In N-Body, however, one-dimensional subarray is being used and *synchronization* method performs close to *kernel arguments*.

Renew Cache Scope: We compare speedup from two alternative methods of renewing cache scope (*communicating* and *private*), under three test cases introduced earlier (GEMM, N-Body, and Jacobi). *Communicating* method shares cache pointers among threads of the thread block and calculates the new scope collaboratively. Slow down under *communicating* method is incurred by thread block synchronizations and read/writes from *shared memory*. *Private* method, however, locally calculates the new cache scope according to Equation 5.1 and 5.2 and avoids debilitating inter-thread communications. As shown in Figure 5.6, *private* method outperforms *communicating* method under all test cases, except under smallest dataset of N-Body. In this case, the number of parallel threads is relatively low and GPU cores complete inter-thread communication very fast (since synchronization instructions are infrequently hindered by other instructions [54]). This makes *communicating* method faster than *private* method in this case. Overall *private* method outperforms *communicating* method by up to 47%.

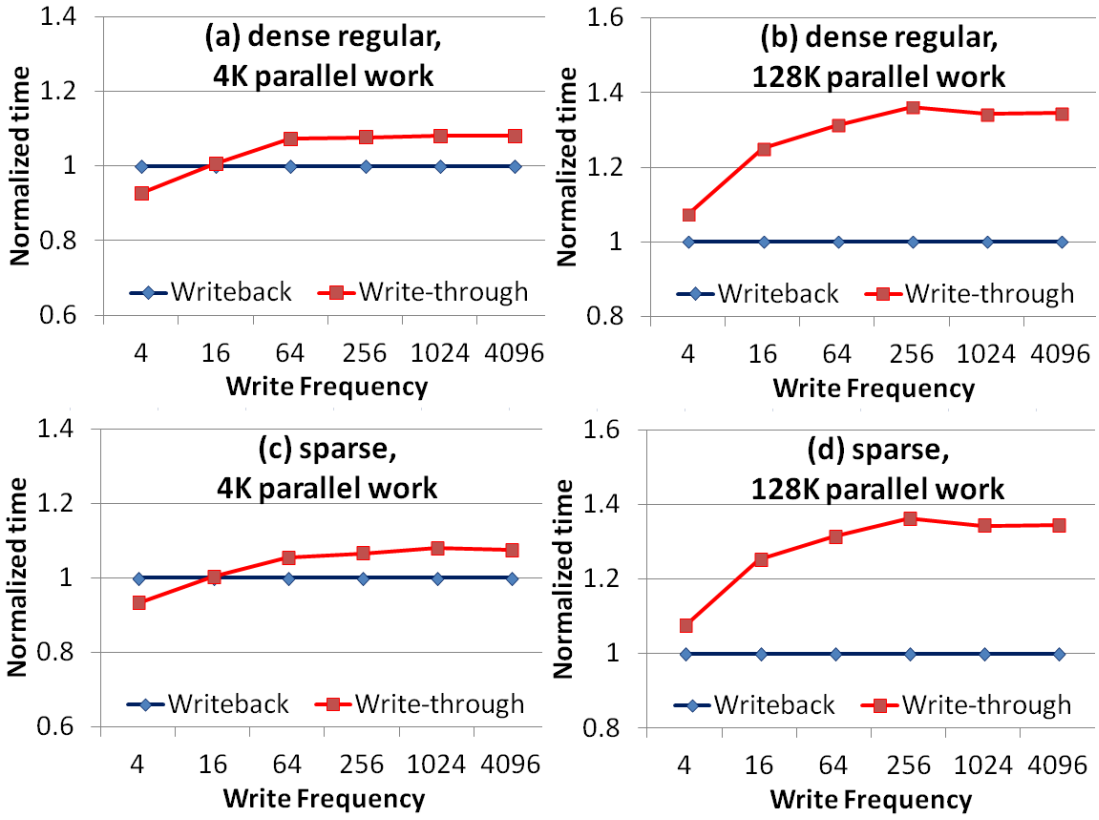


Figure 5.7: Comparing execution time of kernel under various shared memory configurations.

5.4.2 Cache Write

We developed two synthetic workloads to investigate performance of write-back and write-through policies. The first workload's write pattern is *dense* and *regular*. The workload is of 1D Stencil type where each parallel work computes an element in the output array, iteratively. In OpenACC terms, all parallel iterations are active (forming the dense pattern) and consecutive iterations write consecutive words (forming the regular pattern). Every parallel work serially iterates for a certain number of iterations (which is a run parameter) and computes the value of the element iteratively. The second workload is the same as the first, except that only a fraction of threads are active (less than 2%) and only a fraction of serial iterations perform write (less than 2%). This forms the *sparse* pattern.

Parameters of these workloads are parallel iterations (total number of work) and number of serial iterations within the work. The number of serial iterations models the frequency of cache writes. Sweeping this number from 4 to 4096, we measure the

performance of write-back and write-through under various cache access frequencies.

Figure 5.7 compares write-back and write-through under the two synthetic workloads described above (*dense regular* versus *sparse*). Two problem sizes are reported for each workload, 4K and 128K parallel work. We observe a similar trend under both workloads. When parallel work is massive in size (e.g. 128K work), write-back is faster than write-through (Figure 5.7b and 5.7d). This is due to the fact that large amount of threads can perfectly hide the latency of write-back's final write routine. When parallel work is small in size and write frequency is low (e.g. left side of Figure 5.7a and 5.7c), write-through outperforms write-back. For example in Figure 5.7a, write-through is faster when write frequency is lower than 16. Going beyond 16, write-back starts to catch up with write-through. This can be explained by the higher rate of global memory writes that write-through makes. For large write frequencies (e.g. >64), write-through performs numerous redundant writes to global memory. Write-back, in contrast, buffers intermediate written values (in shared memory) and writes them all to global memory once at the end of cache region. This reduces the total global memory writes compared to write-through and saves performance. As presented, the performance gap between write-back and write-through increases from 7% to 34%, as write frequency increases.

5.4.3 Performance Portability

Performance portability is one of the most important motivations of using OpenACC directives. Here we focused on devising efficient implementation of the *cache* directive on the most commonly used platform [66, 11, 57], NVIDIA GPUs. Intuitively, we believe very similar optimization strategies can be followed on other similar architectures, e.g. AMD GPUs [3], to devise an efficient implementation of the *cache* directive. We leave optimization strategies on different platforms to future work.

To show the performance portability across NVIDIA GPUs, here we evaluate our implementation on a different NVIDIA GPU, Quadro K600. In Table 5.4 we report performance improvement from RBI implementation over the baseline OpenACC (without cache) under three benchmarks: GEMM, N-Body, and Jacobi. We limit the evaluations to single dataset per benchmark (largest dataset that could fit in the memory of Quadro K600). For RBI configuration, we assume *private* method for renewing cache scope and *kernel arguments* method for finding sharing width. As shown in the table, improvements are very close. Improvements are slightly larger

Table 5.4: Performance improvement from RBI over the baseline OpenACC (without cache).

	Tesla K20c	Quadro K600
GEMM	238.0%	255.1%
N-Body	198.0%	211.4 %
Jacobi	6.4%	2.5%

under Quadro K600 for GEMM and N-Body benchmarks. This can be explained by the difference in the memory bandwidth of Quadro K600 and Tesla K20c. DRAM memory bandwidth of Quadro K600 is 29 GB/s which is 7.1 times lower than the bandwidth of Tesla K20c (208 GB/s). Accordingly, Quadro K600 is more sensitive to the techniques that optimize memory accesses. The *cache* directive is an example of these techniques and returns higher performance improvement when the memory bandwidth is throttled (e.g. Quadro K600).

5.5 Discussion

5.5.1 EHC in CUDA

In EHC, tag and data arrays should be kept consistent. This limits cache sharing and generally parallelism of software-managed cache operations, specially write operations. For instance, if two threads miss different data and want to fetch both into the same location, synchronization is necessary. The synchronization overhead can be significant as the only way to handle such scenarios is to create a critical section or use atomic operations. Because of this limitation, for performance goals, cache sharing optimizations should be avoided on top of EHC. We exclude EHC from evaluations as we did not find it competitive.

5.5.2 Optimizing RBC

In RBC, *__cache_read* routine is the performance limiting factor, listed in Listing 5.3. Investigating the CUDA assembly of the kernel (in *sass* format), we found that the compiler eliminates branches and instead uses predicates. This, on the positive side, eliminates extra operations for managing the post-dominator stack [19]. On the negative side, all instructions, in both taken and not taken paths of the branch, are

at least fetched, decoded, and issued (some are executed as well). The *nvcc* compiler uses a heuristic to employ predicates or generate control flow statements (we describe this in Section 5.4.2 of [67]). For *__cache_read* routine of RBC, the heuristic finds predicate advantageous. However, the overhead of the predicate version is still huge and the routine is translated to 16 machine instructions. This explains why RBC is slow. We believe further optimizations on RBC should be performed at the machine level.

5.5.3 Alternative cache targets

NVIDIA GPUs have alternative on-chip caches that can be used by OpenACC compiler as the target of the *cache* directive (e.g. constant memory and texture cache) or can be used effortlessly as an alternative to the *cache* directive (L1 cache and read-only cache). Constant and texture memory are limited to read-only data. If the subarray is written in the cache region, constant and texture memory can not store the latest value nor deliver the latest to subsequent requests. In addition, the precision of the application could be affected if texture memory is used. We evaluated the performance impact of L1 and read-only caches separately. We enforced read-only cache using *const* and *__restrict__* keywords and forced the GPU to cache global accesses through *nvcc* compile flags (*-Xptxas -dlcm=ca*) and found out that performance improvements are less than 2%. This suggests that the advantages of using software-managed cache is not limited to reading/writing data from/to faster cache, but also accessing the data in fewer transactions and in a coalescing-friendly way.

5.5.4 Explicit mapping

OpenACC API accepts hints from the programmer to explicitly specify the mapping of loop iterations to different thread blocks (*gang* clause) or the same thread block (*worker* and *vector* clauses). In this case, the compiler should generate a specific mapping of parallel loops to CUDA thread hierarchies, forced by *gang*, *vector*, and *worker* clauses. This can limit the range of compiler optimizations in sharing the cache space among threads. Generally, as long as the mapping enforced by the clauses is a *valid configuration* and *does not have conflict* with the outcome of the compiler pass we propose in Listing 5.5, the compiler proceeds and exploits the sharing opportunity. Invalid configuration is created when the sharing range is larger than the CUDA

shared memory size. This can be enforced by *vector* and *worker* clauses that map loop iterations to threads of one thread block and change the thread block size across x and y dimensions, respectively. The conflict mostly occurs when *gang* clause is used. *gang* clause asks the compiler to map each iteration to a thread block. This can have conflict with the compiler pass we presented in Listing 5.5, if the compiler decides to map this loop to threads of the thread block. In the case of conflict, the compiler can limit the sharing range, e.g. sharing only across one dimension of the grid and ignoring the sharing along the *gang* loop, or even ignoring the sharing optimization, in the worst case.

Alternative cache implementations To the best of our knowledge, currently there are no commercial or open source OpenACC compilers that support the *cache* directive. Therefore, we are unable to compare performance of our implementation to other studies. We studied several compilers (i.e. PGI and Omni) but found none of them supporting the *cache* directive. We compiled the kernels with PGI Accelerator compiler 16.1 and found out that the compiler ignores the *cache* directive and does not generate shared memory CUDA code. We also investigated several open source frameworks, e.g. RoseACC, accULL, and Omni compiler, of which none had an implementation for the *cache* directive.

5.5.5 Cache Coherency

As stated by OpenACC specification [77], it is possible to write an accelerator parallel/kernels region that produces inconsistent numerical results. This is because some accelerators (e.g. GPUs) implement weak memory model. In weak memory model, memory coherency is not supported between operations executed by different threads nor between subsequent operations of a single thread, unless operations are separated by an explicit memory fence. This is the programmer's task to ensure the correctness of the application is not compromised by the weak memory model. Similarly, since variables and subarrays listed in the *cache* directive are part of the memory hierarchy, we assume weak memory model for the *cache* directive as well.

We explain the behavior of our implementation of the *cache* directive under this weak memory model under two major scenarios: i) one thread has the copy, the thread writes to the copy, and all threads attempt to read and ii) multiple threads have copies, multiple threads write to their copy, and all threads attempt to read. The behavior is summarized in Table 5.5. The behavior is very similar to CUDA behavior

when multiple threads attempt to write to the same memory location asynchronously. Here we assume write-back policy for the cache implementation.

The first scenario assumes that one thread has a copy of a data in its cache, and no other thread has a copy of this data. Write operations to this local copy (by the thread) are immediately visible to the thread (marked as #1 in the table). As the thread leaves the cache region, the global memory will be updated by the latest copy available in the cache. While the kernel is running, threads that complete their read access before this update read the old value. Threads that complete their read access after this update read the new value. This behavior is marked as undefined in the table (#2). Once the kernel runs all the threads and completes its execution, the global memory has the new value rewritten by the thread (#3 in the table).

The second scenario assumes that multiple threads have copies of a data in their cache and one or more threads write their local copy. Since our implementation shares the cache among a range of threads (thread block), write operations from other threads to their local copy is visible to the range of threads that share the cache. Accordingly, a thread might read the value written by itself or other threads (#4 and #5 in the table). Our implementation guarantees that one write from the threads updates the cache, but the thread that updates the cache is unspecified. Similar to the first scenario, once a thread leaves the cache region, the global memory will be updated by the latest copy that is in the cache. Once the kernel completes its execution, the global memory has the new value written by a thread, but again the thread is unspecified.

Overall, here the behavior for the cached and uncached data is the same. In either case, if multiple threads write to the same memory location, one thread successfully updates the global copy, but that thread is unspecified. Accordingly, future reads from that location might return undefined value.

5.6 Summary

In this chapter, we studied and addressed the challenges facing the OpenACC *cache* directive in NVIDIA GPUs. We used CUDA *shared memory* as the software-managed cache space for implementing the directive. We presented three different methods and several performance optimizations for implementing the *cache* directive, among which sharing the cache space among multiple threads and parallelizing cache fetch

Table 5.5: Behavior of our weak memory model cache directive implementation under two scenarios: one write multiple reads and multiple writes multiple reads.

Scenario	**During the kernel**		**After the kernel**
	Seen by threads that have a copy	**Seen by other threads**	**Seen by all threads**
Only one thread has a copy and the thread writes to the copy	(#1) new value	(#2) undefined: old value or new value	(#3) the new value
Multiple threads have copies and one or more threads write to the copy	(#4) undefined: old value or new value from any copy	(#5) undefined: old value or new value from any copy	(#6) undefined: the new value from one unspecified thread

and write routines are the most critical. Our results also show that i) sharing the cache among several parallel threads is essential to have a robust performance and ii) write-back cache outperforms write-through policy for the sparse and dense memory patterns. Evaluating under matrix-matrix multiplication, N-Body simulation, and Jacobi method iteration test cases, we presented an implementation that can perform close to hand-written CUDA.

Chapter 6

Software-Managed Cache for OpenACC

In this chapter, we introduce a new OpenACC directive to increase OpenACC flexibility in utilizing software-managed cache (SMC). We first discuss the limitations of the *cache* directive in utilizing software-managed cache. Then we propose a new directive, referred to as *fcw* directive (fetch, communicate, and writeback), to address these limitations. Then we present our experimental results in applying the directive on a wide set of benchmarks. Then we discuss issues related to the use of the fcw directive. Finally, we summarize our findings.

6.1 Limitations of the cache Directive

The *cache* directive allows an OpenACC thread to exploit GPU's SMC. OpenACC programmer can optimize memory accesses by reading and writing from the SMC. However, this is not the only use case of SMC. The *cache* directive dedicates a cache space to each thread and different threads may not share the intermediate results. In CUDA programming model, *certain range* of threads may share their cache space and use each other's intermediate results. This use of SMC is crucial since it saves costly global communications.

In the rest of this section, we investigate 1D stencil example to provide more insight on limitations of the *cache* directive. Listing 6.1 shows 1D stencil in OpenACC[1]. Total

[1]In the listings presented in this section, for the sake of code readability, we exclude the code related to corner case handling.

of *iterations* kernels are launched on the GPU. Each kernel has total of N threads. Each thread in the kernel reads three words and writes one word, for the total of $iterations \times N \times (3+1)$ global memory instructions.

Listing 6.1: 1D Stencil in OpenACC.

```
#define BLOCK_SIZE 256
void stencil(int *src, int *dst, int N){
  #pragma acc kernels loop independent vector(BLOCK_SIZE)
  for(int xidx=0; xidx<N; ++xidx)
    dst[xidx]= (src[xidx-1]+src[xidx]+src[xidx+1])/3;
}
main(){
  for (int t = 0; t < iterations; ++t){
    int *temp = src; src = dst; dst = temp;
    stencil(src, dst, N);
  }
}
```

The same algorithm as Listing 6.1 can be implemented in CUDA. This is presented in Listing 6.2.

Listing 6.2: 1D Stencil in CUDA.

```
#define BLOCK_SIZE 256
__global__ void stencil(int *src, int *dst, int N){
  int xidx = threadIdx+blockIdx.x*BLOCK_SIZE;
  dst[xidx]= (src[xidx-1]+src[xidx]+src[xidx+1])/3;
}
main(){
  for (int t = 0; t < iterations; ++t){
    int *temp = src; src = dst; dst = temp;
    stencil<<<N/BLOCK_SIZE, BLOCK_SIZE>>>(src, dst, N);
  }
}
```

Ghost zone optimization for GPUs has been proposed [59] to reduce kernel launches and global memory accesses. Ghost zone optimization suggests trading global memory accesses for redundant computations. Since memory access is expensive and compu-

tation is cheap in GPUs, this optimization returns significant benefits on the GPUs. Listing 6.3 presents 1D stencil with ghost zone optimization in CUDA. Compared to Listing 6.1 and 6.2, the number of kernel launches is lowered by *ghostzoneSize* times (this lowers the number of global memory reads and writes.). Instead, this version imposes higher number of thread blocks and executed more instructions per thread (less expensive instructions though). In each kernel call, each thread makes two global memory accesses (one read and one write), for the total of $\frac{iterations \times N \times 2}{ghostzoneSize}$. In this algorithm, *ghostzoneSize* is the knob to tune the application for the best performance; the larger the value the lower the kernel calls and the higher the redundant computations, the smaller the value the higher the kernel calls and lower the redundant computation. The key difference between ghost zone implementation and Listing 6.1 and 6.2 is that threads communicate within-the-kernel through the SMC channel instead of across-the-kernels through global memory channel.

Ghost zone optimization may not be implemented in OpenACC using the *cache* directive since *there is not a defined communication model among iterations of a parallel work-sharing loop.* We propose a new directive and well-defined communication model to enable this functionality of SMC in OpenACC. Using the proposed directive, *certain range* of iterations will be able to share data within the kernel through SMC.

6.2 Proposed Directive

The *fcw* directive is applied over a region of code and is only available within kernels/parallel regions. This directive serves as a compiler hint to fetch a data chunk into local software-managed cache (SMC), replace global memory accesses with SMC accesses, allow concurrent accelerator threads to communicate through SMC, and write the local data back to global memory. In Section 6.4, we explain differences between the *fcw* and *cache* directives.

The motivation behind fcw stems from the fact that consequent iterations of a parallel loop often share data. fcw allows neighbor iterations to share intermediate data. It allows fast communication between neighbor iterations through SMC, replacing costly global memory communication/synchronization. In addition, the compiler can mitigate redundant memory fetches by exploiting programmers' hints. The programmer specifies the data that is shared among consequent iterations. This information guides the compiler to fetch common data by few well-coalesced requests, preventing

Listing 6.3: 1D Stencil with ghost zone optimization in CUDA.

```
#define IN_RANGE(x, min, max) ((x)>=(min) && (x)<=(max))
#define BLOCK_SIZE 256
__global__ void stencil_ghostzone(int *src, int *dst, int ghostzoneSize,
    int N){
  __shared__ int prev[BLOCK_SIZE];
  int bx = blockIdx.x; int tx = threadIdx.x;
  int small_block_cols = BLOCK_SIZE-ghostzoneSize*2;
  int blkX = small_block_cols*bx-1; int xidx = blkX+tx;
  if(IN_RANGE(xidx, 0, N-1)) prev[tx] = src[xidx];
  bool computed=false;
  for (int i=0; i<ghostzoneSize; i++){
    computed = false;
    if( IN_RANGE(tx, i+1, BLOCK_SIZE-i-2)){
      computed = true;
      int avg = (prev[W]+prev[tx]+prev[E])/3;
      __syncthreads(); prev[tx] = avg; __syncthreads();
    }
  }
  if (computed) dst[xidx]=prev[tx];
}
main(){
  int smallBlockCol = BLOCK_SIZE-ghostzoneSize*2;
  for (int t = 0; t < iterations; t+=ghostzoneSize){
    int *temp = src; src = dst; dst = temp;
    stencil_ghostzone<<<N/smallBlockCol, BLOCK_SIZE>>>(src, dst,
      ghostzoneSize, N);
  }
}
```

repeated global memory fetches, and reusing data at the SMC bandwidth.

fcw gives the programmer the option to specify the shared data by denoting the range of indexes that need to be cached in SMC. This range can be specified with respect to constant terms or even the induction variable of the outer parallel work-sharing loop. In either case, the data of neighbor iterations may construct a continuous memory chunk. The compiler generates a code to fetch this chunk from global memory to SMC. Future memory accesses within this chunk use SMC bandwidth, instead of global memory. Otherwise, if the range expression is derived from constant terms, the same data can be fetched/shared among all iterations of the loop.

A common scenario occurs when fcw is applied within a parallel loop where the

range identifiers depend on the loop induction variable. In this case, the range of indexes for an iteration may overlap with the range of indexes evaluated on its neighbor iterations. The union of this overlapping data creates a consequent chunk of data. Building on this fact, fcw can be implemented efficiently, if the OpenACC implementation executes OpenACC applications over CUDA API. Under OpenACC to CUDA translations, neighbor iterations of a parallel work-sharing loop usually map to consequent threads of the same thread block [91]. In CUDA terms, a thread block is the collection of certain number of consequent iterations. Therefore, CUDA shared memory can serve efficiently as SMC to share data among the iterations of a parallel loop (or threads of the same thread block.)

In the remainder of this section, we define fcw's programming interface and present examples to clarify the notation. Finally, we present a case study to show how the notation from OpenACC can be translated to CUDA.

6.2.1 Programming Interface

We introduce fcw by the following notation:

#pragma acc fcw SMC_TYPE(arrayName[PIVOT : BEFORE : AFTER])

where the parameters are described below.

arrayName is a variable name pointing to the memory location intended to be cached in SMC.

PIVOT, *BEFORE*, and *AFTER* are guides to identify the range of data required to be cached in SMC. PIVOT denotes the central index. Regardless of BEFORE and AFTER, the compiler generates a code to cache the element in the PIVOT position. BEFORE/AFTER denotes the number of elements before/after PIVOT that should be fetched into SMC. These parameters are best explained through an example, as we show in Table 6.1. In multidimensional arrays, the tuple of {PIVOT, BEFORE, AFTER} can be replicated to point different dimensions of an array.

fcw can have different types which is specified by *SMC_TYPE* clause. Although all fcw types access the specified memory region at the bandwidth of SMC, they differ based on three further operations: i) initializing the SMC data from global memory (fetch), ii) making intermediate writes visible to neighbor iterations (communication channel), and iii) reflecting local changes back to global memory (write-back). Whether necessary to activate each of these three operations or not, there are eight possible fcw types available to programmer. The eight types combine

all possible combinations of fetching data from global memory to SMC (or not), creating communication channels (or not), and writing back data to global memory (or not). We investigated several CUDA benchmarks to find the most common combinations. Based on the studied workloads, we found a subset of five combinations to be the most common among the eight: FETCH_ONLY, CHANNEL_ONLY, FETCH_CHANNEL, CHANNEL_WB, and FETCH_CHANNEL_WB. In FETCH_ONLY, the compiler only fetches the data from global memory to SMC. In this case, no local communication channel is created and writeback are not performed. In CHANNEL_ONLY, the compiler neither fetches data from global memory to SMC nor writes it back to global memory after exiting the region. This mode is only useful for fast interchanging/sharing intermediate data among loop iterations through SMC. FETCH_CHANNEL is similar to CHANNEL_ONLY but also fetches/initializes the corresponding data from global memory to SMC. CHANNEL_WB is similar to CHANNEL_ONLY but also writes the final data back to global memory. FETCH_CHANNEL_WB performs all of the three operations. It fetches/initializes the data into SMC, creates a communication channel among neighbor iterations, allows intermediate writes to be visible to neighbor iterations, and writes the final value of the SMC back to global memory. CHANNEL_ONLY, FETCH_CHANNEL, CHANNEL_WB, and FETCH_CHANNEL_WB create a local communication channel and allow intermediate changes to SMC to be visible to all threads within the same thread block.

6.2.2 Communication Model

fcw allows fast communication through SMC. We refer to this new communication model as inter-iteration communication. Inter-iteration communication allows communication among iterations of a parallel work-sharing loop.

In this model, at the high-level programmer view, all parallel iterations are grouped into communicating thread blocks, in which every iteration belongs to only one communicating thread block. Iterations within the same communicating thread block can share data via SMC. To avoid data hazards, implicit synchronizations are performed after every write to the location marked as SMC. Also the number of iterations belonging to the same communicating thread block, or the communicating thread block size, can be configured.

CHANNEL_ONLY, FETCH_CHANNEL, CHANNEL_WB, and FETCH_CHANNEL_WB

allow various communication patterns. CHANNEL_ONLY creates the basic communication medium. It constructs a communication medium within a communicating thread block using SMC bandwidth. Iterations can share intermediate results through this fast cache. FETCH_CHANNEL, unlike CHANNEL_ONLY, also fetches/initializes SMC from global memory. CHANNEL_WB is the same as CHANNEL_ONLY except it also writes back the final state into global memory. Finally, FETCH_CHANNEL_-WB is different from CHANNEL_WB, as it allows initializing SMC by fetching from global memory. Notice that FETCH_ONLY only allows data reuse at SMC bandwidth and does not construct a communication channel within a communicating thread block.

Since data hazards and race conditions may occur, sharing SMC among iterations of the same communicating thread block needs inter-iteration synchronization. Under inter-iteration communication, there are two kinds of synchronizations: implicit and explicit. For fcw types including FETCH operation, there is an implicit synchronization between iterations fetching data for a thread block. This is implicit and hidden from the programmer as the compiler generates the associated code automatically. Also fcw types that include CHANNEL operation contain implicit synchronization, which is forced by the writes to SMC. For the data chunk that is marked in fcw, the programmer can assume one barrier call before every individual memory write and one after it. We refer to this as implicit synchronization since it is performed implicitly by the compiler, hidden from the programmer. While implicit synchronization satisfies the need for synchronization, programmers can use explicit synchronization to control synchronization selectively. We add the following directive to OpenACC API, allowing manual inter-iteration synchronization: *#pragma acc fcw_barrier*. *fcw_barrier* synchronizes the iterations of the communicating thread block and assures that all previous SMC writes are completed. Notice that *fcw_barrier* can be implemented through CUDA thread block synchronization barriers.

6.2.3 Example

Table 6.1 illustrates two examples of configuring PIVOT, BEFORE, and AFTER parameters. In these examples, *a* is a pointer to a read-only array, targeted to be cached in SMC. As specified by the first parameter, fcw type is FETCH_ONLY. 6.1(a) lists an example where the required data simply includes one element, pointed to by the loop induction variable. In this case, the pivot element is i (the loop induction

Table 6.1: Examples of determining range identifiers to direct the compiler for SMC.

Example	a pivot	a before	a after

(a)
`#pragma acc kernels loop`
`for(i=1; i<(size-1); ++i)`
`#pragma acc fcw FETCH_ONLY(a[i:0:0])`
`{b[i]=a[i];}`
	i	i	i

(b)
`#pragma acc kernels loop`
`for(i=1; i<(size-2); ++i)`
`#pragma acc fcw FETCH_ONLY(a[i:1:2])`
`{b[i]=a[i-1]+a[i]+a[i+2];}`
	i	i-1	i+2

variable) and fcw guides the compiler to fetch only pivot (i:0:0). In 6.1(b), the pivot element is set to i and fcw directs the compiler to fetch a range from one element before the pivot to two elements after the pivot (i:1:2).

In these examples, the required data of consequent iterations construct a continuous memory chunk. For example in 6.1(b), the required data of every consequent 256 iterations is 259 elements. Since the threads of the same thread block execute consequent iterations, using the information provided through fcw, the compiler can identify the data of each thread block and map the corresponding memory chunk into SMC. With a mapping that is invisible to the programmer, the compiler replaces every individual array access with an SMC access. The compiler also generates proper fetch and writeback, based on fcw type provided through SMC_TYPE.

6.2.4 Case Study: Reduction

In this section, we overview our automated code generation through a sample. Our goal is to show how fcw notation can be translated from OpenACC to CUDA. We present a reduction case study which also shows the applicability of fcw in implementing basic parallel algorithms. Listing 6.4 shows the fcw-enhanced OpenACC version. Listing 6.5 and 6.6 show the CUDA version translated from Listing 6.4. Listing 6.5 shows the host code and Listing 6.6 shows the accelerator code.

The code shown in Listing 6.4 calculates the reduction in several kernel launches. The outer data directive copies input arrays (*a[]* and *b[]*) to accelerators memory

Listing 6.4: Reduction in fcw.

```
01:    uint TILESIZE=256, TILESIZEB=8;
02:    #pragma acc data copy(a[0:N],b[0:N])
03:    for(P=N; P>1; P=P/TILESIZE+(((P%TILESIZE)!=0)?1:0)){
04:      int *c=a; a=b; b=c;
05:      #pragma acc kernels present(a,b)
06:      #pragma acc loop independent vector(TILESIZE)
07:      for(i=0; i<P; i++){
08:        int tileid =i>>TILESIZEB;
09:        int offsetid=i&(TILESIZE-1);
10:        int k=N;
11:        #pragma acc fcw FETCH_CHANNEL(a[i:0:0])
12:        {
13:          for(y=TILESIZE/2; y>0; y>>=1)
14:            if((offsetid<y) && ((i+y)<P))
15:              a[i]=a[i]+a[i+y];
16:          if(i==(tileid*TILESIZE))
17:            b[tileid]=a[i];
18:        }
19:      }
20:    }
```

before the data region and copies them back to hosts memory after the region. Within the data region, *kernels* region is translated into a procedure call on accelerator. The kernel reduces every 256 elements (TILESIZE) of *a[]* and stores the summation into *b[]*. The outer loop iterates the kernel launches. In every iteration, the code swaps *a[]* and *b[]* pointers, shrinks the size of arrays by a factor of TILESIZE, and launches the kernel to reduce the new *a[]*. This process relapses until the size of *a[]* shrinks below TILESIZE, returning the final summation in *b[0]*.

In every kernel call, a total of P threads (equal to the current *a[]* size) are launched on the device. The fcw directive guides the compiler to fetch a tile of elements for each thread block into shared memory. Threads of the same thread block calculate the sum of these elements at the bandwidth of SMC and write the final reduced value to *b[]*. The tile size is set to TILESIZE by *vector* clause of the loop directive.

Listing 6.5: Host code of translation of fcw Reduction to CUDA.

```
01:    #define TILESIZE 256
02:    acc_create((void*)a,(N+0)*sizeof(int));
03:    acc_create((void*)b,(N+0)*sizeof(int));
04:    acc_copyin((void*)a,(N+0)*sizeof(int));
05:    acc_copyin((void*)b,(N+0)*sizeof(int));
06:    P = N;
07:    for(P=N; P>1; P=P/TILESIZE+(((P%TILESIZE)!=0)?1:0)){
08:      int *c = a; a = b; b = c;
09:
10:      /* kernel call statement */
11:      {
12:        generated_kernel
13:         <<<((abs((int)((P))-0))/(1))/TILESIZE+1,TILESIZE>>>
14:          ((int*)acc_deviceptr((void*)a),
15:          (int*)acc_deviceptr((void*)b),
16:          N, P);
17:      }
18:      cudaError err=cudaDeviceSynchronize();
19:    }
20:    acc_copyout((void*)a,(N+0)*sizeof(int));
21:    acc_copyout((void*)b,(N+0)*sizeof(int));
```

Listing 6.5 shows the CUDA code obtained by automatic translation of Listing 6.4 OpenACC. data directive memory copies are translated to OpenACC API calls, e.g. *acc_create*, *acc_copyin*, and *acc_copyout*. The code included in the kernel region of Listing 6.4 is translated to a CUDA kernel named *generated_kernel*. For launching this kernel, the thread block size is derived from *vector* clause of the parallel loop. Also the number of thread blocks is the number of loop iterations divided by the thread block size.

Listing 6.6 shows the kernel code translated from the *kernels* region of Listing 6.4. At the beginning lines of the code (line #2), a variable is declared to obtain the global identifier of the thread along the parallel loop. This is followed by the declaration of loops' induction variable (line #3), shared memory space (line#5), and two pointers to locations in global memory (line #6). Loops' induction variables are declared within the kernel locally (allowing the compiler to reserve a register for

each of them). The size of shared memory is the sum of thread block size (TILESIZE), number of elements before the PIVOT (BEFORE field in fcw directive of Listing 6.4, which is 0), and number of elements after the PIVOT (AFTER field in fcw directive of Listing 6.4, which is also 0). These are specified by the programmer via *vector* and *fcw clauses*, respectively. Two pointers, which are assigned later in the code (line #10 to #13), point to the start and end addresses of the global memory data chunk (*a[]*) which is mapped to the shared memory space (*__SMCdata_a*) of each thread block. The start and end pointers point to the first and last threads of the thread block, respectively. The body of parallel loop is injected within the kernel code (line #8 to #35). At beginning code block, each thread calculates the value of the loop induction variable which corresponds to its thread's global identifier (line #3). Next, the *if* statement assures the parallel boundary and guards the loop body from extra threads (line #7). As shown in Listing 6.4, the fcw type is FETCH_CHANNEL. This guides the compiler to initialize the shared memory by fetching the data from global memory. FETCH_CHANNEL also guides to create a channel to allow communication among the threads of the thread block and make intermediate writes locally visible. The body of the loop consists of three code portions: FCW fetch (line #9 to #19), index precalculations (line #20 to #22), and FCW regions. FCW fetch calculates the boundary of the tile assigned to the thread block (line #10 to #13). Line #14 to #18 use parallel threads of the thread block to fetch a data chunk from global memory to shared memory. Compiler statically finds the unique memory indexes accessed within fcw region and injects a code to precalculate their mapping from global space to shared space at runtime (line #20 and #22). To create the communication channel within the fcw region, accesses to *a[]* are replaced by accesses to *__SMCdata_a*. An intra-thread block synchronization call is injected before and after every write to SMC location.

Listing 6.6: Accelerator code of translation of fcw Reduction to CUDA.

```
__global__ void generated_kernel(int* a,int min,int N,int P,int* b){
 int __kernel_getuid_x=threadIdx.x+blockIdx.x*blockDim.x;
 int i=0+(__kernel_getuid_x);
 /* declare the shared memory of a */
 __shared__ int __SMCdata_a[256+0+0];
 int __SMCstartptr_a; int __SMCendptr_a;
 if(i<P){
  int tileid = (i >> TILESIZEB); int offsetid = (i & (TILESIZE - 1));
  { // FCW fetch begins
   __SMCstartptr_a=i-0-threadIdx.x;
   int __ipmacc_stride=blockDim.x;
   bool lastcol= blockIdx.x==(gridDim.x-1);
   __SMCendptr_a=(lastcol)?P-1:blockDim.x+__SMCstartptr_a-1+0;
   int __ipmacc_length=__SMCendptr_a-__SMCstartptr_a+1;
   int kk=threadIdx.x;
   int idx=__SMCstartptr_a+kk;
   if(idx<(P) && idx>=(0)) __SMCdata_a[kk]=a[idx];
   __syncthreads();
  } // end of FCW fetch
  // pointer mapping
  int __SMCidx_a_0_dim1 = i-__SMCstartptr_a;
  int __SMCidx_a_1_dim1 = i+y-__SMCstartptr_a;
  { // FCW region
   for(int y = min; y > 0; y >>= 1){
    if ((offsetid < y) && ((i + y) < P)) {
     int sum = __SMCdata_a[__SMCidx_a_0_dim1] /* replacing a[i]*/
            + __SMCdata_a[__SMCidx_a_1_dim1] /* replacing a[i + y]*/ ;
     __syncthreads();
     __SMCdata_a[__SMCidx_a_0_dim1]= sum;
     __syncthreads();
    }
   }
   if (i == (tileid * TILESIZE))
     b[tileid] = __SMCdata_a[__SMCidx_a_0_dim1] /* replacing a[i]*/ ;
  } // end of FCW region
 }
}
```

6.3 Experimental Results

In this Section, we first investigate performance under fcw directive compared to baseline OpenACC and CUDA. Then we compare the development effort of these implementations. Finally, we investigate fcw's performance sensitivity to thread block size changes.

We use our IPMACC [44] for compiling OpenACC applications and implementing the *fcw* directive. We run evaluations under NVIDIA Tesla K20c GPU. The execution time of the kernel is measured by nvprof [14]. Every number is harmonic mean of 30 independent samples.

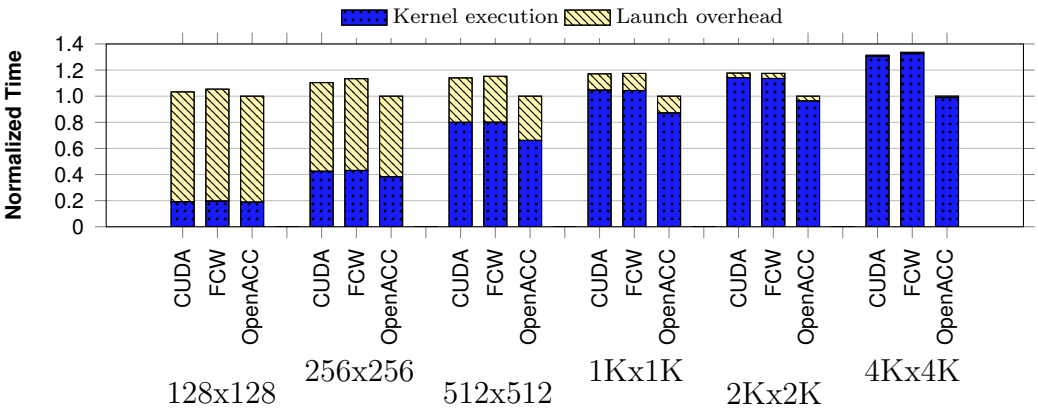
6.3.1 Performance

In this section, we compare the execution time of OpenACC, OpenACC+fcw, and CUDA. Among all the benchmarks investigated in Section 3.3.1, we limit the study to the benchmarks that utilize software-managed cache in their CUDA version. We investigate Hotspot, Pathfinder, Dyadic Convolution, N-Body, Matrix-matrix Multiplication, and Needleman-Wunsch benchmarks. We do not report memory transfer time as memory transfer times are comparable in all three implementations.

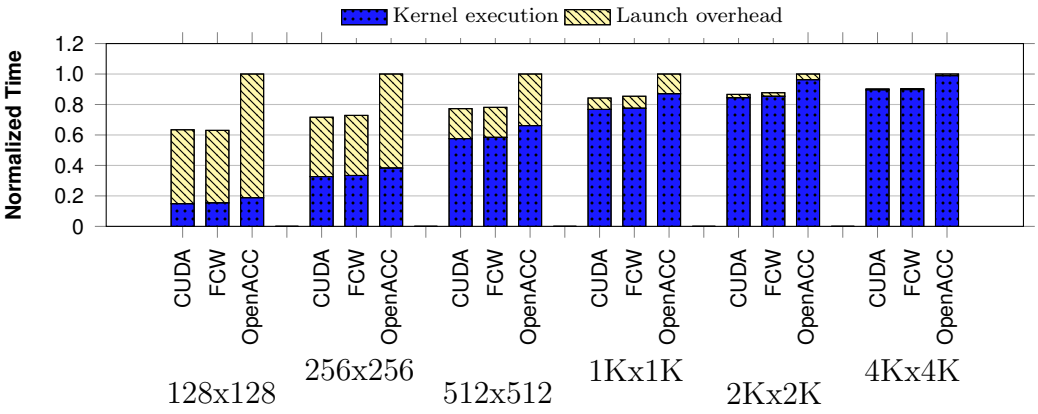
Hotspot

Figure 6.1 compares three implementations of Hotspot under different problem sizes, ranging from 128x128 to 4Kx4K. We run Hotspot for 12 iterations. CUDA and FCW implementations use the same algorithm referred to as *ghost zone optimization* [59]. This algorithm exploits SMC to reduce the number of global synchronization (or kernel launches). In ghost zone optimization, thread blocks *recalculate* their input data, instead of performing *global synchronization* and *reading* the output of other thread blocks. Ghost zone optimizations create halo regions around the tile assigned to each thread block. Halo regions overlap among neighbor thread blocks and calculations for the points located in halo regions is redundant across neighbor thread blocks. Hence, the size of halo region (or amount of local iterations) makes a trade-off between global communications and redundant calculations [59]. Figure 6.1a and 6.1b report the performance under two halo region sizes (or the number of local iterations). Below we first discuss the impact of halo region size on performance. Then we compare the launch and kernel time of FCW to other implementations, followed by the discussion on the Hotspot's chip size.

halo region size. Larger halo region size impacts the performance of ghost zone implementations in two ways. On the positive side, larger halo region size decreases the amount of inter- thread block (or global) communications, reducing kernel launches and global memory accesses. On the negative side, larger halo region size increases the amount of redundant computations. Figure 6.1 reports the numbers



(a) one local iteration for CUDA and FCW.



(b) two local iterations for CUDA and FCW.

Figure 6.1: Kernel execution/launch time of three Hotspot implementations under different problem sizes, ranging from 128x128 to 4kx4k chip sizes. Halo region size of (a) one element and (b) two elements.

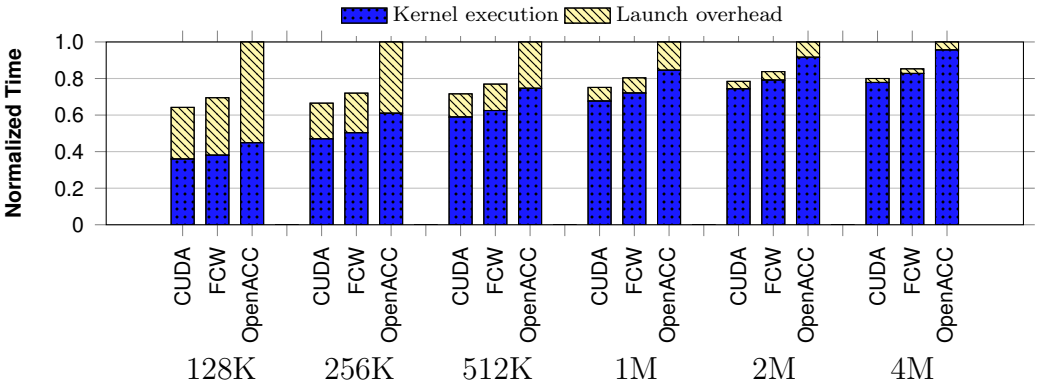
under two halo region sizes, one element and two elements. Comparing FCW under the same problem sizes along Figure 6.1a and 6.1b shows the impact of halo region size on kernel launch time, global synchronization, SMC access overhead, and redundant computations. Below we explain this further.

launch time. For larger problem sizes, e.g. 1K to 4K, kernel launch overhead has insignificant share of the total time. For smaller problem sizes, e.g. 128x128 and 256x256, kernel launch overhead of Hotspot has a significant share in total time. Baseline OpenACC launches several kernels to perform global synchronization across thread blocks. FCW and CUDA save significant kernel launch time by lowering global communications, by a factor proportional to halo region size (or local iterations). This reduction can be seen in 6.1b by comparing FCW and CUDA to OpenACC. In this

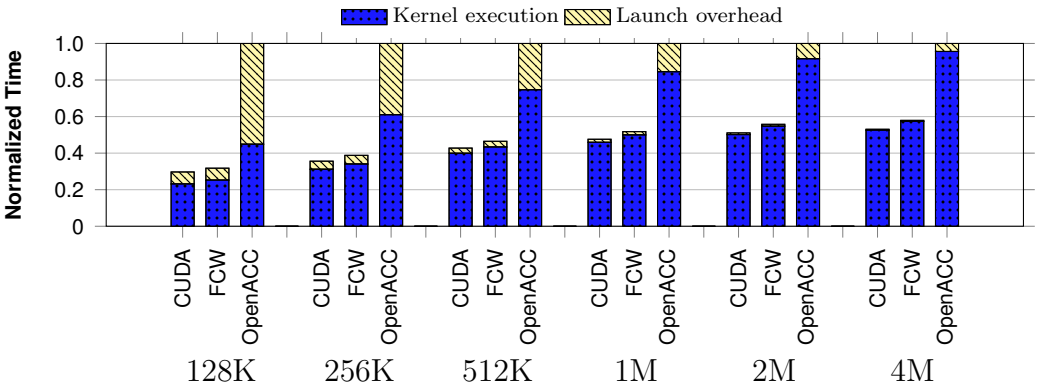
case, FCW and CUDA perform two local iterations and lower the launch time nearly by 2X, compared to OpenACC.

kernel time. Under one local iteration, FCW and CUDA perform slower than OpenACC. This suggests that the overhead of redundant operations dominates the global communication savings. Additionally, FCW and CUDA execute a higher number of dynamic instructions (for accessing the data at SMC), while not taking advantage of SMC for reducing the global communication or reusing the data (since there is only one local iteration). We report one iteration to show the overheads of SMC. Generally, the number of local iterations must be larger than one to allow ghost zone to return performance improvement. Under two local iterations, FCW and CUDA start to perform faster than OpenACC. This suggests that the global communication savings is taking over redundant operations, leading to performance improvement. For the number of local iterations beyond two iterations, we found that redundant computations negate the global communication savings and always degrade performance. 2D halo regions, as used in Hotspot, have higher overhead than 1D regions since they pad the tile along two dimensions, increasing redundant computations 2X more than 1D halo regions. As we report for Pathfinder benchmark, 1D halo regions allow higher improvements under larger halo regions.

problem size. FCW performs significantly better than baseline OpenACC for chip sizes smaller than 256x256. We explain this by separating the discussion of launch and kernel time. On the kernel launch time side, FCW has a huge saving in time which comes from the ghost zone optimization that performs fewer kernel launches. On the kernel time side, the kernel has a total of 16 thread blocks (chip size divided by thread block size which is 16x16) and the available thread-level parallelism is not enough to hide the memory latency effectively. Therefore, it is vital to address the global memory latency using a technique other than multi-threading. FCW takes advantage of SMC to mitigate average memory latency and improves performance. However, when the chip size increases, e.g. in 512x512, thread-level parallelism grows and provides enough threads to allow hardware to hide memory latency effectively. In these cases, immense thread-level parallelism mitigates the performance advantages of SMC in lowering average memory access latency and mitigating the gap between FCW and OpenACC. Putting together the saving of kernel and launch, performance advantages of FCW varies under different problem sizes, ranging from 11% (10% in the kernel and 1% in the launch) in 4Kx4K chip size to 58% (21% in the kernel and 37% in the launch) in 128x128 chip size.



(a) two local iterations for CUDA and FCW.



(b) 12 local iterations for CUDA and FCW.

Figure 6.2: Kernel execution/launch time of three Pathfinder implementations under different problem sizes, ranging from 128K to 4M elements. Halo size of (a) two and (b) 12 elements

Pathfinder

Figure 6.2 compares performance of three Pathfinder implementations. We run the algorithm for a total of 120 iterations. FCW and CUDA use software-managed cache and implement ghost zone optimization. Ghost zone optimizations create halo regions around the tile allowing thread blocks to calculate and reuse intermediate results locally and without global communication. Each bar group reports for a particular problem size for Pathfinder (128K to 4M). Figure 6.2a and 6.2b report the numbers for different halo region sizes (different number of local iterations). Under all configurations, the CUDA implementation performs the fastest and OpenACC performs the slowest, in both kernel execution and kernel launch.

launch. Comparing kernel launch time, FCW performs close to the CUDA version and substantially faster than baseline OpenACC (72% to 8.5X). This is because

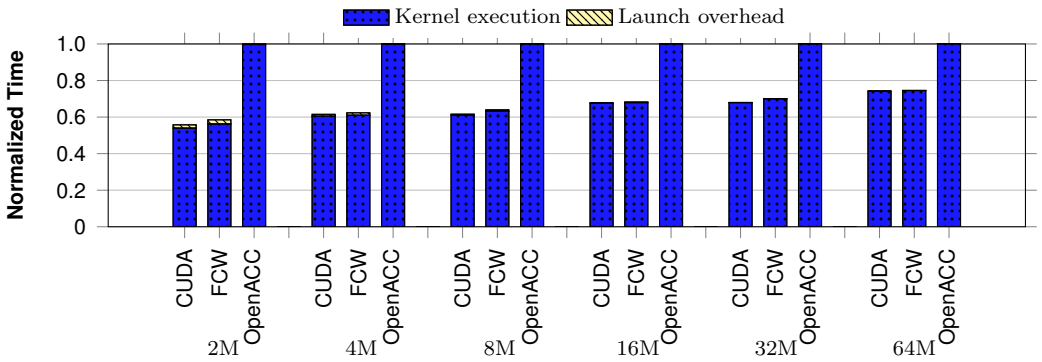


Figure 6.3: Kernel execution and launch time of three Dyadic Convolution implementations: CUDA (CUDA), standard OpenACC (OpenACC), and OpenACC+fcw (FCW). The legend below each group denotes the size of input sequence.

CUDA and fcw implementations launch 2X lower number of kernels than OpenACC in Figure 6.2a (12 times lower than OpenACC in Figure 6.2b).

kernel. Comparing kernel execution time, FCW performs 16% to 79% faster than baseline OpenACC. In FCW, intermediate results are read back by threads of the same thread block, avoiding high number of global memory reads and writes, compared to OpenACC. Comparing CUDA and FCW, FCW performs 6% to 9% slower than the CUDA implementation. For each problem size, the gap between CUDA and FCW kernel execution time narrows down as the number of local iterations grows (comparing the same bar group in Figures 6.2a and 6.2b). This is due to the fact that our automatically generated SMC code imposes higher overhead than the hand-developed CUDA version. Increasing the halo region size (or the number of local iterations) gives rise to the computation body, making the overhead of initial SMC fetch negligible. Hence, the higher number of local iterations the lower the performance gap between CUDA and OpenACC. We found that after increasing the number of local iterations beyond 12, performance improvement saturates. Beyond that point, the redundant calculations of halo regions outweigh the performance advantages of lowering the number of global memory read/writes.

Dyadic Convolution

Figure 6.3 compares performance of OpenACC, CUDA, and OpenACC+fcw under various problem sizes, 2M to 64M elements per input sequence. As can be seen, CUDA and OpenACC+fcw perform comparable while OpenACC performs the slowest.

launch. Contribution of kernel launch time in total time is negligible (nearly 2%

in 2M data).

kernel. Comparing kernel execution time, FCW performs 34% to 77% faster than baseline OpenACC. While baseline OpenACC computes the dyadic convolution directly, CUDA and OpenACC+fcw use Fast Walsch-Hadamard Transformation (FWHT). FCW and CUDA launch several kernels for transforming the input into FWHT domain, calculating dyadic convolution in FWHT domain, and transforming the output back to original domain. Generally, using FWHT algorithm minimizes irregular memory accesses of dyadic convolution and improves performance.

problem size. As the problem size grows, performance advantages of FCW and CUDA decrease. We explain this by the algorithmic difference between FCW/CUDA and OpenACC. Calculation of dyadic convolution in FWHT domain is fairly fast, but transformations of input sequences is the key bottleneck. CUDA/FCW call FWHT three times: twice to transform two input sequences and once to transform back the output sequence. We investigated the breakdown of kernel time and found that the transformations take 94% of the time. By increasing the problem size, this number grows to 96%. Transformation is performed in two kernels and only one of the kernels is optimized to use software-managed cache. The kernel which is optimized by software-managed cache accounts for 26% of transformation time. Increasing the problem size to 64M lowers this number to 20%, emphasizing that the other kernel is the bottleneck under larger problem sizes. Therefore, we can conclude that the performance advantages of CUDA/FCW is lowered under larger problem sizes due to the overhead of FWHT transformation algorithm. Further algorithmic optimization is out of the scope of this work, as we are interested in implementing the same CUDA algorithm using fcw clause in OpenACC.

N-Body

Figure 6.4 compares OpenACC, OpenACC+fcw, and CUDA implementations of N-Body under various input sizes.

launch. CUDA performs faster in kernel launch time since OpenACC and OpenACC+fcw launch two kernels. OpenACC implementations calculate the summation of velocity in one kernel and the new position in another kernel. As we increase the number of bodies from 512 to 16384, kernel execution time dominates the runtime breakdown and kernel launch time becomes negligible.

kernel. Comparing kernel execution time, FCW outperforms OpenACC by 31-

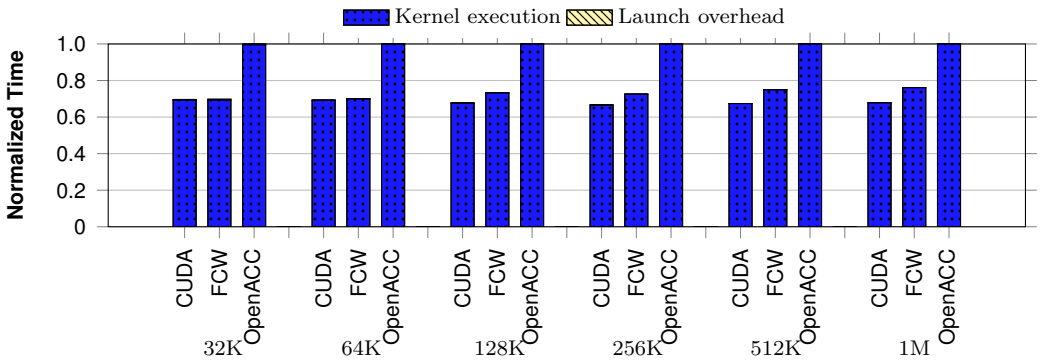


Figure 6.4: Kernel execution and launch time of three N-Body implementations: CUDA (CUDA), standard OpenACC (OpenACC), and OpenACC+fcw (FCW). The legend below each group denotes the number of bodies.

44%, under different problem sizes. CUDA implementation performs faster than OpenACC and FCW. CUDA outperforms OpenACC by 44-47% and OpenACC+fcw by 1-12%.

problem size. As shown in Figure 6.4, increasing the problem size lowers the performance advantages of FCW the over OpenACC implementation. We explain this by the difference between occupancies [2]. In the particular GPU used in this study, OpenACC, CUDA, and FCW kernels run with the occupancy of 75%, 75%, and 38%, respectively. Lower occupancy of FCW becomes a performance degrading issue under larger problem sizes where there are higher number of thread blocks. In this case, FCW runs with a lower occupancy compared to the CUDA version since FCW overestimates the size of required software-managed cache.

Matrix-matrix Multiplication

Figure 6.5 compares the performance of three implementations of matrix multiplication under various input sizes.

launch. Kernel launch time is comparable under different implementations. Also increasing the problem size mitigates the contribution of kernel launch time to total runtime (less than 1%).

kernel. For kernel execution time, same behavior can be seen under different input sizes. CUDA implementation is the fastest and OpenACC implementation is the slowest. FCW outperforms OpenACC by 103%. CUDA implementation out-

[2]Occupancy is the ratio of the number of concurrent active thread blocks to the maximum number of supported thread blocks per core.

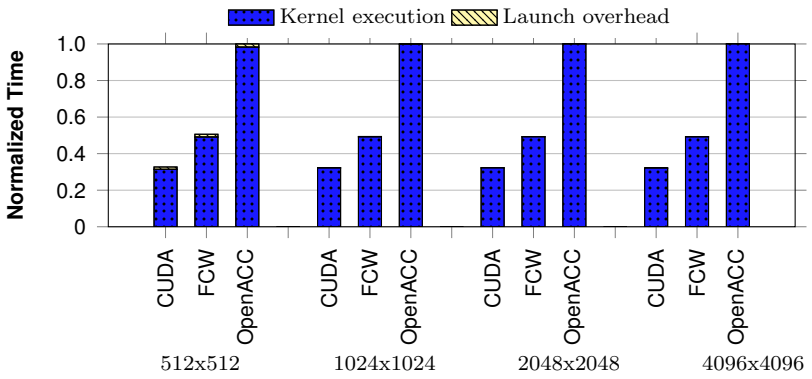


Figure 6.5: Performance of OpenACC (OpenACC), OpenACC+fcw (FCW), and CUDA (CUDA) implementations of matrix multiplication. Each thread in OpenACC version calculates one element in the output and fetches an entire row and column from global memory. OpenACC+fcw and CUDA compute by fetching rows and columns in tiles into shared memory.

performs FCW and OpenACC by 53% and 311%, respectively. FCW and CUDA significantly reduce the global memory traffic, compared to OpenACC. Fetching tiles of input matrices into SMC, FCW and CUDA implementations maximize memory access coalescing. Also these implementations exploit the locality among neighbor threads to minimize redundant memory fetches. Using *nvprof*, we found that FCW tiling reduces DRAM access rate by 3.94X (under 1024x1024 matrices), compared to OpenACC.

Despite the significant improvement of FCW over OpenACC, comparing CUDA and OpenACC+fcw, we still observe a runtime gap (53%). This gap is due to the overhead of our current implementation of fcw. Particularly, CUDA implementation shares a control-flow statement between both tiles of input matrices for fetching the data. However, in FCW implementation, two separate control-flow statements are generated for fetching each of input tiles. In our implementation of fcw, we have few heuristics to identify the opportunities for sharing the control-flow statement of SMC initialization among multiple fcw clauses. For instance, we share a fetch control-flow statement among multiple fcw clauses, if the clauses have the equal value on BEFORE and AFTER in range identifiers. However, in this matrix multiplication example, our heuristics are not capable of detecting the opportunity for sharing the control-flow statement. We leave the study for finding more advanced heuristics to future work.

Needleman-Wunsch

We investigated the opportunity to implement the CUDA algorithm through fcw under Needleman-Wunsch. CUDA algorithm uses one dimensional thread blocks to fetch and writeback a two-dimensional array. Additionally, there is a complex relation between thread identifiers, global memory locations, and shared memory locations. These issues prevented us from implementing the CUDA algorithm with the simple notation of *fcw*. One possible solution maybe to add a dummy parallel loop in OpenACC to create two-dimensional thread blocks and simplify the indexing. However, this solution has two drawbacks. Firstly, it sophisticates OpenACC programming. Secondly, adding a dummy parallel loop increases the number of thread blocks remarkably and may ruin the performance advantages of SMC.

6.3.2 Development Effort

Table 6.2 compares development effort under fcw compared to baseline OpenACC and CUDA, in terms of number of lines of code. Since code modifications for implementing fcw is limited to the kernels region space of code, we only report the number of lines of code for this code region (not whole the code). Under CUDA, we report the number of code lines implementing the same operation. Changes are slight in Pathfinder, Matrix-matrix multiplication, Hotspot, and N-Body and very high in Dyadic Convolution. The overhead in the first group is low since the modification is limited to tiling the data around a loop. By tiling the data around a loop we mean decomposing an iterative loop into two inner and outer loops and applying fcw directive over the inner loop. The overhead in the Dyadic Convolution is significant as major algorithmic changes is required.

In Dyadic Convolution, fcw uses fast Walsh-Hadamard transformation which replaces the single kernel region with three kernels regions, significantly increasing code modification. Among these three kernels, only one of them exploits software-managed cache and most of the code is CUDA-like. Implementing the algorithm in fcw demands higher effort than CUDA. We found that fcw does not have significant advantage in amortizing the SMC fetch instruction, only saving two lines of code. Besides this, fcw performs roughly the same amount of instructions as the CUDA version. fcw code is larger than CUDA as there are single-line directives over the loops to hint the compiler for launching parallel threads. We conclude that using fcw may not be justified in terms of development effort in this case. Appendix A explains how we

Table 6.2: Comparing development effort of baseline OpenACC, fcw, and CUDA implementations in terms of the number of code lines.

benchmark	baseline	fcw	CUDA
Pathfinder	18	41	52
Matrix-matrix multiplication	13	23	30
Hotspot	16	42	66
N-Body	15	20	26
Dyadic Convolution	17	132	99

extend Pathfinder, Matrix-matrix multiplication, Hotspot, and N-Body to implement the fcw version.

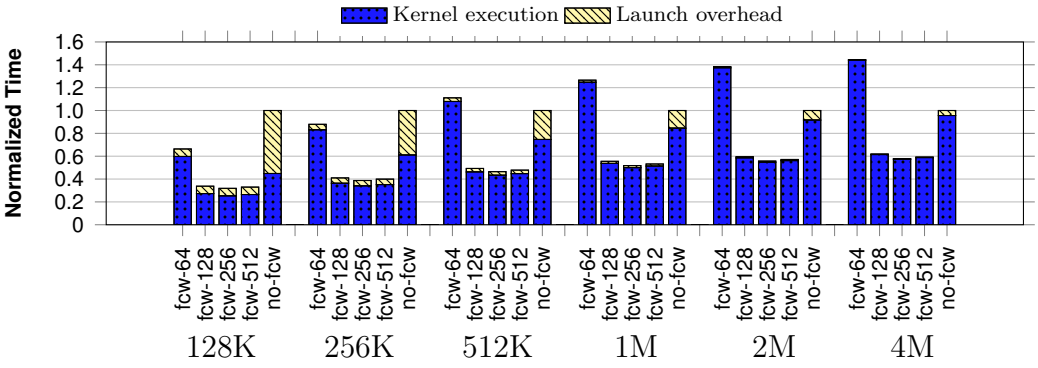
6.3.3 Sensitivity to Vector Size

In this Section, we investigate the sensitivity of our findings to variations in the vector size clause of the loop directive. Since we execute OpenACC applications over CUDA, we use CUDA terminology to explain the performance impact.

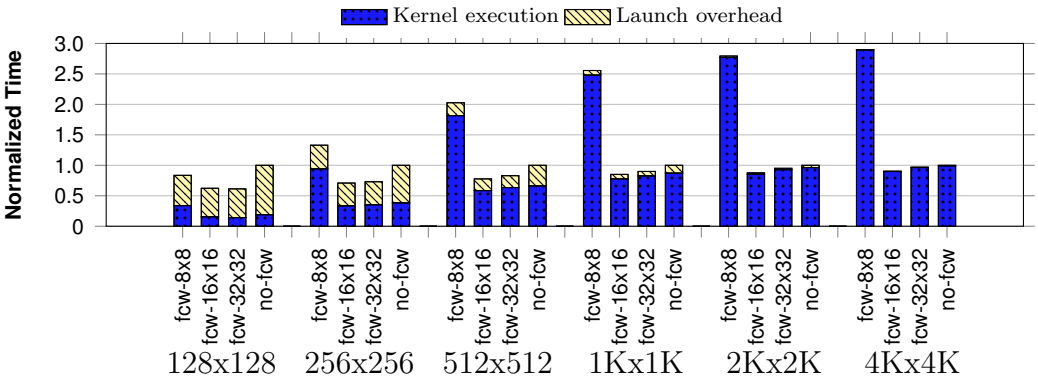
Variations in the vector size impacts the number of communicating loop iterations. This in effect changes the i) number of threads in a thread block and ii) amount of CUDA shared memory used by each thread block. Since each GPU core has a certain capacity in concurrent threads and shared memory per thread block, vector size variations change the number of concurrent thread blocks per GPU core. This is often referred to as *occupancy*.

In Section 6.3.1, we investigated the performance of OpenACC+fcw version of Pathfinder and Hotspot benchmarks under 256 and 16x16 thread block (or vector) sizes (notice that vector size in Hotspot is specified by two numbers as Hotspot has two nested loops). Figure 6.6 presents performance under different thread block sizes. We evaluated occupancy for all the configurations presented in this figure. Using CUDA Occupancy Calculator [70], the occupancy of all configurations are 100%, except Pathfinder with 64 vector size and Hotspot with 32x32 vector size which are 38% and 50%, respectively (the occupancy of baseline OpenACC is also 100% in both cases).

In both benchmarks, very small vector sizes (e.g. 64 in Pathfinder and 8x8 in Hotspot) perform poor. This is explained by the ghost zone optimization characteristic which is exploited in both Pathfinder and Hotspot. For a fixed ghost zone size (which is two in Pathfinder and 12 in Hotspot), small vector sizes increase the overhead of ghost zone operations. In these cases, the overhead of redundant calculations



(a) Pathfinder.



(b) Hotspot.

Figure 6.6: Comparing baseline OpenACC (shown with no-fcw label) and OpenACC+fcw, under different vector sizes and problem sizes. (a) Pathfinder: 12 local iterations and 64, 128, 256, and 512 vector sizes. (b) Hotspot: two local iterations and 8x8, 16x16, and 32x32 vector sizes.

negates the performance gain of software-managed cache significantly leading to very poor performance.

Increasing vector size improves performance, as long as occupancy remains intact. This improvement is caused by the higher efficiency in ghost zone calculation. For example in Pathfinder, the 256 and 512 vector sizes perform best. We see slight performance degradation moving from 256 vector size to 512. This is explained by the higher overhead in thread block synchronization under larger thread blocks. This degradation is more sensible in Hotspot, moving from 16x16 vector size to 32x32. In this case, in addition to the synchronization overhead, 32x32 has lower occupancy which can be a critical issue under larger problem sizes.

6.4 Discussion

6.4.1 Programmer or automatic compiler passes

An important question is whether achieving fcw gains through using a new directive is the most effective approach. In other words is it possible, for example, to achieve similar benefits by employing an alternative compiler. Our proposed fcw requires programmers to provide information for i) base array pointer, ii) fcw type, and iii) the data range that every loop iteration accesses. For the compiler to provide the same benefits, it has to obtain this information automatically. Instead of the first parameter, array pointer, the compiler can blindly search all arrays, which are called within the accelerator region, and investigate the potential for data caching in SMC. The second parameter, fcw type, cannot be decided properly by the compiler. Under fcw types used here, fcw can make inter-iteration communication possible. Since the compiler is blind to the algorithm, such communications may harm correctness. In this situation, the compiler is limited only to assume a safe conservative yet suboptimal FETCH_ONLY type. As for the data range, static compiler passes may be able to determine this information. This is the case when the data range identifiers do not depend on any variable with a runtime-dynamic value. We conclude that under limited circumstances only FETCH_ONLY fcw can be performed by the compiler, without any information from the programmer. CHANNEL_ONLY, FETCH_CHANNEL, CHANNEL_WB, and FETCH_CHANNEL_WB types allow fast inter-iteration communication, which needs the programmer insight and is unlikely to be achieved using compiler passes.

6.4.2 Applicability

fcw is applicable in two ways: implicit or explicit. In implicit mode, programmer relies on FETCH_* (fetch) and *_WB (writeback) clauses of fcw and allows the compiler to implement fetch and writeback in the most efficient way. In explicit mode, programmer passes an array (it might be even a dummy array) in fcw notation (potentially CHANNEL_ONLY type) and relies on the compiler to share intermediate writes among neighbor iterations via software-managed cache. In this case, programmer explicitly fetches the data into SMC or writes the content of SMC back to global memory, in an arbitrary user-defined pattern. The implicit mode is advantageous in terms of development effort. The explicit mode is advantageous in terms of applica-

bility. In our evaluations, we have examples of both cases. We used the implicit mode in Hotspot, Pathfinder, N-Body, and Matrix-matrix Multiplication and the explicit mode in Dyadic Convolution benchmark. We are more interested in implicit mode, since the abstraction and development effort of this mode matches the requirements of OpenACC model. Below we discuss limitations of implicit mode.

We use fcw within parallel work-sharing loops. The data range identifiers of fcw are either constant or varying, in respect to the induction variables of outer containing parallel loops. If the identifiers are constant in respect to induction variables, all parallel iterations need the same range of array elements, fully overlapped between threads. In this case, fcw is applicable if the overlapped range of data fits in the accelerator SMC. Otherwise, if the identifiers are varying in respect to induction variables, parallel iterations potentially demand different array elements. To create a communication channel among the neighbor parallel iterations, i) neighbor iterations must be mapped to the threads of the same thread block and ii) specified fcw range must have an overlap among the threads of the same thread block. Otherwise, neighbor iterations will not be able to communicate, reading intermediate writes of each other. In summary, in order to have communication between neighbor iterations under fcw the following conditions should be met: i) the programmer must perform communication only among neighbor loop iterations, ii) the compiler must map neighbor iterations to threads of the same thread block, iii) the union of data across neighbor iterations should be small enough to fit in the accelerator SMC. If necessary, the number of communicating loop iterations (or the thread block size) can be adjusted to shrink the data size and fit the data in SMC. A similar discussion applies to multi-dimensional arrays marked as fcw.

6.4.3 Implications

Communication thread block size. The programmer may need to control the communicating thread block size or the number of iterations that share one common SMC. She can control this number through the OpenACC vector clause of the outer loop directive. Upon translating OpenACC to CUDA (or OpenCL), the *vector* clause determines the thread block size. In nested parallel loops, a *vector* clause per *loop* can be passed to specify the thread block size along each dimension.

Data hazard in SMC and DRAM. Using fcw, the programmer may develop a code exhibiting SMC write-after-write or write-after-read data hazards. We add the

following restriction to fcw definition to protect fcw under such hazards. Using implicit or explicit synchronizations, every write to a CHANNEL_ONLY, FETCH_CHANNEL, CHANNEL_WB, and FETCH_CHANNEL_WB fcw location is guaranteed to be synchronized across the iterations falling within the same communicating thread block. Multiple writes to the same location are allowed and fcw guarantees to store one write. But the order of writes is not determined (since the threads may execute in an unspecified order on the accelerator). Similarly, there is a possibility for race conditions on global memory. This happens if the range identifiers have an overlap across different communicating thread blocks for CHANNEL_WB and FETCH_CHANNEL_WB. We also add the following restriction to fcw definition to protect fcw in such cases. When there is an overlap between different communicating thread blocks over CHANNEL_WB and FETCH_CHANNEL_WB, one memory write is guaranteed to be stored in global memory. But the order of writes is not determined.

6.4.4 Difference from the cache directive

The difference between the *cache* and *fcw* directive is that *fcw* is a more general directive with wider applicability. The *fcw* directive can fully replace *cache* directive and both will have an identical performance. However, the *fcw* directive can be used in cases where the *cache* directive is not applicable, e.g. exploiting SMC as a channel for communication among iterations.

6.5 Summary

In this chapter, we explained the limitations of the *cache* directive in exploiting SMC. We proposed a new OpenACC directive, referred to as fcw, to allow more flexible use of SMC in OpenACC, nearly close to CUDA programming model. In conjunction with fcw, we introduce a new communication model in OpenACC allowing neighbor iterations of parallel loops to communication and share intermediate data. Using fcw, the OpenACC programmer can fetch a tile of data from DRAM to the software-managed cache, share intermediate data between iterations, and write the final value back to global memory. We studied wide set of benchmarks to show the applicability and performance advantages of fcw compared to OpenACC and CUDA. SMC programming in OpenACC is easier than CUDA for two reasons. First, the shared memory implementation can be obtained in fewer lines of code. Secondly, the com-

piler can automatically inject debugging and verification routines which simplifies debugging. We showed that fcw can narrow down the gap between OpenACC and CUDA significantly.

Chapter 7

TELEPORT: Hardware/Software Alternative To CUDA Shared Memory Programming

Using software-managed cache in CUDA programming provides significant potential to improve memory efficiency. Employing this feature requires the programmer to identify data tiles associated with thread blocks and bring them to the cache explicitly. Despite the advantages, the development effort required to exploit this feature can be significant. Our goal in this chapter is to reduce this effort while maintaining the associated benefits. To this end, we first investigate static precalculability in memory accesses for GPGPU workloads, at the thread block granularity. We show that a significant share of addresses can be precalculated knowing thread block identifiers. We build on this observation and introduce TELEPORT. TELEPORT is a novel hardware/software scheme for delivering performance competitive to software-managed cache programming, but at no extra development effort. On the software side, TELEPORT's static analyzer parses the kernel and finds precalculable memory accesses. We introduce Runtime API calls to pass this information to hardware. On the hardware side, this information is used to fetch the data required for each thread block into shared memory before the thread block starts execution. With this hardware support, TELEPORT outperforms hand-written CUDA code as a result of the associated benefits, e.g. DRAM row locality improvement. Investigating a wide set of benchmarks, we show that TELEPORT improves performance of hand-written implementations, on average, by 32% while reducing development effort by 2.5X. Our

estimations show that the hardware overhead associated with TELEPORT is below 1%.

The remainder of this chapter is organized as follows. In Section 7.1, we overview TELEPORT's organization and briefly evaluate its performance and development effort advantages. In Section 7.2, we show our motivation behind TELEPORT by investigating static precalculability of memory accesses in GPGPU workloads. In Section 7.3, we introduce TELEPORT. In Section 7.4, we overview experimental methodology. In Section 7.5, we investigate the performance and development effort advantages of TELEPORT. In Section 7.6, we estimate hardware overhead of TELEPORT. Finally, Section 7.7 summarizes our findings.

7.1 Overview

Programming the software-managed cache in CUDA involves tremendous development effort. Firstly, the programmer should identify the data to be fetched into the cache. Candidate data are the arrays representing high temporal/spatial locality. Secondly, code should be modified to add an extra array explicitly representing the software-managed cache. To this end, two sets of indexes should be maintained; global and shared memory spaces.

In this chapter, we introduce TELEPORT, a hardware/software mechanism, to offload the shared memory development effort from the programmer to the compiler, while not sacrificing performance. Under TELEPORT, the compiler analyzes CUDA kernels to statically identify the data tiles assigned to each thread block. Later, during runtime, hardware loads the designated tiles into the software-managed cache in advance for each thread block. When both TELEPORT and hand-written CUDA versions implement similar algorithms, TELEPORT can outperform CUDA versions via a unique hardware optimization, improving DRAM row locality.

On the software side, we develop a static analyzer to parse the kernel, identify the candidate arrays, and determine data ranges that each thread block accesses. Extra procedure calls are introduced to pass this information in an abstract form to GPU. The procedure calls configure *preload table* in the hardware, before kernel launch calls. These steps can be fully integrated into the kernel compilation phase.

On the hardware side, a logical preload table per kernel is maintained. Upon dispatching a new thread block to GPU core, the thread block dispatcher issues a burst of memory requests to fetch the thread block's data, using the information in preload

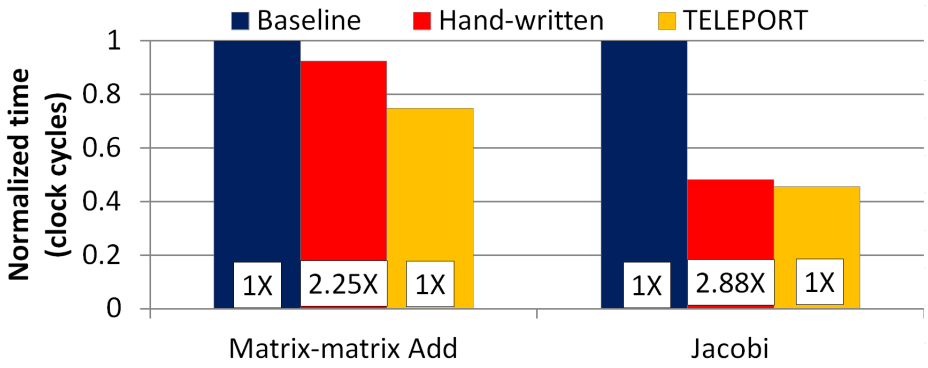


Figure 7.1: Comparing three different implementations of Matrix-matrix Add and Jacobi iteration. Bars report kernel time and numbers below the bar indicate the development effort, normalized to Baseline. (Effort is measured in the number of lines of code.).

table. All threads of the thread block are put on hold till tiles are loaded completely. Putting the thread block on hold also stops threads from issuing redundant memory accesses, avoiding the generation of excessive memory bandwidth traffic.

To take a glance at the performance and development effort advantages of TELEPORT, we present a subset of out findings in Figure 7.1 where we compare three different implementations of two benchmarks: matrix-matrix addition (A + B = C) and Jacobi iterative method (See Section 7.4 for methodology.) The first implementation (Baseline) does not use the software-managed cache. The second implementation (Hand-written) employs the software-managed cache. The third implementation (TELEPORT) analyzes the source code of the Baseline implementation and takes advantage of the opportunities available for using software-managed cache (notice that this implementation relies on hardware support.) Below we explain each benchmark.

Matrix-matrix addition. Under Baseline, every thread calculates one element of the output matrix. While performance is very poor, the development effort is fairly low. Under Hand-written, threads of every thread block collaboratively fetch tiles of A and B to the software-managed cache and calculate the sum. This implementation exploits data locality among threads of the thread block and removes redundant memory fetches within thread block. While performance is very high, Hand-written implementation demands higher development effort compared to Baseline (2.25X greater). Under TELEPORT, development effort is similar to Baseline. During the compile time, the static analyzer parses the Baseline's kernel to specify the ranges of A and B that are assigned to each thread block. Finding opportuni-

ties for caching A and B, the compiler injects API calls before the kernel launch to configure the preload table for the kernel. With hardware support, the input tiles are fed to the thread block during runtime through the software-managed cache. As reported, TELEPORT outperforms Hand-written by 23%. As we explain later, part of this improvement comes from lowering the number of dynamic instructions.

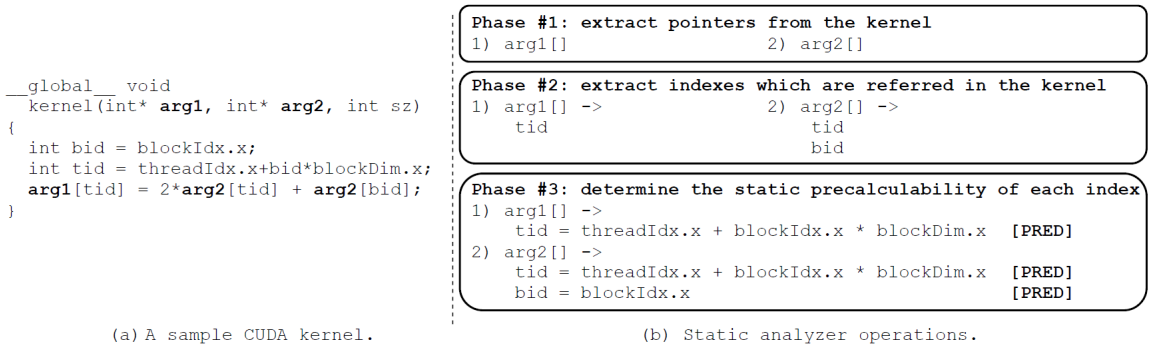
Jacobi iterative method. In this benchmark, every thread calculates one element in the output by applying the smoothing function over nine elements (8 neighbors plus the element itself). This results in a strong spatial data locality among the thread input data as threads use adjacent elements to calculate the output. Under Baseline, threads fetch the elements from global memory separately. This implementation relies merely on memory access coalescing capabilities of hardware [67]. Hand-written fetches a tile of data, covering the input of all collaborating threads, into the shared memory. This lowers the global memory load instructions by nearly 9X (for a tile of $16 * 16$ threads, Baseline performs $16 * 16 * 9$ loads and Hand-written performs $(16 + 1) * (16 + 1) * 1$ loads. But not all of this gain translates to speedup, since Hand-written version needs to access the shared memory for $(16 * 16 * 9) * 2$ times[1].). TELEPORT analyzes Baseline kernel and identifies the input tile associated with collaborating threads. This, combined with hardware support, lowers the development effort of Hand-written by 2.88X and improves its performance by 6%.

Later in this chapter we investigate wider set of benchmarks and show TELEPORT improves performance of Baseline and Hand-written implementations, on average, by 56% and 32%, respectively. We also show TELEPORT lowers development effort by 1.46X to 3.4X, compared to Hand-written. TELEPORT uses the unused space in the software-managed cache of the GPU core as a buffer for storing tiles. The hardware overhead associated with TELEPORT includes the preload table and TELEPORT's controller unit (which are shared among GPU cores), plus an array of tags per GPU core for indexing the software-managed cache. Our estimations show that the hardware overhead is below 1%.

7.2 Motivation

In this Section, we investigate the static precalculability of memory accesses in CUDA kernels. We limit the analysis to global memory space, since this is the space con-

[1]Multiplied by 2 to account for one store and one load.



(a) A sample CUDA kernel. (b) Static analyzer operations.

Figure 7.2: Example to clarify the static analyzer operations.

tributing to performance the most. We developed a static analyzer parsing one CUDA for C kernel at a time. For each kernel, the static analyzer performs three phases. First, the analyzer forms a list of variable names which are listed in the kernel's arguments as pointer variables. These pointers essentially point to a location in the global memory. Within the kernel, each pointer is treated as array. Second, for each of these arrays, the static analyzer extracts the array indexes which are referred to in the kernel body. Finally, the static analyzer examines static precalculability of the value of each index. Figure 7.2 clarifies the three phases of our static analyzer. Below we explain our definition of static precalculability and report the findings of static analysis under different CUDA kernels[2].

7.2.1 Static Precalculability

We classify array indexes into *statically precalculable*, *quasi-static precalculable*, and *non-precalculable*, based on the static precalculability of the index value. We declare the precalculability of index values based on the operators (add, multiply, shift, etc.) and terms (variables or constants) forming the index expression. We identify precalculability in two steps; processing operators and processing terms.

Processing operators. We consider an index as non-precalculable if it is composed of any operator other than addition, subtraction, and multiplication. As we clarify later, this reduces hardware complexity for address calculations on precalculable addresses.

Processing terms. If the index is not found non-precalculable in the previous step, then the analyzer examines the terms of the expression to check precalculability

[2]Hereafter, by array we refer to every array identified in first phase of static analysis. Also array index is an array index extracted in the second phase.

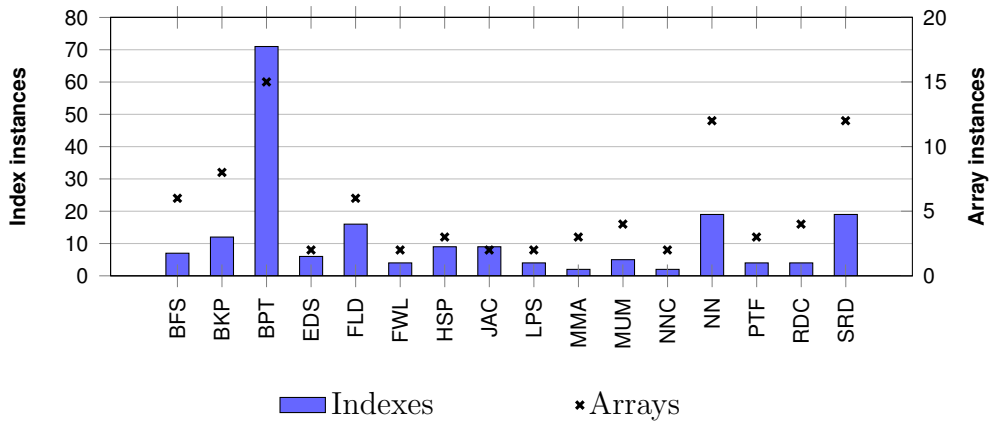


Figure 7.3: The number of arrays and indexes identified by the static analyzer.

(either statically or quasi-static). The index is statically precalculable, if the terms have constant values (e.g. *a[0]* or *a[blockDim.x]* in CUDA). The index is quasi-static precalculable, if the memory index term depends on at least one built-in CUDA thread identifier (*threadIdx* or *blockIdx*). Since the thread and thread block identifiers are known at the time of dispatching the thread block, we refer to this type as quasi-static precalculable. The index is non-precalculable, if the memory index term depends on a runtime variable. A runtime variable can be a memory location (e.g. *a[b[0]]*) or a control-dependent variable (e.g. *a[condition ? 1:0]*).

7.2.2 Findings

Here we report static precalculability findings under 16 benchmarks (for methodology refer to Section 7.4), as measured by our static analyzer. Figure 7.3 reports the number of unique global memory arrays found in the kernels. These arrays are declared as pointer variables in the kernel's arguments. The figure also reports the number of array indexes found in the kernel[3]. In the case of multiple kernels in the benchmark (which is the case for BKP, BPT, NN, RDC, and SRD), we report the summation of arrays and indexes which are found in each kernel. As shown, the number of arrays ranges from two (in NNC benchmark) to 15 (in BPT) while the number of array indexes ranges from two (in HSP) to 71 (in BPT).

Figure 7.4 complements Figure 7.3 and reports the breakdown of array indexes into statically precalculable, quasi-static precalculable, and non-precalculable. As reported, none of the array indexes are found statically precalculable in the evaluated

[3]For indexes, we only report memory reads.

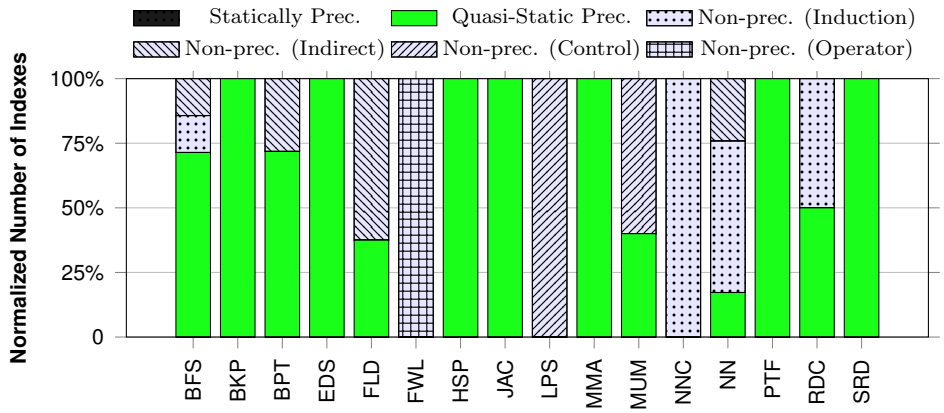


Figure 7.4: Breakdown of array indexes into statically precalculable, quasi-static precalculable, and non-precalculable. Non-precalculable indexes either depend on induction variable (Induction), another memory load (Indirect), a control statement (Control), or use a sophisticated operator (Operator).

kernels and a significant portion of array indexes are found quasi-static precalculable. There are four subcategories of non-precalculable indexes: i) induction variable dependence, ii) indirect addressing, iii) control dependent, and iv) sophisticated operation. In NNC, NN, and RDC, non-precalculable array indexes depend on a loop induction variable. While we list them as non-precalculable in the breakdown, they may also be considered as quasi-static precalculable, if the boundaries of the loop are precalculable (which is the case for these benchmarks[4].) For instance, in NN, 17 non-precalculable array indexes depend on an induction variable, whose range can be evaluated statically. In BPT and FLD, non-precalculable array indexes depend on a load from another array. In LPS and MUM, non-precalculable array indexes depend on control statements. In FWL, non-precalculable array indexes use the AND operator (&). In BFS, one of the non-precalculable array indexes depends on an induction variable and the other non-precalculable index depends on a load from another array.

Generally, a large portion of array indexes are found to be quasi-static precalculable (up to 100% in many benchmarks). Quasi-static precalculable indexes only depend on CUDA *threadIdx* and *blockIdx* variables which are evaluated at the time of dispatching the thread block. Hence, during runtime, a range of the data that each thread block accesses can be precalculated. Hereon we build on this observation and propose TELEPORT hardware/software scheme to take advantage of the

[4]Here minimum and maximum values of the induction variables can be obtained from the loop statements statically. Then we can relax the induction dependency constraint and evaluate the indexes as quasi-static precalculable as the range of induction variables are evaluated statically.

opportunity.

7.3 TELEPORT

7.3.1 Software Side

The software side of TELEPORT includes i) static analyzer (which we described in Section 7.2) and ii) an API for passing static analysis information from CUDA applications to GPU hardware, at the kernel launch time. This information is passed for the array indexes which are marked as precalculable by static analysis. The information guides the GPU to fetch data chunks required by each thread block, after dispatching the thread block and before issuing any instruction from individual threads of the thread block. The API passes the information to the GPU in an abstract form. Below, we first discuss the essence of information which is needed for precalculating the demand addresses and introduce the API to pass that information. Secondly, we present an example to clarify the API usage.

API

If the index is marked as precalculable, static analysis evaluates the array index as an expression of i) thread identifiers and ii) constant terms. During the peiod of parsing this expression, we only evaluate thread IDs to determine the index value. Knowing the thread IDs that belong to each thread block, the compiler statically specifies the domain of values that index may have within each thread block. The domain of values of an index is the range between minimum and maximum values that index may have. Thread blocks have different IDs and, accordingly, the minimum and maximum varies for each thread block. Since thread block IDs are evaluated during runtime, the compiler passes the essential information to hardware and allows hardware to calculate the minimum and maximum at runtime. Below we elaborate on the essential information that has to be passed to hardware.

In order to determine the data chunk of each thread block, the static analyzer looks for the *minimum* and *maximum* value of the array index for each thread block. The compiler evaluates the index expression to specify its minimum and maximum values based on the fact that in CUDA there are four three-dimensional thread identifier variables: *gridDim*, *blockDim*, *blockIdx*, and *threadIdx*. The index expression can be reduced to only one variable (*blockIdx*) provided that: i) *gridDim* and *blockDim* are

constant terms within a particular kernel and can be evaluated as a constant value, ii) Lowest and highest values of *threadIdx* are constant, and iii) *blockIdx* is determined at runtime upon dispatching the thread block

Applying the above assumptions to any precalculable array index (statically precalculable or quasi-static precalculable) simplifies the index to an expression of constant values and only one runtime variable (*blockIdx*). In general, precalculable array index expression can be presented in the polynomial form of Equation 7.1.

$$v = a_0 + \sum_{i=1}^{n} (a_i \times bIdx^i) \tag{7.1}$$

where a_i for $0 \leq a_i \leq n$ are life-time constant terms, $bIdx$ is *blockIdx*, and n is the degree of polynomial. If an array index is constant (or statically precalculable) then a_i for $1 \leq a_i \leq n$ is zero. An array index is affine constraint of degree one if a_i for $2 \leq a_i \leq n$ is zero. Similarly, array indexes with affine constraint of higher degrees are possible. To simplify API and hardware support, here we limit TELEPORT to arrays having indexes with affine constrains of degrees one or lower (the degree of array index can be identified by the static analyzer by simply searching for the number of *blockIdx* variables that are multiplied. In the evaluated benchmarks, we found all quasi-static precalculable in the first degree or linear polynomial.)

For every precalculable array index, the static analyzer finds a_0 and a_1 by parsing the dependency graph. It returns the pair of a_0 and a_1 to specify minimum and maximum values of index, based on varying values of *threadIdx*. Then, the proposed API passes two (a_0, a_1) pairs, one for the minimum and the other for the maximum, to guide the GPU in specifying the lowest and highest values of the thread block's data. On the hardware side, this information is used to dynamically calculate the minimum and maximum values of indexes, provided that the value of the last unevaluated variable, *blockIdx*, is known at the time of dispatching the thread block. Minimum/maximum point to the beginning/end of the range of addresses that threads of thread blocks may request. After calculating the range of address, the hardware sends a burst of read requests to the memory controller. This fetches the entire range into a buffer on the GPU core that the thread block is dispatched to. The core starts issuing instructions from this thread block as soon as the entire range is fetched.

The proposed API prototype is shown in Listing 7.1. This procedure sets registers on the GPU before kernel launch and simplifies calculating the address of continuous

Listing 7.1: Proposed API for passing per- thread block preload information.

```
__host__ cudaError_t
cudaSetCTATracker(void *baseptr, size_t typesize,
  unsigned int minbidxp, unsigned int minbidyp,
  unsigned int minbidzp, unsigned int minoffset,
  unsigned int maxbidxp, unsigned int maxbidyp,
  unsigned int maxbidzp, unsigned int maxoffset);
```

data chunk of each thread block required by TELEPORT. TELEPORT controller unit then calculates the range of data using the following linear equations:

$$\begin{aligned} min = baseptr + typesize \times (blockIdx.x \times minbidxp \\ + blockIdx.y \times minbidyp \\ + blockIdx.z \times minbidzp + minoffset) \end{aligned} \tag{7.2}$$

$$\begin{aligned} max = baseptr + typesize \times (blockIdx.x \times maxbidxp \\ + blockIdx.y \times maxbidyp \\ + blockIdx.z \times maxbidzp + maxoffset) \end{aligned} \tag{7.3}$$

Example

In this section, we explain the mechanism of passing information from the static analyzer to the GPU via runtime API. We overview a test case; BFS benchmark.

Listing 7.2 presents the kernel of BFS in CUDA, as available in Rodinia [13]. The static analyzer lists the following procedure arguments (or global memory pointers) as candidates for loading early: *nodes*, *edges*, *mask*, *visited*, *cost*, and *over*. It parses the kernel body and extracts all array reads from these arrays. Table 7.1 shows the list of identified read accesses. The first and second columns list array names and the corresponding lines of the code in Listing 7.2, respectively. The third column reports the index expression of the array access. Based on the terms and operators composing the expression, the fourth column reports whether the index is precalculable or not. 5 of 7 indexes are marked as precalculable. Two array accesses are found non-precalculable since static analysis is unable to simplify the index expression to only thread identifiers and constant values, e.g. line #14. For precalculable array indexes, static analysis extracts the minimum and maximum possible values of the

Listing 7.2: CUDA kernel in BFS.

```
__global__ void
 Kernel(Node* nodes,
   int* edges, bool* mask,
   bool* visited, int* cost,
   bool* over, int no_of_nodes)
{
 int tid = blockIdx.x*blockDim.x + threadIdx.x;
 if( tid<no_of_nodes && mask[tid]){
  mask[tid]=false;
  visited[tid]=true;
  for(int i=nodes[tid].starting;
   i<(nodes[tid].no_of_edges+nodes[tid].starting);
   i++){
   int id = edges[i];
   if (! visited[id]){
    cost[id]=cost[tid]+1;
    mask[id]=true;
    *over=true;
   }
  }
 }
}
```

index. This returns the minimum and maximum, if the index expression is an affine expression of degree one.

There are three unique precalculable array indexes in this example which are evaluated to single common index expression. According to the terminology of Equation 7.1, *blockIdx.x* is $bIdx^i$, *blockDim.x* is a_1, and *threadIdx.x* is a_0 in this expression. Replacing minimum and maximum values of *threadIdx.x* in the expression returns the minimum and maximum values of the index expression. Minimum and maximum values of *threadIdx.x* are 0 and $blockDim.x - 1$, respectively. This information is passed to GPU through *cudaSetCTATracker* API calls, one call per unique index.

Listing 7.3 shows the *cudaSetCTATracker* calls passing static analysis information to the GPU just before the kernel launch. *cudaFlushCTATracker* informs the GPU to flush and clear prior information of the CUDA stream on preload table, indicating a new kernel is about to be launched. Subsequent *cudaSetCTATracker* invocations pass loading information for one continuous data chunk at a time. In Listing 7.3, there are

Table 7.1: Output of static analysis determining the precalculable array indexes, affine index expressions of degree one, and the minimum and maximum value of index.

Array name	Line#	Extracted index expression	Index Status	min/max	
				a1 (multiplier)	a0 (offset)
mask nodes nodes nodes cost	8 11 12 12 16	(blockIdx.x* blockDim.x) +threadIdx.x	precalculable	blockDim.x	0/blockDim.x
edges	14	i	Induction	-	-
visited	15	edges[i]	Indirect	-	-

Listing 7.3: API calls for passing static analysis information to GPU.

```
1  cudaFlushCTATracker();
2  cudaSetCTATracker((void*)mask, sizeof(bool),
3    threads.x, 0, 0, 0,              // MIN
4    threads.x, 0, 0, threads.x); // MAX
5  cudaSetCTATracker((void*)nodes, sizeof(Node),
6    threads.x, 0, 0, 0,              // MIN
7    threads.x, 0, 0, threads.x); // MAX
8  cudaSetCTATracker((void*)cost, sizeof(int),
9    threads.x, 0, 0, 0,              // MIN
10   threads.x, 0, 0, threads.x); // MAX
11 Kernel<<< grid, threads, 0 >>>(nodes,
12   edges, mask, visited,
13   cost, over, no_of_nodes);
```

three *cudaSetCTATracker* calls to specify the data region for *mask*, *nodes*, and *cost*. For each call, the first argument specifies the base address of the array. The second argument passes the size of each array element in bytes. Next eight arguments pass *blockIdx* products and offsets so the GPU can calculate the minimum and maximum addresses for each thread block at runtime. As stated earlier, the GPU calculates the minimum and maximum addresses using Equation 7.2 and 7.3.

7.3.2 Hardware Side

Below we explain hardware modifications of TELEPORT.

Preload table

The proposed API calls allow the compiler (or optionally the programmer) to pass the preload information to hardware. Hardware uses this information to dynamically load the data regions associated with each thread block. To maintain this information in the hardware, we propose enhancing thread block dispatching unit with a table; refers to as preload table. Each row of preload table stores the information passed by a single API call. Each row registers base pointer, data type size, minimum products, and maximum products.

TELEPORT controller unit

The proposed controller unit is shared among all thread blocks and resides in the thread block dispatching unit. This unit reads every valid row of preload table to load the data region associated with the ready-for-dispatch thread blocks. Upon dispatching every thread block, the controller reads each row of the preload table, calculates the boundary of data region of the row using Eq. 3 and 4, and issues a burst of requests per row to load the data region. Requests are issued from the thread block dispatcher unit to the memory controllers. In each request, the controller attaches the network address of the target GPU core (the core on which the thread block is dispatched) as the destination. This allows the network to route the reply packets from DRAM directly to the target core. On the core, the loaded data is stored in buffer.

Data buffer

TELEPORT extends GPU cores with a logical buffer to store the data loaded via the TELEPORT controller unit. The buffer stores the loaded data for outstanding thread blocks of the core. The buffer is also responsible to count the number of received packets for each newly scheduled yet-stalled thread block and signal the warp scheduler to activate the thread block once all the packets from the TELEPORT controller unit are received at the core's end. Physically, this buffer can be a dedicated cache or any of the already available caches on the core (e.g. data cache or software-managed cache). In this work, we use the unused space of the existing CUDA shared memory as the buffer. An extra tag array is maintained along the shared memory to create a set-associative cache out of shared memory. The LRU cache replacement

Table 7.2: GPGPU-sim configurations for modeling GTX 480.

GPU chip	
GPU cores	15
Memory controllers	6
Sub partition / memory controller	2
GPU core	
L1 Cache	16KB, 32 sets, 4-way
Shared Memory	48KB, 32 banks
# of Threads	1536
Maximum concurrent thread blocks	8
# of registers	32768 32-bit
L2 Cache / sub partition	
Size	64KB, 64 sets, 8-way

policy is used when needed[5].

Load Concurrency

There might be few instructions ready to execute between the time that it takes for TELEPORT to load the entire thread block's data and when a thread actually demands the data. The GPU core can execute these instructions (in parallel to the pending memory requests issued by TELEPORT's controller unit) and hide the TELEPORT's loading delay. However, since overlapping the thread block progress and TELEPORT's loading increases the hardware complexity, we decided to simply stall the entire thread block until loading is completed.

7.4 Experimental Methodology

Modeling TELEPORT. We develop a static analyzer that highlights precalculable array indexes. While this can be accomplished automatically by the compiler, currently we manually reform this information to the proposed API calls and inject them to the benchmarks' source code. We use GPGPU-sim 3.2.2 [7] for modeling both hardware and software sides of TELEPORT. We model a hardware similar to NVIDIA GTX 480 as the baseline GPU of this study [69]. Simulation details are listed in Table 7.2.

[5]We leave investigation of alternative replacement policies to future work.

Benchmarks. We used 16 benchmarks from Rodinia [13], GPGPU-sim [7], NVIDIA GPU Computing SDK [68], and three third-party applications (edge-detection by sobel filter (EDS), Jacobi iteration (JAC), and matrix-matrix add (MMA)) in this work. We selected these 16 benchmarks as they are not merely compute-bounded and show tangible performance improvement under the ideal zero-latency memory machine. We model an ideal machine by assuming a perfect L1 cache which has a hit rate of 100%.

Evaluations. We measure performance in execution time (clock cycles) when comparing different implementations of the same benchmark (whenever different implementations come with unequal number of instructions.) Otherwise, we use IPC (instructions per cycle) as the performance metric. We assume a 64-entry preload table and a 48KB 96-way associative cache tag for TELEPORT, unless stated otherwise.

7.5 Experimental Results

We investigate performance and development effort advantages of TELEPORT, comparing three implementations of the benchmarks (comparing TELEPORT to two other implementations):

- Baseline: This implementation does not use software-managed cache. This should be prone to inefficient memory accesses.
- Hand-written: This implementation exploits software-managed cache to minimize off-chip memory accesses.
- TELEPORT: Built on top of Baseline, static analyzer parses the kernel and extracts precalculable array accesses. Using this information, *cudaSetCTA-Tracker()* calls are injected before the kernel launch call to set preload table in hardware.

Since evaluating each benchmark requires excessive amount of development effort to develop a Hand-written implementation, here we limit the evaluations to subset of the benchmarks. We limit our study to five benchmarks with extensive use of software-managed cache: EDS, HSP, JAC, MMA, and PTF benchmarks.

The rest of this section is organized as follows. In Section 7.5.1, we present performance improvements and development effort savings under TELEPORT. In

Table 7.3: Comparing development effort of TELEPORT to Hand-written shared memory version. Development effort is measured in code lines.

	TELEPORT	Hand-written	Improvement
EDS	13	19	1.46
HPS	21	51	2.43
JAC	8	23	2.88
MMA	4	9	2.25
PTF	10	34	3.4

Section 7.5.2, we investigate impacts on DRAM row locality and accesses. In Section 7.5.3, we discuss interactions between TELEPORT hardware and software.

7.5.1 Performance & Development Effort

Table 7.3 compares development effort of TELEPORT to Hand-written. We measure development effort in code lines. As reported, development effort improvements range from 1.46X to 3.4X. Below, we investigate the performance aspect.

Figure 7.5 compares execution time of these benchmarks under Baseline, Hand-written, and TELEPORT implementations. Baseline and TELEPORT execute the exact same kernel code. The number below the bar group reports the ratio of dynamic instructions saved by Baseline and TELEPORT, compared to Hand-written.

As shown, the proposed approach consistently improves performance compared to Baseline. This improvement is minor under PTF and significant in other benchmarks.

Generally, TELEPORT has four performance advantages over Hand-written implementation. Firstly, it executes less number of *dynamic instructions*, since TELEPORT controller unit removes explicit read/writes from the shared memory space. Secondly, the controller unit issues the burst of memory requests in advance, effectively lowering average *memory access latency*. Thirdly, the controller unit issues requests for the same DRAM row back-to-back, potentially improving *DRAM row locality*. Finally, TELEPORT delivers higher *GPU core occupancy* (compared to Hand-written), since it does not demand allocating shared memory space statically (unlike Hand-written). Below we discuss each benchmark specifically.

Under **EDS**, TELEPORT outperforms both Baseline and Hand-written. Most of the speedup comes from memory latency improvements. The input data for the entire thread block are fetched altogether, adequately earlier than the real demand. We

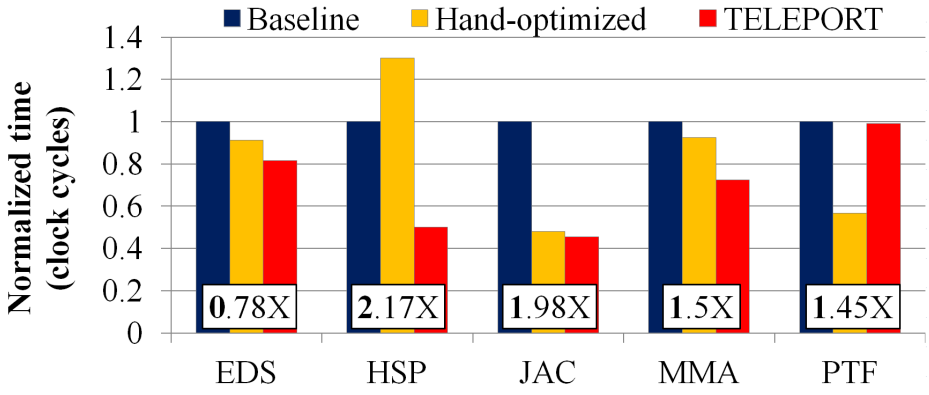


Figure 7.5: Comparing performance of TELEPORT to Baseline and Hand-written versions. The numbers below the bar group show the ratio of dynamic instructions under Hand-written over TELEPORT.

found that DRAM row locality of TELEPORT is 1.9X and 2.7X greater than Hand-written and Baseline, respectively. Meanwhile, Hand-written executes lower dynamic instructions that TELEPORT. This might seem ironic since Hand-written executes more load/stores from/to shared memory. This is explained by observing the kernel code of these two implementations. Comparing kernel code of TELEPORT (and also Baseline) to Hand-written, the latter removes large number of logical and control-flow instructions (Hand-written returns dark pixel from shared memory, instead of checking the boundary and assuring the index falls within the range of the tile in global memory, as Baseline and TELEPORT do). Baseline, Hand-written, and TELEPORT have the same occupancy in this case, 100%.

Under **HSP**, each thread block reads two 2D input tiles and writes one 2D output tile. Hand-written performs slower than other implementations as its occupancy is fairly low (50%), limited by the thread block registers usage. Compiler uses extra registers for loading/storing from/to shared memory. Also TELEPORT has advantages in executing lower dynamic instructions than Hand-written (by 2.17X).

Under **JAC**, TELEPORT delivers 2.2X speedup over Baseline. TELEPORT also outperforms Hand-written by 6% for executing 1.98X lower dynamic instructions. Baseline, Hand-written, and TELEPORT have the same occupancy in this case, 100%.

Under **MMA**, Hand-written and TELEPORT both improve the efficiency of memory accesses using software-managed cache. Comparing these two, Hand-written executes 50% more dynamic instructions. Although these extra instruction only involve two loads from shared memory and two stores to shared memory, this degrades perfor-

mance by a significant amount since the instruction sequence of threads is relatively short. Baseline, Hand-written, and TELEPORT have the same occupancy in this case, 100%.

Under **PTF**, Baseline and TELEPORT perform close. Hand-written improves performance of Baseline through algorithmic modifications. PTF is 1D stencil kernel iterating for 36 times. Hand-written reduces the total number of iterations through ghost zone optimizations [59]. This amortizes several operations in one kernel launch, which significantly improves performance. Baseline, Hand-written, and TELEPORT have the same occupancy in this case, 100%.

7.5.2 DRAM Row Locality & Accesses

The TELEPORT controller unit generates a burst of memory requests for a contiguous data region. This traffic pattern can improve DRAM row locality by mitigating row changes. However, this will not necessarily turn into a faster DRAM, since TELEPORT may simultaneously increase the total number of memory requests. This is the case when i) the percentage of memory accesses covered by TELEPORT is low or ii) the controller unit loads the entire data range while the thread block sparsely accesses the data. Generally, to have a faster DRAM with TELEPORT, we aim to i) keep memory demand as low as the baseline and ii) deliver high row locality at DRAM. Below we inspect these aspects of TELEPORT.

Figure 7.6 presents the total number of DRAM accesses under three implementations. As reported, TELEPORT total accesses are approximately equal to Baseline, for up to 2% more than Baseline (under JAC). In JAC, TELEPORT loads two extra rows and two extra columns for each tile to cover boundaries. Although all the data is not required by the threads, this covers all input data that thread block demands. The extra data fetches increase DRAM requests under TELEPORT by 2%, compared to Baseline. Hand-written version significantly reduces DRAM accesses under HSP and PTF. In these cases, Hand-written version applies ghost zone optimizations [59] to consolidate several kernels in one kernel, reducing global reads and writes.

Figure 7.7 reports the average DRAM row locality under three implementations. *Average DRAM row locality* is defined as the ratio of total row accesses to total row changes. As shown, TELEPORT generally improves DRAM row locality by reducing row changes. To reduce row changes, we use the following two techniques in hardware. First, the controller unit issues requests from single data region (or single entry of

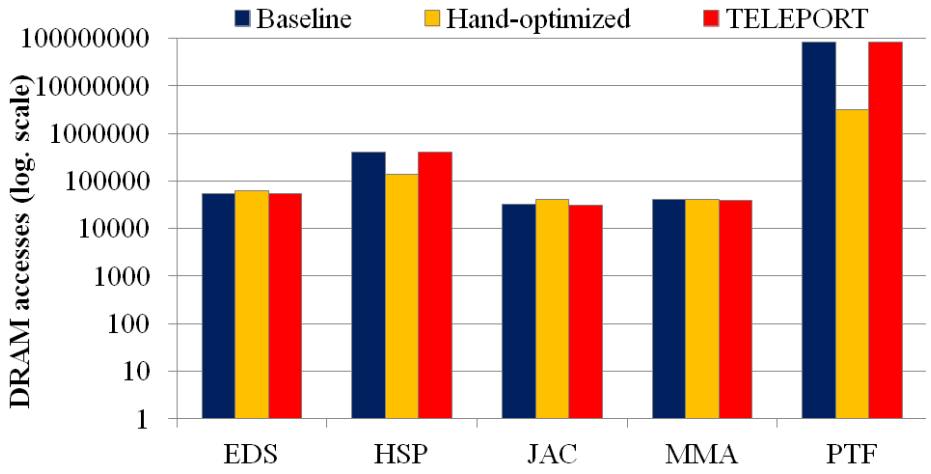


Figure 7.6: Comparing total DRAM accesses of Baseline, Hand-written, and TELEPORT implementations.

preload table or *cudaSetCTATracker()* call) back-to-back, avoiding early interleaving. Second, the memory controller prioritizes the controller unit requests over the requests coming from GPU cores. As reported, the combination of these two techniques lowers DRAM row changes by up to 4.84X. The only exception is PTF benchmark where TELEPORT degrades the row locality of Baseline. Unlike other benchmarks, PTF works on one dimensional arrays. TELEPORT fetches two input arrays into software-managed cache at the begining. One array is used at the begining and the other array is used at the end. Baseline fetches the array on demand and the locality among concurrent thread blocks improves DRAM locality. Under TELEPORT, however, the fetch process of the first array is interleaved with the fetch of the second array. This increases row changes and degrades DRAM row locality. One solution to this issue is to avoid fetching the second array to software-managed cache. In general, TELEPORT can be tuned (preferably automatically) to only include the arrays that do not incur extra row changes. Various aspects of the kernel should be inspected to identify these arrays, namely the kernel size, the location of array in the kernel (e.g. early at the begining or in the end) etc. We leave the study of adaptive TELEPORT to future work.

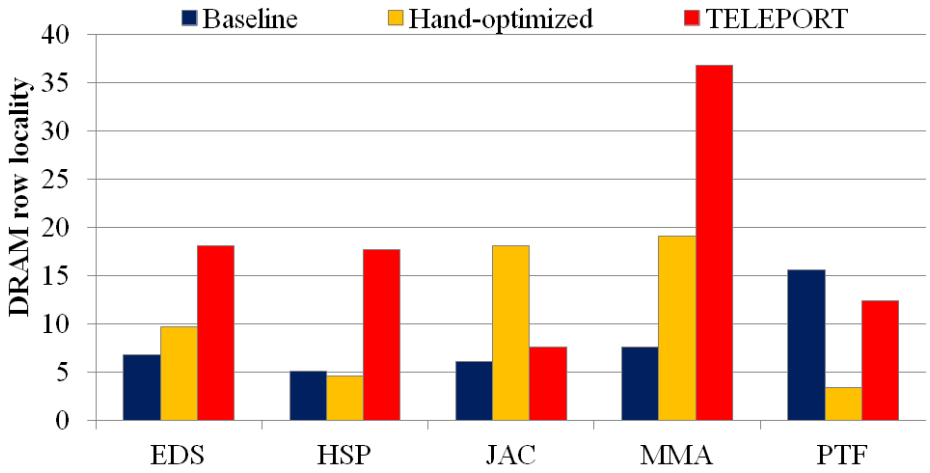


Figure 7.7: Comparing average DRAM row locality of Baseline, Hand-written, and TELEPORT implementations.

7.5.3 Hardware-Software Interactions

While TELEPORT hardware and software are closely coupled, this is interesting to investigate interactions between hardware and software and specify how much gain is pertained to either side. One way to approach this problem is to tune hardware and software configurations and evaluate how these changes would impact the overall performance.

On the software side, TELEPORT can be tuned for the best performance by adjusting the *cudaSetCTATracker* calls. Adjustments include optimizing the number of calls to *cudaFlushCTATracker* and tunning the range of data to be fetched by each of the calls. In either case, adjustments alter the memory demand imposed by TELEPORT. This accordingly impacts the pressure on global memory bandwidth and preload buffers. These adjustments can be done manually by the programmer or, preferably, by compiler heuristics.

On the hardware side, TELEPORT can be tuned for the best performance by adjusting i) thread block concurrency and ii) DRAM scheduling policy. As we discussed earlier, we evaluated a version of TELEPORT that i) stalls thread blocks until all data is loaded into the buffer and ii) always assigns higher priority to TELEPORT requests upon DRAM scheduling. An alternative thread block concurrency policy is to interleave thread block execution with the TELEPORT loading data into the buffer. An alternative DRAM scheduling policy is to assign equal priorities to all types of memory requests.

7.6 Hardware Complexity

We used CACTI 6.5 [62] in 40 nm to estimate the size of cache tag arrays and global preload table. We estimated the area of a 4-set 96-way tag array to be 0.009 mm^2. Multiplying this number by GPU cores, the total area overhead of cache tag arrays is 0.136 mm^2 $(15 * 0.009)$. Assuming 21 bytes per entry for preload table (1-byte for datatype size, 4-byte for pointer, 8X 2-byte for min/max product/offsets) and 64 entries per table, preload table's size is nearly 1344 bytes. Using CACTI, we found the area of preload table 0.004 mm^2.

Also TELEPORT controller unit needs 6 16-bit integer multipliers and 6 16-bit integer adders to calculate minimum and maximum range of data for one thread block in single cycle. Using high-performance ALUs proposed by [58], each ALU occupies 0.071 mm^2 in 90 nm. This ALU occupies roughly 0.0144 mm^2 in 40 nm, scaled with 0.198 $((\frac{40nm}{90nm})^2)$ scaling factor. Assuming 12 ALUs of this kind, the controller unit area is 0.173 mm^2.

Compared to GTX480 die size (529 mm^2), preload table, cache tag arrays, and the controller unit impose less than 1% overhead.

7.7 Summary

CUDA programmers exploit shared memory space to reduce off-core traffic. Although shared memory may deliver huge performance improvement, it imposes significant development effort. We proposed TELEPORT as a hardware/software scheme for addressing performance and productivity in GPGPUs. TELEPORT is motivated by our observation on the precalculability of memory accesses in CUDA kernels. We presented our motivation that a large share of memory accesses in CUDA kernels is statically precalculable, at the thread block granularity. This means that data tiles assigned to thread blocks can be exactly determined by knowing thread block identifiers. On average, and compared to hand-written programs, TELEPORT improves performance by 32% and lowers development effort by 2.5X.

Chapter 8

Related Work

8.1 OpenACC

It is highly desirable to maximize the performance of GPUs while minimizing the development effort. This fact drove the rise of high-level programming models for GPUs [24, 9, 48, 6, 25]. In 2011, OpenACC programming API integrated features from these high-level programming models and introduced an standard for directive-based accelerator programming [75]. In OpenACC programming model, programmers can take advantage of GPUs (and, accelerators in general) by adding few directives to the existing serial C/Fortran code. There are three advantages in using OpenACC: i) saving a huge development efforts for porting applications to accelerators, ii) applying a wide range of optimization techniques automatically at the compile time, and iii) maintaining code portability across various accelerators.

Reyes et al. [81] introduce an open-source tool, named accULL, to execute OpenACC applications on accelerators. The tool consists of a source to source compiler and a runtime library. The compiler translates OpenACC notations to the runtime library routines. The runtime library routines are implemented in both CUDA and OpenCL. Tian et al. [91] introduce an OpenACC implementation integrated in OpenUH [51]. They evaluate the impact of mapping loop iterations over GPU parallel work-items. Lee and Vetter [49] introduce a framework for compiling, debugging, and profiling OpenACC applications.

Hoshino et al. [28] investigate the impact of memory layout on the performance of NVIDIA Kepler, Intel XeonPhi, and Intel Xeon processors, under directive-based programming languages. They found that having structure-of-arrays is much more

efficient than array-of-structures under Kepler and XeonPhi, while it has minor impact on the performance of Xeon. They explain this by relatively smaller cache of Kepler (110 Bytes per hardware thread) and XeonPhi (128 KBytes per hardware thread), compared to Xeon (1048 KBytes per hardware thread). They also introduce a new directive allowing programmer to change the data layout of multi-dimensional arrays.

Herdman et al. [27] compare performance of *parallel* and *kernels* constructs under various vendor implementations of OpenACC. They found that most vendors focus on one of these constructs. Comparing quickest construct of the vendors, their performance variations found to be below 13%. They also found 15% to 20% gap between OpenACC and CUDA on NVIDIA GPUs.

Nakao et al. [63] introduce XACC as an alternative to MPI+OpenACC programming model to harness the processing power of cluster of accelerators. XACC offers higher productivity since XACC abstractions reduce the programming efforts. Under small and medium problem sizes, XACC performs up to 2.7 times faster than MPI+OpenACC. This higher performance comes from the PEACH2 interface that XACC communicates through. PEACH2 performs faster than GPUDirect RDMA over InfiniBand under data transfer size of below 256KB. Increasing the problem size, XACC and MPI+OpenACC perform comparable, since the latency of PEACH2 and GPUDirect RDMA over InfiniBand would be equal.

Murai et al. [61] propose an extension to OpenACC to facilitate multi-accelerator processing. Language extension includes *on_device* clause for data, kernels, and loop directives.

Xu et al. [101] evaluate the performance of Kirchhof Migration under OpenACC. They found OpenACC code executed on the GPU is 20.54X faster than the code executed on single 10-core CPU. They also found that gang serialization may degrade performance significantly. Instead of serialization and broadcast, they suggest to perform the computations redundantly.

Kraus et al. [33] investigated the opportunity to improve the performance of CFD workloads through OpenACC. They applied several CUDA-like optimizations at the OpenACC level, including texture cache and occupancy optimizations. They apply texture memory optimization by declaring variables as constant. They alter the streaming multiprocessor's occupancy by specifying vector_length (or thread-block size). They found that the optimal occupancy is the point with higher cache hit rate, since the CFD workloads tend to work on large working sets. They also transform

array-of-structures to structure-of-arrays to optimize memory layout (returned nearly 52% performance improvement).

Govett et al. [23] compare the performance three different OpenACC implementations under NIM workload. They perform three optimizations in their own implementation, called F2C-ACC. Among these optimizations, they found that variable demotion technique can improve performance significantly. Variable demotion avoid transfer of entire dimension of array when only certain indecies are accessed. This can decrease the memory transfer time and also allow generation of more efficient kernel code. For instance, variable demotion on a 1D array, where possible, can replace global memory array accesses with scalar or register accesses.

Wienke et al. [97] compare the performance and development cost of two OpenACC applications to their OpenCL equivalent. In terms of modified code lines, they found OpenACC demands 6.5X lower development effort compared to OpenCL. Meanwhile, they found the best-effort performance gap is of 2.5X. They observed that the OpenACC inflexibility in exploiting software-managed cache causes this large performance gap. Williams et al. [98] study different applications to evaluate the performance gap between OpenACC and CUDA implementations. Niemeyer and Sung [65] investigate the OpenACC and CUDA performance in CFD applications. They show that the performance gap between OpenACC and CUDA narrows down to a negligible value for large data sets. Herdman et al. [27] compare the performance and programming productivity of OpenACC, OpenCL, and CUDA. Under hydorodynamics applications, they found that OpenACC outperforms CUDA and OpenCL on average by 1.35X and 1.24X, respectively. In terms of words of code, they conclude OpenACC applications are 11.9X and 8.67X more productive for programming than OpenCL and CUDA, respectively. Additionally, they found OpenACC applications easier to debug and verify, compared to CUDA and OpenCL.

8.2 GPU Micro-benchmarking

Wong et al. [100] introduced a set of micro-benchmarks, revealing wide set of micro-architectural details on NVIDIA GTX280 GPU, including the latency and throughput of different instructions and cache hierarchy of different memory spaces. They also investigated the behavior of intra- thread block synchronization and intra-warp branch divergence.

Zhang et al. [104] developed a set of micro-benchmarks to understand the impact

of VLIW utilization on the power consumption of ATI GPUs. They found that increasing VLIW utilization from 20% to 100% can increase the power consumption by nearly 33%.

Anderson et al. [4] used micro-benchmarking to understand shared memory, global memory, and intra- thread block synchronization in NVIDIA Fermi architecture. They found that the achievable bandwidth of shared memory is 85.4% and global memory is 75%. They also found that the latency of shared memory and global memory are 27 and 570 cycles, respectively. They showed that address calculation overhead in Fermi architecture significantly increases the shared memory access time, while the address calculation overhead is negligible for global memory accesses. They also presented that the latency of intra- thread block synchronization depends heavily on the thread block size, if the thread block size is below 256 threads.

Aji et al. [1] developed a set of micro-benchmarking to investigate the performance impact of partition camping, contention of memory requests on the same memory controller, under NVIDIA GTX 280. They found that partition camping may degrade performance by 7X. They same micro-benchmark cannot be used under the GPUs with L1 or L2 cache, since the redundant traffic injected by the micro-benchmark would be captured by the cache.

8.3 Software-Managed Cache for GPUs

Li et al [50] compare the performance of software-managed and hardware-managed caches in NVIDIA GPUs. They study several test cases and compare two CUDA implementations. The first implementation uses shared memory and the second implementation tiles the computation and allows hardware-managed cache to store the tiles implicitly. They found three advantages in using software-managed cache. First, shared memory has higher bandwidth and less serialization compared to hardware-managed cache. This can be explained by the multi-banked design of shared memory. Second, software-managed implementation has much higher memory-level parallelism, fetching all the data before the real request. Third, the programmer can use shared memory to avoid conflict miss and low associativity problems which may harm performance in hardware-managed caches. On the other hand, they also found two advantages in using the hardware-managed cache. First, it executes lower number of dynamic instructions, since the software-managed cache version has extra operations for interchanging data between shared memory and register file. Second,

hardware-managed cache implementation may have higher occupancy since the number of thread blocks per SM may not be limited by the shared memory usage.

Fang et al. [17] proposed ELMO APIs to lower OpenCL development effort in utilizing local memory. They explained challenges in introducing a high-level API for local memory fetch, writeback, and communication. They also investigated performance of various implementation alternatives. They found 1.3X to 3.7X performance improvement over the baseline (without local memory). They do not report development effort advantages over hand-written version.

CUDA programmers control threads of thread blocks to optimize both data and computation patterns. This can fail under scenarios where optimizing data and computation patterns simultaneously is not possible. CudaDMA API [8] aims to improve both performance and productivity by decoupling data and computation patterns. CudaDMA allows splitting thread block into compute and DMA warps. DMA warps manage the movement of data between shared memory and DRAM and compute warps are responsible for computation.

Lee and Vetter [49] introduce *openarc* directive to allow OpenACC programmer to map OpenACC arrays to CUDA memory spaces, including shared and texture memory spaces. They do not investigate the effectiveness of their proposal for these mappings. Based on the short introduction that they present, we believe their proposal for utilizing shared memory is different from our *fcw* directive in two ways. Firstly, while the *openarc* directive needs programmer to separate shared memory array and corresponding global memory array in the code, the *fcw* directive separates the arrays automatically, based on the information presented by the programmer. Secondly, while the *openarc* directive allows fine-grained control to OpenACC programmer to perform fetch, synchronization, and writeback, *fcw* directive handles these operations implicitly. Based on these differences, we consider the *fcw* directive as a high-level proposal for utilizing SMC and openarc as a low-level fine-grained control over SMC.

OpenACC provides the *tile* clause and *cache* directive to hint the compiler on using software-managed caches. As follows we discuss the capabilities and limitations of these clauses in exploiting them as shared memory of CUDA (or local memory of OpenCL). The *tile* is a clause used by loop directive to split the loop into two loops; outer and inner. This clause is intended to specify a boundary for splitting the loop calculations over different gangs. Programmers do not contribute to tiling adjustments as the compiler carries the necessary tiling optimizations. Although this clause introduces the tiling concept of CUDA/OpenCL in OpenACC programming

model, there are still two limitations in employing it for utilizing SMC effectively. First, unlike CUDA/OpenCL, the *tile* does not provide enough information to allow the compiler exploit the software-managed cache as an inter-thread communication medium. Second, developers cannot control the software-managed cache at the programming level via this clause. The *tile* directive improves the locality of accesses and assists the accelerator in fitting the working set of gangs in the small-sized automatic hardware cache. The *cache* directive can be used within work-sharing loops to fetch a memory region into the highest level of accelerator's cache hierarchy. It can mark a subset of array elements to exploit locality efficiently. Although the compiler can use this hint to reduce costly global memory accesses, there are opportunities that cannot be captured by the *cache* directive. For example, as we have shown in Chapter 6, communication between neighbor iterations of a work-sharing loop, similar to the communication across the threads within a CUDA thread block, is not possible under the *cache* directive. Sharing intermediate memory writes among neighbor iterations can reduce global memory communications significantly. In summary, the *tile* clause and *cache* directive do not allow communication through SMC and they can be considered as a specific type of *fcw* directive, FETCH_ONLY. A previous work [28] also found the same limitation in the *cache* directive.

8.4 Prefetching

Cache fetch algorithms are either demand fetch or prefetch methods [87]. Demand requests are actual memory requests needed by the application. Prefetch requests speculate actual accesses and load them into the cache in advance. Prefetching mechanisms can solely be a hardware approach, not modifying the application code, or accept hints from the software. While prefetching mechanisms are independent from the application flow, they are still tightly related to and can be imagined as a concurrent process predicting memory accesses of the main process. Overviewing the prefetching literature since 1978[1], four major challenges should be addressed for prefetching to achieve performance improvment: prefetching timeliness (distance), accuracy, excessive bandwidth usage, and cache pollution. i) timeliness: a prefetch request should be initiated with proper timing so it can lower the overall memory access latency. If initiated too early, data might be evicted in the cache before be-

[1]We refer readers to study [86] for work prior to this.

ing used, returning no benefit. If initiated too late, and near the time that demand fetch happens, prefetching will not reduce memory latency. To address this, previous work propose to set the prefetching distance statically for the next Nth access [12, 20] or adjust the distance adaptively [103, 55]. ii) accuracy: a prefetching mechanism should assure the addresses (of future demand fetches) are being predicted with high accuracy. This might be trivial for simple patterns (e.g. iterating through an array sequentially [32]), but involves a complicated compiler pass on sophisticated patterns (e.g. pointer-chasing patterns [56].) iii) excessive bandwidth: prefetching increases the memory bandwidth usage of the application significantly. This can negate performance, if prefetching is wasting the bandwidth with wrong predictions [2]. Prior work suggest various heuristics to estimate prefetching usefulness and drop/filter useless prefetching out [105, 88, 46, 15, 16, 53]. iv) cache pollution: If the prefetched data share cache space with the demand fetch data, there is a risk that prefetching may evict lines from the working set of the application. To address this, using a dedicated prefetch buffer or cache partitioning are suggested [80, 22, 83].

Prefetching in GPU computing has also been investigated. Ryoo et al. [82] investigated software prefetching in the matrix multiplication test case. They found prefetching advantageous so long register pressure does not degrade occupancy. Lee et al. [47] evaluate several hardware prefetching mechanisms. Generally, they found that memory patterns are highly predictable. They also found out that a significant challenge stems from excessive memory bandwidth usage and lack of prefetch timeliness. They introduced a threshold-based heuristic to address these challenges. Jog et al. [31] report how integrating the warp scheduler into the prefetching mechanism unchains the real performance potential behind prefetching in GPGPUs. They show that conventional warp schedulers keep warps at a close pace, accessing nearby cache blocks during short intervals. If warps are scheduled far apart, then warps can potentially prefetch for each other. They show this careful warp scheduling combined with a simple prefetcher can yield significant speedup. Sethia et al. [84] used prefetching as a technique to improve energy-efficiency. They exploited prefetching to improve memory latency hiding. In addition, they lowered thread-level parallelism to save energy, while maintaining performance. Jeon et al. [29] found that the memory access pattern of the threads within the same thread block is strongly strided. In order to predict memory accesses of the thread block, their main challenge is to find the base address and stride value. To calculate base addresses, the warp scheduler prioritizes a single warp (from each thread block) to run ahead of the rest of the warp. The

stride value is calculated by subtracting memory addresses that are issued from two successive warps upon executing the same instruction. Lakshminarayana and Kim [35] propose an approach for predicting the address of load instructions that depend on another load instruction. They tune the work for GPGPUs and address cache pollution by using spare registers' of the threads for prefetching peace requirements.

The approach taken by TELEPORT is fundamentally different from prefetching, as unlike prefetching, TELEPORT deals with *precalculating (absolute accuracy)* and not *speculation*. Accuracy is not a concern in TELEPORT as it precisely precalculates the range of data that will be accessed within each thread block. This is achieved in two steps: a software step that finds the range as a function of thread block identifier and a hardware step that evaluates the range by assigning values to thread block identifiers.

Prefetching and TELEPORT are orthogonal techniques and therefore can be employed in the same system. Under such circumstances, the prefetching mechanism can search TELEPORT's buffer before issuing a prefetch request. This lowers prefetching memory bandwidth usage as the prefetch request might have hit in the buffer already.

8.5 DRAM Efficiency

Yuan et al. [102] studied DRAM row locality in GPGPUs. They found that the memory traffic generated by cores has a very high row locality. However, these requests are reordered on the network on chip (NoC) that connects GPU cores to memory controllers. Built on this observation, they suggest NoC optimizations to deliver a performance close to that of complex DRAM schedulers.

Jog et al. [30] observed that when a DRAM row is open, sooner or later, most columns are read from the row. They suggest prefetching more columns than the demand (adaptively set between 8 to 16) into the L2 cache when a row is open. They show that while this can increase the latency of demand fetch stream, the overall impact on performance is positive as row-conflicts are reduced.

Chapter 9

Conclusions and Future Work

9.1 Conclusion

Over the past decade, GPGPUs have become a standard compute-accelerator in high-performance computing centers. While developing naïve version of an application in low-level CUDA-like programming models is affordable, optimizing the application in such models can be very expensive (in terms of development effort) since the optimization space is gigantic. Various optimization techniques have been proposed, well tested, and matured in low-level programming models and it is the time to apply such optimizations by compiler automatically or by hints. This is the reason behind the invent of high-level accelerator programming models. OpenACC programming API, for example, allows the programmer to port serial C/Fortran code to GPU by adding few directives. High-level programming models mainly rely on compiler innovations to apply well-tested optimizations on the code and perform close to hand-written low-level equivalents. In this dissertation, we analyzed performance limitations of high-level programming models, compared them to low-level programming models, and proposed solutions to address these limitations.

We started by evaluating the performance gap between high-level and low-level programming models, OpenACC versus CUDA specifically. We developed our in-house OpenACC compiler, referred to as IPMACC. IPMACC translates OpenACC for C files to CUDA files and uses NVIDIA nvcc compiler for compiling CUDA files and generating GPU binaries. IPMACC is publically available on github [37]. Our goal in developing IPMACC was to implement OpenACC as near to CUDA version as possible. This creates a platform to accurately comment on the performance gap

between OpenACC and CUDA. We used IPMACC for comparing OpenACC and CUDA versions of 10 benchmarks. We reported the breakdown of the time spent in i) data movements between CPU and GPU, ii) launching kernels on the GPU, and iii) running the kernels on the GPU. We showed that the OpenACC version of the application can be up to 2.6X slower than the equivalent CUDA version. We found that most benchmarks spend roughly the same amount of time on data movements under both OpenACC and CUDA. However, comparing the kernel launch and runtime, we found that low-level software-managed cache optimizations that are applied in CUDA versions are the main reasons behind large performance gaps between OpenACC and CUDA. Motivated by this observation, we proposed solutions to address this limitation.

Initially, we investigated potential benefits in OpenACC programming API available by utilizing GPU software-managed cache. The *cache* directive in OpenACC hints the compiler to cache a subarray in the highest level of the cache. We suggested three different implementations of the *cache* directive, plus various optimizations on top of each implementation. We found cache sharing optimization essential for boosting the cache directive's performance. To tune our cache implementation for best performance, we derived a set of micro-benchmarks to understand global memory and software-managed cache of the GPUs. For instance, we found that adding a small padding during allocation of software-managed cache vastly resolves bank conflicts. We used these findings in optimizing our cache directive (and generally our OpenACC) implementation. We studied the configuration space of our proposal under three different benchmarks. We showed that our best implementation is able to significantly shrink the performance gap between OpenACC and CUDA, accelerating the baseline OpenACC version by up to 2.4X. We also measured the development effort advantages of OpenACC over CUDA in software-managed cache programming. In terms of number of statements, we found that OpenACC comes with 29% lower development effort than CUDA.

Although the *cache* directive allows utilizing software-managed cache in OpenACC, the directive does not offer the full functionality associated with the software-managed cache. In CUDA, for instance, threads may communicate through software-managed cache locally. This local communication is very beneficial to the overall performance as it reduces the total number of global memory accesses and kernel launches. In OpenACC, however, the communication model among loop iterations is not defined and iterations may not communicate through software-managed cache.

We identified and explained the limitations of the *cache* directive in fully utilizing the software-managed cache potential. To address these limitations, we proposed to extend the OpenACC API with a new directive, referred to as the *fcw* directive. We introduced a communication model along with the *fcw* directive that allows a certain range of subsequent iterations to communicate through the software-managed cache. The *fcw* directive and *cache* directive can be used interchangeably for caching a range of read-only data in the software-managed cache and have the exact same performance and development effort. However, the applicability of the *fcw* directive is wider than the *cache* directive. The *fcw* directive can be used for exchanging data among loop iterations through software-managed cache. We applied the *fcw* directive to six different benchmarks to should how this local communication can unleash OpenACC performance and reduce the performance gap between OpenACC and CUDA. For example, under Hotspot benchmark, we showed that the *fcw* directive improves the performance of the baseline OpenACC (that does not use software-managed cache) by up to 2.8X and lowers the performance gap between OpenACC and CUDA down to 10% (from 206%).

While the *cache* and *fcw* directive lower the development effort of software-managed cache programming, they still require the developer to analyze the code and find the caching opportunity. In Chapter 7, we showed that static compiler passes can automatically identify very high percentage of the caching opportunities in GPGPU. We proposed a hardware/software mechanism, referred to as TELEPORT, to fully automate the software-managed cache programming to exploit these caching opportunities. TELEPORT starts with analyzing the static CUDA kernel code at compile time and identifying caching opportunities. TELEPORT reads the array indexes and if it can resolve the array index in a particular format, then it considers the array for caching. We introduced an API call to capsulate this information and pass it to the GPU hardware. These API calls are injected before the kernel launch call to pass the information to the GPU, right before the kernel launch. During runtime, the range associated with each thread block is calculated and fetched into the GPU core that the thread block is scheduled on. We evaluated TELEPORT under diverse set of benchmarks, showing both opportunities and limitations of the mechanism. We compared TELEPORT to the baseline (without software-managed cache) and the hand-written CUDA version optimized for software-managed cache. Compared to hand-written version, TELEPORT lowers the development effort by 46% to 188%. Compared to the baseline, TELEPORT improves the performance by 1% to 120%.

TELEPORT does not replace CUDA software-managed cache programming. Because TELEPORT is limited by the intelligence of the static compiler pass it may not always be possible to use it to implement complex software-managed cache algorithms. We showed this limitation in our evaluation by investigating PTF benchmark. In summary, if TELEPORT is able to implement the same caching algorithm as CUDA, it results in the following four advantages over CUDA: i) executing lower number of dynamic instructions, ii) improving DRAM row locality by sending subsequent column reads in a burst, iii) reducing memory latency by sending requests in advance, and iv) improving GPU core occupancy.

9.2 Moving Forward

In Chapter 6 we introduced a novel directive to allow controlling software-managed cache through OpenACC API. We believe there is room for performance improvement in the proposed directive by optimizing the implementation further. For example, synchronizations overhead of fetching tiles to software-managed cache can be significant if there are multiple tiles to fetch (e.g. matrix-matrix multiplication). Static heuristics are required to figure out if the tiles are equal in length and if they can be fetched using a single common loop. Currently, our only heuristic is if the range identifiers are the same for two fcw clauses, they can be merged and fetched via a single common loop. However, in matrix-matrix multiplication sample, the heuristic is unable to find the opportunity for merging the two tile fetching loops in one common loop. This is because the heuristic is unable to statically conclude that the two tiles are equal in length. Improving this heuristic can enhance performance further.

Static analyzer in TELEPORT parses the kernel and finds arrays with the caching opportunity. In the implementation of TELEPORT introduced in Chapter 7, all of these arrays are considered for caching and API calls are injected to cache these arrays during runtime. However, this might not be optimal way to cache *all* these arrays at the *beginning*. Firstly, caching all arrays can cause buffer overflow. This can be avoided by *selectively* caching the arrays. One heuristic, for example, can be to prioritize arrays based on the access frequency. Secondly, all arrays may not be needed at the beginning. Instead, arrays can be scheduled to be cached slightly in advance, not necessarily at the beginning. This also helps avoiding the buffer overflow and reduces the cache space demand at the beginning. We believe there is an opportunity to improve TELEPORT by tuning the API calls.

Appendix A

Code Modification for fcw Directive

A.1 Pathfinder

Listing A.1 and A.2 show baseline and fcw implementations of Pathfinder. The fcw version implements the ghost zone algorithm and differs from the baseline in four ways. Firstly, the most outer loop in the baseline is divided into inner and outer loops. Secondly, the number of iterations of the parallel loop is increased to allow enough parallel threads for redundant ghost zone calculations. Also part of the code maps the loop inductive variable to the output index. Thirdly, fcw directive is added over the most inner loop. The fcw type is FETCH_CHANNEL which allows intermediate writes to be visible to neighbor threads. Finally, the final state of the local copy (in the software-managed cache) is written back by the loop iterations that were active in recent iterations. True value of the boolean variable, named *computed* in the code, indicates that the thread was active in the last inner iteration of the ghost zone and should write the final value back to global memory.

Listing A.1: Pathfinder baseline OpenACC.

```
#pragma acc data copyin(src[0:cols],data[0:rows*cols]) copyout(dst[0:cols])
{
  for (int t = 0; t < rows-1; t++) {
    temp = src;
    src = dst;
    dst = temp;
```

```
    #pragma acc kernels
    #pragma acc loop private(min) independent
    for(int n = 0; n < cols; n++){
      min = src[n];
      if (n > 0)
        min = MIN(min, src[n-1]);
      if (n < cols-1)
        min = MIN(min, src[n+1]);
      dst[n] = (data[(t+1)*cols+n]) + min ;
    }
  }
}
```

Listing A.2: Pathfinder fcw OpenACC.

```
#define TILESIZE 256
#define TILESIZELOG 8
#pragma acc data copyin(src[0:cols],data[0:rows*cols]) copyout(dst[0:cols])
{
  for (int t=0; t < rows-1; t+=inner_iter){
    temp = src;
    src = dst;
    dst = temp;
    int pad=((cols/(TILESIZE-2*inner_iter))+1)*inner_iter*2;
    #pragma acc kernels
    #pragma acc loop private(min) independent vector(TILESIZE)
    for(int np=0; np<(cols+pad); np++){
      int n=np-((np>>TILESIZELOG)*2*inner_iter+inner_iter);
      int localid=np&(TILESIZE-1);
      int val=0;
      bool computed=false;
      #pragma acc fcw FETCH_CHANNEL(src[n:0:0])
      {
        for(int i=0; i<inner_iter; i++){
          int row=t+i+1;
          if(row<rows){
            computed=false;
```

```
            if(localid>i && localid<(TILESIZE-i-1) && n>=0 && n<cols){
              min = src[n];
              computed=true;
              if (n > 0)
                min = MIN(min, src[n-1]);
              if (n < (cols-1))
                min = MIN(min, src[n+1]);
              val = (data[row*cols+n]) + min ;
              src[n]=val;
            }
          }
        }
      }
      if(computed){
        dst[n]=val;
      }
    }
  }
}
```

A.2 Matrix-matrix Multiplication

Listing A.3 and A.4 show baseline and fcw implementations of Matrix-matrix Multiplication. The fcw implementation differs from the baseline in two ways. Firstly, the most inner loop is decomposed into inner and outer loops allowing a different tile of data to be processed each time. Secondly, the fcw directive is added over the inner loop to cache the active tile.

Listing A.3: Matrix-matrix multiplication baseline OpenACC.

```
#pragma acc data pcopyin(a[0:LEN][0:LEN],b[0:LEN][0:LEN])
    pcopyout(c[0:LEN][0:LEN])
#pragma acc kernels
#pragma acc loop independent
for (i = 0; i < LEN; ++i) {
  #pragma acc loop independent
  for(j=0; j<LEN; j++){
```

```
    float sum=0;
    for(m=0; m<LEN; m++){
      sum += a[i][m]*b[m][j];
    }
    c[i][j]=sum;
  }
}
```

Listing A.4: Matrix-matrix multiplication fcw OpenACC.

```
#define MIN(a,b) ((a<b)?a:b)
#pragma acc data pcopyin(a[0:LEN][0:LEN],b[0:LEN][0:LEN])
    pcopyout(c[0:LEN][0:LEN])
#pragma acc kernels
#pragma acc loop independent vector(16)
for(i=0; i<LEN; ++i){
  #pragma acc loop independent vector(16)
  for(j=0; j<LEN; ++j){
    float sum=0;
    for(l=0; l<LEN; l+=16){
      int offseti=(i&(16-1))+l;
      int offsetj=(j&(16-1))+l;
      #pragma acc fcw FETCH_ONLY(a[i:0:0:offsetj:0:0],b[offseti:0:0:j:0:0])
      if(j<LEN && i<LEN){
        for(m=l; m<MIN(l+16,LEN); m++){
          sum += a[i][m]*b[m][j];
        }
      }
    }
    if(j<LEN && i<LEN){
      c[i][j]=sum;
    }
  }
}
```

A.3 Hotspot

Listing A.5 and A.6 show baseline and fcw implementation of Hotspot. Modifications are very similar to the modifications in Pathfinder (Section A.1) as both benchmarks implement the ghost zone algorithm. Hotspot is more complex than Pathfinder, since Hotspot implements the two-dimensional ghost zone. Here fcw code differs from the baseline in five ways in implementing the two-dimensional ghost zone algorithm. Firstly, iterations of the outer row and column parallel loops are increased to supply enough threads for the redundant ghost zone calculations. Also the output identifier of each thread is calculated in early lines of each loop. Secondly, the fcw directive is added over the inner loop to cache intermediate writes. The fcw type is FETCH_CHANNEL which makes intermediate writes visible to neighbor loop iterations. Thirdly, an inner loop is added to perform local iterations. Fourthly, a guard is added over the calculation body of the inner loop to specify the threads which must be active in each iteration of the ghost zone. Lastly, the final value of the software-managed cache is written back to global memory.

Listing A.5: Hotspot baseline OpenACC.

```
#pragma acc kernels present(temp, power, result)
#pragma acc loop independent
for (r = 0; r < row; r++) {
  #pragma acc loop independent
  for (c = 0; c < col; c++) {
    int S=(r==(row-1))?row-1:r+1;
    int N=(r==(0))?0:r-1;
    int W=(c==(0))?0:c-1;
    int E=(c==(col-1))?col-1:c+1;
    delta = (stepCap) * (power[r][c] +
        (temp[S][c] + temp[N][c] - 2.0*temp[r][c])*rRy +
        (temp[r][E] + temp[r][W] - 2.0*temp[r][c])*rRx +
        (amb_temp - temp[r][c])*rRz);
    result[r][c] =temp[r][c]+ delta;
  }
}
```

Listing A.6: Hotspot fcw OpenACC.

```
int dimrow=(row+((2*innerIter)*(row/(16-2*innerIter)+1)));
int dimcol=(col+((2*innerIter)*(col/(16-2*innerIter)+1)));
#pragma acc kernels present(temp, power, result, written)
#pragma acc loop independent vector(16)
for (rs=0; rs<dimrow; rs++) {
  int r=rs-((rs>>4)*2*innerIter+innerIter);
  #pragma acc loop independent vector(16)
  for (cs=0; cs<dimcol; cs++) {
    int c=cs-((cs>>4)*2*innerIter+innerIter);
    double new_temp;
    bool compute=false;
    int S=(r==(row-1))?row-1:r+1;
    int N=(r==(0)) ?0:r-1;
    int W=(c==(0)) ?0:c-1;
    int E=(c==(col-1))?col-1:c+1;
    #pragma acc fcw FETCH_CHANNEL(temp[r:0:0:c:0:0],power[r:0:0:c:0:0])
    {
      for(int iter=0; iter<innerIter; iter++){
        compute=false;
        if( (r>=0) && (c>=0) && (r<row) && (c<col) &&
            (((rs&(16-1))>iter) && ((cs&(16-1))>iter)) &&
            (((rs&(16-1))<(16-iter-1))&&((cs&(16-1))<(16-iter-1))) ){
          compute=true;
          delta = (stepCap) * (power[r][c] +
              (temp[S][c] + temp[N][c] - 2.0*temp[r][c]) * rRy +
              (temp[r][E] + temp[r][W] - 2.0*temp[r][c]) * rRx +
              (amb_temp - temp[r][c]) * rRz);
          new_temp= delta + temp[r][c];
        }
        if(iter==(innerIter-1))
          break;
        if(compute){
          temp[r][c] = new_temp;
        }
      }
    }
    if(compute){
```

```
      result[r][c] = new_temp;
    }
  }
}
```

A.4 N-Body

N-Body simulation has two kernels where one of them can be optimized to take advantage of software-managed cache. Listing A.7 and A.8 show this kernel under baseline and fcw implementations. There are two differences between the baseline and fcw. Firstly, the most inner loop is decomposed to inner and outer to perform operations on tiles. Secondly, the fcw directive is added over the inner loop to cache the active tile in the software-managed cache.

Listing A.7: N-Body baseline OpenACC.

```
#pragma acc data pcopyin(m_force[0:m_numBodies],m_pos[0:m_numBodies])
#pragma acc kernels
#pragma acc loop independent
for (int i = 0; i < m_numBodies; i++){
  float3 acc;
  acc.x=0;
  acc.y=0;
  acc.z=0;
  float4 pos_p=m_pos[i];
  int j = 0;
  for(j=0; j< m_numBodies; j++){
    bodyBodyInteraction(&acc, pos_p, m_pos[j], m_softeningSquared);
  }
  m_force[i] = acc;
}
```

Listing A.8: N-Body fcw OpenACC.

```
#define TILESIZE 256
#define TILESIZELOG 8
#pragma acc data pcopyin(m_force[0:m_numBodies],m_pos[0:m_numBodies])
```

```
#pragma acc kernels
#pragma acc loop independent
for(int i=0; i<m_numBodies; i++){
  float3 acc;
  acc.x=0;
  acc.y=0;
  acc.z=0;
  float4 m_pos_local = m_pos[i];
  for(int tile=0; tile<=((m_numBodies>>TILESIZELOG)+1); tile++){
    int bound = bnd1<m_numBodies?((tile+1)*TILESIZE):m_numBodies;
    #pragma acc fcw FETCH_ONLY(m_pos[(tile*TILESIZE):0:TILESIZE])
    for(int j=(tile*TILESIZE); j<bound; j++){
      bodyBodyInteraction(&acc, m_pos_local, m_pos[j], m_softeningSquared);
    }
  }
  m_force[i]=acc;
}
```

Bibliography

[1] Ashwin M. Aji, Mayank Daga, and Wu-chun Feng. Bounding the effect of partition camping in gpu kernels. In *Proceedings of the 8th ACM International Conference on Computing Frontiers*, CF '11, pages 27:1–27:10, New York, NY, USA, 2011. ACM.

[2] A. R. Alameldeen and D. A. Wood. Interactions between compression and prefetching in chip multiprocessors. In *2007 IEEE 13th International Symposium on High Performance Computer Architecture*, pages 228–239, Feb 2007.

[3] AMD Inc. AMD graphics cores next (GCN) architecture. *Available: https://www.amd.com/Documents/GCN_Architecture_whitepaper.pdf (Accessed: 2017-07-12).*

[4] M.J. Anderson, D. Sheffield, and K. Keutzer. A predictive model for solving small linear algebra problems in gpu registers. In *Parallel Distributed Processing Symposium (IPDPS), 2012 IEEE 26th International*, pages 2–13, May 2012.

[5] Jorg Arndt. *Matters computational*. Springer, Chapter 23, 2011.

[6] Eduard Ayguadé, Rosa M. Badia, Pieter Bellens, Daniel Cabrera, Alejandro Duran, Roger Ferrer, Marc Gonzàlez, Francisco Igual, Daniel Jiménez-González, Jesús Labarta, Luis Martinell, Xavier Martorell, Rafael Mayo, Josep M. Pérez, Judit Planas, and Enrique S. Quintana-Ortí. Extending openmp to survive the heterogeneous multi-core era. *International Journal of Parallel Programming*, 38(5):440–459, 2010.

[7] A Bakhoda, G.L. Yuan, W.W.L. Fung, H. Wong, and T.M. Aamodt. Analyzing CUDA workloads using a detailed GPU simulator. In *Performance Analysis of Systems and Software, 2009. ISPASS 2009. IEEE International Symposium on*, pages 163–174, April 2009.

[8] Michael Bauer, Henry Cook, and Brucek Khailany. Cudadma: Optimizing gpu memory bandwidth via warp specialization. In *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '11, pages 12:1–12:11, New York, NY, USA, 2011. ACM.

[9] Stéphane Bihan, Georges-Emmanuel Moulard, Romain Dolbeau, Henri Calandra, and Rached Abdelkhalek. Directive-based heterogeneous programming–a GPU-accelerated RTM use case. In *Proceedings of the 7th international conference on computing, communications and control technologies*, 2009.

[10] Bill Dally. Challenges for future computing systems. *Keynote speech at The 10th HiPEAC. Available: http://www.cs.colostate.edu/%7Ecs575dl/Sp2015/Lectures/Dally2015.pdf (Accessed: 2017-07-12).*

[11] Claudio Bonati, Enrico Calore, Simone Coscetti, Massimo D'Elia, Michele Mesiti, Francesco Negro, Sebastiano Fabio Schifano, and Raffaele Tripiccione. Development of scientific software for hpc architectures using openacc: The case of lqcd. In *Proceedings of the 2015 International Workshop on Software Engineering for High Performance Computing in Science*, SE4HPCS '15, pages 9–15, Piscataway, NJ, USA, 2015. IEEE Press.

[12] David Callahan, Ken Kennedy, and Allan Porterfield. Software prefetching. In *Proceedings of the Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS IV, pages 40–52, New York, NY, USA, 1991. ACM.

[13] Shuai Che, Michael Boyer, Jiayuan Meng, David Tarjan, Jeremy W. Sheaffer, Sang-Ha Lee, and Kevin Skadron. Rodinia: A benchmark suite for heterogeneous computing. In *Proceedings of the 2009 IEEE International Symposium on Workload Characterization (IISWC)*, IISWC '09, pages 44–54, Washington, DC, USA, 2009. IEEE Computer Society.

[14] NVIDIA Corp. Profiler's user guide: nvprof. *Available: http://docs.nvidia.com/cuda/profiler-users-guide/#nvprof-overview (Accessed: 2017-07-12).*

[15] E. Ebrahimi, O. Mutlu, and Y. N. Patt. Techniques for bandwidth-efficient prefetching of linked data structures in hybrid prefetching systems. In *2009 IEEE*

15th International Symposium on High Performance Computer Architecture, pages 7–17, Feb 2009.

[16] Eiman Ebrahimi, Onur Mutlu, Chang Joo Lee, and Yale N. Patt. Coordinated control of multiple prefetchers in multi-core systems. In *Proceedings of the 42Nd Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO 42, pages 316–326, New York, NY, USA, 2009. ACM.

[17] Jianbin Fang, A.L. Varbanescu, Jie Shen, and H. Sips. Elmo: A user-friendly api to enable local memory in opencl kernels. In *Parallel, Distributed and Network-Based Processing (PDP), 2013 21st Euromicro International Conference on*, pages 375–383, Feb 2013.

[18] Kayvon Fatahalian and Mike Houston. A closer look at gpus. *Commun. ACM*, 51(10):50–57, October 2008.

[19] Wilson W. L. Fung, Ivan Sham, George Yuan, and Tor M. Aamodt. Dynamic warp formation and scheduling for efficient gpu control flow. In *Proceedings of the 40th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO 40, pages 407–420, Washington, DC, USA, 2007. IEEE Computer Society.

[20] I. Ganusov and M. Burtscher. Future execution: a hardware prefetching technique for chip multiprocessors. In *14th International Conference on Parallel Architectures and Compilation Techniques (PACT'05)*, pages 350–360, Sept 2005.

[21] Michael Garland, Scott Le Grand, John Nickolls, Joshua Anderson, Jim Hardwick, Scott Morton, Everett Phillips, Yao Zhang, and Vasily Volkov. Parallel computing experiences with cuda. *IEEE Micro*, 28(4):13–27, July 2008.

[22] Binny S. Gill and Dharmendra S. Modha. Sarc: Sequential prefetching in adaptive replacement cache. In *Proceedings of the Annual Conference on USENIX Annual Technical Conference*, ATEC '05, pages 33–33, Berkeley, CA, USA, 2005. USENIX Association.

[23] Mark Govett, Jacques Middlecoff, and Tom Henderson. Directive-based parallelization of the nim weather model for gpus. In *Proceedings of the First Workshop on Accelerator Programming Using Directives*, WACCPD '14, pages 55–61, Piscataway, NJ, USA, 2014. IEEE Press.

[24] The Portland Group. PGI Fortran & C accelerator programming model. *Available: https://www.pgroup.com/lit/whitepapers/pgi_accel_prog_model_1.2.pdf (Accessed: 2017-07-12).*

[25] T. D. Han and T. S. Abdelrahman. hicuda: High-level gpgpu programming. *IEEE Transactions on Parallel and Distributed Systems*, 22(1):78–90, Jan 2011.

[26] Mark Harris. Optimizing parallel reduction in CUDA. *Available: http://developer.download.nvidia.com/assets/cuda/files/reduction.pdf (Accessed: 2017-07-12).*

[27] J.A. Herdman, W.P. Gaudin, S. McIntosh-Smith, M. Boulton, D.A. Beckingsale, A.C. Mallinson, and S.A. Jarvis. Accelerating hydrocodes with openacc, opecl and cuda. In *High Performance Computing, Networking, Storage and Analysis (SCC), 2012 SC Companion:*, pages 465–471, Nov 2012.

[28] Tetsuya Hoshino, Naoya Maruyama, and Satoshi Matsuoka. An OpenACC Extension for Data Layout Transformation. In *Proceedings of the First Workshop on Accelerator Programming Using Directives*, WACCPD '14, pages 12–18, Piscataway, NJ, USA, 2014. IEEE Press.

[29] Hyeran Jeon, Gunjae Koo, and Murali Annavaram. Cta-aware prefetching for gpgpu. *Computer Engineering Technical Report CENG-2014-08*, 2014.

[30] Adwait Jog, Onur Kayiran, Nachiappan Chidambaram Nachiappan, Asit K. Mishra, Mahmut T. Kandemir, Onur Mutlu, Ravishankar Iyer, and Chita R. Das. Owl: Cooperative thread array aware scheduling techniques for improving gpgpu performance. *SIGPLAN Not.*, 48(4):395–406, March 2013.

[31] Adwait Jog, Onur Kayiran, Asit K. Mishra, Mahmut T. Kandemir, Onur Mutlu, Ravishankar Iyer, and Chita R. Das. Orchestrated scheduling and prefetching for gpgpus. In *Proceedings of the 40th Annual International Symposium on Computer Architecture*, ISCA '13, pages 332–343, New York, NY, USA, 2013. ACM.

[32] Norman P. Jouppi. Improving direct-mapped cache performance by the addition of a small fully-associative cache and prefetch buffers. In *Proceedings of the 17th Annual International Symposium on Computer Architecture*, ISCA '90, pages 364–373, New York, NY, USA, 1990. ACM.

[33] J. Kraus, M. Schlottke, A. Adinetz, and D. Pleiter. Accelerating a c++ cfd code with openacc. In *Accelerator Programming using Directives (WACCPD), 2014 First Workshop on*, pages 47–54, Nov 2014.

[34] David Kroft. Lockup-free instruction fetch/prefetch cache organization. In *Proceedings of the 8th Annual Symposium on Computer Architecture*, ISCA '81, pages 81–87, Los Alamitos, CA, USA, 1981. IEEE Computer Society Press.

[35] N. B. Lakshminarayana and H. Kim. Spare register aware prefetching for graph algorithms on gpus. In *2014 IEEE 20th International Symposium on High Performance Computer Architecture (HPCA)*, pages 614–625, Feb 2014.

[36] A. Lashgar and A. Baniasadi. A case against small data types in gpgpus. In *Application-specific Systems, Architectures and Processors (ASAP), 2014 IEEE 25th International Conference on*, pages 108–113, June 2014.

[37] Ahmad Lashgar. IPMACC. *Available: http://github.com/lashgar/ipmacc/ (Accessed: 2017-07-12).*

[38] Ahmad Lashgar. Micro-benchmarking CUDA GPUs. *Available: https://github.com/lashgar/microbenchmark (Accessed: 2017-07-12).*

[39] Ahmad Lashgar and Amirali Baniasadi. Rethinking prefetching in gpgpus: Exploiting unique opportunities. In *To appear in proceedings of 17th IEEE International Conference on High Performance Computing and Communications*, HPCC 2015, New York, NY, USA, 2015.

[40] Ahmad Lashgar and Amirali Baniasadi. Employing software-managed caches in openacc: Opportunities and benefits. *ACM Trans. Model. Perform. Eval. Comput. Syst.*, 1(1):2:1–2:34, February 2016.

[41] Ahmad Lashgar and Amirali Baniasadi. OpenACC cache directive: Opportunities and optimizations. In *Proceedings of the Third International Workshop on Accelerator Programming Using Directives*, WACCPD '16, pages 46–56, Piscataway, NJ, USA, 2016. IEEE Press.

[42] Ahmad Lashgar and Amirali Baniasadi. Efficient implementation of OpenACC cache directive on NVIDIA GPUs. *To appear in the International Journal of High Performance Computing and Networking (IJHPCN), Special Issue on High-level Programming Approaches for Accelerators*, 2017.

[43] Ahmad Lashgar, Ebad Lashgar, and Amirali Baniasadi. Understanding outstanding memory request handling resources in gpgpus. In *To appear in proceedings of The Sixth International Symposium on Highly Efficient Accelerators and Reconfigurable Technologies*, HEART 2015, Boston MA, USA, 2015.

[44] Ahmad Lashgar, Alireza Majidi, and Amirali Baniasadi. IPMACC: Translating OpenACC API to OpenCL. In *In poster session of The 3rd International Workshop on OpenCL (IWOCL)*, IWOCL 2015, Stanford University, California, USA, 2015.

[45] Ahmad Lashgar, Ebad Salehi, and Amirali Baniasadi. A case study in reverse engineering gpgpus: Outstanding memory handling resources. *SIGARCH Comput. Archit. News*, 43(4):15–21, April 2016.

[46] Chang Joo Lee, Onur Mutlu, Veynu Narasiman, and Yale N. Patt. Prefetch-aware dram controllers. In *Proceedings of the 41st Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO 41, pages 200–209, Washington, DC, USA, 2008. IEEE Computer Society.

[47] Jaekyu Lee, Nagesh B. Lakshminarayana, Hyesoon Kim, and Richard Vuduc. Many-thread aware prefetching mechanisms for gpgpu applications. In *Proceedings of the 2010 43rd Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO '43, pages 213–224, Washington, DC, USA, 2010. IEEE Computer Society.

[48] Seyong Lee and Rudolf Eigenmann. OpenMPC: Extended OpenMP programming and tuning for GPUs. In *Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '10, pages 1–11, Washington, DC, USA, 2010. IEEE Computer Society.

[49] Seyong Lee and Jeffrey S. Vetter. OpenARC: Extensible OpenACC Compiler Framework for Directive-based Accelerator Programming Study. In *Proceedings of the First Workshop on Accelerator Programming Using Directives*, WACCPD '14, pages 1–11, Piscataway, NJ, USA, 2014. IEEE Press.

[50] Chao Li, Yi Yang, Hongwen Dai, Shengen Yan, F. Mueller, and Huiyang Zhou. Understanding the tradeoffs between software-managed vs. hardware-managed

caches in gpus. In *Performance Analysis of Systems and Software (ISPASS), 2014 IEEE International Symposium on*, pages 231–242, March 2014.

[51] Chunhua Liao, Oscar Hernandez, Barbara Chapman, Wenguang Chen, and Weimin Zheng. Openuh: An optimizing, portable openmp compiler: Research articles. *Concurr. Comput. : Pract. Exper.*, 19(18):2317–2332, December 2007.

[52] E. Lindholm, J. Nickolls, S. Oberman, and J. Montrym. Nvidia tesla: A unified graphics and computing architecture. *Micro, IEEE*, 28(2):39–55, March 2008.

[53] Fang Liu and Yan Solihin. Studying the impact of hardware prefetching and bandwidth partitioning in chip-multiprocessors. In *Proceedings of the ACM SIGMETRICS Joint International Conference on Measurement and Modeling of Computer Systems*, SIGMETRICS '11, pages 37–48, New York, NY, USA, 2011. ACM.

[54] Yuxi Liu, Zhibin Yu, Lieven Eeckhout, Vijay Janapa Reddi, Yingwei Luo, Xiaolin Wang, Zhenlin Wang, and Chengzhong Xu. Barrier-aware warp scheduling for throughput processors. In *Proceedings of the 2016 International Conference on Supercomputing*, ICS '16, pages 42:1–42:12, New York, NY, USA, 2016. ACM.

[55] Jiwei Lu, Howard Chen, Pen-Chung Yew, and Wei-Chung Hsu. Design and implementation of a lightweight dynamic optimization system. *J. Instruction-Level Parallelism*, 6, 2004.

[56] Chi-Keung Luk and Todd C. Mowry. Compiler-based prefetching for recursive data structures. *SIGOPS Oper. Syst. Rev.*, 30(5):222–233, September 1996.

[57] Stefano Markidis, Jing Gong, Michael Schliephake, Erwin Laure, Alistair Hart, David Henty, Katherine Heisey, and Paul Fischer. Openacc acceleration of the nek5000 spectral element code. *Int. J. High Perform. Comput. Appl.*, 29(3):311–319, August 2015.

[58] S.K. Mathew, M.A. Anders, B. Bloechel, Trang Nguyen, R.K. Krishnamurthy, and S. Borkar. A 4-ghz 300-mw 64-bit integer execution alu with dual supply voltages in 90-nm cmos. *Solid-State Circuits, IEEE Journal of*, 40(1):44–51, Jan 2005.

[59] Jiayuan Meng and Kevin Skadron. Performance modeling and automatic ghost zone optimization for iterative stencil loops on gpus. In *Proceedings of the 23rd International Conference on Supercomputing*, ICS '09, pages 256–265, New York, NY, USA, 2009. ACM.

[60] S. Moy and J. Lindholm. Across-thread out of order instruction dispatch in a multithreaded graphics processor, June 23 2005. US Patent App. 10/742,514.

[61] Hitoshi Murai, Masahiro Nakao, Takenori Shimosaka, Akihiro Tabuchi, Taisuke Bokut, and Mitsuhisa Sato. XcalableACC - a Directive-based Language Extension for Accelerated Parallel Computing. In *Proceedings of the Supercomputing Conference poster*, SC '14 poster session, Piscataway, NJ, USA, 2014.

[62] Naveen Muralimanohar, Rajeev Balasubramonian, and Norm Jouppi. Optimizing nuca organizations and wiring alternatives for large caches with cacti 6.0. In *Proceedings of the 40th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO 40, pages 3–14, Washington, DC, USA, 2007. IEEE Computer Society.

[63] Masahiro Nakao, Hitoshi Murai, Takenori Shimosaka, Akihiro Tabuchi, Toshihiro Hanawa, Yuetsu Kodama, Taisuke Bokut, and Mitsuhisa Sato. XcalableACC: Extension of XcalableMP PGAS Language Using OpenACC for Accelerator Clusters. In *Proceedings of the First Workshop on Accelerator Programming Using Directives*, WACCPD '14, pages 27–36, Piscataway, NJ, USA, 2014. IEEE Press.

[64] John Nickolls, Ian Buck, Michael Garland, and Kevin Skadron. Scalable Parallel Programming with CUDA. *Queue*, 6(2):40–53, March 2008.

[65] Kyle Niemeyer and Chih-Jen Sung. Recent progress and challenges in exploiting graphics processors in computational fluid dynamics. *Journal of Supercomputing*, 67(2):528, February 2014.

[66] Matthew Norman, Jeffrey Larkin, Aaron Vose, and Katherine Evans. A case study of cuda fortran and openacc for an atmospheric climate kernel. *Journal of Computational Science*, 9(1):1–6, Jul 2015.

[67] NVIDIA Corp. CUDA C Programming Guide. *Available: http://docs.nvidia.com/cuda/cuda-c-programming-guide/ (Accessed: 2017-07-12).*

[68] NVIDIA Corp. CUDA Downloads. *Available: https://developer.nvidia.com/cuda-downloads (Accessed: 2017-07-12).*

[69] NVIDIA Corp. GeForce GTX 480. *Available: http://www.geforce.com/hardware/desktop-gpus/geforce-gtx-480/specifications (Accessed: 2017-07-12).*

[70] NVIDIA Corp. NVIDIA CUDA occupancy calculator. *Available: http://developer.download.nvidia.com/compute/cuda/CUDA_Occupancy_calculator.xls (Accessed: 2017-07-12).*

[71] NVIDIA Corp. Nvidia's next generation CUDA compute architecture: Kepler GK110. *Available: http://www.nvidia.ca/content/PDF/kepler/NVIDIA-Kepler-GK110-Architecture-Whitepaper.pdf (Accessed: 2017-07-12).*

[72] NVIDIA Corp. Tesla K20 GPU Accelerator. *Available: http://www.nvidia.com/content/pdf/kepler/tesla-k20-passive-bd-06455-001-v07.pdf (Accessed: 2017-07-12).*

[73] J. P. Lars Nyland and Mark Harris. *Gpu Gems 3: chapter 31.* Addison-Wesley Professional, first edition, 2007.

[74] L. Nyland, J.R. Nickolls, G. Hirota, and T. Mandal. Systems and methods for coalescing memory accesses of parallel threads, March 5 2013. US Patent 8,392,669.

[75] OpenACC. The OpenACC application programming interface version 1.0. *Available: https://www.openacc.org/sites/default/files/inline-files/OpenACC_1_0_specification.pdf (Accessed: 2017-07-12).*

[76] OpenACC. The OpenACC application programming interface version 2.0. *Available: https://www.openacc.org/sites/default/files/inline-files/OpenACC_2_0_specification.pdf (Accessed: 2017-07-12).*

[77] OpenACC. The OpenACC application programming interface version 2.5. *Available: https://www.openacc.org/sites/default/files/inline-files/OpenACC_2pt5.pdf (Accessed: 2017-07-12).*

[78] PathScale. Modified rodinia benchmark suite. *Available: https://github.com/pathscale/rodinia (Accessed: 2017-07-12).*

[79] M. Pharr and W.R. Mark. ispc: A spmd compiler for high-performance cpu programming. In *Innovative Parallel Computing (InPar), 2012*, pages 1–13, May 2012.

[80] P. Reungsang, Sun Kyu Park, Seh-Woong Jeong, Hyung-Lae Roh, and Gyungho Lee. Reducing cache pollution of prefetching in a small data cache. In *Computer Design, 2001. ICCD 2001. Proceedings. 2001 International Conference on*, pages 530–533, 2001.

[81] Ruyman Reyes, Ivan López-Rodríguez, Juan J. Fumero, and Francisco de Sande. accull: An openacc implementation with cuda and opencl support. In *Proceedings of the 18th International Conference on Parallel Processing*, Euro-Par'12, pages 871–882, Berlin, Heidelberg, 2012. Springer-Verlag.

[82] Shane Ryoo, Christopher I. Rodrigues, Sam S. Stone, Sara S. Baghsorkhi, Sain-Zee Ueng, John A. Stratton, and Wen-mei W. Hwu. Program optimization space pruning for a multithreaded gpu. In *Proceedings of the 6th Annual IEEE/ACM International Symposium on Code Generation and Optimization*, CGO '08, pages 195–204, New York, NY, USA, 2008. ACM.

[83] Vivek Seshadri, Samihan Yedkar, Hongyi Xin, Onur Mutlu, Phillip B. Gibbons, Michael A. Kozuch, and Todd C. Mowry. Mitigating prefetcher-caused pollution using informed caching policies for prefetched blocks. *ACM Trans. Archit. Code Optim.*, 11(4):51:1–51:22, January 2015.

[84] Ankit Sethia, Ganesh Dasika, Mehrzad Samadi, and Scott Mahlke. Apogee: Adaptive prefetching on gpus for energy efficiency. In *Proceedings of the 22Nd International Conference on Parallel Architectures and Compilation Techniques*, PACT '13, pages 73–82, Piscataway, NJ, USA, 2013. IEEE Press.

[85] John Shalf, Sudip Dosanjh, and John Morrison. Exascale computing technology challenges. In *Proceedings of the 9th International Conference on High Perfor-*

mance Computing for Computational Science, VECPAR'10, pages 1–25, Berlin, Heidelberg, 2011. Springer-Verlag.

[86] A. J. Smith. Sequential program prefetching in memory hierarchies. *Computer*, 11(12):7–21, Dec 1978.

[87] Alan Jay Smith. Cache memories. *ACM Comput. Surv.*, 14(3):473–530, September 1982.

[88] S. Srinath, O. Mutlu, H. Kim, and Y. N. Patt. Feedback directed prefetching: Improving the performance and bandwidth-efficiency of hardware prefetchers. In *2007 IEEE 13th International Symposium on High Performance Computer Architecture*, pages 63–74, Feb 2007.

[89] John A. Stratton, Christopher Rodrigues, I-Jui Sung, Nady Obeid, Li-Wen Chang, Nasser Anssari, Geng Daniel Liu, and Wen-Mei W. Hwu. Parboil: A revised benchmark suite for scientific and commercial throughput computing. *Center for Reliable and High-Performance Computing (2012).*

[90] Akihiro Tabuchi, Masahiro Nakao, and Mitsuhisa Sato. A source-to-source openacc compiler for cuda. In *HeteroPar'2013*, 2013.

[91] X. Tian, R. Xu, Y. Yan, Z. Yun, S. Chandrasekaran, and B. Chapman. Compiling a high-level directive-based programming model for gpgpus. In *Proceedings of the 26th International Workshop on Languages and Compilers for High Performance Computing*, LCPC 2013, 2013.

[92] Top500.org. Green500 list for november 2016. *Available: https://www.top500.org/green500/lists/2016/11/ (Accessed: 2017-07-12).*

[93] Top500.org. TOP500 list for november 2016. *Available: https://www.top500.org/lists/2016/11/ (Accessed: 2017-07-12).*

[94] Y. Torres, A Gonzalez-Escribano, and D.R. Llanos. Understanding the impact of cuda tuning techniques for fermi. In *High Performance Computing and Simulation (HPCS), 2011 International Conference on*, pages 631–639, July 2011.

[95] V. Volkov and J.W. Demmel. Benchmarking gpus to tune dense linear algebra. In *High Performance Computing, Networking, Storage and Analysis, 2008. SC 2008. International Conference for*, pages 1–11, Nov 2008.

[96] Vasily Volkov. Better performance at lower occupancy. *GPU Technology Conference 2010 (GTC 2010). Available: http://www.nvidia.com/content/GTC-2010/pdfs/2238_GTC2010.pdf (Accessed: 2017-07-12).*

[97] Sandra Wienke, Paul Springer, Christian Terboven, and Dieter an Mey. Openacc: First experiences with real-world applications. In *Proceedings of the 18th International Conference on Parallel Processing*, Euro-Par'12, pages 859–870, Berlin, Heidelberg, 2012. Springer-Verlag.

[98] David Williams, Valeriu Codreanu, Po Yang, Baoquan Liu, Feng Dong, Burhan Yasar, Babak Mahdian, Alessandro Chiarini, Xia Zhao, and Jos Roerdink. Evaluation of autoparallelization toolkits for commodity gpus. In *Parallel Processing and Applied Mathematics*, volume 8384, pages 447–457. Springer, 2014.

[99] C.M. Wittenbrink, E. Kilgariff, and A Prabhu. Fermi GF100 GPU architecture. *Micro, IEEE*, 31(2):50–59, March 2011.

[100] H. Wong, M.-M. Papadopoulou, M. Sadooghi-Alvandi, and A. Moshovos. Demystifying gpu microarchitecture through microbenchmarking. In *Performance Analysis of Systems Software (ISPASS), 2010 IEEE International Symposium on*, pages 235–246, March 2010.

[101] Rengan Xu, Maxime Hugues, Henri Calandra, Sunita Chandrasekaran, and Barbara Chapman. Accelerating kirchhoff migration on gpu using directives. In *Proceedings of the First Workshop on Accelerator Programming Using Directives*, WACCPD '14, pages 37–46, Piscataway, NJ, USA, 2014. IEEE Press.

[102] George L. Yuan, Ali Bakhoda, and Tor M. Aamodt. Complexity effective memory access scheduling for many-core accelerator architectures. In *Proceedings of the 42Nd Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO 42, pages 34–44, New York, NY, USA, 2009. ACM.

[103] Weifeng Zhang, Brad Calder, and Dean M. Tullsen. A self-repairing prefetcher in an event-driven dynamic optimization framework. In *Proceedings of the International Symposium on Code Generation and Optimization*, CGO '06, pages 50–64, Washington, DC, USA, 2006. IEEE Computer Society.

[104] Ying Zhang, Yue Hu, Bin Li, and Lu Peng. Performance and power analysis of ati gpu: A statistical approach. In *Networking, Architecture and Storage (NAS), 2011 6th IEEE International Conference on*, pages 149–158, July 2011.

[105] X. Zhuang and H. h. S. Lee. Reducing cache pollution via dynamic data prefetch filtering. *IEEE Transactions on Computers*, 56(1):18–31, Jan 2007.