ABSTRACT

Title of Dissertation: ON EFFICIENT GPGPU COMPUTING FOR INTEGRATED HETEROGENEOUS CPU-GPU MICROPROCESSORS

Daniel Gerzhoy
Doctor of Philosophy, 2021

Dissertation Directed by: Professor Donald Yeung
Department of Electrical
and Computer Engineering

Heterogeneous microprocessors which integrate a CPU and GPU on a single chip provide low-overhead CPU-GPU communication and permit sharing of on-chip resources that a traditional discrete GPU would not have direct access to. These features allow for the optimization of codes that heretofore would be suitable only for multi-core CPUs or discrete GPUs to be run on a heterogeneous CPU-GPU microprocessor efficiently and in some cases- with increased performance.

This thesis discusses previously published work on exploiting nested MIMD-SIMD Parallelization for Heterogeneous microprocessors. We examined loop structures in which one or more regular data parallel loops are nested within a parallel outer loop that can contain irregular code (e.g., with control divergence). By scheduling outer loops on the multicore CPU part of the microprocessor, each thread launches dynamic, independent instances of the inner loop onto the GPU, boosting GPU utilization while simultaneously parallelizing the outer loop.

The second portion of the thesis proposal explores heterogeneous producer-consumer data-sharing between the CPU and GPU on the microprocessor. One advantage of tight integration – the sharing of the on-chip cache system – could improve the impact that memory accesses have on performance and power. Producer-consumer data sharing commonly occurs between the CPU and GPU portions of programs, but large kernel sizes whose data footprint far exceeds that of a typical CPU cache, cause shared data to be evicted before it is reused.

We propose Pipelined CPU-GPU Scheduling for Caches, a locality transformation for producer-consumer relationships between CPUs and GPUs. By intelligently scheduling the execution of the producer and consumer in a software pipeline, evictions can be avoided, saving DRAM accesses, power, and performance. To keep the cached data on chip, we allow the producer to run ahead of the consumer by a certain amount of loop iterations or threads. Choosing this "run-ahead distance" becomes the main constraint in the scheduling of work in this software pipeline, and we provide a method of statically predicting it.

We assert that with intelligent scheduling and the hardware and software mechanisms to support it, more workloads can be gainfully executed on integrated heterogeneous CPU-GPU microprocessors than previously assumed.

ON EFFICIENT GPGPU COMPUTING FOR INTEGRATED HETEROGENEOUS CPU-GPU MICROPROCESSORS

by

Daniel Gerzhoy

Dissertation submitted to the Faculty of the Graduate School of the
University of Maryland, College Park in partial fulfillment
of the requirements for the degree of
Doctor of Philosophy
2021

Advisory Committee:
Professor Donald Yeung, Chair/Advisor
Professor Bruce Jacob
Professor Ankur Srivastava
Professor Manoj Franklin
Professor Alan Sussman

Acknowledgments

Thank you to all the people who made this possible. First, thank you to my advisor Donald Yeung for guiding my research. I am a better engineer having worked under you. Thanks to Ankur Srivastava for being a great professor to TA and grade for, and a good mentor and advisor on top of that. Thanks to my entire panel for passing me! And thanks to the entire staff and faculty of the ECE department at Maryland, and all other staff across the university as well.

Thanks to my peers who worked with my during my time in grad school. Thanks to Mike Zuzak for helping me in my first year get started on research, and providing the seed for the beginning of my work. Thanks to Xiaowu Sun for working with me on the first half of my work, we made a great team. To Devesh Singh and Candace Walden, thanks for helping me broaden my horizons with our reading group, and generally just cooperating with lab management with me. It's sad my final year with you guys was so remote, but I wish you all the best of luck.

Thank you to my friends and family for keeping me sane and happy during a challenging half-decade. To my parents Olga and Val for loving me endlessly, and providing me with an excellent education prior to grad school. To my brother Gene for being my role-model and pushing me to get my PhD when I was doubting.

Table of Contents

Chapter 1: Introduction

1.1 CPU-GPU Integration Trends and Consequences

When Graphics Processing Units (GPUs) were first created, they were highly specialized fixed-function accelerators used to perform real-time rendering and other display-related tasks, typically located on a daughter-card or separate module (but never on the same die as the CPU). The GPU was a separate accelerator or sub-computing system used by the the CPU to perform graphics related tasks. As such, the ecosystem of drivers, operating-system controls, programming techniques, and supporting software evolved assuming the GPU was a discrete, and not co-equal system.

As these *discrete GPU*s advanced, they began to replace some fixed-function hardware with programmable *shader-cores* that could be used flexibly for tasks in the graphics rendering pipeline. It then became clear to the general computing community that the GPU and its programmable shader-cores could be used for more than just graphics, and thus *General Purpose GPU* (or *GPGPU*) computing was born [1]. GPGPU computing maps tasks with extremely parallel structures to execute as "kernels" on the GPU and "launches" them to the discrete card using programming interfaces like Nvidia's *CUDA*. On this CPU-GPGPU machine, a *Het-*

erogeneous System, highly parallel code could under the right circumstances achieve much more performance than that same code executing on a CPU alone. However, though the systems were used together, the GPU had to be managed as a separate entity.

Treating the GPU as a separate entity from a software perspective was necessary until recently because these physically discrete GPUs did not provide important ease-of-use features that CPU hardware had included for many years: cache coherence and a unified virtual memory. The GPU had its own "main" memory (modern discrete GPUs still do for performance reasons) which was physically separate from the CPU main memory. One could not access these addresses from CPU user-code, and thus data had to be transferred over via explicit copy commands. Coherent sharing of data between the CPU and GPU was enforced by the same mechanism. NVidia introduced "Unified Memory" in the 6th version of CUDA. This made it possible for the GPU to integrate into the CPU's virtual memory system, automatically copying data from one physical memory to the other [2]. The integration of the two systems with this single unified virtual address space remains the norm for discrete GPUs to this day. It enabled more complex codes to run on the GPU, and made easier the writing of software for it. However, the physical separation between the two chips still remained. Though explicit copies could be eliminated, they would implicitly be executed by the runtime system. The chips still had separate main memories, and coherence still needed to be managed for tasks requiring complex data-sharing.

These virtually integrated heterogeneous systems broaden the scope of codes

that can be launched to the GPU as kernels, but lacking physical integration they still face some limitations. To launch a kernel to the GPU, data must be transferred from the CPU's physical memory over an external bus (like PCIe) to the GPU's physical memory. As the term "launch" implies, there is a significant latency cost associated with such an action, and therefore, discrete GPUs require massive or "coarse-grained" parallelism to amortize their startup latencies and improve on the performance of the same code executing on the CPU [3].

Since 2011, the trend for processor manufacturers has been to produce *heterogeneous microprocessors* in which a CPU and a GPU are integrated on the same die. For example, Intel [4] and AMD [5] have done so for x86 processors, and Apple [6] has done so for ARM-based SoCs. This physical integration provides the CPU and GPU the ability to share physical resources like main memory and on-chip coherent caches. Utilizing shared resources closely located on chip, the cores are able to compute on data "in place" either through shared main memory or shared caches – thus increasing the speed of communication between the cores and eliminating unnecessary off-chip traffic.

Because integrated GPUs enjoy high-speed communications with the CPU, the latency overhead of launching kernels to the GPU is decreased. Thus they are capable of exploiting a finer granularity of parallelism compared to discrete GPUs. This allows integrated GPUs to accelerate a wider range of loops. Furthermore, a shared cache between CPU and GPU allows the cores to access the same data blocks without extra accesses to main memory. Reduced accesses to DRAM have the potential to decrease the energy usage of workloads running on these types of

chips.

The gradual integration of the GPU with the computing system that CPUs were the center of made using the GPU simpler, less costly in terms of performance, and gave the GPU access to shared resources.

1.2 Challenges in Optimizing for Integrated GPUs

Together, flexibility to off-load finer-grained loops coupled with enhanced data-sharing will enable programmers to map and optimize more complex programs onto integrated CPU-GPU heterogeneous microprocessors. As researchers try to accelerate more complex programs, a major challenge will be effectively parallelizing and optimizing these irregular codes for heterogeneous microprocessors. GPUs are typically used to accelerate large data parallel loops with regular parallelism. The conventional approach is to parallelize such loops and schedule them as kernels onto the GPU one at a time while a single CPU core responsible for launching the kernels idles while the GPU completes the work. Any pre or post-processing done by the CPU executes entirely before or after the GPU kernel runs. For programs that contain large regular loops that dominate execution time and workloads meant for discrete GPUs, this approach works well.

But for more complex programs executing on heterogeneous microprocessors, running large regular loops one at a time on the integrated GPU can lead to poor performance, wasted memory accesses, and/or leave potential performance gains on the table.

One problem is that while complex programs do exhibit GPU-friendly loops, the amount of parallelism can vary significantly. In many cases, individual loops may contain only modest levels of parallelism. Such smaller loops can still be profitably off-loaded onto integrated GPUs, especially given the low communication overheads discussed earlier. But, they may achieve lower speedups, and if executed one at a time, each loop cannot fully utilize the GPU's cores [7]. Another problem is that complex programs tend to contain code with control divergence and irregular memory access patterns that can perform poorly on GPUs. These irregular code regions can account for significant portions of execution time. If they are run serially on a single CPU core, not only are the CPU cores underutilized but also Amdahl's law will limit the performance gains that are possible.

A second problem is wasted memory accesses caused by capacity evictions in a shared cache. When loops are large enough, they will evict data from the cache that they themselves accessed before they finish executing in full. If this loop produces data that a subsequent loop consumes, this means that shared data will need to be fetched into cache multiple times. This wastes energy on DRAM accesses, and potentially decreases performance. These loops can be any combination or ordering of dependent CPU loops or GPU kernels.

1.3 Contributions

This thesis makes several contributions to optimizing GPGPU computations on integrated heterogeneous CPU-GPU microprocessors as part of two overarch-

ing techniques. For complex programs containing SIMD loops of smaller sizes we increase utilization of the cores of heterogeneous microprocessors with a novel parallelization technnique, Nested MIMD-SIMD Parallelization. And, for programs with larger loops that share data between CPU-GPU producers and consumers, Pipeline Scheduling for Caches takes advantage of the coherent cache system using a cache-aware software-hardware scheduling technique to keep shared data on-chip.

Nested MIMD-SIMD Parallelization

To solve the first problem and achieve higher performance for complex programs, it is necessary to expose greater amounts of parallelism such that the GPU and CPU cores available in a heterogeneous microprocessor are both more fully utilized. On the GPU side, given the problem of smaller regular loops, one way to boost parallelism is to off-load multiple loops onto the GPU simultaneously when possible. Recently, researchers have investigated concurrent kernel launch [8, 9] which makes exposing such multi-kernel GPU parallelism possible. On the CPU side, given the problem of serial irregular code, parallelizing the irregular code regions so they can run on multiple CPU cores is needed to address Amdahl's law. In other words, *heterogeneous parallelism*—- or, parallelization of CPU code regions with GPU code regions—is necessary to achieve high performance.

Chapter 3 will present a new parallelization technique which exploits nested loop structures in which an inner regularly parallel loop is nested inside an outer parallel region[10]. The inner loops can be gainfully accelerated and executed on

"Single-Instruction, Multiple Data" (SIMD) GPU cores. The outer regions, which can either be parallel loops or exhibit task-parallelism, could contain irregular code that performs poorly on GPUs. Thus this outer region is scheduled onto the "Multiple Instruction, Multiple Data" (MIMD) CPU cores. The technique – called nested MIMD-SIMD parallelization – achieves the goal of providing both multi-kernel GPU parallelism and CPU parallelism at the same time, increasing utilization of the chip.

We show this technique commonly increases performance in practice. The nested loop structure occurs frequently in programs parallelized by OpenMP, where complex coarse-grained parallel regions often contain smaller fine-grained parallel loops nested within them that can be executed gainfully on the GPU [11]. And in some cases where complex parallel code regions contain fine-grained loops with both regular parallel and serial non-parallel portions, using "loop fission" we can extract these serial portions, creating regular parallel loops suitable for acceleration on the GPU.

Because some of the SIMD regions within the outer MIMD loops are small, they require low-latency kernel launch. While this is trivial to achieve with a simulator, real hardware lacks the capability to launch kernels without significant launch latency. For some of the smaller SIMD regions in the programs we evaluared, this was detrimental to performance. Thus, in order to evaluate our technique on real hardware we also implemented a low-latency launch system to bypass the unecessary slowdowns imparted on kernel-launch by a GPGPU runtime system that cares little for launch latencies.

We evaluate seven OpenMP programs on both an Intel Core i7-6700, and on a

cycle-accurate simulator, gem5-gpu. The technique was able to speedup performance of the seven OpenMP programs providing a 16.1x and 8.67x speedup over sequential computing on a simulator and a physical machine, respectively. Our technique beats CPU-only parallelization by 4.13x and 2.40x, respectively, and GPU-only parallelization by 2.74x and 2.26x, respectively. Compared to the next-best scheme (either CPU- or GPU-only parallelization) per benchmark, our approach provides a 1.46x and 1.23x speedup for the simulator and physical machine, respectively.

Pipelined CPU-GPU Scheduling for Caches

To solve the problem of wasted memory accesses, Chapter 4 proposes Pipelined CPU-GPU Scheduling for Caches. By scheduling producers and consumers together in a pipelined fashion, whether they be CPU loops or GPU kernels, a reduction in superfluous DRAM accesses can be achieved. The pipeline consists of a chain of two or more producers/consumers which operate in lock-step. The amount of computation done by each stage per "epoch" is defined by how far ahead a producer is allowed to execute ahead of a consumer. This "run-ahead distance" (RAD), determines the total cache-footprint of the epoch. If that footprint is larger than the cache system can sustain, data will be evicted to main memory before it can be used. When the consumer stage gets to that data, superfluous memory accesses will occur. Pipelined Scheduling aims to keep the RAD small enough such that this overflow does not occur, and data stays in the cache system - "in-place" - until it is reused.

The lock-step operation of the pipeline requires fine-grain synchronization, which the heterogeneous microprocessor provides. Using the GPU model created by AMD in the new gem5 simulator, including an emulated driver upon which the real AMD GPGPU driver-stack ROCM relies, we have created a system which implements this lock-step pipeline in a practical and realistic way. Furthermore, we have found and adapted seven benchmarks that exhibit this producer/consumer data relationship using this system.

By varying the RAD, we vary the cache footprint of each producer/consumer relationship, and can observe the benefits of executing these benchmarks in a pipelined fashion. We show that decreasing the RAD below a certain threshold drastically decreases the number of DRAM accesses the producer-consumer relationship incurs. This reduces energy usage, and for consumers that are latency sensitive (CPU consumers) this translates into a performance benefit, and consequently, further energy usage reduction.

We evaluated seven GPGPU programs using the new gem5 cycle-accurate APU simulator. These workloads came both from benchmark suites like Rodinia aimed at evaluation of traditional GPGPU platforms [12], and those aiming at evaluating modern collaborative computing models between the CPU and GPU like HeteroMark [13]. We also evaluated one OpenMP workload evaluated in the first module. The loop fission transformation we use to extract more parallelism creates a producer-consumer sharing pattern that we were able to optimize using our pipelined scheduling technique [10].

Using our technique we were achieved a maximum 27.4% reduction in total

DRAM energy averaged across the seven workloads we evaluated. This DRAM energy savings was a consequence of both a 30.4% reduction in superfluous DRAM accesses, and a 26.8% reduction in execution time achieved by our techniques intelligent scheduling.

The rest of this thesis is organized as follows. Chapter 2 provides background on GPGPU architecture and programming models, useful throughout this work. Chapter 3 presents the contributions related to the first technique we developed, Nested MIMD-SIMD Parallelization. Chapter 4 proposes the second optimization technique, Pipeline Scheduling for Caches. Chapter 5 concludes the thesis and discusses future work.

Chapter 2: GPGPU Background

This thesis uses both software and hardware techniques to improve performance for Heterogeneous integrated GPU microprocessors. This section will serve to give a background on General-Purpose GPU (GPGPU) programming models, programming interfaces, and micro-architecture.

2.1 Vendors and Nomenclature

The GPGPU ecosystem consists of a variety of vendors releasing their own architectures, application programming interfaces (APIs), and drivers - with their own nomenclature for each. In this thesis we utilize several platforms for qualitative evaluations, and as such often encountered differing names for identical or similar constructs. Table 2.1 shows the vendors, architectures and associated APIs used. First, Nvidia's *Fermi* architecture is associated with their *Compute Unified Device*

Vendor	Architecture	API	SIMT Core	ThreadGroup
Nvidia	Fermi	CUDA	Streaming Multiprocessor (SM)	Warp
Intel	GEN9	OpenCL	Execution Unit (EU)	SIMD-Width
AMD	GCN3	HIP	Compute Unit (CU)	Wavefront

Table 2.1: The GPGPU vendors, architectures, APIs, and hardware nomenclatures used in this thesis. Note that architectures and names change with generation; these are just examples, though they are the architectures we use in this thesis.

Architecture (CUDA) API. Next, Intel's GEN GPU architecture interfaces with an Intel-implemented version of the open source *OpenCL* specification by the Khronos Group. Finally, AMD's GCN3 architecture uses *Heterogeneous-compute Interface for Portability* (HIP). Though each architecture and API is unique, they are fundamentally similar. This chapter will generalize the discussion of micro-architecture and software interface, referencing specific names when it is relevant to the discussion.

2.2 Programming Model

Since GPUs have unique architectures with distinct instruction set architectures (ISAs) and traditionally GPUs have been located on daughter cards, GPGPU code must be managed and executed by specialized APIs which interface with the operating system and drivers. GPGPU software can be understood through three fundamental constructs: GPU Kernels, Thread-blocks, and Grids.

GPU Kernels. Simply put, the primary use-case of a GPU for general-purpose compute tasks is to parallelize a loop. Figure 2.1a shows a simple affine loop adding two arrays together, a perfect candidate for GPU acceleration. To execute on the GPU the loop is transformed into what is called a "kernel," shown in figure 2.1b, which is akin to function in normal CPU code. However, unlike a function in CPU code, which is as easy to execute as a branch in the CPU, a GPU kernel must be "launched" to the GPU via the API and system calls. Figure 2.1c shows this launch call. Depending on the implementation the launch can pass

```
for( int i = 0; i < iterations; i++ )
    C[i] = A[i] + B[i];
```

(a) Affine Loop

```
__global__ void vector_add(int *A, int *B, int *C) {
    int i = blockIdx.x * blockDim.x + threadIdx.x;
    C[i] = A[i] + B[i];
}
```

(b) Kernel Code

```
vector_add<<<nBlocks, TPB>>>(A, B, C);
```

(c) Kernel Launch

Figure 2.1: GPGPU Code Example

through software queues in both user-space and kernel-space, as well as hardware queues on the GPU before it is executed. All of this amounts to a certain amount of latency overhead associated with a GPU call. In order to make transforming a loop worthwhile, the speedup gained by executing it on the GPU as a kernel, must be greater than the latency overhead of a launch.

Thread-Blocks and Grids The iterations in the original loop are divided into a *Grid* of *Thread-Blocks* that make up the kernel. Each individual loop iteration is a thread. Thread-Blocks contain a constant number of threads during the execution of a kernel, and the thread and block id's that determine array indexing are accessed through special API calls. Figure 2.2 shows this hierarchy.

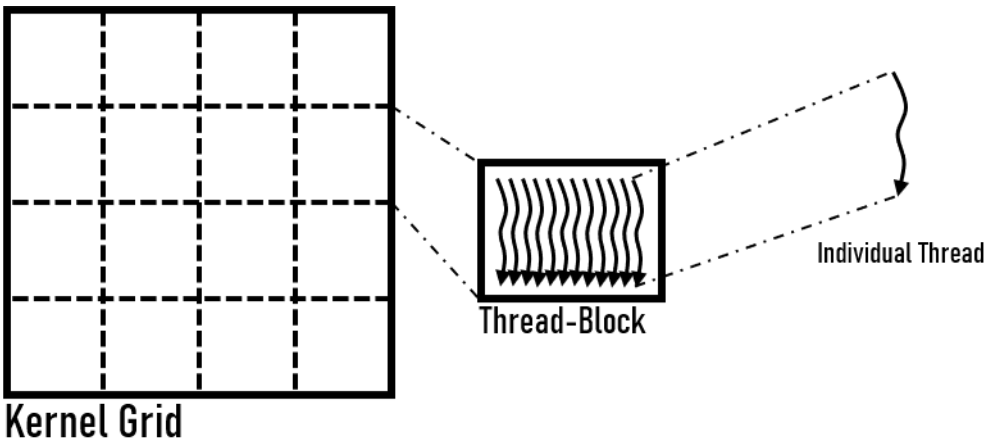


Figure 2.2: Hierarchy of a kernel: A kernel is comprised of a grid which contains thread-blocks, which are groups of individual threads.

2.3 GPU Architecture

Whereas a CPU is capable of executing a single complex branch-divergent instruction stream efficiently, GPGPUs are throughput-oriented processors capable of massive parallelism for simpler less branch-divergent codes. They exhibit high instruction-throughput, much more memory-level parallelism, and have higher memory bandwidth requirements than CPUs. GPU kernels typically have limited temporal locality which encourages GPU caches to be small relative to the total number of threads GPUs are capable of executing simultaneously [14].

Compute Units. Figure 2.3 shows the structure of a GPU. GPUs are organized into groups of Compute Units (CUs). Each compute unit contains a number of Single Instruction, Multiple Data (SIMD) functional units that are the source of the GPUs ability to execute massive amounts of compute simultaneously. However, GPUs are not just a SIMD architecture. Instead CUs employ a Single Instruction, Multiple Threads (SIMT) paradigm, where individual threads execute in lock-step.

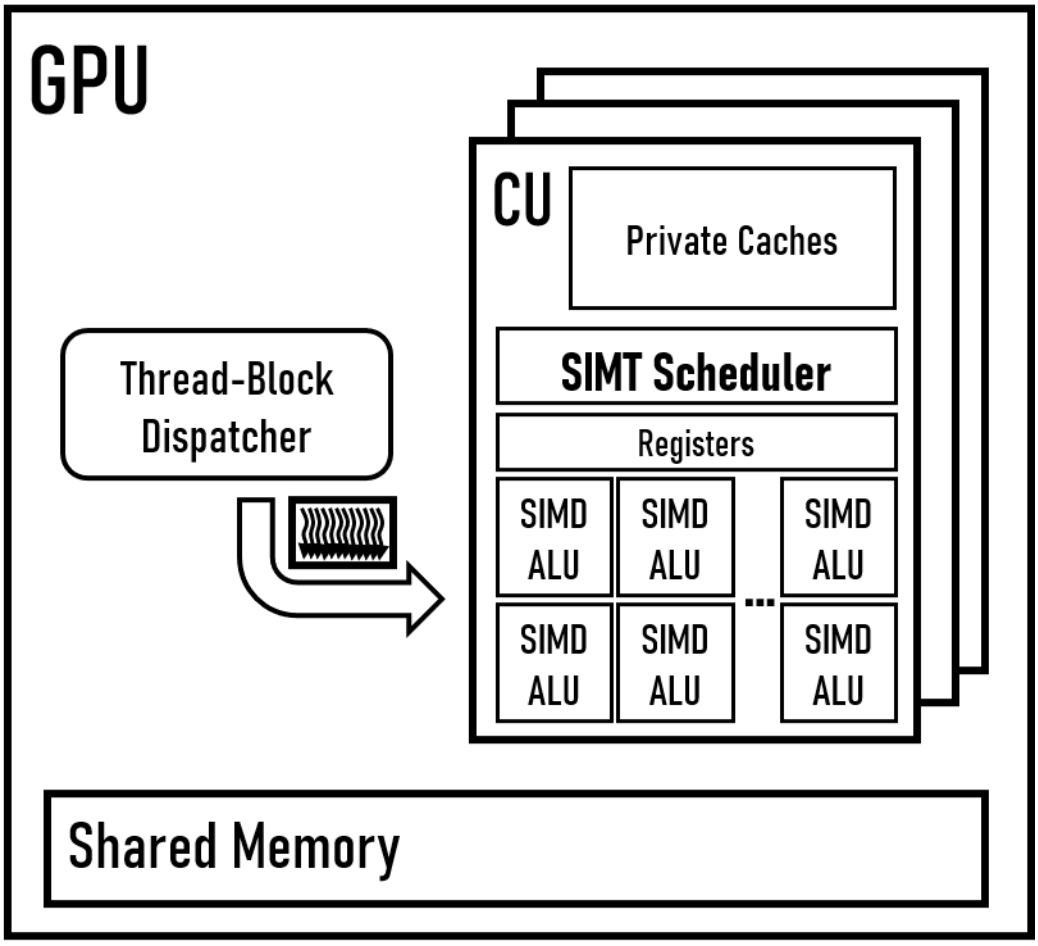


Figure 2.3: GPU containing Thread-block dispatcher and Compute Units (AMD Nomenclature) . Nvidia and Intel equivalents would be "Streaming Multiprocessor"(SM) and "Execution Unit" (EU).

When one of these groups encounters a stall due to memory access, the CU's SIMT scheduler simply schedules a new group to execute while the first group waits for the memory system to return the required data, akin to a context switch. A CU can thus hide memory latency with extreme effectiveness, making GPGPUs very tolerant to memory system latency. Table 2.1 presents the different names for these SIMT cores for different vendors. Compute Units are an AMD term, Execution Units for Intel, and Streaming Multiprocessor for Nvidia.

There are several names for the groups of threads executing in lock-step as well. In Table 2.1 these equivalent terms are AMD wavefronts, Nvidia warps, and Intel SIMD-widths, and they vary in size from architecture to architecture. For simplicity we will refer to them as "warps" in this chapter and the scheduler the schedules them the "SIMT Scheduler."

Not all codes executed on the GPU are as affine and simplistic as the example in figure 2.1. When intra-warp branch-divergence occurs, *i.e.* an If-Else statement where individual threads within a warp take different paths, the warp must serialize execution. The scheduler masks off the diverging threads, only executing the threads that branch together simultaneously. This serialization is a serious blow to the throughput of the GPU, and it is advisable to avoid as much as possible. However, it is important to note that while it is a performance hit *a GPU is capable of executing such code.* This is one of the things that distinguishes the GPU's SIMT architecture from SIMD architectures.

Each thread-block that makes up a kernel's grid is likewise comprised of a number of these warps, and thus a thread-block serves as a pool of warps from

which the CU's SIMT scheduler may schedule work to the functional units.

The GPU also performs best when memory accesses from a single warp access contiguous memory addresses, that way accesses to the same cache-line can be "coalesced," reducing pressure on the cache system. When warps within a thread access different cache lines, this is called "memory divergence" and can impact performance negativily, just as intra-warp control flow divergence does.

Thread-block Dispatcher. Each thread-block is scheduled to an individual compute unit by a dispatcher unit whose job it is to decide where each block goes and how many get to execute per CU. A block may only be dispatched to a CU if there is room for it. The SIMT scheduler has a limited number of slots, *i.e.* the number of warps it can context switch between. Additionally, the CU has a limited number of shared physical resources like registers and shared memory. Each kernel has unique usage characteristics determined by the code and compiler. The dispatched takes into account these differences to schedule the blocks.

Memory System. A GPU's memory system must be able to keep up with the extreme memory-level parallelism of the CUs. Discrete GPUs have specially designed high-bandwidth DRAM [15], and high-bandwidth low-capacity caches. Each compute unit can have several different types of caches. Instruction caches are often shared by multiple CUs owing to the limited number of instructions needed across the GPU.

Early GPGPU had only explicitly managed scratch-pad memories to act as caches with coherence likewise manually managed by the programmer. Scratch-pad memories live on as explicitly managed "Local" memory (Local in the context of

a single thread-block). However, modern GPUs access "global memory" through a coherent cache hierarchy shown in the figure. We go into detail about GPU coherence as it relates to our work in chapter 4.

Chapter 3: Nested MIMD-SIMD Parallelization

This chapter presents our parallelization scheme for heterogeneous microprocessors based on *nested parallelism*: *Nested MIMD-SIMD Parallelization*. We exploit the features of modern heterogeneous microprocessors that integrate a GPU with a CPU, to map more complex codes compared to the traditionally massively parallel codes that have run on GPUs in the past. By utilizing low-cost communication provided by the GPU's on-chip proximity to the CPU, we enable smaller SIMD loops to be gainfully executed on the GPU as kernels. Concurrent kernel launch feeds multiple the GPU work from multiple parallel CPU threads, increasing the *spatial* utilization of the GPU for SIMD loops too small to fill the GPU on their own. Our technique benefits codes with larger SIMD loops and substantial CPU work by allowing the parallel CPU threads to take turns running kernels on the GPU, while they themselves execute their code, increasing the *temporal* utilization of the GPU.

The rest of this Chapter follows as so: Section 3.1 introduces our nested MIMD-SIMD parallelization technique. Then, Section 3.2 presents a loop fission transformation that exposes more opportunities to offload kernels onto the GPU. Section 3.3 discusses the experimental methodology of our simulation study, with Section 3.4 presenting the simulation results. Section 3.5 presents a low-latency

launch system we developed in order to realize our technique on real hardware. Next, Sections 3.6 and 3.7 provide the methodology and results, respectively, of hardware study. Finally, Section 3.8 discusses related work, and Section 3.9 concludes the chapter.

3.1 Nested MIMD-SIMD Parallelization

The kinds of workloads traditionally used to evaluate GPU performance spend the majority of their time on massive data parallel loops [12, 16], with comparatively little code outside of these data parallel regions that make up a significant portion of execution time. These loops are off-loaded or "launched" to the GPU as GPU "kernels", incurring a latency overhead. On discrete GPU platforms, this launch latency overhead is high and thus the kernels must have massive parallelism in order for the reduction in execution time achieved by running on the GPU to be larger than the overhead. These massively parallel GPU kernels fully utilize all of the thread-contexts of the GPU, and thus can be launched sequentially, *i.e.*one at a time, onto the GPU.

Figure 3.1a illustrates this *single-loop SIMD parallelization* scheme wherein the program starts running code serially in a single CPU thread, launches a kernel to the GPU, and stalls while that kernel executes. In traditional workloads, the amount of time spent on the off-loaded loop, W_{SIMD} is much larger (*i.e. nthreads* in Figure 3.1a is large) than the time spent on the other code regions, W_{MIMD}. The large *nthreads* contributes to large speedups attained on the (discrete) GPU,

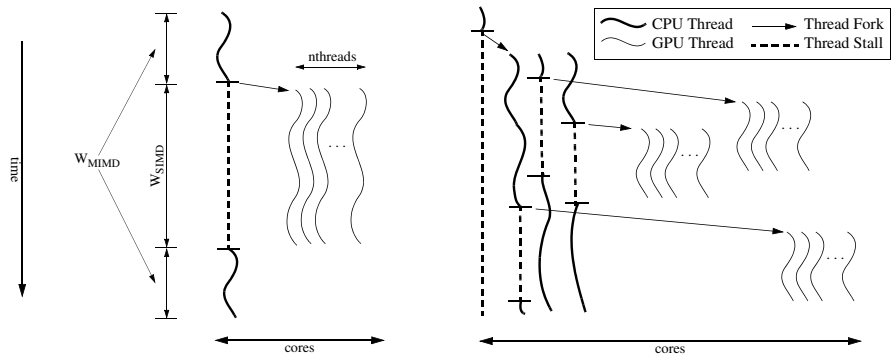


Figure 3.1: CPU and GPU concurrency from a). single-loop SIMD parallelization and b). nested MIMD-SIMD parallelization. W_{SIMD} is the amount of work in each loop parallelized for the GPU, W_{MIMD} is the amount of work outside of each GPU loop, and *nthreads* is the number of GPU threads from each GPU loop.

as the kernel can effectively utilize the GPU's parallel hardware. This relative size difference between W_{SIMD} and W_{MIMD} means that serially executing the code outside of the GPU loops, *i.e.*the CPU completes its work after the GPU finishes, does not degrade performance much.

In this work, we make an attempt to move away from traditional GPU workloads with their relative simplicity and raw parallelism, to map more complex and irregular programs. For one, the programs we investigate can contain SIMD loops with smaller amounts of work, or small W_{SIMD}. Normally, transforming such loops into kernels and executing them on discrete GPUs would result in performance degradation due to the launch overhead. On integrated GPUs that provide low-cost communication, gainfully off-loading these finer-grained loops becomes possible.

In addition to having smaller W_{SIMD}, these non-traditional workloads that we target may have small-sized regular loops that when transformed into GPU kernels will have small *nthreads*. Thus, individual instances of these small loops may not

exhibit sufficient parallelism to fully utilize the GPU. As we discuss in Chapter 2, GPUs consist of *streaming multiprocessors* or *SMs*[1] that are SIMT cores that employ hardware multithreading to schedule many threads to their many hardware thread-contexts. If the SM is not provided enough threads by the kernel to fully occupy all of its hardware contexts, then those contexts are left idle; in other words, *spatial underutilization* occurs. Thus, under single-loop parallelization, one small GPU kernel does not fully exploit all of the compute capability that the GPU has available, and may leave potential gains on the table.

Another issue with traditional single-loop SIMD parallelization of the more complex programs we are interested in is that they can spend more of their execution time on the non-GPU portions of the code, *i.e.*$W_{MIMD} \approx W_{SIMD}$. Much of code in these more complex regions can cause intra-warp thread divergence making them ill suited for execution on the GPU, as this will serialize the executing warps. Under single-loop parallelization, these complex regions of code are limited by Amdahl's Law and gain no benefit from executing on a multi-core chip like the one we are interested in. This leaves the CPU hardware underutilized and performance gains from CPU multithreading unclaimed. Further, when these non-SIMD regions are executing, the GPU is left idle until the end of the W_{MIMD} period - in other words, *temporal underutilization* occurs.

To enable gainful utilization on GPUs for more complex and irregular programs, our work explores the exploitation of different types of parallelism from multiple code structures simultaneously– in other words *heterogeneous parallelism.*

[1] Also known as Compute Units and Execution units. See Table 2.1.

Our approach executes SIMD code appropriate for SIMD hardware on the GPU, in concert with executing non-SIMD parallel code (that is, irregular, MIMD code) on the MIMD CPU cores. One code construct that can provide such heterogeneous parallelism is *nested parallelism*. Within the more complex programs we are interested in, we look for regular parallel loops nested within an irregular outer loop,or other parallel structure such as a task-parallel work queue. We call our new parallelization technique *nested MIMD-SIMD parallelization*.

Figure 3.1b illustrates our technique. Like Figure 3.1a, the program begins execution serially on a host thread. When the code reaches a nested MIMD-SIMD loop, it spawns multiple threads and schedules them to the other cores of the CPU corresponding to the iterations of the irregular outer loop. Again, the irregularity of these regions means that they cannot be gainfully executed in their totality on the GPU due to characteristics like data-dependent control statements and other irregular code structures that perform poorly on the GPU. However, when these threads reach the SIMD regions of code nested within the MIMD outer loops, each thread launches a kernel to the GPU, allows it to run to completion, and then returns to executing irregular code. Such nested MIMD-SIMD parallelization can increase the performance of heterogeneous microprocessors in two possible ways. First, launching multiple dynamic GPU kernels from each CPU thread provides more parallelism that boosts the GPU's spatial and temporal utilization. Second, the portions of the code outside of the SIMD regions (W_{MIMD}) execute in parallel on the multi-core CPU, where they would otherwise have been executed serially under single-loop SIMD parallelization.

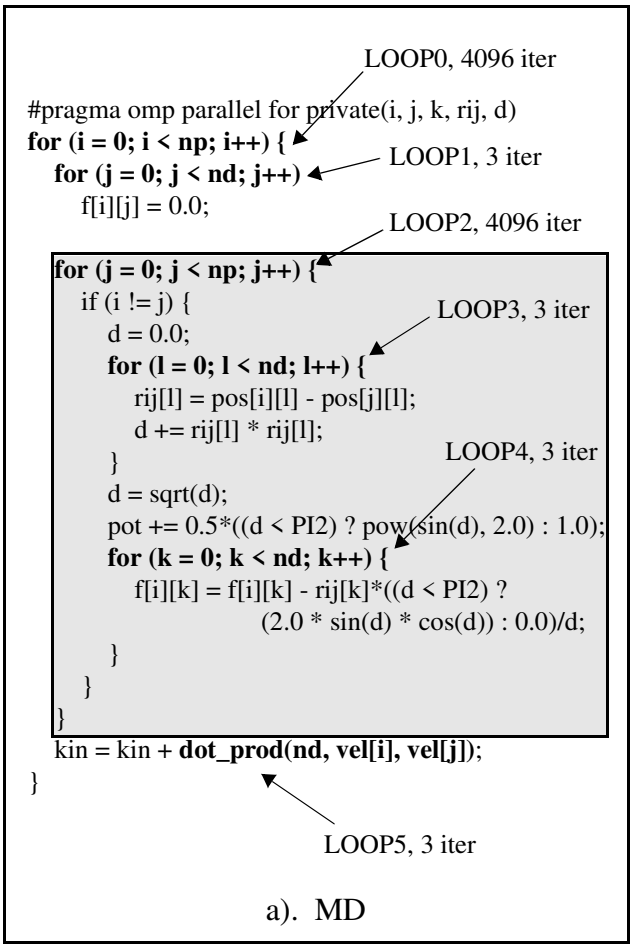


Figure 3.2: Code example from the MD bemchmark exhibiting opportunities for nested MIMD-SIMD parallelization. Our technique schedules the parallel outer loops (OpenMP pragmas) on CPUs, and the parallel inner loops (shaded boxes) on GPUs.

3.1.1 Code Examples

We conducted a survey of several programs, looking for hierarchical heterogeneous parallelism for our technique to exploit. We found that OpenMP programs often provide the kind of parallel code structures and characteristics that nested MIMD-SIMD parallelization exploits. Using OpenMP, programmers can conveniently annotate code with compiler directives to parallelize code for CPUs and

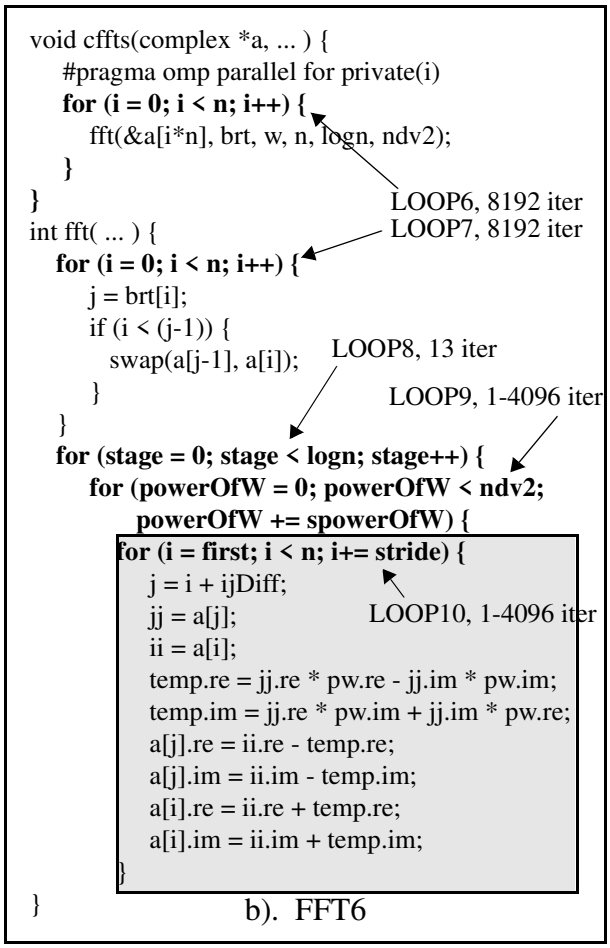


Figure 3.3: Code example from the fft6 benchmark exhibiting opportunities for nested MIMD-SIMD parallelization. Our technique schedules the parallel outer loops (OpenMP pragmas) on CPUs, and the parallel inner loops (shaded boxes) on GPUs.

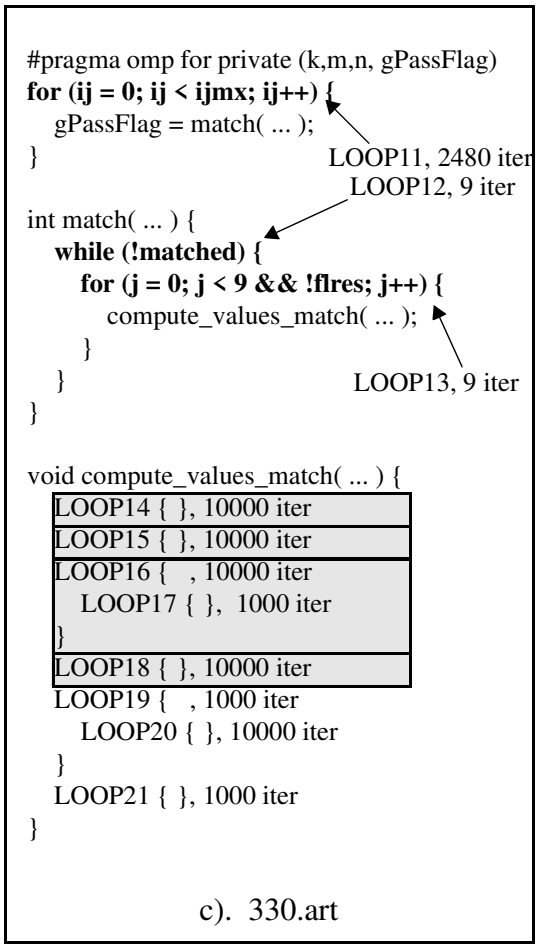


Figure 3.4: Code example from the 330.art benchmark exhibiting opportunities for nested MIMD-SIMD parallelization. Our technique schedules the parallel outer loops (OpenMP pragmas) on CPUs, and the parallel inner loops (shaded boxes) on GPUs.

GPUs [11]. The ease of use has made OpenMP one of the most popular programming environments, and so there are numerous OpenMP programs in existence. OpenMP programs which parallelize for the GPU are a relatively recent development compared to OpenMP for CPU parallelism, and as such there are fewer examples. Regardless, these GPU OpenMP workloads would be functionally similar to workloads coded in CUDA *et al.* and would similarly be prone to the same kind of single-loop SIMD parallelism that we wish to avoid. For these reasons, we focused on investigating OpenMP programs for CPUs. Many of these parallel CPU programs parallelize complex loop nest structures that would be challenging or impossible to accelerate on GPUs. Thus, with these complex CPU workloads, we begin with only *half* of the kind of parallelism we are looking for, it having been exposed by the programmer. But, by looking more closely we find the other half, regular loops, nested within those complex structures if they exist.

Figures 3.2, 3.3, and 3.4 present three examples of programs that exhibit nested MIMD-SIMD parallelism, MD, FFT6, and 330.art, respectively. MD and FFT6 are from the OpenMP source code repository [17], and 330.art is from the SPEC OMP 2001 benchmark suite [18]. In each example, we show a loop nest structure where the outer loop is explicitly parallelized by a "#pragma omp" compiler directive. The iterations of these outer-most loops are the source of the CPU parallelism that we exploit in our technique– in other words the outer loop iterations are executed in parallel on the CPU. We also show all of the loops nested within the outer loops, labeling each of these nested loops, and indicate their iteration counts. The regular loops with SIMD characteristics are highlighted in shaded boxes (for 330.art in

Figure 3.4, inside the "compute_values_match()" function, there are eight loops – we omit their contents to fit them all in the figure).

The inner loops showcased in these examples have some interesting characteristics that are relevant to our technique. First, as previously discussed, the number of loop iterations is an important factor to consider. When a loop is transformed into a GPU kernel, the number iterations in the loop translates directly to *nthreads*. Thus, if a loop has a trivial number of iterations, like LOOP3 at three iterations, the resulting kernel will have only 3 total threads. This will not be nearly enough parallelism to fully utilize the thread contexts on any GPU. The GPUs we evaluate in Sections 3.4 and 3.7 have 24,000 and 5,000 hardware thread-contexts, respectively. Additionally, small thread-count will typically mean small W_{SIMD}, so these small loops will not have enough execution time to mitigate the impact of launch overhead. (The relationship between *nthreads* and W_{SIMD} is not a hard and fast rule, but for the programs we evaluate, the relationship holds).

Another characteristic of the loops in these examples relevant to our discussion here is irregularity. As we discuss in Chapter 2, Irregular loops, *i.e.*those with control flow-divergence or memory divergence, can perform poorly on the GPU. LOOP2 and LOOP7 in Figures 3.2 and 3.3 for instance both exhibit control-flow divergence. LOOP2 however, is nevertheless transformed into a kernel in our evaluation, while LOOP7 is passed over. This is due to the number of control flow divergences. LOOP2 will only have one thread diverge per kernel launch so that only one warp in the entire grid will have to serialize. Meanwhile, LOOP7's control flow divergence is data dependent, rather than iteration dependent, and thus

could cause many serializations to occur. Other loops in the programs we studied exhibit memory divergence. LOOP10 in Figure 3.3 has strided array accesses, and thus experiences some performance degradation as well, though ultimately it can be gainfully executed on the GPU.

In summary, criteria for loops to be gainfully executed on the GPU are: enough *nthreads* to fully utilize the GPU, large enough W_{SIMD} to overcome launch latency, minimal control-flow divergence to prevent warp-serialization, and minimal memory divergence to minimize the number of memory requests to the cache memory system.

Programs with nested MIMD-SIMD parallelism can have many nesting levels, with nesting occurring across function calls, and SIMD loops can be found anywhere in the hierarchy. FFT6 and 330.art (Figures 3.3 and 3.4) are examples of this. In FFT6, the outer loop parallelizes 8192 iterations, with each calling the "fft" function in parallel. Inside the "fft" function, the SIMD loop is found under two more layers of loops. In 330.art, the SIMD loops (LOOP14, LOOP15, LOOP17, and LOOP18) are found under not one, but three layers of functions and loops. This has a number of implications for nested MIMD-SIMD parallelization. First, the SIMD loops (the GPU kernels) can execute multiple times in a single outer loop thread, presenting opportunity for temporal utilization of the GPU. Second, complex nesting structures nested within the outer loop contribute to a larger W_{MIMD} from Figure 3.1. Again, as evidenced by LOOP7 and LOOP12, many of these regions contain control-flow divergence that cannot be gainfully executed on the GPU, and therefore are part of W_{MIMD}. Under single-loop SIMD parallelization, W_{MIMD} would not be parallelized, and performance gains would be limited due to Amdahl's

Law. This is the $W_{MIMD} \approx W_{SIMD}$ problem alluded to in Figure 3.1.

In the OpenMP programs we evaluated, we found that parallel loops nested within an explicitly parallelized outer loop were a fairly common feature.[2] Given that some of these parallel inner loops can be executed gainfully on the GPU for the reasons we outline, these OpenMP programs commonly provide the kind of heterogeneous nested MIMD-SIMD parallelism that our technique seeks. By leveraging CPU parallelism, we can launch multiple instances of these inner loops on the GPU at the same time (if they have sufficiently small *nthreads*), increasing spatial utilization of the GPU. In cases where one SIMD loop (GPU kernel) fully utilizes the GPU's thread-contexts, other outer loop iterations will inevitably spread out temporally, executing their S_{MIMD} computations in parallel to the GPU's execution of the kernel and other CPU threads. This addresses Amdahl's Law and also continually feeds the GPU with work, increasing temporal utiliation.

As we uncover parallelism in these OpenMP programs, for our technique to be successful we must carefully choose the loops that we transform into kernels and launch to the GPU. As we have discussed, some loops have small *nthreads* and/or irregular code that can cause the resultant GPU kernels to exhibit a slowdown when executed on the GPU. However, this is difficult to quantify just by code inspection. Whereas CPU parallelism can generally be assumed to provide linear speedup with the number of cores utilized, GPU speedup is more complex. Launch overhead, branch divergence, memory divergence, utilization, register usage, and

[2]By parallel, we mean the nested loop either has no cross-iteration dependences, or it computes a reduction.

more can have significant effects on the throughput of a GPU kernel. Thus, to determine if a loop can be gainfully executed on the GPU, it must be directly assessed *by executing it as a kernel on the GPU*. This assessment could be performed statically or dynamically. For example, a dynamic assessment may employ inspector-executor techniques to decide at runtime whether or not to off-load (the program could compile two versions of the loop: one CPU loop, and one GPU kernel launch). Dynamic assessment could save programmer effort, be more adaptable to changes at runtime, and be more modular across varying architectures. In our work, we employ a static approach. We manually determine which inner loops to launch to the GPU as kernels and do not adapt at runtime.

3.1.2 Speedup Analysis

We also wish to quantify the potential speedups of nested MIMD-SIMD parallelization. With the CPU and GPU working in concert, their individual speedups will combine. Let us consider an OpenMP region's execution time, T, when executed fully on a *single* CPU thread (*i.e.*no CPU or GPU parallelism), that we wish to speed up with our technique. Assume that applying MIMD parallelism to the OpenMP region (the outer loop) will result in a speedup of S_{MIMD}. Next, let us assume that the execution time of the SIMD loop(s) inside the region is a fraction of the serial execution time: f_{SIMD} (f_{SIMD} is related to $\frac{W_{SIMD}}{W_{SIMD}+W_{MIMD}}$ from Figure 3.1, but is based on execution time rather than computational work). Then, if we apply SIMD parallelism to the f_{SIMD}, we assume we will achieve S_{SIMD} speedup. By

applying the combination of the two types of parallelism, *i.e.*nested MIMD-SIMD parallelization, we can estimate overall speedup as:

$$Speedup = \frac{T}{\left(\frac{T\times(1-f_{SIMD})+\frac{T\times f_{SIMD}}{S_{SIMD}}}{S_{MIMD}}\right)} = \frac{S_{MIMD}}{(1-f_{SIMD})+\frac{f_{SIMD}}{S_{SIMD}}} \quad (3.1)$$

To combine the two speedups into one, the above equation is similar to Amdahl's Law. Just as Amdahl's Law applies speedup to only the portion of the code that is parallel, the unsimplified equation applies S_{SIMD} to only the fraction of the execution time spent on SIMD code, f_{SIMD}. (resulting in the execution time of single-loop SIMD parallelism). Then, given that GPU accelerated execution time S_{MIMD} is applied.

Equation 3.1 is not perfect at predicting the speedup of our technique; instead serves as an upper bound estimate of speedup. As we have discussed, our technique boosts GPU utilization by launching multiple kernels dynamically from multiple CPU threads. Equation 3.1 takes this into account via the multiplication of S_{MIMD} with the entirety of $\frac{1}{(1-f_{SIMD})+\frac{f_{SIMD}}{S_{SIMD}}}$. However, this assumes that all of the kernels from each CPU thread can fit onto the GPU, which is not always the case. For example, two kernels with *nthreads* greater than the number of available hardware thread contexts on the GPU would cause contention for those contexts if launched simultaneously– simply put, the GPU will be full. However, if f_{SIMD} is low enough that the GPU is not always occupied during one thread's execution, other threads may fill that time with their kernels, boosting temporal utilization and overcoming

the spatial contention caused by too high of an *nthreads* value. Another issue that may cause deviation from the ideal of Equation 3.1 is contention for cache resources. When the working sets from multiple threads of execution combine, this can sometimes cause thrashing if the combined working sets are too large. In our evaluation, we see examples of these deviations from the estimated speedup. Nevertheless, the above equation still provides valuable intuition on the expected gains.

3.2 Enabling Further SIMD Parallelization

In our study of OpenMP workloads to identify nested MIMD-SIMD parallelism, we came across some examples where all of the nested loops in the region were serial, *i.e.* they had cross-iteration dependences, and thus could not execute on the GPU. In some cases, this was the point at which we would move on to other workloads. In others, however, while a nested loop had some cross-iteration dependences, it also had regular data parallel computations independent of those dependent computations. One potential reason for this is that arrays are ubiquitous, and computations on them are often SIMD in nature–*i.e.*, the same operation is performed on every array element. A programmer may, if he or she has no intention of parallelizing the SIMD computations, write a single loop that combines the SIMD and serial computations (after all, one loop is easier to write than two). We can *extricate the regular parallel computations from the serial ones, creating two separate loops*. This *loop fission* results in both MIMD and SIMD parallelism for

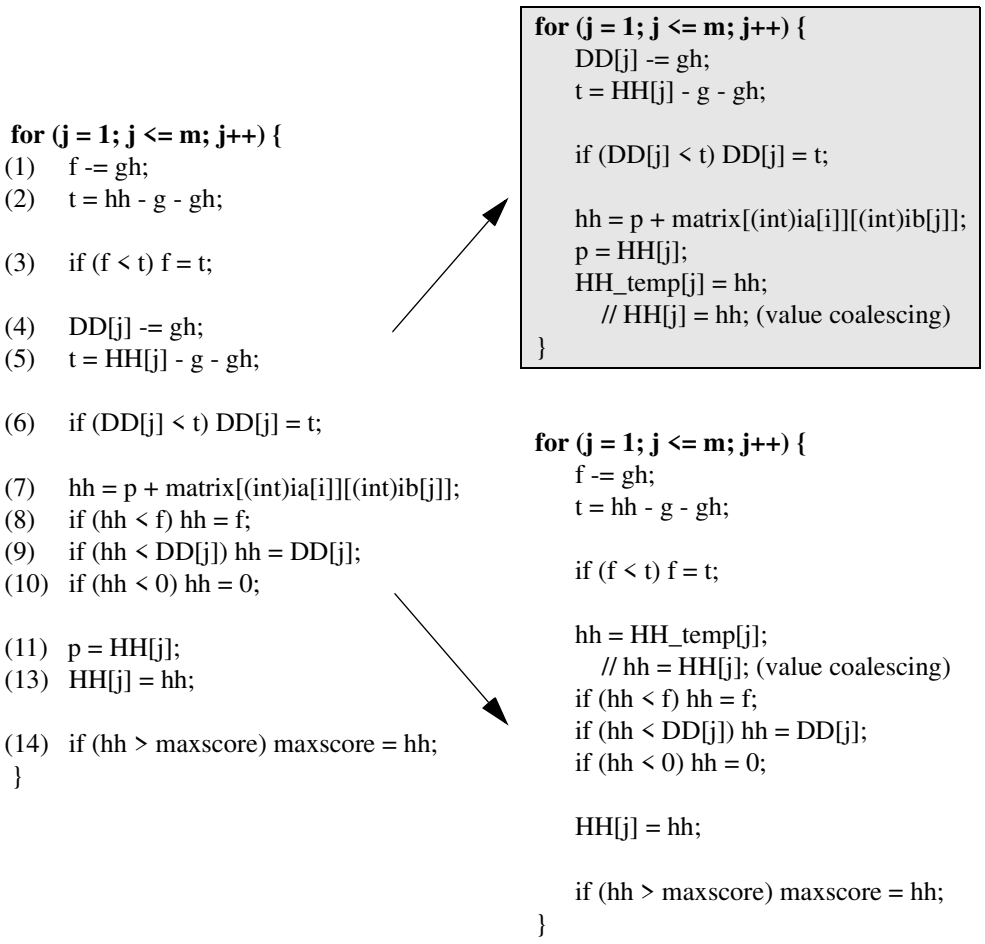


Figure 3.5: Loop fission performed on an inner loop from the 358.botsalgn benchmark. The transformation produces a regular parallel loop (shaded box) that can be off-loaded to the GPU, and a serial loop that remains on the CPU.

our technique to exploit.

To carry out loop fission, we identify the code statements that do and do not participate in the loop-carried dependences within the loop and separate them into two groups. We then copy the loop header from the original loop to each of these groups, and form two new loops, one SIMD and one serial, each which performs the same number of iterations as the original loop. Loop fission has been used before to extract parallelism from non-parallel loops [19], sometimes for vectorization purposes [20]. Our technique similarly extracts code meant to be transformed into

GPU kernels and launched to the GPU. In our work, we perform this transformation manually.

Figure 3.5 illustrates our loop fission transformation. In the figure, we show a code example taken from the 358.botsalgn benchmark which is part of the SPEC OMP 2012 benchmark suite [21]. The left half of the figure shows the original, untouched, serial loop that is nested inside an OpenMP parallel region (the rest of the OpenMP loop is not shown). This loop computes the result of two arrays "DD" and "HH." "DD" can be computed entirely in parallel since its computations (lines #4 and #6 of Figure 3.5) have no loop carried dependences. The computation of "HH," on the other hand, is on the whole serial. The first part of its computation which produces the temporary scalar "hh" (line #7) is parallel. The conditional update of that initial value (line #8) is serial because it depends on "f," which in turn depends on all previously computed "hh" values (lines #2 and #3). Thus, even though it has parallel computations, on the whole this loop is serial.

The right half of Figure 3.5 shows the result of the loop fission transformation: two loops. The first loop, in a shaded box, is a regular data-parallel loop now amenable to kernel transformation and execution on the GPU. It contains the computation of "DD" and the parallel portion of "HH." The loop below it is the serial loop that must be executed on the CPU. Notice that there is an additional array called "HH_temp." Since we split the computation of the "HH" into two parts, parallel and serial, the intermediate result (that of the parallel portion of the computation) must be communicated from one loop to another. We use "HH_temp" for this purpose.

Unfortunately, there is no free lunch. Using a temporary array to enable parallelism increases the overall data footprint of our two new loops when compared to the original loop. Previous work [22] has proposed *value coalescing* which can eliminate this extra memory overhead. Value coalescing utilizes free array elements already in the loop structure to communicate intermediate values rather than using a new array. For example, in Figure 3.5, once used in the parallel loop, each "HH" array element ("HH[j]") can be used to pass the intermediate value to the serial array instead of the temporary array, eliminating the memory overhead. The two commented lines of code in Figure 3.5 illustrate this value coalescing optimization.

Though it is a good tool to have, value coalescing cannot always be applied. The cases where it cannot be must utilize temporary arrays to pass data between parallel and serial loops generated by loop fission. Fortunately, the benefits of GPU acceleration outweighed the added overhead from the temporary array for the benchmarks we evaluate in Section 3.4, and the loop fission transformation was profitable. In general though, profitability of loop fission should be evaluated on a case-by-case basis.

Finally, we note that the temporary variable resultanting from the loop fission transformation we present here constitutes a producer-consumer relationship between the parallel loop (the producer) and the serial loop (the consumer). Once the parallel loop is transformed into a kernel, this relationship becomes heterogeneous. In Chapter 4, we evaluate how heterogeneous producer-consumer data sharing such as this can utilize the shared cache system of an integrated heterogeneous CPU-GPU microprocessor efficiently to communicate shared data on-chip. Indeed, one

of the benchmarks we evaluate for nested MIMD-SIMD parallelism, 372.SmithWa [21], is used to evaluate the locality transformation we discuss in Chapter 4.

3.3 Simulator Study Methodology

The rest of this chapter presents the results of our quantitative evaluation of our nested MIMD-SIMD parallelization technique. First, we perform an evaluation using the cycle accurate gem5-gpu simulator [23], and gain a deep understanding of our technique's performance. We follow that with an evaluation on a physical machine, demonstrating that our technique also works on a real integrated CPU-GPU platform.

3.3.1 Simulator Study Methodology

CPU		GPU	
Number of cores	4	Number of SMs	16
Clock rate	2.6 GHz	Clock rate	900 MHz
Issue width	8	Number of SPs per SM	32
Issue queue size	64	Warp size	32
Reorder buffer size	192	Maximum warps per SM	48
L1 I/D cache (private)	32 KB/64 KB	L1 cache (private per SM)	128 KB
L2 cache (private)	2 MB	L2 cache (shared by all SMs)	2 MB
Main Memory			
Each channel			64-bit, 1.848 GHz, DDR3
Total bandwidth, 4 channels			110 GB/s

Table 3.1: Simulation parameters used in the experiments. The modeled heterogeneous microprocessor resembles an Intel integrated chip containing a Core i7 CPU and an Iris Pro Graphics 580 GPU.

Table 3.1 presents the configuration we use for the gem5-gpu simulator. We model a heterogeneous CPU-GPU chip with four out-of-order superscalar CPU cores

and a 2-level cache hierarchy. The GPU contains 16 streaming multiprocessors (SMs), each with a warp size of 32 threads, at 48 maximum warps per SM, leading to $16 * 32 * 48 = 24,576$ concurrent thread-contexts in total. We employ the "fusion" configuration in which the CPU and GPU enjoy a unified shared address space visible from both types of cores. Coherence is also maintained between the CPU and GPU L2 caches by a MOESI cache coherence protocol, and all the cores share a DDR3 main memory system.

The configuration in 3.1 was chosen to be similar to an Intel heterogeneous microprocessor–*e.g.* a Core i7 CPU with an Iris Pro Graphics 580 GPU [24]. However, gem5-gpu models Nvidia GPU micro-architecture, so slight differences occur. The simulator executes C code on the CPU and CUDA code on the GPU. Though execution of OpenCL code was possible, the implementation in gem5-gpu was less mature than that of CUDA, limiting what we could do with it. For example, the OpenCL implementation did not support "fusion" mode for unified shared addressing which is important to the accurate evaluation of integrated CPU-GPU microprocessors.

As we have discussed, low-latency kernel launches are important to gainful acceleration of regular loops as kernels on the GPU. To this end we made a few modifications to the simulator to support low latency launches. We set the launch latency to 4.5 μs, a fairy aggressive target reported in the literature [25]. We also consider even lower launch latencies for benchmarks with higher launch latency sensitivity.

We also modifed the simulator to support simultaneous GPU kernel launches

from different CPU threads. As previously discussed, our technique boosts spatial utilization of the GPU by launching multiple kernels from multiple CPU threads, and an inability to do so would be detrimental to performance. We examine the effects of this *concurrent kernel launch* in Section 3.4.4.

Benchmark	Suite	Input	# SIMD	Lines Changed	Sim
MD	OMP repo	4096	1	69	1.65
FFT6	OMP repo	8192	1	60	1.83
330.art	OMP2001	ref	4	158	3.15
358.botsalgn	OMP2012	ref	2	191	4.82
359.botsspar	OMP2012	ref	3	92	60.4
367.imagick	OMP2012	ref	1	38	21.4
372.smithwa	OMP2012	ref	1	68	1.47

Table 3.2: OpenMP programs used in the experimental evaluation. Simulated instruction counts, labeled "Sim," are reported in billions.

3.3.2 Benchmarks

Using our modified gem5-gpu simulator, we evaluate nested MIMD-SIMD parallelization. We surveyed several OpenMP benchmarks across different benchmark suites to identify those benchmarks that exhibit the nested parallel structures our technique targets. About half the benchmarks we surveyed exhibited such nested parallel structures. This yielded seven benchmarks, which are listed in Table 3.2. The benchmarks are MD and FFT6 from the OpenMP source code repository, 330.art from the SPEC OMP 2001 suite, and 358.botsalgn, 359.botsspar, 367.imagick, and 372.smithwa from the SPEC OMP 2012 suite. Input sizes of $N = 4096$ particles and $N = 8192$ points were used for MD and FFT6, respectively. For all of the SPEC benchmarks, the ref inputs were used.

The gem5-gpu simulator we used runs in system emulation mode, where stubs of system calls are used rather than a full operating system. So, we manually ported the OpenMP pragmas appearing in our benchmarks to pthreads, and then linked the pthread code against the M5 threads library [26] which provides pthreads compatibility in system emulation mode. We did this for the most important OpenMP region from each benchmark. (In many benchmarks, there is only one OpenMP region). Most of the time, the OpenMP region we ported is a loop with load-balanced iterations. For these benchmarks, we statically partitioned the loops across the M5 threads. One exception is 359.botsspar which performs blocked LU decomposition on a sparse matrix. If different parts of the matrix are statically assigned to threads, then load imbalance can occur depending on the fill pattern of the matrix. For 359.botsspar, we instead created a work queue, and dynamically distributed work to the M5 threads. (This emulates the OpenMP code for 359.botsspar which uses dynamic scheduling across the threads). The outer parallel region in 359.botsspar is the most irregular across all our benchmarks, and is the closest to exhibiting task-level parallelism.

After we created pthreads versions of our benchmarks, we examined the loops nested within the OpenMP regions to identify those with regular parallelism. (As described in Section 3.1). In the case of two of the benchmarks we evaluated, we extracted regular parallelism from otherwise irregular loops with our loop fission transformation from Section 3.2. In Table 3.2, the column labeled "# SIMD" reports the number of regular data parallel loops identified for each benchmark, including those uncovered via loop fission.

From here, we transformed these regular loops, written in C, to kernels, written in CUDA. We then compiled the C and CUDA code with GCC and NVCC, respectively, and linked them together with NVCC. In all cases we used the highest level of optimization possible. The "Lines Changed" column in Table 3.2 reports the lines of CUDA and C code added to the OpenMP region.

Finally, we ran the compiled benchmarks on our modified gem5-gpu simulator. We fast-forwarded to the parallel OpenMP region we are interested, in which we collected detailed statistics. The last column of Table 3.2 reports the number of instructions (in billions) executed during detailed simulation.[3]

For MD and 367.imagick, we were able to simulate the entire OpenMP region, but for other benchmarks, the OpenMP regions were too large to simulate to completion within a reasonable timeframe. Instead, for the rest of the workloads we chose a representatiuve portion of each OpenMP region to simulate. For FFT6, 330.art, 358.botsalgn, and 372.smithwa, each iteration of the OpenMP outer loop performs very similar computations for every iteration of the loop. So, for those benchmarks we simply simulated several iterations from the beginning of the loop. For 359.botsspar, the openMP region performs partial pivoting and sub-matrix elimination underneath the pivot element. Unfortunately, the computation is different for different pivots, in particular affecting the amount of work done by the GPU relative to the CPU, *i.e.* f_{SIMD}. We identified the pivot for which the f_{SIMD} is representative of the OpenMP region as a whole, and simulate the computation

[3]These instruction counts correspond to executing the simulated portion of the OpenMP region on a single CPU thread.

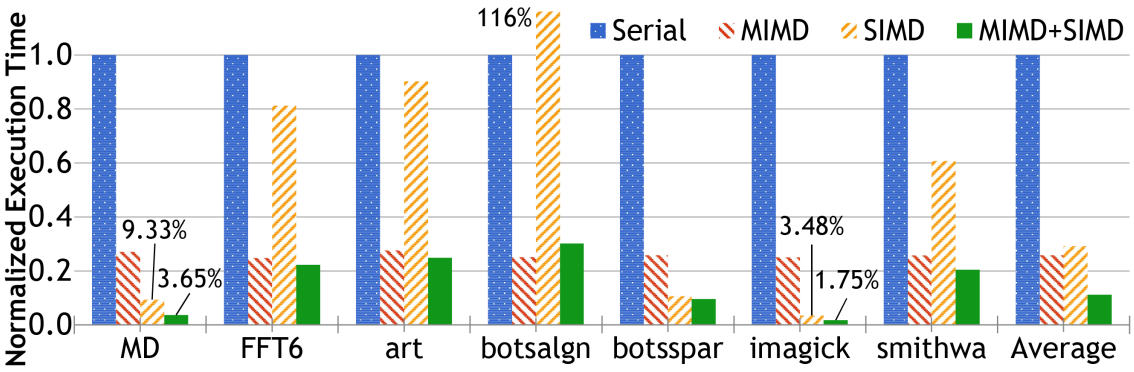


Figure 3.6: Normalized execution time for no parallelization (Serial), CPU parallelization (MIMD), single-loop SIMD parallelization (SIMD), and nested MIMD-SIMD parallelization (MIMD+SIMD).

associated with that pivot alone.

3.4 Simulator Study Results

This section presents the results of the simulator study we performed to evaluate nested MIMD-SIMD parallelization. It begins by presenting the main results in Section 3.4.1, and follows with a detailed performance breakdown in Section 3.4.2 to better understand the results. Next, we evaluate how scaling the number of CPU cores in our configuration impacts our gains in Section 3.4.3. Finally, in Section 3.4.4 we determine the impact that concurrent kernel launch has on our technique's performance.

3.4.1 Main Result

Figure 3.6 presents the main results from our simulator-based evaluation. In the figure, we first show the results of four execution schemes applied to each of our seven benchmarks, and the averages of those. First, the "Serial" bars show the

execution time of each benchmark without any parallelism. That is, the regular inner-loops executing serially on the CPU, and the outer OpenMP loops executing sequentially on one thread. We normalize each benchmark's results in Figure 3.6 to this result, and thus the "Serial" bars all have the value of 1.0. We follow this by successively applying three parallelization schemes, the "MIMD" bars show the result of applying CPU-only parallelization, the "SIMD" bars show the result of applying GPU-only parallelization (single-loop SIMD parallelization from 3.1), and finally the "MIMD+SIMD" bars show the result of our nested MIMD-SIMD parallelization technique.

The "MIMD" bars in Figure 3.6 reflect the original OpenMP parallelization performance of each benchmark (translated to pthreads for the simulator), *i.e.* CPU-only parallelism. Most of the "MIMD" bars show a normalized exection time near 0.25. The speedup for these "MIMD" bars compared to the "Serial" is reported in Table 3.3 under the "S_{MIMD}" column. The Table shows the benchmarks achieve between 3.63x and 4.03x speedup (FFT6 achieves a slight superlinear speedup due to the increased cache capacity that comes with scaling). On average "S_{MIMD}" is 3.87x, close to the perfect linear speedup one would expect from four CPU cores. These results show how the original programs were well suited to being parallelized with OpenMP, as they achieve good performance on a multicore CPU.

Next, the results of single-loop SIMD parallelization, *i.e.*launching GPU kernels from a single CPU thread, show that this parallelization scheme can achieve good speedup as well. In Figure 3.6, the "SIMD" bars report execution time of the single-loop SIMD case normalized to the serial case. On average we achieve a

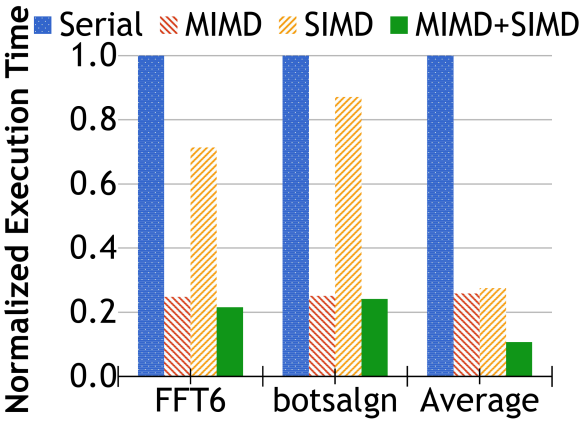


Figure 3.7: Normalized execution time assuming user-level kernel launch (0.1 μs latency). The "Average" bars include the other 5 benchmarks from Figure 3.6.

70.8% reduction execution time and a 3.43x speedup reported in Table 3.3. However, these results are not as consistent across individual benchmarks as the CPU-only parallelization results. For MD, 359.botsspar, and 367.imagick, there are very large gains–between 89.3% and 96.5% reduction in execution time compared to the "Serial" bars. For the other benchmarks, however, the gains are more modest and show between 9.8% and 39.3%reduction, with 358.botsalgn actually showing an *increase* in execution time of 16.2%. What these results tell us is that in general, exploiting single-loop SIMD parallelization in OpenMP programs is beneficial, but that the complex nature of these codes may cause variability in GPU performance across different benchmarks.

One of the sources of this variability is sensitivity to kernel launch latency. As we discussed in section 3.1, some of the loops in our OpenMP benchmarks exhibit a small W_{SIMD}. It is therefore crucial for the gainful execution of these loops as kernels on the GPU that the system have a lower kernel launch latency. The results in Figure 3.6 assume our baseline launch latency of 4.5 μs [25]. While this is an

aggressively low launch latency, it is still unfortunately too high for FFT6 and 358.botsalgn which are the two most latency sensitive benchmarks in our study. Figure 3.7 shows the result of a much more aggressive 0.1 μs launch latency. To simulate this, we set the system-call handler delay in the simulator to zero, leaving only the CUDA library launch stub. In Figure 3.7, the "SIMD" bars show an improved 28.7% execution time reduction (compared to only 18.8% in Figure 3.6), and 358.botsalgn's execution time *increase* turns into an execution time *decrease* of 12.8%. (The other benchmarks do not benefit from a 0.1 μs launch latency). This increases the average speedup from 3.43x in Figure 3.6 to 3.55x in Figure 3.7.

Finally, we combine both types of parallelization, SIMD and MIMD, with our nested MIMD-SIMD parallelization technique. The bars labeled "MIMD+SIMD" in Figures 3.6 and 3.7 report normalized execution time when the nested GPU loops are off-loaded in parallel from multiple CPU threads simultaneously. The last column in Table 3.3 labeled "Actual" reports the speedup of the "MIMD+SIMD" bars over the "Serial" bars.

In Figure 3.6, we see our nested MIMD-SIMD parallelization scheme achieves large gains, providing execution time reductions between 75.1% and 98.3% compared to the "Serial" bars. Specifically, Table 3.3 reports our nested MIMD-SIMD parallelization scheme provides 27.4x, 10.47x, and 57.1x speedup for MD, 359.botsspar, and 367.imagick, respectively. For 330.art and 372.smithwa, our technique provides 4.01x and 4.89x speedup, respectively. And for FFT6 and 358.botsalgn, we provide 4.50x and 3.32x speedup, respectively. As before, the last two benchmarks are the most sensitive to kernel launch overhead. When using a 0.1 μs launch latency, as

is done in Figure 3.7, these speedups increase to 4.63x and 4.13x, respectively, as reported in Table 3.3. On average, nested MIMD-SIMD parallelization (assuming 0.1 μs launches for FFT6 and 358.botsalgn) provides a 16.1x speedup over the serial execution time.

These results show that not only does nested MIMD-SIMD parallelization provide large speedups, but that it also beats CPU-only and GPU-only parallelization schemes by leveraging MIMD parallelism on the CPU and SIMD parallelism on the GPU simultaneously. In particular, nested MIMD-SIMD parallelization outperforms CPU-only parallelization (the MIMD bars in Figures 3.6 and 3.7) by 4.13x on average, and outperforms GPU-only parallelization (the SIMD bars) by 2.74x on average. Notice, in Figures 3.6 and 3.7, nested MIMD-SIMD parallelization is always the best parallelization scheme (except for 358.botsalgn in Figure 3.6 due to launch latency sensitivity), but the next-best scheme is benchmark dependent. For MD, 359.botsspar, and 367.imagick, the next-best scheme is single-loop SIMD parallelization, but for the other benchmarks, the next-best scheme is OpenMP parallelization. Averaged across all the benchmarks, our nested MIMD-SIMD parallelization scheme beats the next-best scheme by 45.8%.

3.4.2 Performance Breakdown

This section presents detailed results to break down and better understand the gains of nested MIMD-SIMD parallelization that were observed in our simulation study. First, Figure 3.8 isolates the benefits of transforming regular inner loops in

Bench	S_{MIMD}	S_{SIMD}	f_{SIMD}	UpperB	Actual	CPU	GPU
MD	3.70	9.19	0.99	33.9	27.4	0.12	0.43
FFT6	4.03	2.85	0.32	5.09	4.63	0.71	0.038
art	3.63	52.1	0.12	4.09	4.01	0.99	0.0031
botsalgn	3.99	2.72	0.38	5.24	4.13	0.67	0.011
botsspar	3.87	9.40	0.98	30.5	10.5	0.12	0.40
imagick	4.00	22.4	0.99	85.2	57.1	0.018	0.98
smithwa	3.90	22.3	0.55	8.19	4.89	0.95	0.16
average	3.87	17.3	0.62	24.6	16.1	0.51	0.29

Table 3.3: S_{MIMD}, S_{SIMD}, and f_{SIMD} parameters for our benchmarks. The UpperB and Actual columns report upper-bound and actual speedup, respectively, of nested MIMD-SIMD parallelization. The last two columns report CPU and GPU utilization. (Parameters for FFT6 and botsalgn reflect the results shown in Figure 3.7).

our OpenMP regions into kernels, and executing them on the GPU. In other words, it shows if a loop can be gainfully executed on the GPU. To determine these benefits, we individually measure the execution time of the original regular loops on the CPU, that is, excluding the rest of the OpenMP regions outside of these loops. We then individually measure the execution time of the resultant kernels on the GPU and compare the results to the original loop, resulting in the *per-kernel performance gain*. The "Serial" and "SIMD" bars in Figure 3.8 show this per-kernel gain, with the "SIMD" bars normalized to the "Serial" bars. Notice, 330.art, 358.botsalgn, and 359.botsspar contain multiple regular loops inside their OpenMP regions; these are measured and presented separately. In Table 3.3, the "S_{SIMD}" column reports the per-kernel speedup (for the benchmarks with multiple kernels, it reports the time-weighted average per-kernel speedup). Second, the vertical labels in Figure 3.8 report *nthreads* from Figure 3.1–*i.e.*, the number of threads that occupy the GPU each time the kernel is launched from a single CPU thread. Lastly, the fourth column of Table 3.3 reports f_{SIMD} from Equation 3.1–*i.e.*, the fraction of end-to-end serial

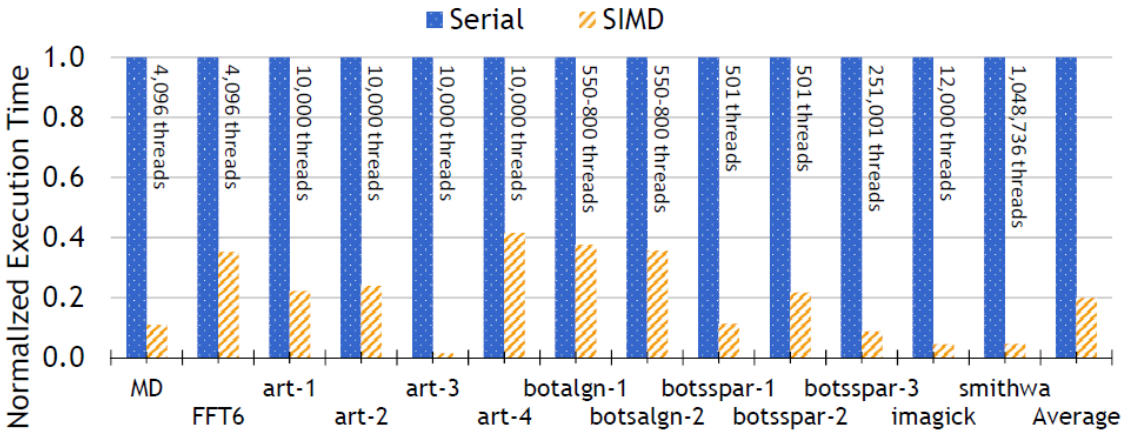


Figure 3.8: Normalized execution time for individual off-loaded loops running on a single CPU thread (Serial) and on the GPU (SIMD). Vertical labels indicate *nthreads* for each kernel. (Results for FFT6 and botsalgn's kernels assume a 0.1 μs launch latency).

execution time of the overall OpenMP region that is spent in the regular loop(s).

These results separate our benchmarks into four categories based on how our method works to improve their performance. As we discussed in Section 3.1, our nested MIMD-SIMD parallelization technique boosts performance by launching multiple kernels from multiple CPU threads to the GPU. This boosts spatial and temporal utilization of the GPU, while simultaneously parallelizing complex code on the CPU. The four categories are based on how dominant the spatial and temporal utilization aspects of our technique are for each benchmark.

3.4.2.1 Category 1: Spatial Utilizers

First, The MD and 367.imagick benchmarks comprise the category of *spatial utilizers*. Both MD and 367.imagick contain a single kernel for which the GPU achieves a large gain, as shown in Figure 3.8. Table 3.3 reveals these kernels enjoy a 9.19x and 22.4x speedup, respectively. Moreover, Table 3.3 also shows virtually

all of the time is spent in these kernels: $f_{SIMD} = 0.99$ in both benchmarks. This is why the two benchmarks' "SIMD" bars in Figure 3.6 perform so well: the large per-kernel speedup translates directly into end-to-end performance gains because of the large f_{SIMD}. The figure also reports MD and 367.imagick's kernels have $nthreads = 4{,}096$ and 12,000, respectively. In comparison, our simulated GPU from Table 3.1 can support up to 24,576 threads. This means that multiple GPU kernels from each of these benchmarks can fit onto the GPU at one time. Because f_{SIMD} is so high in both benchmarks there is very little delay between kernel launches, and therefore each CPU thread will have a GPU kernel launched at any given time. For 367.imagick, we expect 2 simultaneous kernels since its *nthreads* is about half the number of available GPU hardware threads. This is why in Figure 3.6 the performance of nested MIMD-SIMD parallelization exceeds the performance of single-loop SIMD parallelization by roughly 2x. For MD, a smaller *nthreads* of 4,096 would ostensibly permit the maximum 4 kernels (launched from the 4 CPU threads) to run on the GPU. Unfortunately, the high frequency of kernel launches in MD creates contention for the GPU's hardware launch queue which limits the number of simultaneous kernels. Nevertheless, Figure 3.6 still shows a 2.58x speedup for nested MIMD-SIMD parallelization over single-loop SIMD parallelization.

3.4.2.2 Category 2: Temporal Utilizers

This second category is comprised of one benchmark, 372.smithwa, which is a *temporal utilizer*. 372.smithwa has a large per-kernel speedup, 22.3x–as shown

in Table 3.3–that is diminished by a smaller f_{SIMD}, 0.55–also shown in Table 3.3. For 372.smithwa, we used loop fission to extract regular parallelism, Loop fission tends to result in smaller f_{SIMD} because it introduces a serial loop executed on the CPU that is comparable in size to the SIMD loop it exposes for the GPU. This results in reasonably good (single loop) "SIMD" bars in Figure 3.6. Notably, 372.smithwa's *nthreads* is over one million, as its label in Figure 3.8 reports. So, even a single kernel can fill all of the GPU's thread contexts. For nested MIMD-SIMD parallelization, this means that kernel launches from each CPU thread will serialize with the kernel launches from the other threads, because the GPU is full. In other words– 372.smithwa cannot increase spatial utilization by launching multiple simultaneous kernels because the GPU is fully spatially utilized by a single CPU thread's kernel launch. Fortunately, this serialization has the side effect of staggering the execution of the CPU threads. What this means is that the CPU threads will end up taking turns completely filling the GPU, *i.e.*the kernel launches spread out in time on the GPU, increasing the GPU's temporal utilization. Also, the f_{SIMD} of 0.55 means that 45% of the execution time of the serial case is complex CPU code. Under nested MIMD-SIMD parallelization, this CPU code is parallelized as well, so both types of parallelism are being exploited. These reasons explain 372.smithwa's performance under nested MIMD-SIMD parallelization in Figure 3.6.

3.4.2.3 Category 3: Spatial and Temporal Utilizers

FFT6, 330.art, and 358.botsalgn comprise the third benchmark category – those benchmarks that boost both spatial and temporal utilization. In this category, Figure 3.8 shows the per-kernel gains are mixed. 330.art exhibits very strong per-kernel performance (kernel art-3, which is 330.art's most dominant kernel by far, achieves a 76.4x speedup) but the other two benchmarks exhibit more modest per-kernel performance. Table 3.3 shows 358.botsalgn's per-kernel speedup is 2.72x (averaged over its two kernels, which achieve 2.66x and 2.81x speedup), while FFT6's per-kernel speedup is 2.85x.

Another issue is that Table 3.3 also shows these benchmarks' f_{SIMD} is small, between 0.12 and 0.38. Both FFT6 and 330.art exhibit smaller f_{SIMD} due to significant amounts of code outside of their inner SIMD loops. For 358.botsalgn, like 372.smitha, we used loop fission to expose the nested SIMD loops, resulting in smaller f_{SIMD}. Due to their smaller f_{SIMD}, the per-kernel gains for FFT6, 330.art, and 358.botsalgn are diminished by Amdahl's Law. This is why these benchmarks exhibit the lowest single-loop SIMD performance in Figures 3.6 and 3.7. On the bright side, nested MIMD-SIMD parallelization still performs well. Because f_{SIMD} is small, these benchmarks spend a lot of time in the outer OpenMP loops, so they benefit significantly from CPU parallelization. On top of that, the GPU also contributes some gain. Similar to MD, Figure 3.8 shows these benchmarks have an *nthreads* (between 550 to 10,000) that is smaller than the 24,576 threads the GPU can handle– in other words, the GPU has the spatial capacity to handle multiple

instances of these kernels. However, the smaller f_{SIMD} results in a lower kernel launch frequency, so kernels may also spread out in time on the GPU instead, boosting temporal utilization. This is particularly evident for 330.art, which having 10,000 threads per kernel, may only launch roughly two kernels at any given time. But because it has such a low f_{SIMD} (0.12), it spreads these kernel launches out temporally in addition to spatially, on the GPU. Thus, for these benchmarks the GPU has the capacity, both spatially and temporally, to support kernel launches from multiple CPU threads.

3.4.2.4 Category 4: SIMD Dominant

Finally, 359.botsspar comprises the last benchmark category. Similar to MD, Figure 3.8 shows significant per-kernel gains for 359.botsspar, and Table 3.3 shows $f_{SIMD} = 0.98$. This is why 359.botsspar achieved such excellent single loop SIMD parallelization performance in Figure 3.6, in accordance with Amdahl's Law. Notice our nested MIMD-SIMD parallelization technique achieves very little gain over single-loop SIMD parallelization in Figure 3.6. To explain this, we again look to the spatial and temporal utilization of this benchmark. Like 372.smithwa, one of its kernels, botsspar-3, has enough threads (251K) to fully spatially utilize the GPU by itself. This would not be an issue if f_{SIMD} were low, and the kernels could spread out temporally, but that is not the case. Its two other kernels, botsspar-1 and botsspar-2, have a much smaller thread count at 501 threads each. These kernels can and, to some extent, do boost the GPU's spatial utilization. However, the

problem is these kernels are dominated by the larger kernel, which accounts for 85% of the execution time. Although multiple botsspar-1 and botsspar-2 kernels can overlap in the GPU, there is only a small opportunity for this since most of the time is spent in the botsspar-3 kernel. For 359.botsspar, the main thing that matters is GPU performance on individual botsspar-3 kernels The benchmark is dominated by SIMD performance, with a small amount of spatial and temporal GPU utilization boosting.

3.4.2.5 Upper-Bound Speedup

From the S_{MIMD}, S_{SIMD}, and f_{SIMD} values in Table 3.3, we compute the upper-bound speedup for each benchmark using Equation 3.1. This result is shown in the column labeled "UpperB" in Table 3.3 alongside the "Actual" columns which report the speedup of the simulated result. We confirm that the upper-bound speedup is always higher than the actual speedup. As we discussed in Section 3.1, Equation 3.1 does not model the contention for the GPU's hardware resources–in particular, the contention for the 24,576 thread-contexts our simulation models. In benchmarks where contention is limited through either the exploitation of spatial or temporal capacity, Equation 3.1 is a good predictor of actual speedup. (The benchmarks of category two, FFT6, 330.art, and 358.botsalgn comprise this group). MD has some limited contention as we discussed above but still, the benchmark is able to have close to 3 simultaneous kernels. So, the average error between the upper-bound and actual speedups for these four benchmarks is only 15.6%.

In the case of 359.botsspar and 372.smithwa, significant contention for GPU's thread contexts occurs. As discussed above, only a single instance of the large kernel from these two benchmarks (kernels botsspar-3 and smithwa from Figure 3.8) can run simultaneously. 359.botsspar's two smaller kernels (botsspar-2 and botsspar-2) may be able to fit simultaneously, eeking out an improvement over the single loop SIMD case and some of the contention in 372.smithwa is ameliorated by its kernels spreading out temporally, but in neither case do these improvements overcome the contention fully. Finally, 367.imagick is similarly limited by the GPU's hardware thread contexts, though to a lesser degree. At 12,000 threads, two instances of this benchmark's kernel can run simultaneously. Unlike 330.art above, the f_{SIMD} for 367.imagick is very high (.99), and it cannot spread out temporally like 330.art can. Thus, for these three contention limited benchmarks, there is a large stall component (CPU threads waiting for the GPU) that is not modeled in Equation 3.1. As a result, we see large errors in Table 3.3 of 192%, 67.5%, and 49.2% when comparing the upper-bound and actual speedups for 359.botsspar, 372.smithwa, and 367.imagick, respectively.

3.4.3 Processor Utilization and CPU Scaling

The last two columns of Table 3.3, "CPU" and "GPU", report the extent to which our nested MIMD-SIMD parallelization technique utilizes both types of cores in the heterogeneous microprocessor. For the CPU utilization, we report the fraction of time the CPU is busy averaged over the 4 CPU cores. For the GPU, we account

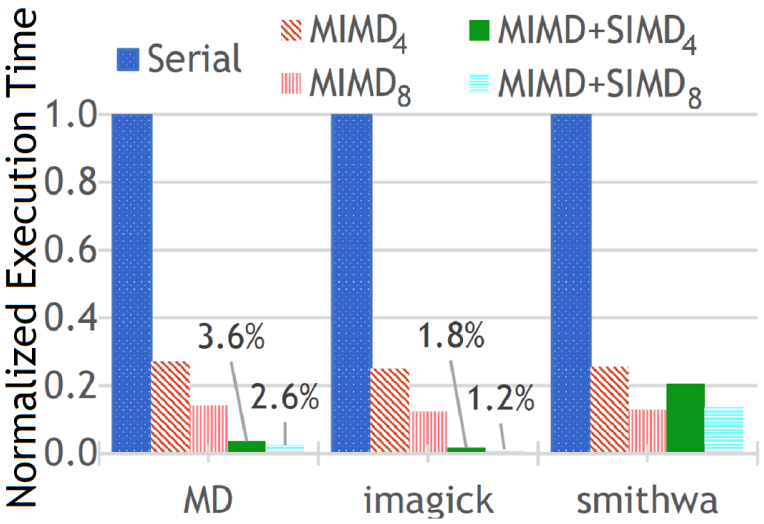


Figure 3.9: Impact of scaling to 8 CPU cores.

for both the spatial and temporal elements of the utilization by taking the average of the fractional occupancy of the GPU's thread contexts at every cycle. Because execution on the GPU blocks the off-loading CPU thread's forward progress, and because the GPU may not be fully spatially utilized for any given kernel exection, the sum of the two statistics is usually less than or equal to 1.0. 372.smithwa is the one exception to this rule, as the kernel instances spread out temporally on the GPU, so too do the regions of code executing in parallel on the CPU, keeping both occupied more often. These results naturally vary with f_{SIMD}: when f_{SIMD} is large, GPU (SIMD) utilization is higher and CPU utilization is lower, whereas when f_{SIMD} is small, the opposite is true. However, the relationship does not follow directly because f_{SIMD} is computed based on the serial (CPU-only) execution time whereas the utilization is computed over parallel (CPU and GPU) execution and includes GPU spatial underutilization. In general, these results show that the CPU is more fully utilized than the GPU on the benchmarks we studied.

Given that GPU utilization is lower than CPU utilization, we tried increasing the number of CPU cores to 8 while keeping the GPU the same to see if we could improve performance further. Figure 3.9 hows the result of this experiment for MD, 367.imagick, and 372.smithwa. In the figure, we plot the same "Serial," "MIMD," and "MIMD+SIMD" bars from Figure 3.6 (the latter have been renamed to "MIMD_4" and "MIMD+SIMD_4"), but we also plot "MIMD_8" and "MIMD+SIMD_8" which increase the number of CPU cores to 8 for the corresponding experiments. (The GPU configuration remains unchanged from Table 3.1).

First, unsurprisingly, the MIMD_8 bars in Figure 3.9 provide roughly another 2x speedup compared to the MIMD_4 bars, further demonstrating the effectiveness of (CPU-only) OpenMP parallelization in our benchmarks already discussed for Figure 3.6. More interestingly, the MIMD+SIMD_8 bars show that our nested MIMD-SIMD parallelization technique continues to scale for both MD and 367.imagick, providing an additional 1.38x and 1.50x speedup, respectively, over MIMD+SIMD_4. Increasing the number of CPU cores not only speeds up the CPU computation, it also increases the off-load frequency. As reported in Table 3.3, the GPU utilization for MD is only 43%; this leaves room for more kernels from the additional four threads, resulting in performance gains. 367.imagick is the only benchmark for which the GPU is already fully utilized at 4 CPU cores (98% GPU utilization in Table 3.3). As such, a net performance gain from launching more GPU kernels to an already full GPU is unexpected. However, the larger number of simultaneous GPU kernels increases the GPU's working set size, which, although it increases cache-miss frequency, allows for a higher degree of memory latency tolerance overall. In other

words, although we are not able to exploit any more TLP on the GPU, improved MLP (latency tolerance) improves performance for 367.imagick.

Lastly, for 372.smithwa, the GPU is also underutilized (only 16% busy according to Table 3.3), so like MD we get scaling from MIMD+SIMD$_4$ to MIMD+SIMD$_8$, but the gain is not that large and the same performance can be achieved by the (CPU-only) MIMD$_8$ bar. As previously mentioned, 372.smithwa is one of the benchmarks for which we performed loop fission; this results in a producer-consumer data relationship between the fissioned GPU kernel and the CPU loop where a significant amount of data is communicated between the CPU and the GPU. At 4 CPU cores, the data is small enough to fit into the GPU's cache and therefore the communication happens on chip. At 8 cores, the cache footprint becomes too large, and the data spills to DRAM. The CPU must then wait a longer latency for the data to be fetched from DRAM, and since the CPU is a latency intolerant architecture, it loses performance.

In chapter 4 we study this heterogeneous producer-consumer relationship in detail, and propose a new optimization to improve perfomrance. 372.smithwa is one of the workloads used to evaluate the new technique that we propose.

3.4.4 Concurrent Kernel Execution

When there are sufficient GPU thread contexts available, and contention does not occur, nested MIMD-SIMD parallelism can result in multiple kernel instances executing on the GPU simultaneously– in other words, GPU spatial utilization is

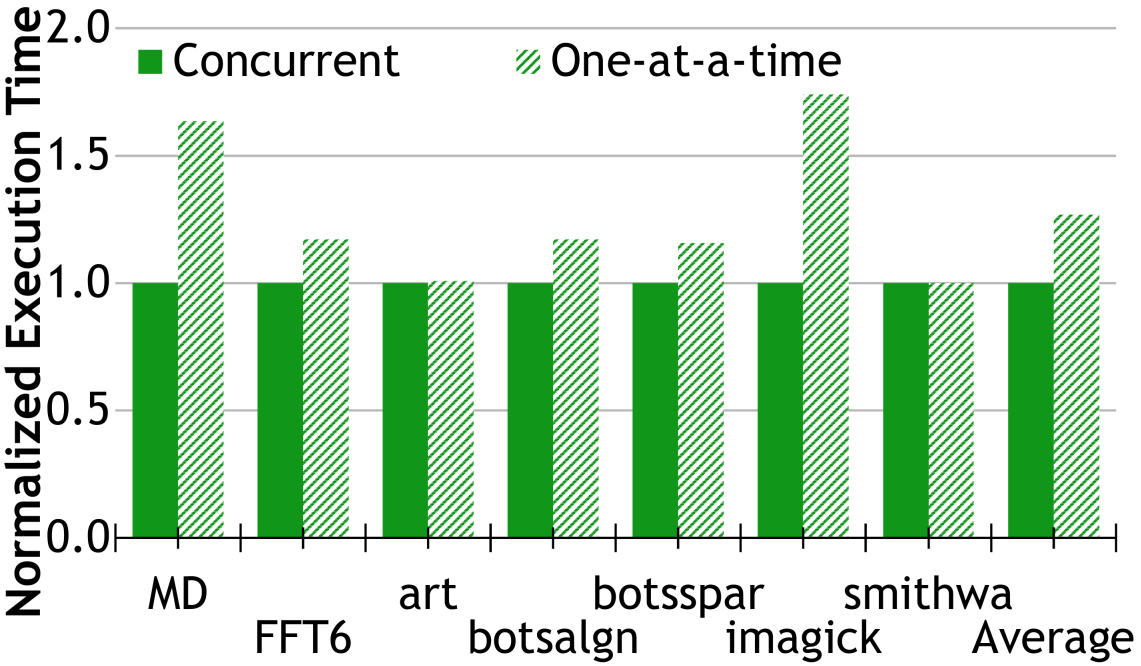


Figure 3.10: Impact of concurrent kernel execution.

increased. In order to evaluate to what extent our technique benefits from increased spatial utilization we re-run the experiments of Section 3.4.1 but turn concurrent kernel scheduling off. Specifically, we rerun "MIMD+SIMD" result, without kernel concurrency. Consequently, if two kernels launched from parallel CPU threads arrive at the GPU simultaneously, we serialize them –*i.e.*, each kernel runs to completion before we allow the next kernel to start. The "One-at-a-time" bars in Figure 3.10 show this result. These bars show execution time normalized to the original "MIMD+SIMD" bars from before, which are labeled as "Concurrent" in this Figure.

Figure 3.10 shows that a lack of concurrent kernel execution degrades performance for MD and 367.imagick significantly. This comes at no surprise since these benchmarks made up the "spatial utilizers" category from Section 3.4.2; a lack of concurrent kernel execution effectively blocks any boost to spatial utilization

that our nested MIMD-SIMD technique could provide. Therefore, for the second category, "temporal utilizers," comprised solely of 372.smithwa, it is similarly unsurprising that one-at-a-time scheduling makes no difference. 372.smithwa has one kernel with an *nthreads* that completely utilizes the GPU on its own, so multiple instances of the kernel will never execute concurrently regardless.

Skipping to the final category from Section 3.4.2 for now, 359.botsspar similarly has a kernel with a very large *nthreads* that will dominate the GPU's thread contexts, and serialize kernel execution. However, it also has two smaller kernels that will be able to concurrently execute at times when the larger kernel is not executing. This results in a slight degradation from one-at-a-time scheduling in Figure 3.10.

Finally, we come to the category of spatial and temporal utilizers, which includes FFT6, 330.art, and 358.botsalgn. These workloads have smaller *nthread*, so there would be space on the GPU for concurrent execution to occur. Indeed, these benchmarks experience a slight degradation from one-at-a-time scheduling, though much less than MD and 367.imagick. This is because these benchmarks have a small f_{SIMD}. The smaller the f_{SIMD}, the less frequently kernels will be launched from each CPU thread, reducing the chances that kernel launches will collide and serialize under one-at-a-time scheduling. Overall, these results show how our technique can benefit the performance of of the workloads we evaluated by boosting spatial utilization of the GPU when those workloads launch GPU kernels from multiple threads at the same time.

3.5 Enabling Low-Latency Launch on Hardware

With a detailed understanding of our technique's performance characteristics in the fully controllable simulation platform, Gem5-gpu, we now set out to show how Nested MIMD-SIMD Parallelization could be practically and gainfully utilized on a real hardware platform. However, one of the major challenges in realizing speedup on real hardware is the high latency penalty for launching GPU kernels on standard platforms.

As we have discussed earlier in this Chapter, a low launch-latency is necessary to gainfully offload small SIMD loops for some OpenMP programs. The drastic performance improvement for the FFT6 and botsalgn benchmarks between Figure 3.6 (4.5 μs) and Figure 3.7 (0.1 μs) showed this fact quantitatively. Standard Kernel launches can have latencies of 100μs or more, depending on the kernel and the system. This is far in excess of the modest 4.5 μs we show in the Figure 3.6. Clearly this would not be suitable for FFT6 and botsalgn.

Large launch latencies are a consequence of the CPU needing to communicate the parameters of a kernel with the GPU through layers of runtime and operating system queues. Figure 3.11 shows this datapath. For traditional, massively parallel, GPGPU programs executed on discrete GPU platforms, this is a relatively small issue. The launch latency is amortized over the long-running kernel, and for discrete GPUs it must travel across a PCI-express bus which adds its own latency. In an integrated CPU-GPU Heterogeneous Microprocessor, however, we eliminate this long I/O latency and communicate through the on-chip cache. Thus, the latency

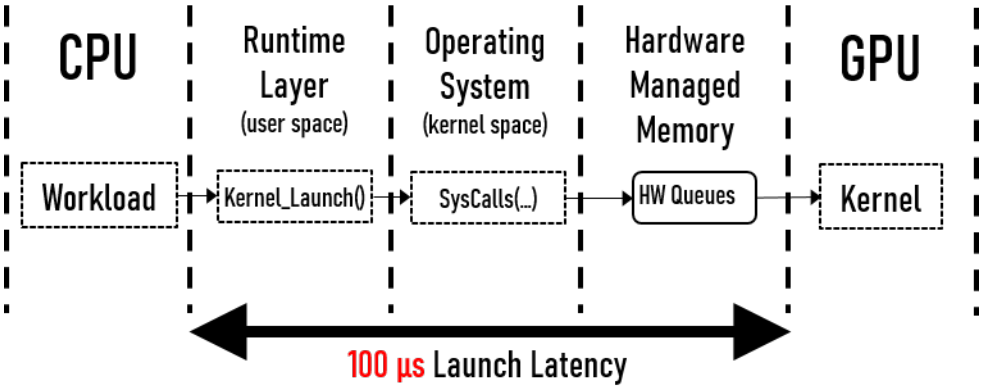


Figure 3.11: Normal kernel launch datapath. The workload running on the CPU calls a kernel launch from the runtime system, which loads parameters into GPU memory via system calls in the operating system layer. The kernel begins executing after 100s of μs

added by the operating system is untenable, and must somehow be eliminated.

To bypass the operating system, we take advantage of modern heterogeneous processors' ability to enforce memory consistency and coherence between the CPU and GPU [27]. This is achieved using C++ atomic memory accesses, which trigger a fence, updating memory locations in both the CPU and GPU's caches. With this capability, we implement a GPU launch daemon and lightweight custom runtime system similar to the "Instant Mode" mechanism proposed in previous work [28].

This entails a special "launch daemon" kernel launched by the main thread ahead of the execution of the MIMD loop we are interested in, incurring the normal launch latency cost once. The launch daemon runs on the GPU for the full duration of the MIMD loop, and can be used many times to execute work on the GPU. Upon launch it enters a loop that atomically polls a communication buffer, waiting for a trigger to inform it that work is available (*i.e.*a kernel launch) or for the signal to exit. The launch daemon kernel contains all of the code of the normal kernel

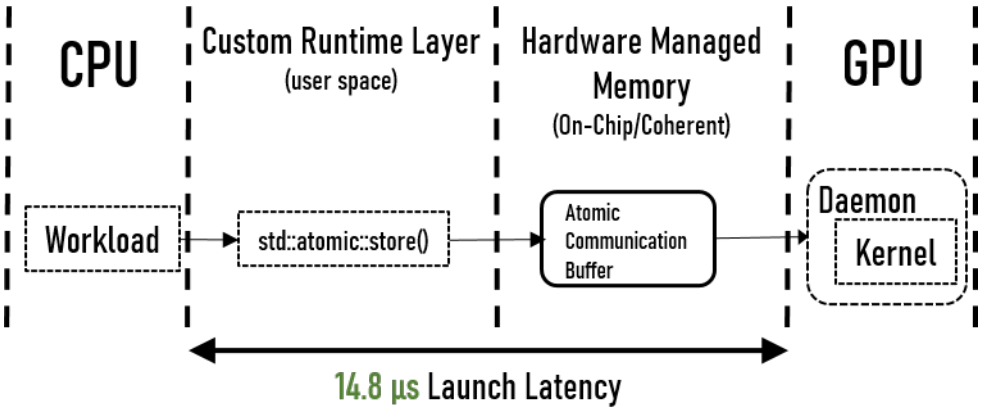


Figure 3.12: Kernel Launch Via Launch Daemon. The workload running on the CPU calls C++ atomics to populate a communication buffer located in coherent on-chip memory. The daemon polls this buffer waiting for parameters to be loaded triggering kernel execution. This datapath bypasses the operating system, and takes 14.8μs on average.

wrapped in the polling loop, executing the correct GPU code with the correct parameters. Figure 3.12 shows the datapath through which the CPU uses this atomic communication buffer to directly write the parameters the kernel needs to execute and then trigger that execution. All of this atomic communication bypasses any operating system calls, software queues, and normal kernel launch runtime layers, significantly reducing launch latency from 100μs down to 14.8μs on average for the workloads we evaluated.

Our ability to implement such a simple workaround to drastically reduce launch latency goes to show that the extremely high cost of launches inherent to normal GPGPU runtime systems is unnecessary and in cases like our technique, detrimental to performance. Our launch daemon is practical and improves performance under real conditions; it is valuable in its own right. However, architects and system designers should be motivated to find a better way for CPUs to communi-

CPU		GPU	
Number of cores	4	Number of SMs	24
Clock rate	3.4-4.0 GHz	Clock rate	350-1150 MHz
ISA	x86+AVX512	ISA	GEN Assembly
		Warp size	32
	32/32 KB	Max warps per SM	7
L1 I/D cache (private)	256 KB	L1/2 cache	N/A
L2 cache (private)		L3 I cache (shared)	768 KB
		L3 D cache (shared)	1.5 MB
Last-Level Cache (shared by CPU and GPU)			8 MB
Main Memory			
Each channel			64-bit, 2.133 GHz, DDR4

Table 3.4: Hardware parameters for the physical Intel Core i7-6700 with an HD Graphics 530 integrated GPU. Although the CPU cores have AVX512 instructions, we do not exploit them in our experiments.

cate with GPUs for such a fundamentally important mechanism like kernel launch. With an improvement to their integration in this space, GPUs could be more akin to "co-processors" to the CPU, complementing their increasing importance in the computing ecosystem.

3.6 Hardware Study Methodology

Our physical machine study is performed on an Intel Core i7-6700 whose parameters are listed in Table 3.4. This chip is largely similar in architecture to the one simulated in the above study, with the major distinction being a total of 5,376 concurrent threads possible in the GPU, considerably fewer than the 24.5K threads in the simulated GPU, and a shared last-level cache between the GPU and CPU.

For the simulation study, our GPU kernels were implemented in CUDA. But the integrated Intel GPU in our physical machine only works with OpenCL. Hence, we ported all of our benchmark kernels to OpenCL 2.0. Unfortunately, we found the

kernel launch latency using the standard software queue interface in our OpenCL implementation incurs an overhead of at least 100μs. Because the performance of our benchmarks is highly sensitive to the launch latency, we had to address this issue before we could achieve performance gains on the physical machine.

With the GPU launch daemon we describe in Section 3.5, the launch latency reduces from 100μs down to 14.8μs on average. This is a significant improvement, but is still higher than the 4.5μs baseline (and 0.1μs aggressive) launch latency assumed for the simulation study. Normal launch latencies were measured using built-in OpenCL event statistics. Latencies for the low-latency launch mechanism were measured by executing a "Null" kernel many times, and measuring how long full execution took, including the execution of the launch daemon. Essentially, the workload would load parameters to trigger the launch daemon and the launch daemon would enter its execution phase, but would then immediately return to polling the communication buffer without executing any of the "payload" code. Each workload had unique numbers and sizes of parameters, and thus their launch overhead would vary.

3.6.1 Benchmarks

The same benchmarks listed in Table 3.2 are used for our physical machine study. We ported these benchmarks' CUDA kernels to OpenCL, added our launch daemon mechanism, and compiled the resulting C++/OpenCL code with GCC and Intel's OpenCL SDK using the highest level of optimization possible.

Unfortunately, we had trouble porting 358.botsalgn. This benchmark is the most sensitive to launch latency due to the small size of its GPU kernels. Although we were able to get performance gains on 358.botsalgn with a 0.1μs launch latency (see Section 3.4), the 14.8μs launch latency on our physical machine resulted in significant performance degradation. We were unable to get performance gains on the physical machine for 358.botsalgn by simply translating our CUDA implementation to OpenCL.

Upon closer examination of the 358.botsalgn code, we found a software transformation that can mitigate high launch latency. In each outer loop iteration of 358.botsalgn, multiple kernels are launched to the GPU. These kernels are actually *independent* and can execute concurrently. (This is not generally true across our benchmarks, but occurs in 358.botsalgn). In the CUDA implementation for our simulator, we only exploit concurrent kernels across CPU threads; each CPU thread still executes all of its kernels serially which exposes the launch latency at every kernel launch. We modified the 358.botsalgn code so that each CPU thread launches multiple independent kernels concurrently, stalling only after all launched kernels complete. This hides much of the launch latency and provides even more kernel-level parallelism for the GPU to exploit. With this software optimization, the single-loop SIMD parallelization scheme for 358.botsalgn speeds up by 2.10x, and our nested MIMD-SIMD parallelization scheme speeds up by 2.32x on the physical machine. In our performance evaluation below, we assume 358.botsalgn uses this software transformation.

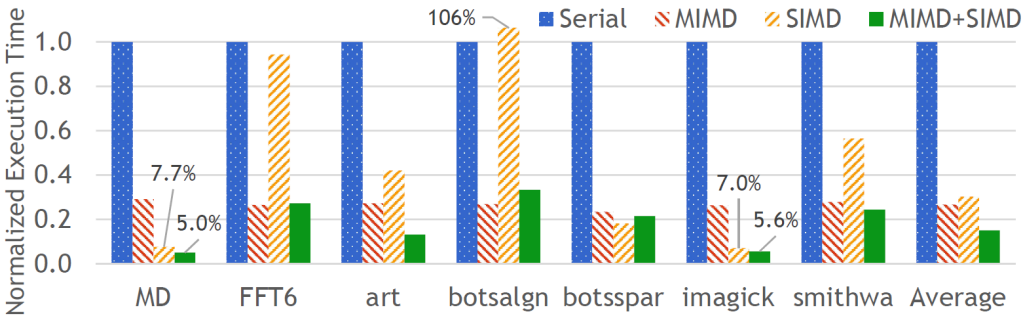


Figure 3.13: Normalized execution time for no parallelization (Serial), CPU parallelization (MIMD), single-loop SIMD parallelization (SIMD), and nested MIMD-SIMD parallelization (MIMD+SIMD) on the physical machine.

3.7 Hardware Study Results

Figure 3.13 presents the results from our physical machine study. In the figure, the "Serial," "MIMD," "SIMD," and "MIMD+SIMD" bars are identical to the corresponding bars from Figure 3.6, except they report the execution time on the physical machine rather than the simulator. All execution times are normalized to each benchmark's "Serial" bars.

Comparing the "MIMD" and "Serial" bars in Figure 3.13, we see the same behavior for CPU parallelization as in the simulator study: all of the "MIMD" bars are near 0.25, providing a speedup of 3.76x (compared to 3.87x on the simulator) averaged across all the benchmarks. Thus, CPU parallelization is essentially as effective on the real hardware as it is on the simulator.

Figure 3.13 also shows single-loop SIMD parallelization (the "SIMD" bars) performs well on the physical machine, too. Like our simulator results, the performance shown by the "SIMD" bars in Figure 3.13 varies across the benchmarks. These benchmarks show a similar pattern of results. In particular, since we have

much less control over the launch latency of the hardware, the benchmarks sensitive to launch latency (ff6 and 358.botsalgn) do worst than the others, even accounting for the smaller sized GPU. Overall, however, single-loop SIMD parallelization achieves a 3.30x speedup over serial execution.

Similarly, the "MIMD+SIMD" bars in Figure 3.13 show our parallelization scheme performs well on the physical machine, but not as well as on the simulator. Averaged across all benchmarks a 8.67x speedup is achieved. Moreover, our parallelization scheme outperforms CPU-only parallelization by 2.4x on average, and it outperforms GPU-only parallelization by 2.3x on average. Except for 358.botsalgn where CPU-only parallelization is slightly better, and for 359.botsalgn where GPU-only parallelization is slightly better, nested MIMD-SIMD parallelization is the best scheme. Averaged across all the benchmarks, nested MIMD-SIMD parallelization beats the next-best scheme by 1.23x. But on the simulator, the speedup over serial is 16.0x (instead of 8.67x) and the speedup over the next-best scheme is 1.47x (instead of 1.23x).

One reason why the simulator results are better is because the GPU in our physical machine supports less parallelism than the GPU in our simulator (5,376 threads versus 24,576 threads). At the same time, the physical machine's CPU runs faster (between 3.4 GHz to 4.0 GHz) compared to the simulator's CPU (2.6 GHz). Both tend to reduce the relative gain for off-loading computations from the CPU onto the GPU. Also, the physical machine's higher launch latency is a contributor, especially for benchmarks with a small W_{SIMD}. This is the reason for the poorer performance in 358.botsalgn on the physical machine compared to the simulator.

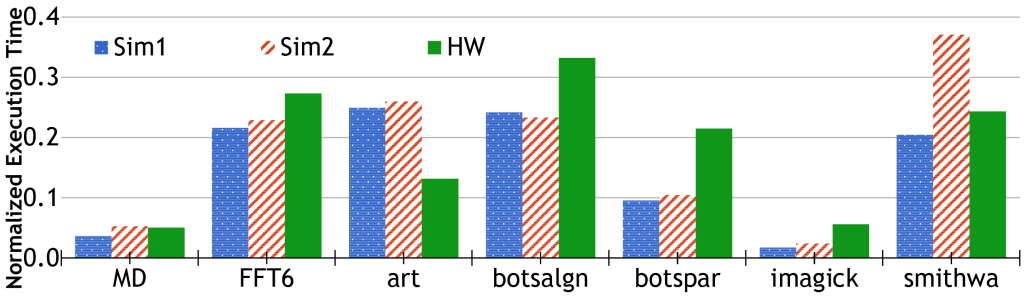


Figure 3.14: Normalized execution time for the MIMD-SIMD parallelization technique running on two configurations of the simulator and the physical machine. The first simulator configuration follows Table 3.1 while the second is closer to the physical machine's configuration.

(Even with a software optimization for 358.botsalgn, the "SIMD" bar is 6% worse than the "Serial" bar in Figure 3.13). Lastly, the simulator employs more total last-level cache (10 MB) compared to the physical machine (8 MB); and, the simulator's cache memories, including the GPU's caches, all run at 2 GHz. On the physical machine, the GPU shared L3 cache runs at the speed of the GPU which ranges from 350 MHz to 1150 MHz. This increases the time that the CPU must wait for GPU results to be transferred to the CPU after a GPU kernel has completed.

Figure 3.13 demonstrates our parallelization scheme can provide real performance gains on actual systems. And, taken together, Figures 3.6/3.7 and 3.13 show our approach can be effective across different platforms with varying hardware capabilities.

3.7.1 Simulator Validation

In this section, we compare our physical machine results against our simulation results from Section 3.4 in detail. As discussed in Section 3.7, our simulations result in a higher speedup for our nested MIMD-SIMD parallelization technique than do

the results from the physical machine. In Figure 3.14 we show the "MIMD+SIMD" bars from the simulator results ("Sim1" in the figure) and the hardware results ("HW" in the figure), side by side. For FFT6 and 358.botsalgn the "Sim1" bars use the low-latency launch results from Figure 3.7. The bars retain their original normalization to their respective "Serial" bars from their original figures, so rather than a comparison of absolute execution times, we are comparing the relative gains achieved on the two platforms. In almost every benchmark, the simulator shows a larger gain (smaller normalized execution time) compared to the physical machine. Averaged across all the benchmarks, the difference between the Sim1 and HW bars is 36%.

In order to close try and gap between the simulation results and the hardware measurements, we adjusted the simulation parameters to more closely match (where possible) the physical machine's specification. Firstly, we reconfigured the simulator to have the same number of total thread-contexts as the physical machine, 5,376. Second, we increased the CPU's clock rate to 3.4GHz, and decreased the speed of all the caches down to 1 GHz. Lastly, we reduce the CPU's private L2s and the GPU's shared L2 to 1.5 MB each, totally 7.5 MB, in order to more closely match the physical machine's 8 MB LLC. Unfortunately, there were differences between the simulator configuration and the hardware's specification that we could not correct. Namely, that the physical machine has a shared LLC through which the CPU and GPU's L2s may pass data. The simulator does not support a shared cache above the CPU and GPU's L2s in the cache hierarchy, though it does enforce coherence between the two. As we mention in Section 3.6 we were unable to find documentation on the physical

machine's coherence protocol, so there are likely differences in coherence enforcement between the simulator and the physical machine that will affect performance.

The bars labeled "Sim2" in Figure 3.14 report the normalized execution time for nested MIMD-SIMD parallelization on the reconfigured simulator. For MD, FFT6, 359.botsspar, and 367.imagick, the Sim2 bars more closely matched the HW bars than did the Sim1 bars, *i.e.*the new configuration improved agreement between the simulator and the physical machine. For 330.art and 358.botsalgn, the Sim2 bars diverged slightly further away from the HW bars, but for 372.smithwa, they diverged significantly more. One possible reason for the more significant disagreement of the 372.smithwa result is the increased GPU-to-CPU data traffic resultant of the loop fission transformation we use to extract parallelism for 372.smithwa, and the large size of this data. It is however difficult to tell without a concrete understanding of the hardware's coherence protocol. Overall, the difference between Sim2 versus HW improved slightly, 32% down from 36% difference between Sim1 and HW. To improve the agreement of the simulation and the physical machine further, it would be necessary to address the fundamental functional limitations of the gem5-gpu simulator that we outlined above.

3.8 Related Work

Research into the performance benefits of heterogeneous microprocessors has been conducted before. Daga *et al.* [29] and Spafford *et al.* [30] evaluate AMD's Fusion architecture [5]. They both examine how CPU-GPU integration in the Fusion

architecture addresses the communication bottlenecks faced by discrete systems [3]. However, they only consider traditional GPU workloads, such as the SHOC [1] and HPC Challenge [31] benchmark suites. They do not look for new and potentially more complex codes that become enabled by CPU-GPU integration.

The work by Arora *et al.* [32] recognizes that heterogeneous microprocessors are capable of handling a wider range of programs, so they consider a mix of traditional GPU workloads (Rodinia [12]) as well as benchmarks from the SPEC CPU 2000/2006 suites [33]. The latter contain more irregular codes exhibiting a variety of data parallel loops, which is also the focus of this portion of the thesis. However, Arora is concerned mainly with CPU performance, observing that CPUs will execute much more serial code as parallel loops are off-loaded onto the GPU. This chapter and the research it is based on in contrast concerns itself with the performance of both types of cores, *CPU and GPU*, with a new parallelization technique that leverages the different cores in concert.

Besides studying the benefits of heterogeneous microprocessors, several research efforts have developed parallelization techniques to make more effective use of both CPU and GPU cores simultaneously. The research presented in this chapter was published in ACM Transactions on Architecture and Code Optimization (TACO, 2019) [10]. This work in turn was based on our own prior work [34] that first introduced the idea of mapping nested parallelism to integrated CPU-GPU chips. Compared to the first paper [34], this chapter presents a number of additional contributions made in the TACO paper [10]. The second phase of research added loop fission for extracting SIMD portions from serial loops, the experimental

evaluation (using both the cycle-accurate simulator and the physical machine) was completely new, and the low-latency launch system used for the physical machine study.

Preceeding our work, several researchers have proposed scheduling iterations from the same data parallel loop across both CPU and GPU cores [35–37]. As we showed in this chapter, however, such *homogeneous parallelism* from a single thread cannot keep the GPU (let alone both the CPU *and* GPU) fully utilized given the smaller loops that can be extracted from more complex loops, a lack in both the spatial and temporal dimensions of utilization. Exploiting homogeneous parallelism from a single loop only also means that the other cores of the CPU go unutilized. Heterogeneous parallelism on the other hand uses multiple CPU cores simultaneously. Thus our approach exposes greater parallelism for both the GPU and CPU.

Wende *et al.* [38] demonstrate a CPU-GPU parallelization scheme on the GLAT molecular thermodynamics code. Similar to our work, their approach extracts parallelism from different loops for execution on CPU and GPU cores. But whereas they identify distributed parallel loops, we exploit nested parallel loops. More importantly, their study is specific to a single benchmark only. In contrast, we look at a greater number of benchmarks. We also argue that our approach will be relevant for many irregular codes parallelized using OpenMP, which encompasses a very large number of programs.

Our work is also related to dynamic parallelism. Modern GPUs allow a threads within a GPU kernel to launch another GPU kernel [39]. Researchers have devel-

oped several parallelization schemes around this idea of internal kernel launches, including dynamic thread block launch (DTBL) [7], nested parallelism in CUDA (CUDA-NP) [40], and lazy nested parallelism (LazyNP) [39]. In DTBL, the child threads are dynamic instances of the original kernel launched by the CPU–*i.e.*, the parallelism occurs within recursive code. In CUDA-NP and LazyNP, the child threads belong to parallel loops nested within the original kernel launched by the CPU. Like our parallelization scheme, CUDA-NP and LazyNP also exploit nested parallelism. However, DTBL, CUDA-NP, and LazyNP only schedule loops on the GPU. This *Homogeneous* parallelism only takes advantage of the GPU, leaving the CPU under-utilized. Our nested MIMD-SIMD parallelization technique also runs on the CPU, supporting MIMD loops that would otherwise perform poorly on the GPU, or code with task-level parallelism which cannot run on GPUs.

In addition to these parallelization schemes, there has also been work that focuses primarily on new kernel launch mechanism. Our technique requires multi-kernel launch, or in other words: kernel launches from multiple threads. Prior work has investigated mechanisms for off-loading multiple simultaneous kernels onto the GPU [8, 9], either from a single CPU thread or from multiple CPU threads. While our techniques rely on these mechanisms, our main focus is on identifying the parallel idioms–*e.g.*, nested parallel loops–that give rise to multiple simultaneous kernels. Previous research has focused more on the launch mechanisms themselves.

Finally, instead of using the GPU to accelerate regular inner loops as we propose in our work, it is also possible to do so via SIMD instruction extensions available in today's CPUs [41], like SSE and AVX. SIMD instruction extensions have lower

overhead than GPU kernel launches, so they do not incur the off-loading costs that exist in our techniques. However, SIMD instruction extensions cannot exploit as much parallelism due to the much narrower datapath in the CPU compared to the massively parallel (even in integrated chips, by comparison) GPU. It is possible to combine nested MIMD-SIMD parallelization with SIMD instruction extensions, using the latter for smaller inner loops that are the most sensitive to kernel launch overhead. This could provide even higher performance than our techniques applied alone. In this research, we focused on understanding nested MIMD-SIMD parallelization by itself. We leave its combination with SIMD instruction extensions for future work.

3.9 Conclusions

This portion of the thesis presented *Nested MIMD-SIMD Parallelization*, a new parallelization technique that leverages both the CPU and GPU in Heterogeneous integrated CPU-GPU microprocessors. Given the increasing ubiquity of these types of chips in low-power devices like mobile phones, we argue that traditional parallelization paradigms are not sufficient to fully utilize all of the hardware features provided by these chips. This includes both the heterogeneity of the cores and the low-cost communication between the cores afforded to them by an on-chip coherent cache.

We show that seven OpenMP benchmarks contain a mix of MIMD parallelism suitable for the multi-core CPU and SIMD parallelism suitable for the GPU, with

the SIMD regions nested within the MIMD regions. In some cases we use "loop fission" to extract SIMD parallel regions from non-SIMD regions, creating two separate loops. In all cases we transform the benchmarks' SIMD regions into GPU kernels, and with low-latency kernel launch concurrently launch multiple instances of them on the GPU from multiple CPU threads. In this way we increase spatial and temporal utilization of the GPU, while at the same time utilizing the CPU for computations when the GPU is not executing.

We conducted an in-depth evaluation of these seven benchmarks on a cycle accurate simulator with an aggressively sized GPU, as well as a real physical machine with a more modest sized GPU. For the real machine we implemented a low-latency launch system that enabled us to gainfully execute fine-grained SIMD loops on the GPU.

On the simulator our results show that exploiting nested MIMD-SIMD parallelism provides a 16.1x speedup over serial execution. On the physical machine, the gain of this technique drops slightly, to 8.81x speedup. The drop is primarily due to the modest size of the real hardware compared to the simulated chip. Our parallelization scheme beats CPU-only parallelization by 4.13x and 2.40x on the simulator and physical machine, respectively. Compared against the next-best scheme (either CPU-only or GPU-only parallelization) across all our workloads, nested MIMD-SIMD parallelization provides speedups of 1.46x and 1.27x on the simulator and physical machine, respectively. Thus we demonstrate that this technique is viable for a range of heterogeneous microprocessors with varying capabilities.

Chapter 4: Pipelined CPU-GPU Scheduling for Caches

One of the most potentially salient features of integrated heterogeneous microprocessors is the coherent cache system. In typical multi-core CPU computing, the cache system serves as an important sharing resource between the cores who may access the same data, with the hardware coherence system facilitating this sharing by taking the burden of managing coherence from the programmer.

Contemporary GPU hardware has some coherence mechanisms, though due to the lack of temporal locality in GPUs, it often has limited usefulness. GPUs can often exacerbate the challenges in designing scalable coherence systems as part of large server-scale compute systems [42].

Heterogeneous chips, found more often than not in mobile devices, are architected at a smaller scale than chips meant for server or workstation use – containing fewer cores of both types and thus less cache. At this smaller scale, tightly integrated coherence is quite practical. Since the integration of GPUs onto heterogeneous chips, research has been done into how to include the GPU in the cache system with some success. Modern chips such as the Intel Core series and AMD Fusion APUs, for example, provide cache coherence between the CPU and GPU cores [24, 42, 43].

As we showed in Chapter 3 the reduced overhead of launching work to the

GPU that integration provides more work will find its way from the CPU to the GPU. In cases where only a portion of the work gets GPU acceleration, this leaves the CPU and GPU in the position where they must collaborate. Indeed, benchmark suites like Chai and Hetero-Mark present collaboration models where CPUs and GPUs work together in a variety of ways [13, 44]. Even traditional GPU benchmark suites like Rodinia present workloads where the two types of cores collaborate, migrating computation from the CPU to the GPU and back. Data dependencies in this computation migration cause Producer-Consumer data sharing, where data is passed between stages of computation that have migrated to the CPU or GPU [12, 16].

The status quo of the computation migration and resulting producer-consumer sharing in many of these workloads is for the producer to run its work to completion, followed by the consumer finishing out the computation, in-effect: serial scheduling. In a system without a coherent shared cache, this works well enough. But in a system with a coherent LLC this execution pattern results in redundant and wasted DRAM accesses when the cache footprint of the producer is larger than the size of the LLC, which is often the case. The producer loads the shared data into the cache system, and then evicts the same shared data before completion. Then when the consumer accesses the shared data, it must read it from main memory rather than from the LLC. This extra DRAM access wastes energy, and increases response latency slowing down latency intolerant CPU consumers.

To exploit the coherent LLC of heterogeneous microprocessors and retain shared data "in-place" in the cache system, we propose *Pipelined CPU-GPU Schedul-*

ing for Caches. This locality transformation schedules the producer and consumer stages concurrently to increase temporal reuse of shared data. We can exploit the fact that while GPU kernels are massively parallel, they still typically access a dataset in a linear streaming fashion with the first thread-blocks executed consuming and producing at the beginning of the dataset, and the last thread-blocks executed at the end of the dataset. Since the same is true for the iterations of a CPU loop, we can overlap CPU and GPU execution, creating a software pipeline in which a consumer is fed the data it needs as it is produced, on-chip and in-place.

Previous work has exploited pipelined CPU-GPU for increased parallelism [45]. But this kind of pipeline also allows us to control how far ahead of the consumer the producer runs. By tuning this "run-ahead" distance to be small enough, we can fit the data shared by the producer and consumer into the cache system of the heterogeneous microprocessor. Thus, the communication between the CPU and GPU happens on-chip, improving performance, and reducing energy spent on DRAM accesses - increasing overall efficiency.

The contributions of this work are:

- A qualitative analysis of how Producer-Consumer sharing utilizes the cache system under both naive and pipelined schedules.
- A realistic control scheme using industry standard protocols.
- A simulator study showing that our technique improves performance and power for seven workloads, on average reducing the number of DRAM accesses by 30.4% , execution time by 26.84% , and saving 27.4% total DRAM

energy.

The rest of this chapter is organized as follows: Section 4.1 will heterogeneous cache coherence, heterogeneous applications, and explore naive scheduling, illustrating how it fails to exploit the coherent LLC. We then present how pipeline scheduling corrects this deficiency. Section 4.2 describes the experimental methodology used for our quantitative simulation study. Section 4.3 presents the results of said study. Section 4.4 discusses related work. Finally, Section 4.5 concludes the chapter.

4.1 Background

4.1.1 Heterogeneous Cache Coherence

Homogeneous multiprocessors (those containing no GPU cores, only CPU cores) have enjoyed hardware managed coherence for many years now [42]. This coherence mechanism is a well understood, highly performant, and very convenient abstraction to a programmer. Most programmers need not worry about managing coherence and can write efficient code without much hassle. Heterogeneous cache coherence and GPU code are a different story.

In the early days of GPGPU coding, with discrete GPUs, the GPU and CPU had separate address spaces and any data had to be explicitly copied between a CPU pointer and a GPU pointer, a significant burden to the programmer. Modern GPUs implement shared virtual memory (SVM), enabling "zero-copy" buffer transfers, through page-locked memory allocated specially at runtime [2].

Hardware mechanisms that fully implement the kind coherence found in homo-

geneous systems are difficult to achieve in heterogeneous systems. Firstly, keeping the many private caches in a GPU coherent is a problem of scale. The more coherent cores in the protocol, the more expensive it is to implement. This can be mitigated by keeping only a few, or even just one private GPU cache coherent and requiring that the GPU writes through this private cache rather than keeping dirty data that only writes back on eviction and needs to be probed and invalidated. Secondly, difficulty stems from the divergent access patterns of the CPU and GPU. GPUs tend to have and extremely high throughput and as a result access significantly more memory than the CPU in the same amount of time. If the CPU and GPU share an LLC the GPU will completely thrash the CPU's data, leaving the CPU to access that data from main memory - something for which the latency intolerant CPU will suffer.

With this access heterogeneity in mind, proposals for heterogeneous coherence in the literature suggest ways to limit the GPU's access to shared caches (*i.e.*the LLC) [42, 46, 47]. Though implementation details of heterogeneous coherence protocols are sparse, due to intellectual property concerns, these mechanisms certainly exist. For instance, AMD and Intel both implement coherence, including system-wide atomic access on their integrated CPU-GPU microprocessors [5, 24]. LLCs in these heterogeneous coherence protocols are generally non-inclusive, and avoidance strategies like dis-allowing the GPU and CPU to cache the same blocks or allowing the GPU to bypass the LLC entirely, are used to manage the different coherence and consistency models of heterogeneous systems [42, 46, 47]. This strategy would the the GPU from thrashing the LLC by limiting data sharing through the cache

system. It would also however, cause producer-consumer sharing between the GPU and CPU to occur entirely off-chip, an undesirable result. For the evaluation of our technique, which thoughtfully schedules work on the GPU and CPU in order to avoid interference that is detrimental to performance, the coherence protocol allows the CPU and GPU to share the LLC and manage the data therin.

Caching Considerations and Speculative DRAM Access. Another avoidance method in the context of CPU-GPU coherence is victim caching [48]. An victim-cache LLC will only store data when it is written back (evicted) or written-through from the cores. To avoid polluting the LLC a GPU cache may choose not to write back clean cache lines. Considered in isolation this should be an effective strategy considering the GPU's limited reuse. However, avoiding using the LLC in this manner may incur an LLC miss for the CPU should it need that cache line later, and thus a performance degradation if the cache line comes from DRAM. Should the CPU attempt to access the cache line before the line is evicted from the GPU's L2 the data can be snooped from there instead of DRAM for a reduced latency penalty. However, the LLC may speculatively access the DRAM in parallel to probing the other caches in order to reduce access time for the latency sensitive CPU. While normally effective at reducing latency, in such a case the speculative access is wasted, unnecessarily draining energy [49]. Such wasted speculative DRAM accesses could be avoided by ensuring LLC hits. Read-only data shared between the cores or even write-after-read data could be loaded into the LLC the first time it is accessed from DRAM. Subsequent accesses, assuming no capacity evictions, would be LLC hits.

Speculative DRAM accesses in combination with exclusivity and a write-back

policy can also affect the data path in the other direction, CPU to GPU. Even if the CPU's L2 cache will eventually write back a resident line to the LLC, with an exclusive LLC an access from the GPU would incur a miss and speculative DRAM access. In this case the inefficiency of the missed speculative DRAM access is more pronounced - the latency tolerant GPU would not have cared about getting the block faster. In this case the combination of an exclusive cache and speculative DRAM access do disservice to the memory system. L2 cache capacity, as well as associativity may have an effect on this phenomenon. By reducing total or set capacity, the L2 would be more likely to evict the shared block to the LLC before the GPU reads it, at which time it would be an LLC hit. A final more drastic solution might be to treat accesses from the GPU and CPU differently, not speculatively accessing DRAM for the GPU. This solution however would likely require adding significantly more state to the directory controller.

Rather than avoiding the shared cache because of the inherent data-access heterogeneity of GPGPU and CPU codes, in section 4.1.4 we will demonstrate how to thoughtfully execute them concurrently such that they benefit from cache-coherence, while not stepping on each other's toes. In the following section we discuss GPGPU workloads and their data-sharing characteristics.

4.1.2 Heterogeneous Producer-Consumer Sharing

As GPGPU computing increasingly became a more important tool for many fields interested in high-performance computing, to evaluate the performance of the

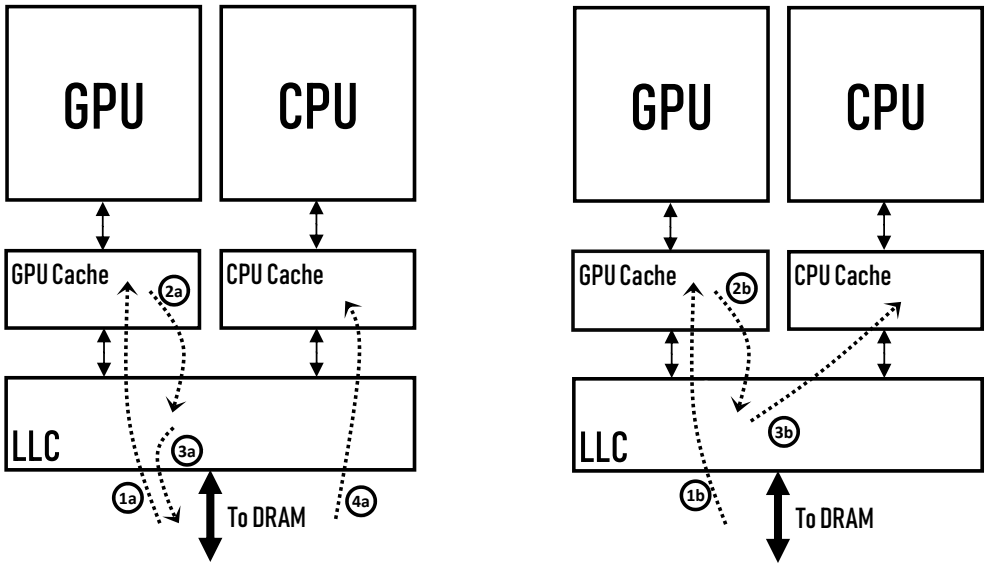
evolving architecture, GPU benchmark suites such as Rodinia [12] and Parboil [16] came into being. Developed at a time when the CPU and GPU were more loosely coupled, benchmarks from these suites primarily evaluate the raw throughput of the GPU. Though they place less importance on collaboration between the two types of core, they do often include CPU work or I/O done by the CPU that causes data migration between the CPU and GPU at a course granularity. In other words: as computation migrates from one core to the other the CPU or GPU produces data that the other core then consumes – producer-consumer data sharing.

Recently, several benchmark suites which take advantage of modern GPU memory features found in integrated GPGPU systems, such as system-wide coherence and atomic memory accesses, have been released. Hetero-Mark [13], Hetero-Sync [50], and Chai [44] all recognize the importance of collaboration between the CPU and GPU in a more tightly coupled system and many of these benchmarks also exhibit computation migration and Producer-Consumer data sharing.

The common thread between both the older and newer cohorts of GPGPU benchmarks is a lack of care for how the shared data gets from the producer to the consumer when the computation migrates. Ideally, in a integrated GPU system with a shared Last-Level-Cache (LLC), data would efficiently migrate on-chip. However, these applications schedule the computation migration naively: the producer runs to completion followed by the consumer. As data-footprint of most GPGPU kernels far outstrips the size of the cache system, the produced data spills out to DRAM before it can be consumed.

```
#define TPB 256 // Threads-per-block == Block Size

__global__ void GPU_Producer(int *A) {
    int i = blockSize() * blockId() + threadId();
    A[i] = ...; //Write to A
}

void CPU_Consumer(int *A, int iters) {
    for ( int i = 0; i < iters; i++ ) {
        ... = A[i]; //Read from A
    }
}

void main() {
    //Total GPU Threads == CPU Iterations
    int nThreads = 1048576;

    //Number of GPU threadblocks
    int nBlocks = nThreads/TPB;

    //Array of size 4MB
    int * A = malloc(nThreads * sizeof(int));

    //Launch producer kernel to GPU
    GPU_Producer<<<nBlocks, TPB>>>(A);

    //Blocks on GPU kernel completion
    deviceSynchronize();

    CPU_Consumer(A);
}
```

Figure 4.1: Producer Consumer Sharing Code Example



(a) Shared data evicted from LLC before rereference. (b) Shared data Hits in LLC at rereference.

Figure 4.2: Shared Data Communication Pattern

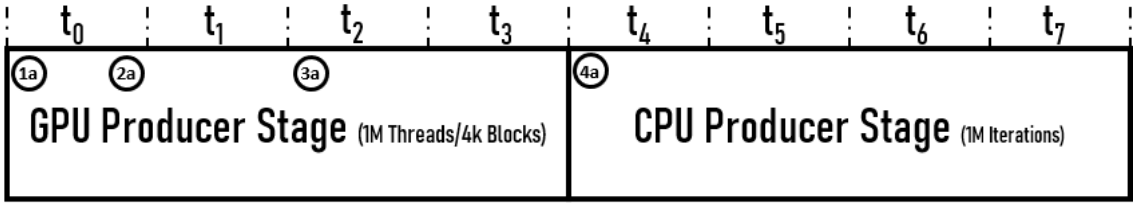


Figure 4.3: Naive Serial Scheduling

4.1.3 Naive Scheduling

Figure 4.1 shows an example of an application exhibiting producer-consumer sharing. The *GPU_Producer* kernel writes an integer array A, and subsequently the *CPU_Consumer* function reads it. The number of threads ($nThreads$) in the kernel is 1,048,576 (1M), and writes one integer per thread. It has a total cache footprint of 4 MB. The loop inside *CPU_Consumer* has the same number of iterations as the GPU has threads, reading the same 4 MB of data.

In this example the producer is naively scheduled to execute after the producer completes its work, demonstrating worst-case temporal-reuse. Figures 4.3 and 4.2a show the consequences of this naive scheduling in a system with a 2 MB shared LLC with a Least-Recently Used replacement policy.

During time t_0 the kernel executes the first 1024 thread-blocks of the computation. ①ₐ represents the compulsory miss that reads a cache-block **B** from the DRAM into the GPU's private cache. Next, at ②ₐ, the kernel writes the data through to the LLC. At this point the producer has finished with B, and the consumer may now read it. However, the GPU continues to execute and during time t_2 the 2 MB LLC reaches capacity **B** is evicted at ③ₐ before the consumer stage can read it. This pattern of evictions and compulsory repeats for all the blocks until the CPU consumer stage begins executing at time t_4. Here, ④ₐ represents the DRAM access needed to fetch the block that was evicted at ③ₐ. Not only is this a redundant read of the same cache-line **B** from DRAM a waste of energy, but for the latency sensitive CPU it decreases performance; the CPU wastes cycles stalling for memory to arrive from DRAM, rather than from the lower latency on-chip LLC.

4.1.4 Locality Aware Scheduling

Let us now consider a solution to what is effectively a capacity miss problem. We could of course increase the size of the LLC to accommodate the entire cache footprint of the GPU kernel. This way, when the GPU completes execution and the computation migrates to the CPU, the data the CPU consumer needs is still on chip. Obviously this unsophisticated solution is not the right answer, but it leads us to consider a better solution: scheduling the consumer to execute as data becomes available for it, before the LLC reaches capacity and must evict the desired data.

We can accomplish this by controlling at a finer granularity how many thread-blocks execute and synchronizing them with their associated CPU loop iterations to keep the temporal reuse-distance of the shared data small. With this finer granularity scheduling, a producer

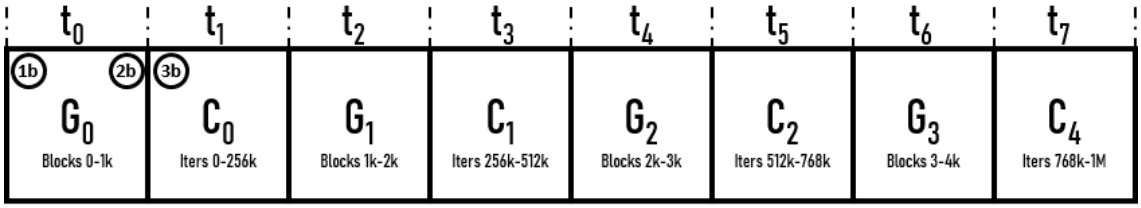


Figure 4.4: Locality Aware Fine-grained Serial Scheduling

runs ahead of a consumer by a certain distance. This "run-ahead distance" (RAD) affects the amount of data that a producer generates before a consumer begins to use that data, and the amount of data the consumer is allowed to use. With a sufficiently small run-ahead distance producer-consumer data migrates efficiently on-chip. In figures 4.4 and 4.5 the same producer-consumer pair are split into subsets according to the RAD. The blocks labeled $G_{0\text{-}3}$ and $C_{0\text{-}3}$ are the subset of GPU thread-blocks that produce data and CPU iterations that consume that same data, respectively. These form a software pipeline with the producer stage's output feeding into the consumer stage's input.

First Pass: Locality Aware Serial Pipeline Scheduling

With the stages divided to reduce the size of their cache footprints, a first-pass scheduling method is to execute the producer and consumer sections that consume the same data one after the other, switching back and forth between the two to fully process all cache-lines before they are evicted from the cache. Figures 4.4 and 4.2b show the effects of this method for the same code example in figure

2.1.

The RAD is chosen such that all of the data from both the producer and consumer fit in the LLC. The footprints per-threadblock of each stage can be calculated as the sum of both the shared data and exclusive data together:

$$F_{stage} = S_{exclusive} + S_{Shared} \tag{4.1}$$

Then calculating the footprint as a function of RAD:

$$F_{stage}(RAD) = F_{stage} * RAD \tag{4.2}$$

For the RAD in figure 4.4, 1024 blocks, the footprints of each subdivision $G_{0\text{-}3}$ and $C_{0\text{-}3}$ of the stages are 1 MB each.

For Locality-Aware Serial Pipeline Scheduling, in order for no capacity evictions to occur the footprint from both the producer and consumer must be considered. The simple example in figure 2.1 has no exclusive data. If there were other data arrays accessed by either stage, they would need to be accounted for as well, and the RAD

would be sized for the larger of the two footprints:

$$F_{SerialPipe} = \qquad Max(F_{Producer}, F_{Consumer}) \qquad (4.3)$$

Using these equations, with an run-ahead-distance of 1024 thread-blocks (equaling 256*1024 threads), we can calculate that in figure 4.4, 1024 GPU thread-blocks (256k threads) and 256k CPU iterations together have a cache footprint of 1MB. Equaling to half the size of the 2MB LLC.

With this RAD, at time t_0, the GPU executes the subset of thread-blocks G_0 equal to the RAD. As before, ①b represents the compulsory miss, and ②b the write-through from the GPU's private cache to the LLC. Next, at time t_1 things change: the data the CPU must read has not yet been evicted and ③b represents an LLC **hit** by the CPU. The data stayed on-chip throughout the time it was active, and can now that it has been consumed, can be freely evicted. Not only was a DRAM access and its associated energy drain saved, but because the CPU is latency sensitive the CPU consumer stage receives an execution time reduction owing to the lower latency of access from

the LLC than from main memory. This execution time reduction compounds with the access energy savings, reducing the total background energy of the main memory.

In the reverse case of a CPU producer with a GPU consumer, performance characteristics are different. The latency tolerant GPU would not receive an execution time reduction from the on-chip data migration. Instead, the benefit of keeping data on-chip would be solely DRAM access energy savings.

Second Pass: Locality Aware Parallel Pipeline Scheduling

The next logical step in the optimization of this kind of producer-consumer pipeline relationship is to execute the stages in parallel as shown in figure 4.5. Not doing so would be foolish considering that no dependencies (at least in this simple example) exist between non-correlating blocks (*i.e.*GPU thread index correlate to CPU iteration). Because of both stages are executing at the same time, to guarantee no capacity contention for the LLC, the sum of the footprints of the

producers and consumers must be considered:

$$F_{ParallelPipe} = \sum^{Stages} F_{Stage}(r) = F_{Producer}(r) + F_{Consumer}(r) \quad (4.4)$$

Thus with the same RAD of 1024 thread-blocks, the maximum footprint of the parallel pipeline during any given time step in figure 4.5 is 2MB, the size of the LLC. Because the RAD and thus the data footprint are sized to 2 MB, data passes between the producer and consumer through the cache in the same way as in figure 4.4. ③ₐ, ③ᵦ, and ③꜀ correlate to their counterparts in figure 4.2b and 4.4 as the compulsory miss by the producer, write-through, and LLC hit by the consumer, respectively. Thus same reduction in DRAM accesses, and latency are achieved - with their associated energy savings. Along with these savings, the execution time reduction that results of parallelizing the software pipeline further decreases the background energy usage of the DRAM.

Finally we note that this software pipeline is not limited to one producer-consumer pair. It can be extended to chains of producers and consumers of both types, introducing complexity in the sharing

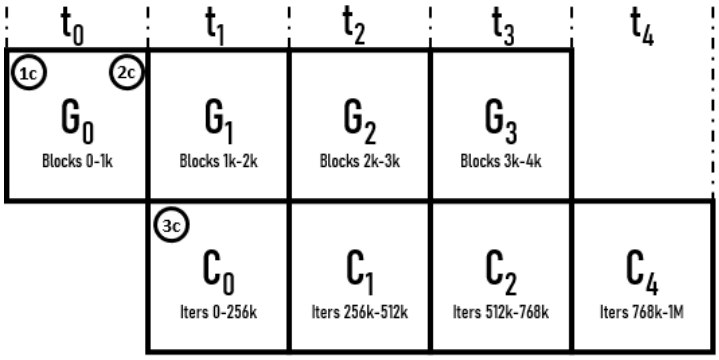


Figure 4.5: Locality Aware Fine-grained Pipeline Scheduling

patterns we will discuss in section 4.1.6.

Optimal Run-ahead Distance Determining the optimal RAD can be achieved by a simple profiling of the stages. Given some kind of helpful compiler directive like those in OpenMP [11], a compiler could detect arrays indexed by the main loop variable *i.e.*the full global id of the kernel including the block offset and thread id. During execution the OMP (or other) run-time system would take into account the size of the cache, and calculate a Run-ahead Distance:

$$RAD = \frac{LLCSize}{F_{Pipe}} \tag{4.5}$$

This equation serves as a good starting point to calculate an optimal run-ahead distance. However as we will discuss in section 4.1.6, access patterns and private L2 sizes complicate calculating the perfect value.

In 4.3.2 we conduct a run-ahead distance sensitivity study to explore the effects of these access patterns on the energy and execution time savings our technique achieves.

4.1.5 Synchronization Granularity and Control Methods

Pipeline scheduling through the control of run-ahead distance could be achieved by a number of means of varying complexity depending on the needs of the application space and the size of the cache-system available. In terms of application dependence: the amount of memory accessed per GPU thread-block (or equivalent CPU iterations) plays an important role in determining how course or fine the granularity of synchronization needs to be. A smaller cache "footprint-per-thread" (FPT) allows many threads to execute per stage and still fit into cache, *e.g.* a courser granularity of scheduling. Stages with a larger FPT require fewer threads to execute per synchronization in order to fit the overall footprint into the LLC, necessitating finer-granularity synchronization. The size of the LLC likewise affects the synchronization granularity of the pipeline. A larger LLC gives more room to breath and allows for a courser granularity of synchroniza-

tion. Conversely, a small LLC will require finer granularity synchronization for the same FPT. With this relationship between application FPT, cache size, and synchronization granularity in mind we considered where best to implement our technique. We found that for the applications we surveyed and the sizes of LLCs in available heterogeneous microprocessors today, synchronizing at the GPU thread-block granularity was sufficient.

Our implementation lies at the hardware-software interface. A run-time system executes on the CPU, taking care of controlling the RAD for CPU stages of the pipeline, signaling the RAD to a modified thread-block dispatcher, and synchronizing the CPU and GPU. Normally the thread-block dispatcher schedules as much of a kernel's pool of thread-blocks as will fit onto the GPU. Then when the kernel completes all of its thread-blocks, the GPU signals completion. We modify the dispatcher to wait to schedule thread-blocks to the GPU until it is instructed to schedule an RAD's worth of thread-blocks. The GPU then signals when this RAD completes. As a bonus this method offers flexibility in terms of controlling the footprint-per-thread by

controlling the size of each thread-block (and thus the kernel grid) in software. This worked well for our purposes, however thread-block size is often difficult to change for functional or performance reasons. It would be possible to achieve a similar kind of hardware-software interface synchronization with a tweak to the wave-front (warp in Nvidia nomenclature) scheduler inside each compute unit of the GPU. This could offer a finer granularity of control, down to the individual thread level without changing the kernel's grid.

Section 4.2 will go into greater detail about the implementation of the software runtime and hardware dispatch-control systems for our implementation.

For applications with extreme properties that cause a FPT large enough to require sub-thread granularity synchronization more complex hardware methods could be explored. Hardware methods like Full-empty bits have been proposed as an method of synchronization aimed at reducing launch latencies for discrete GPUs [25]. They could similarly be used to control the rate at which a producer-consumer pair execute to keep them in lock-step and avoid violating data dependen-

cies. These however have significant hardware costs, both in redesign and the area-overhead of adding full-empty bits to the cache. Another, less invasive method is to use the virtual memory system and inject full/empty bits at the page level to control the flow of the pipeline. This method would require pre-processing of producer-consumer regions to determine their data footprint, and would be unsuitable to control-flow divergent code.

On the other side of the gulf of granularity a fully software based solution could be used for extremely low FPT or for normal FPT programs with large LLCs. Instead of relying on hardware, we could use techniques like kernel fission to split kernels and overlap them with their associated CPU iterations [51]. While the convenience and cost of a software solution is alluring, we found that a more balanced hardware-software interface solution was appropriate for the applications we investigated.

4.1.6 Dependency Patterns

The type of dependence between two accesses of a cache block are fundamentally important to the performance of the cache, and

the functioning of the coherence protocol. In other words, the amount of Read-Read *(RR)*, Read-Write *(RW)*, Write-Read *(WR)*, and Write-Write *(WW)* dependence relationships existing between producer-consumer pairs can affect the cache performance of workloads.

In combination with these four dependence types, we also must consider the core-type of the producer and consumer. Homogeneous sharing, *i.e.* GPU-to-GPU (GG) and CPU-to-CPU (CC) may use private L2s to communicate depending on data size and L2 slicing. Heterogeneous sharing, *i.e.* CPU-to-GPU *(CG)* and GPU-to-CPU *(GC)* must make use of the LLC and snooping for on-chip sharing. These together make four stage-sharing dependency types: GG, GC, CG, and CC.

Read-Producer Patterns. Read-first patterns (RR and RW) suffer from having an exclusive victim-cache LLC. GC_RR (GPU producer, CPU consumer, Read-Read) and GC_RW sharing in particular leave potential DRAM access savings on the table, both in terms of energy and access latency for the latency sensitive CPU. When the GPU reads a shared block, because the GPU's L2 generally does not

write clean data back to the LLC [42], that block is lost upon eviction. The CPU will then miss in the LLC and access the block from DRAM, incurring an energy and execution time cost. However, if the temporal locality of the two accesses of said block are close enough because of our technique and correctly sized run-ahead distance, then the block may yet be in the GPU's L2 from which the CPU can snoop it, and save on the latency of waiting for DRAM. However, the energy cost of an LLC miss is still incurred as the controller will speculatively access DRAM on an LLC miss. Similarly GG_RR and GG_RW patterns, when scheduled naively or with an RAD that causes the cache footprint to exceed the size of the L2 cache, will not cache their reads in the LLC, leading to redundant DRAM accesses (but no slowdown).

In the case of CPU-first patterns in this category (CG_RR and CG_RW, CC_RR CC_RW) the CPU's L2 policy will evict clean lines to the LLC. So in the case of RAD small enough to not overfill the LLC yet large enough to cause blocks to be evicted from the L2, there will be an LLC hit, eliminating any speculative read for that line.

An exclusive victim caching policy for the LLC, or in other words

a lack of "read-inclusivity," is a consequence of heterogeneous coherence protocols avoiding allowing the CPU and GPU to share resources, in fear of the GPU thrashing the CPU's data. However with our technique, carefully controlled throttling of the GPU keeps this thrashing from occurring, while retaining performance.

Write-Producer Patterns. Write-first CPU-producer patterns (CG_WR, CG_WW, CC_WR, CC_WW) behave similarly to read-first pattern as the CPU private caches have a write-back policy. The size of the RAD relative to the CPU's private L2 therefore has a significant effect on when shared data ends up in the LLC. At smaller RADs, data will stay in the the L2 and when eventually consumed will come from there, causing a speculative (and wasted) read. Larger RADs will evict some of the stage's shared data (the stage's data footprint less the size of the L2) to the LLC where it can be consumed with no DRAM read.

Write-first GPU-producer patterns (GC_WR, GC_WW, GG_WR, GG_WW), do not have this issue. The GPU writes though to the LLC, and does not suffer from private cache spilling in the way that

write-back caches do.

4.2 Methodology

A revitalization project has recently begun to update Gem5 and create an active, healthy development community around it [52]. This new Gem5 includes in it a realistic integrated GPU model developed by AMD based on their Graphics Core Next 3 (GCN3) GPU architecture, following the guidelines in the HSA design manual [43]. Heterogeneous System Architecture (HSA) is a set of cross-vendor specifications widely used in the industry for a variety of devices [53]. Implementing the HSA standard means that the new Gem5 is far more representative of how a real GPU's hardware and software operate compared to the the driver emulation done in gem5-gpu. Thus for this phase of research we have chosen to forego using the out-dated gem5-gpu, and switch to this new simulator.

This section describes the experimental methodology for the full quantitative results to be discussed in 4.3. It begins with a description of the modeling parameters used, the reasons behind them, and the

architecture of the cache hierarchy. It goes on to discuss the software architecture of Gem5's GPU driver system, and the customizations necessary to control the GPU model with sufficient precision to achieve our desired performance enhancements.

Next, we discuss the software run-time system created to interface between the workloads we use to evaluate our technique. As before, any workloads must first be tailored to allow them to execute properly on the simulator. A specialized software interface written in C++ allows users to easily encapsulate the CPU and GPU regions of code that can be pipelined together. The interface can then be called upon to execute the pipeline, keeping DRAM access minimization and overall performance in mind.

Finally section 4.2.3 discusses the workloads used in our quantitative evaluation of the technique.

4.2.1 Model Configuration

Table 4.1 shows the configuration we used in the evaluation of our technique on Gem5. The terminology used for GPU attributes is that of AMD, as outlined in Chapter 2. We based the configuration off of

the Ryzen 3 2XXXU series of APUs. Like the configurations from the previous chapter, the modeled chip has 4 out-of-order cpu cores, and a modestly sized GPU. Since gem5 does not model Dynamic Voltage and Frequency Scaling (DVFS), we chose clock speeds for both the CPU and GPU in the middle of the range for the Ryzen 2200U chip.

As with the previous iteration of the simulator, the GPU model must run in Syscall Emulation mode, meaning there is no true operating system. This means that system calls are spoofed inside the simulator. This leads to some limitations which will be discussed in section 4.2.3. The coherent cache system is however modeled with fidelity, and we believe this to be the most important component necessary to evaluate our technique's performance accurately. DRAM Power is modeled using libDRAMPower [54].

Cache Hierarchy. Unlike Gem5-gpu which did not model a shared Last-Level-Cache (LLC), the GPU_VIPER coherence protocol implements an LLC through which efficient on-chip data-sharing can be achieved [43]. Each CPU has private L1 I/D caches, with every two CPUs grouped together in "core-pairs" sharing an L2 cache. The

GPU's Compute Units (CUs) share an L1 instruction cache known as a Sequencer Cache (SQC), while each CU has a private L1 data cache known as a Texture Cache per Pipe (TCP). CUs share the GPU's L2 cache known as a Texture Cache per Channel (TCC) [43]. As our configuration has 4 CPU cores and 3 CUs, there are 3 L2 caches in the system.

The focal point for this work – the LLC – serves as an exclusive victim cache for the GPU and CPU L2s, controlled by a stateless directory-based memory controller. On LLC misses, the controller issues a speculative access to DRAM in parallel with snooping the L2s, reducing access latency for lines not found in the L2s. This is beneficial to CPU performance, though marginally useful for the latency tolerant GPU. In terms of L2 write policies, the CPU L2 caches employ a standard write-back policy to the victim cache LLC. In contrast, the GPU L2 cache engages in avoidance strategies to keep the GPU from thrashing the LLC. In the context of producer-consumer sharing patterns in which the CPU and GPU collaborate; these avoidance strategies are detrimental to performance. Specifically

the GPU's stores write through to main memory unless told to use the L3 on write-through. Since we are interested in GPU-CPU sharing data on-chip, we enable writing to the L3. GPU fills bypass the LLC entirely, being routed directly to the TCC and then simply invalidated on eviction, rather than being written back to the LLC. This presents a problem for data dependences that follow GPU reads. In section 4.1.6, we discussed in detail how the differences in writing policies between the GPU and CPU can interact with victim caching and speculative DRAM access to affect DRAM power and performance. In order to combat this issue, we modified the LLC's policy to cache reads upon arrival from DRAM, before sending a copy to the requesting L2. Section 4.3.3 presents the savings we achieved by doing so.

CPU		GPU	
Number of cores	4	Number of CUs	3
CPU Clock rate	2.95 GHz	GPU Clock rate	1100 MHz
Issue width	8	SIMD Units per CU	4
Issue queue size	64	SIMD size	16
Reorder buffer size	192	Wavefront size	64
L1-I cache (private per core)	32 KB	Wavefront slots	10
L1-D cache (private per core)	64 KB	L1 (TCP) Size (Private/CU)	128 KB
L2 cache (shared per core-pair)	2 MB	L2 (TCC) Size (Shared)	256 KB
L3 Cache (LLC)			4MB
Main Memory			8 GB DDR4 16x4 (64 bit) @ 2400 MHz

Table 4.1: Simulation parameters used in the experiments. The modeled heterogeneous microprocessor resembles an Ryzen 2XXXU series APU.

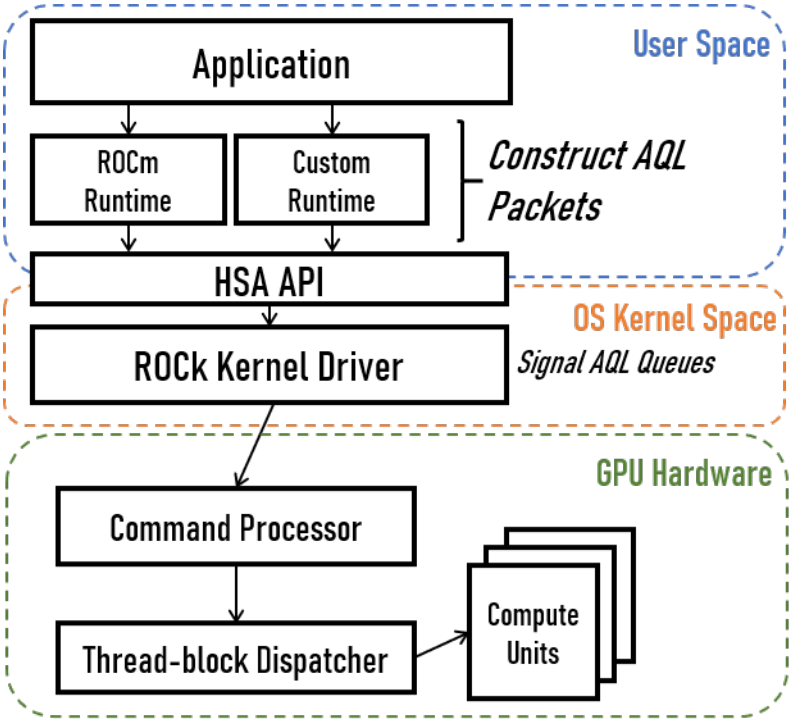


Figure 4.6: HSA Compliant Driver Stack and customizations for Gem5.

4.2.2 Driver Stack Architecture

As previously mentioned, Gem5 now models AMD's GCN3 Architecture. Running in tandem is AMD's Radeon Open Compute Platform (ROCm), which serves as the HW-SW interface between the workloads and the GPU. Figure 4.6 shows the driver stack that the application uses to communicate and synchronize with the GPU. ROCm communicates from the user space to the emulated Kernel Fusion Driver in kernel space (ROCk) by sending command Architected Queuing Language (AQL) packets conforming to the HSA specifica-

tion through software queues that map to hardware queues on the GPU. The emulated kernel receives the packets and sends them to the GPU's command processor (CP) which executes various functions according to the packet type and sends back a completion signal when the task has been completed.

For instance, when a user calls a kernel launch through ROCm, it sends a *kernel dispatch packet* containing the location of the kernel's code in memory along with its parameters and an additional completion signal to the GPU command processor. The command processor then instructs the hardware scheduler to schedule the kernel's workgroups (blocks) to the GPU's compute units, and signals that the kernel has been launched. Finally, when the last workgroup of the kernel is completed, the GPU sends the kernel completion signal back to user space via the kernel driver.

Custom Scheduling Controller. In order to achieve our goal of scheduling workgroups synchronized with the CPU in a pipeline fashion, we take advantage of a type of HSA command packet, an *agent dispatch packet*, which contains fields that the CP can read in order

to determine what the application wants the CP to do. We then customized the CP and hardware dispatcher to respond to two key commands from the agent packet. The first command, INJECT_SIGNAL, injects a custom HIP signal created by the software interface and associates that signal with a kernel id. If the hardware dispatcher sees that a custom signal has been injected for a particular kernel id, when that kernel is launched with a normal kernel launch packet, it will not schedule any workgroups for execution on the GPU.

When the application desires workgroups to execute on the GPU, it sends the second type of command, FWD_PROGRESS. This command instructs the dispatcher to execute a given number of workgroups rather than all of the kernel's workgroups. The number of workgroups executed in this fashion can be varied by the application in user space to control the cache footprint of the GPU. When the last of these workgroups is completed, the CP sends back the custom signal given to it from the injection command packet.

This custom scheduling method relies on the same underlying communication protocol that the ROCm userspace layer uses to con-

trol the lower-level layers. However because the kind of packets we need to send are not standard, we do not use the software layer that comes in ROCm for anything other than launching kernels. Using the HSA API introduces complexity to actually writing the code for our optimization technique. Given this complexity of the underlying control scheme, we created a software interface that abstracts much of that complexity away from the direct view of the programmer trying to use our optimization in a practical setting. The user need only set wrap the producer-consumerstages with the interface class, and call the pipeline.

Overhead. Using this interface to synchronize with the GPU introduces a small overhead. The alternative method is to drive our software pipeline with a new GPU kernel for every synchronization. As we have discussed in detail, however, GPU Kernel launches are expensive. Indeed, we measure for each benchmark we evaluate the overhead of synchronizing with the GPU using our method and the overhead of launching a new GPU kernel. The launch latency was at least an order of magnitude larger than our method for every workload

measured.

4.2.3 Workloads

To evaluate Pipelined CPU-GPU Scheduling for Caches on the platform we have described, we surveyed several Heterogeneous computing benchmark suites and found seven suitable workloads containing producer consumer relationships between CPU loops and GPU kernels. These workloads are listed in Table 4.2. The table lists the workload's name, source, input size, and ordering of the CPU and GPU stages.

Benchmark	Suite	Input	Stage Order
CEDT	Chai	2146 x 3826 video	GGCC
BE	Hetero-Mark	1080p video	CG
EP	Hetero-Mark	8192 Creatures	CGC
DWT2D	Rodinia	1125x2436 image	CG
Kmeans	Rodinia	512K Objects, 34 Features	GC
LavaMD	Rodinia	1000 boxes, 100 Particles per box	CG
372.smithwa	OMP2012	ref - 1048576	GC

Table 4.2: Workloads used in the experimental evaluation of Pipeline Scheduling for Shared Data Cache Locality.

From the Chai benchmark suite, we have a single benchmark *CEDT*: the "Task-Partitioning" version of Canny Edge Detection. This benchmark is singular in that it has a chain of 4 producer/consumers, two GPU stages, followed by two CPU stages. The provided

input for this benchmark, a clip from a cartoon, was at extremely low resolution, 626x354. At this resolution the cache footprint of the stages would not be large enough to fill a modern LLC. Thus, we chose to use a clip at a more appropriate modern resolution, 2146x3926 (4K) [44].

Hetero-Mark provides two benchmarks to our list [13]. *BE*: Background Extraction, in which a video is passed frame by frame from the CPU and passed to the GPU. We use the default input for this benchmark, a 1080p (1920 x 1080 pixels) black & white video. The second workload from this benchmark suite is *EP*: Evolutionary Programming. EP has 4 distinct stages, of which the first three were amenable to optimization with our technique. For input we have one island of 8,192 creatures, and every iteration half of the island's creatures are eliminated.

The Rodinia benchmark suite is often used to evaluate GPGPU architectures and has a wide variety of benchmarks, 3 of which were suited to our pipelining optimization. *DWT2D* (Discrete Wavelet Transform) is a broadly used digital signal processing technique. The

first step of this program is for the CPU to perform file I/O and load an image into memory. Then the GPU processes the image once, and passes the result to an iterative process that runs on the GPU, eventually producing the transform. For this benchmark, we elected to measure only the region where file I/O and pre-processing occur, and not the iterative part that would dominate execution time. The reasoning for this is that file I/O and pre-processing are common operations that occur in many image related workloads, and similar code could be embedded in many applications. We believe analyzing this case is a worthwhile endeavor. As an input, we use a 1125x2436 pixel image, the size of an iPhone screen.

Next, from Rodinia is *Kmeans*. In Kmeans, a GPU stage is the producer for a CPU stage consumer. In addition to the Write then Read producer-consumer relation we have thus far targeted, Kmeans features a read-read data-sharing relationship between the GPU and the CPU. This exposes the interesting deficiency in the coherence protocol, that the LLC acts as a victim-cache for the L2s, and the GPU L2 does not write back clean blocks. We discussed this "read-inclusivity"

issue in Section 4.1.6 and in Section 4.3.3 we discuss the impact it has on Kmeans and other workloads. For input we use a generated input file with 524,288 objects and 34 features, slightly larger than the standard input of 494,020 to be an even multiple of 1024.

The last Rodinia benchmark that we showcase is *lavaMD*. In this workload a CPU stage initializes key data structures that the GPU then uses to perform a 3D molecular dynamics simulation. We use the standard input of 1000 boxes, with 100 particles per box.

The last benchmark we present in this section is the 372.smithWaterman benchmark from SPEC OMP 2012 used to evaluate our Nested MIMD-SIMD Parallelization technique in Chapter 3. Indeed, due to the loop splitting technique we used to expose parallelism, data passes between the GPU producer and the CPU consumer. Rather than using nested parallelism, we instead pipeline the GPU and CPU stages with our new method. We use the same input as before, 1048576.

Order Matters. Table 4.2 specifies the order of the stages in each benchmark. This is relevant because as we will show in Section 4.3, CPU consumers tend to get a significant execution time reduction

when compared to GPU consumers. This is because CPUs are latency sensitive, while GPUs are latency tolerant. While there are energy savings from the reduction in DRAM accesses in both cases, in the case of CPU consumers, execution time reduction translates to reduced DRAM self-refresh energy: compounding the energy savings for our technique. Benchmarks that contain CPU consumers are: CEDT, EP, Kmeans, and 372.smithwa.

Benchmark Criteria. The reasoning for choosing our benchmarks is four-fold. First of all, a workload must have at least one GPU kernel and CPU loop of roughly the same number of threads/iterations that pass data between them. Many benchmarks we surveyed were completely dominated by GPU kernels, with no shared data to speak of. Second, the data footprint of the shared data must be a large percentage of the total cache footprint of the stages it is passed between: optimizing for a small fraction of the data would be inefficient. Third, the full result of the producer stage cannot be needed when the consumer stage launches. For example, a stage that performs a reduction, where the resultant value is needed in the next stage. Finally, in

some cases while the previous constraints were met, the input size of the workload was not sufficient to fill the LLC, and the workload was hard-coded to expect an input of a particular type.

In general our technique is well suited to streaming workloads which pass data between heterogeneous stages and access that shared data at the same rate in terms of their iterations or threads. We believe that these types of workloads are both relatively common from our survey of workloads, and relatively easy to achieve when writing new benchmarks that may take advantage of our technique.

4.3 Results

This section presents the results of a quantitative study of our technique simulated on Gem5. We begin by presenting the overall gains in terms of reduced DRAM accesses, execution time, and total DRAM energy. We follow with a detailed performance breakdown by examining the results of a run-ahead distance sensitivity study. Finally, we consider the effects of LLC read-inclusivity on our technique.

4.3.1 Main Result

Figure 4.7 presents the main result from our evaluation of our technique on the simulation platform described in section 4.2. It shows the optimal run-ahead distance (blue bars) normalized to the "serial" case ("1.0" red bars) where the producer execution rate is not throttled at all. In the serial case, producers execute in their entirety before consumers begin, referencing their full cache-footprint, and spilling producer-consumer data out of the cache system. The optimal case was chosen based on total DRAM energy savings, which we show in addition to the total DRAM accesses (reads and writes), and the execution time - all normalized to the serial case.

By controlling the run-ahead distance, we are able to achieve an average 30.4% reduction in DRAM accesses in the optimal case. Every benchmark we evaluated was able to achieve a significant reduction in DRAM accesses due to the producer-consumer communication occurring on-chip and via the cache hierarchy. This reduction in memory traffic leads to a corresponding savings in DRAM energy,

The set of benchmarks also achieved a 26.84% reduction in ex-

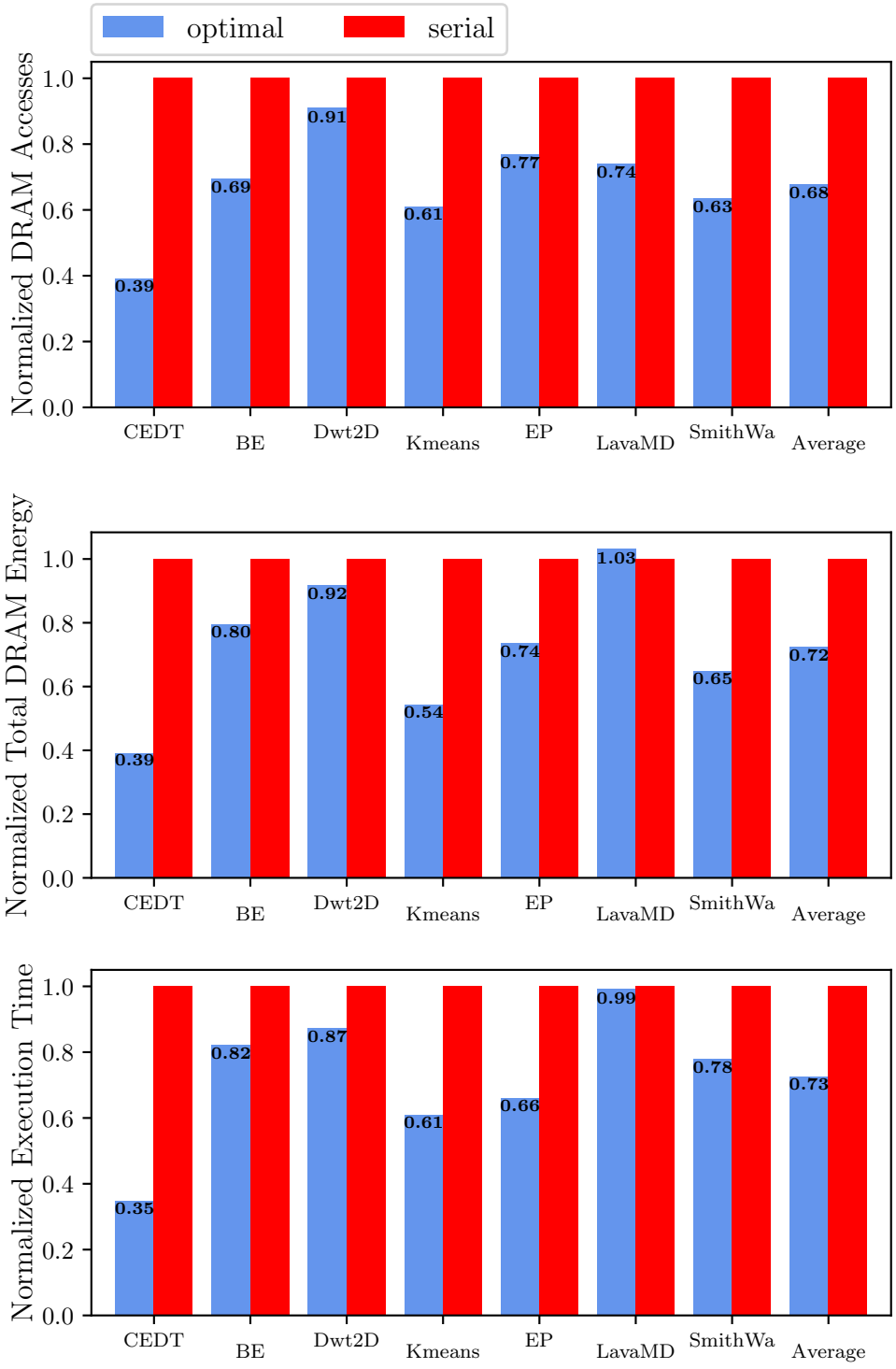


Figure 4.7: DRAM Accesses, Execution Time, and Total DRAM Energy usage of the optimal Run-ahead Distance (Blue) normalized to the Serial Case (Red).

ecution time. Workloads with CPU consumer stages (CEDT, EP, Kmeans, SmithWa) benefited strongly from a tight run-ahead distance, achieving an average 38.8% execution time reduction. Data meant for CPU consumer stages stayed in the cache, reducing access latency, something which the latency intolerant CPU cores benefit from greatly. On the other hand, those workloads with GPU consumers (BE, dwt2D, LavaMD) did not receive as much performance gain, achieving a less substantial 10.9% reduction in execution time. This is to be expected since the GPU consumers are naturally more latency tolerant.. The speedup of this subset of benchmarks is due primarily to software pipelining overlap, and not due to the locality improvement from our technique. One benchmark included in this latter set, LavaMD, achieves only a 0.74% savings in execution time because the GPU stage's execution time in the serial case is 52x larger than the CPU stage's execution time, leaving little execution time to overlap. The other workloads have closer ratios between the stages' execution times, between 1.1x and 4.7x. This simply means that our technique benefits these workloads primarily through the energy ben-

efit that comes from a reduction in DRAM accesses, factoring into the total DRAM energy.

To wit, we were able to achieve a 27.4% reduction in total DRAM energy on average. This includes the energy saved from the reduction in DRAM accesses, as well as the refresh, activation, pre-charge, and their associated background energies. Again, we see that the GPU-First patterns perform better than their CPU-First counterparts: a 41.3% reduction in energy on average compared to 8.8% . Notably, LavaMD actually receives an *increase* in total DRAM energy compared to the serial case. While the access energy (Read, Write, Activation, and PreCharge energy) goes down proportionally to the savings in accesses, the background energies, namely precharge and activation background energies increase when we apply our technique. LavaMD performs a stencil calculation where each each block within the calculation accesses its neighbors and has non-contiguous data structures to keep track of details about its neighbors. When our technique is applied, data structures for non-contiguous blocks are accesses, leading to banks waiting in activated and precharged states for longer.

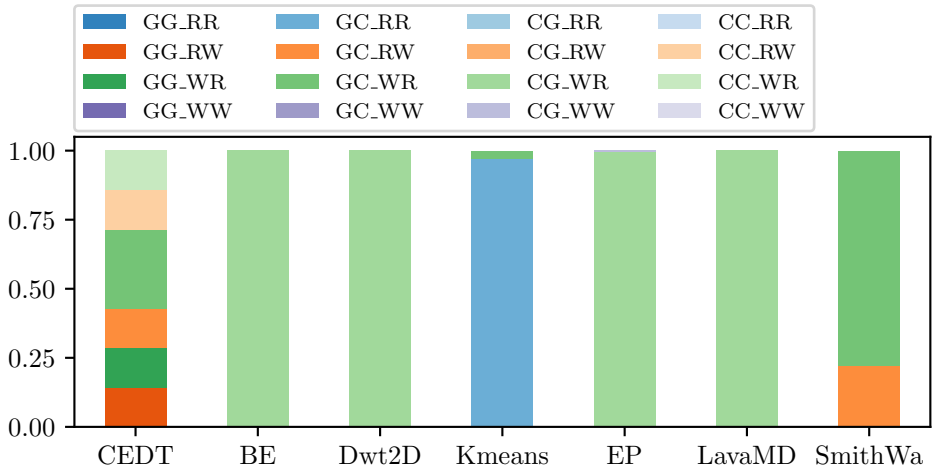


Figure 4.8: Heterogeneous producer-consumer sharing patterns for each workload. Bars show percentage data size of each type of sharing pattern relative to the all shared data in a workload.

4.3.2 Run-Ahead Distance Sensitivity Studies

Choosing the correct run-ahead distance is an important factor in optimizing performance and power usage with our technique. This section presents the results of sensitivity studies for each workload wherein the run-ahead distance, and thus the cache-footprint of the stages, is increased up to its limit (the "serial" case in Figure 4.7).

Each workload is uniquely sensitive to changes in RAD depending its footprint-per-thread (FPT, section 4.1.5) and dependency patterns. Figure 4.8 shows for each benchmark what proportion of the total shared data fell into the sixteen categories we outline in Section 4.1.6.

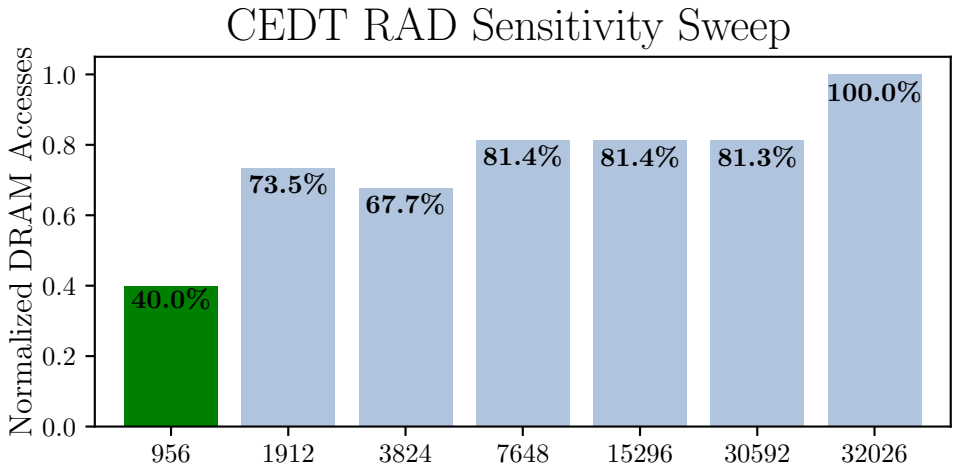
4.3.2.1 CEDT Sensitivity Study

Pattern	Stage Footprints						
	S0	S1	S2	S3	TPB	FPT	RAD
GGCC	2	3	3	2	256	10	1639

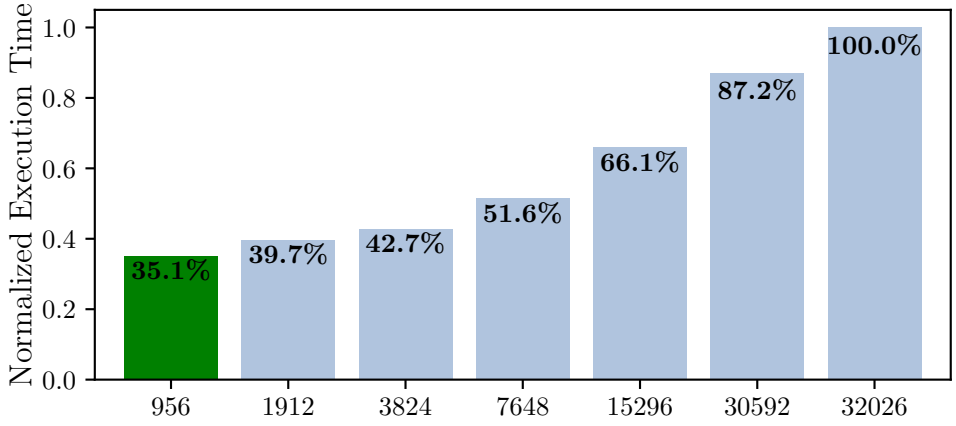
Table 4.3: CEDT Profile. Each stage's footprint is in bytes per thread (or bytes per CPU iteration). Using Equation 4.5, the threads-per-block (TPB) and footprint-per-thread (FPT) are used to calculate potentially optimal run-ahead distance (RAD) in terms of GPU thread-blocks using 4 MB LLC size in our configuration.

Figure 4.9 presents a sweep of RAD values in number of GPU thread-blocks (x-axis) in each plot. The optimal RAD value used in the main result is highlighted in green and was chosen based on normalized total DRAM energy usage.

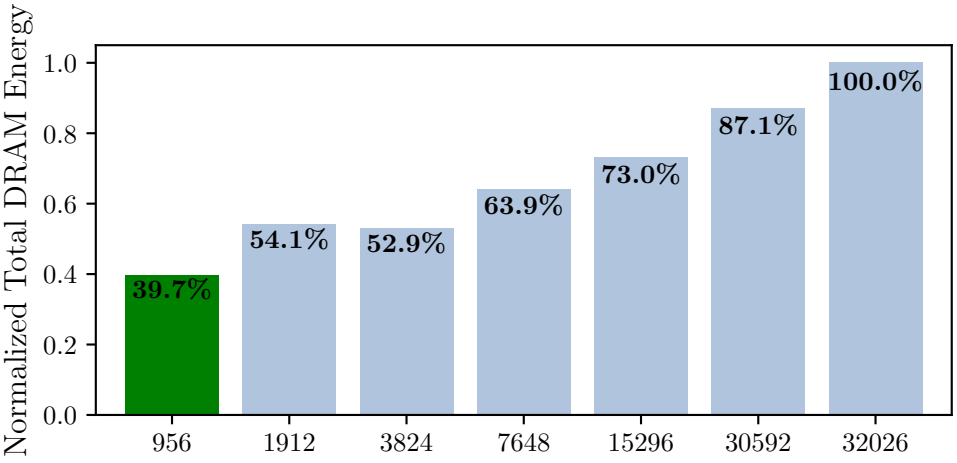
CEDT's profile is presented in Table 4.3. Each stage contributes a number of uniquely accessed bytes per thread, totally to 10 bytes for every GPU thread executed in the pipeline. With the threads-per block(TPB) of 256 we can calculate using equation 4.5 that a potentially optimal RAD will be at 1639 thread-blocks. Under the pipelined schedule that our technique applies, CEDT must be run using a multiple of 956 thread-blocks due to some dependencies caused by a stencil calculation (this naturally excludes the maximum number of blocks, 32,026, where-in each stage is run serially). Therefore, the optimal



(a) Normalized DRAM Accesses Sensitivity



(b) Normalized Execution Time Sensitivity



(c) Normalized DRAM Energy Sensitivity

Figure 4.9: CEDT Run-Ahead Distance (RAD) sensitivity study. X-axis shows RAD in GPU thread-blocks.Result of optimal RAD (Min Energy) shown in green.

RAD should be 956, the largest possible RAD below our predicted value.

Figure 4.9c shows that this prediction bears out. The optimal case for CEDT is 956 thread-blocks, with the normalized total DRAM energy being lowest at this point. We can consider the total DRAM accesses in Figure 4.9a to explain the optimal point. We can see that again the optimal case is the RAD we predicted, 956 thread-blocks. However, we can see that several larger RAD values achieve savings (1912 and 7648 thread-blocks). This is due to the CPU L2s providing extra space (2MB each) for the working set to reside in, keeping the footprints of the CPU stages from evicting shared data. The rest of the non-serial cases (15,296 and 30,592 thread-blocks) do not divide evenly into the total number of blocks for this input, 32026. Thus, once one epoch of the pipeline completes, the remainder will have far fewer blocks to complete, making for a pipeline with a small cache footprint.

The dependencies for this workload are for the most part straightforward, with the only dependency of note being a CC_WR depen-

dency between the two CPU Stages, S2 and S3. The last stage, S3, will miss in the LLC because some of the shared data has not yet been evicted from the other CPU's L2, causing a speculative read to DRAM. 11% of the accesses in the optimal case are speculative reads.

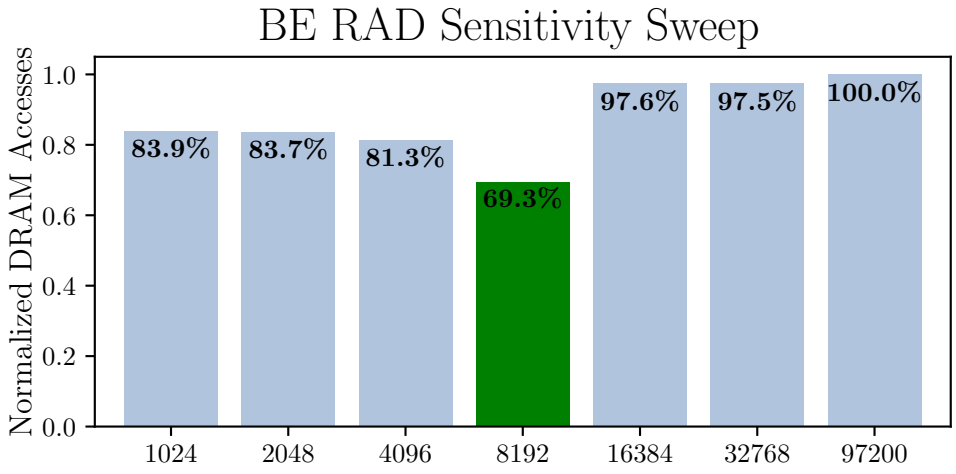
4.3.2.2 BE Sensitivity Study

Pattern	Stage Footprints				
	S0	S1	TPB	FPT	RAD
CG	6	6	64	12	5462

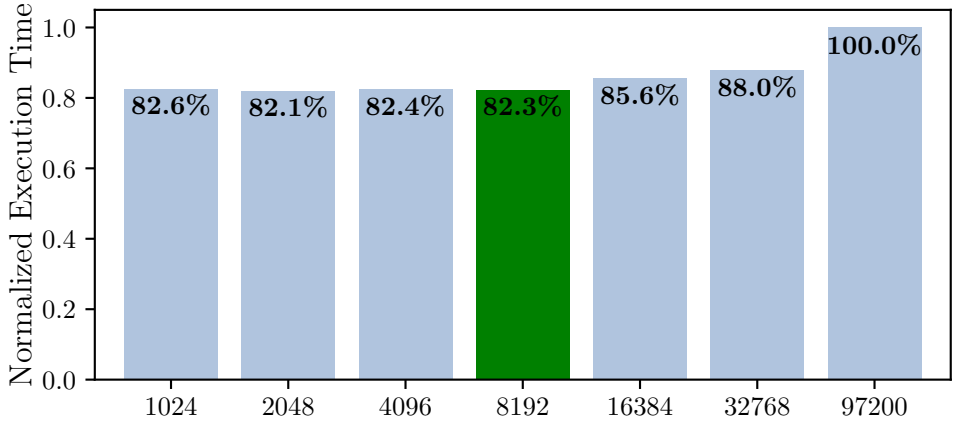
Table 4.4: BE Profile. Each stage's footprint is in bytes per thread (or bytes per CPU iteration). Using Equation 4.5, the threads-per-block (TPB) and footprint-per-thread (FPT) are used to calculate potentially optimal run-ahead distance (RAD) in terms of GPU thread-blocks using 4 MB LLC size in our configuration.

Figure 4.10 presents a sweep of RAD values for BE, up to its maximum, 97200 (serial scheduling). Table 4.4 shows the profile of this benchmark. It has 2 stages, a CPU and then a GPU stage that together have a FPT of 12. Using Equation 4.5 we estimate an optimal RAD value of 5462 thread-blocks.

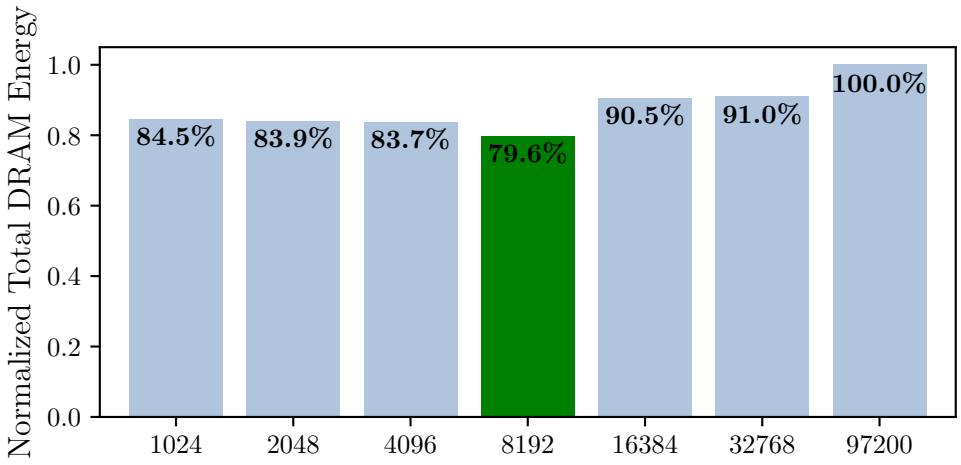
In Figure 4.10c, we see that the Normalized DRAM Energy usage, is lowest when using a RAD of 8192 thread-blocks, which is larger than our prediction. In fact, the smaller RAD values, 1024 - 4096 thread-blocks actually exhibit degraded performance. This is particularly



(a) Normalized DRAM Accesses Sensitivity



(b) Normalized Execution Time Sensitivity



(c) Normalized DRAM Energy Sensitivity

Figure 4.10: BE Run-Ahead Distance (RAD) sensitivity study. X-axis shows RAD in GPU thread-blocks. Result of optimal RAD (Minimum Energy) shown in green.

pronounced in Figure 4.10a, whichy shows the Normalized DRAM accesses. This is due to 100% of the shared data for this workload being a CG_WR case, as shown in Figure 4.8. At small RAD values, data that the GPU consumes resides in the CPU's L2, causing a speculative read. Ultimately, the data comes from the CPU's L2 cache via snooping. For instance, at a RAD of 4096 thread-blocks, the CPU's L2 Cache footprint is 1.5 MB. At the optimal case of 8192, the footprint of the CPU Stage is 3 MB, leading the CPU to write back shared data to the LLC, ultimately causing the GPU accesses to be hits.

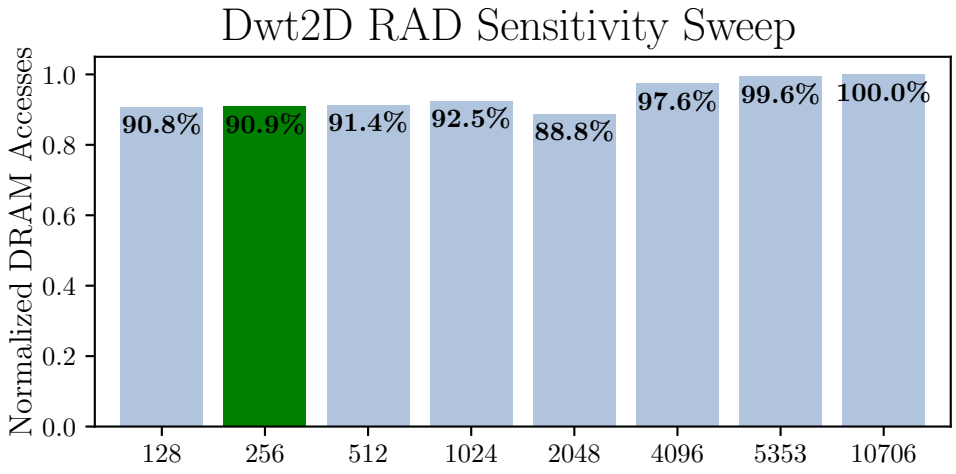
4.3.2.3 Dwt2D Sensitivity Study

Pattern	Stage Footprints				
	S0	S1	TPB	FPT	RAD
CG	3	15	256	12	911

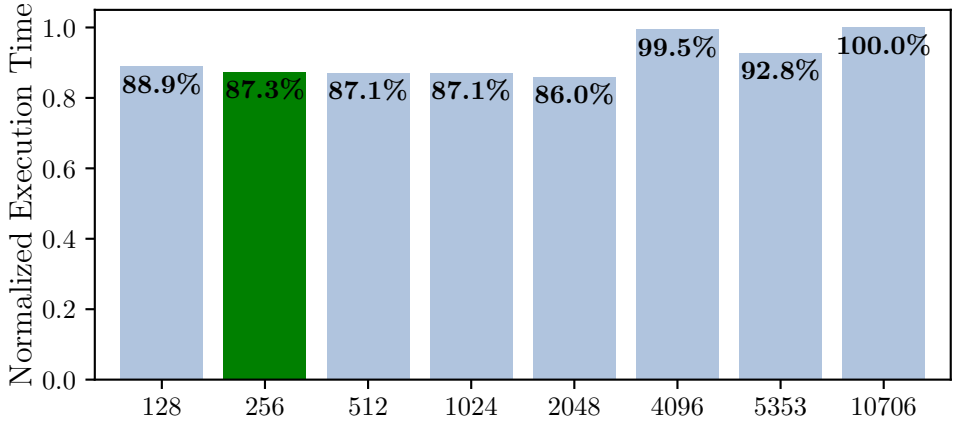
Table 4.5: DWT2D Profile. Each stage's footprint is in bytes per thread (or bytes per CPU iteration). Using Equation 4.5, the threads-per-block (TPB) and footprint-per-thread (FPT) are used to calculate potentially optimal run-ahead distance (RAD) in terms of GPU thread-blocks using 4 MB LLC size in our configuration.

Dwt2D's profile is reported in Table 4.5. Using Dwt2D's FPT of 21 bytes, we calculate an estimated optimum RAD of 911 thread-blocks.

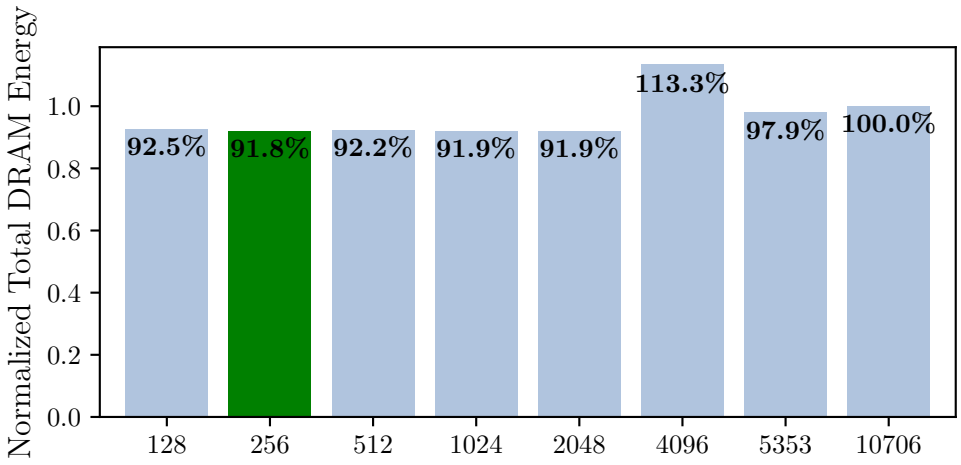
What we see in Figure 4.11c, is that the optimal energy usage



(a) Normalized DRAM Accesses Sensitivity



(b) Normalized Execution Time Sensitivity



(c) Normalized DRAM Energy Sensitivity

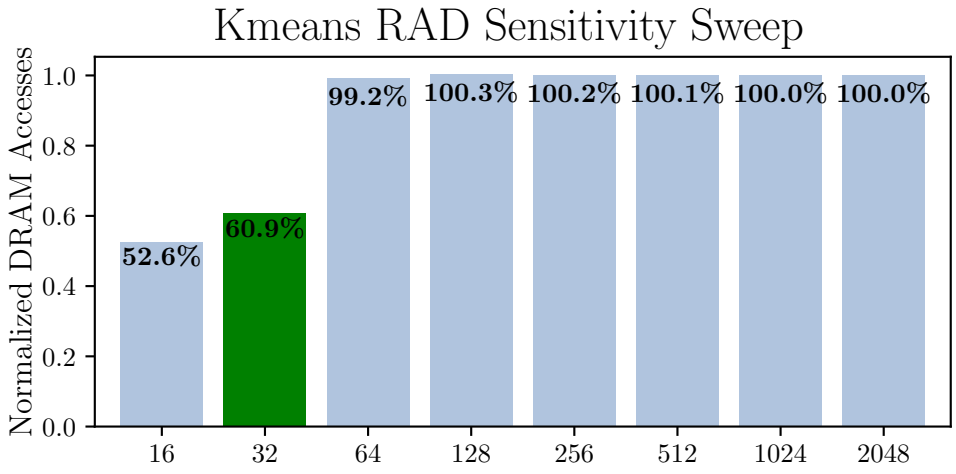
Figure 4.11: DWT2D Run-Ahead Distance (RAD) sensitivity study. X-axis shows RAD in GPU thread-blocks. Result of optimal RAD (Min Energy) shown in green.

falls significantly below our estimate, but that the estimated RAD's performance is extremely close to the optimum, and thus our estimate was pretty good. However, we can also see that RAD values significantly higher than our estimate also did well.

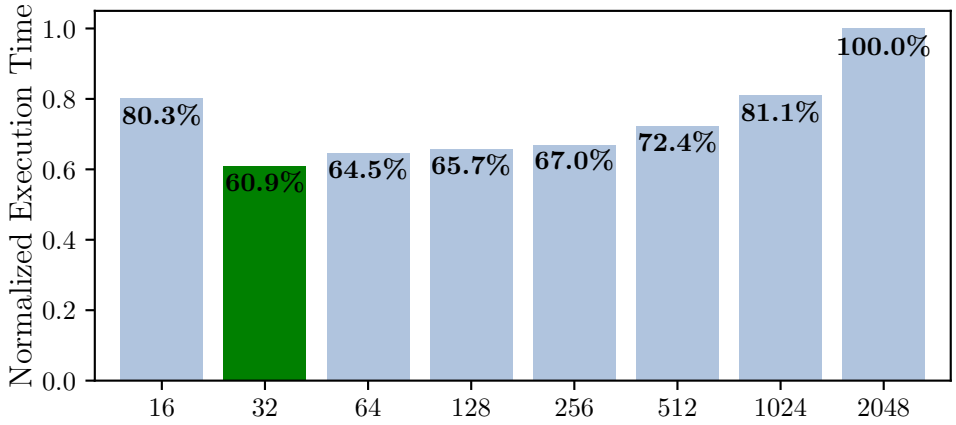
In Figure 4.11a, we see that a RAD of 2048 actually achieves the best access savings overall. To explain this, we must look to the dependency pattern within Dwt2D from Figure 4.8, that is: 100% CG_WR. Dwt2D is a CPU-to-GPU benchmark in which the shared data will first be written to the CPU's L2 until it is evicted to the LLC. Additionally, the GPU stage executes significantly faster than the CPU stage at the same RAD (2.4x faster) and so during one epoch of the software pipeline, the GPU will consume data and write its own data to the LLC before the CPU evicts its data. This race condition works in our favor by keeping the shared data in the CPU's L2 until later when the GPU cannot evict it erroneously.

4.3.2.4 Kmeans Sensitivity Study

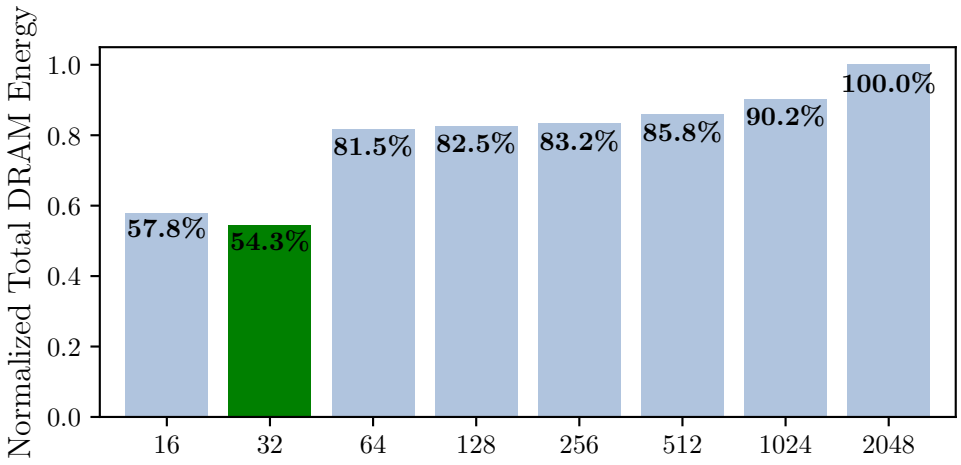
The profile for Kmeans is shown in 4.6. It consists of a GPU to CPU pipeline with a moderately sized FPT of 284 Bytes. Using



(a) Normalized DRAM Accesses Sensitivity



(b) Normalized Execution Time Sensitivity



(c) Normalized DRAM Energy Sensitivity

Figure 4.12: KM Run-Ahead Distance (RAD) sensitivity study. X-axis shows RAD in GPU thread-blocks. Result of optimal RAD (Minimum Energy) shown in green.

Pattern	Stage Footprints				
	S0	S1	TPB	FPT	RAD
GC	140	144	256	284	58

Table 4.6: Kmeans Profile. Each stage's footprint is in bytes per thread (or bytes per CPU iteration). Using Equation 4.5, the threads-per-block (TPB) and footprint-per-thread (FPT) are used to calculate potentially optimal run-ahead distance (RAD) in terms of GPU thread-blocks using 4 MB LLC size in our configuration.

Equation 4.5, we predict that the optimal RAD for Kmeans should fall near 58 thread-blocks. Figure 4.12 presents the results of the RAD sensitivity study conducted for Kmeans, and in Figure 4.12c we see the optimal energy value is at 32 thread-blocks with just 54% of the energy of the serial case (2048 thread-blocks). The next RAD step, 64 thread-blocks, sees a large jump to 82% of the total DRAM energy in the serial case. In Figure 4.12a we see that the 64 thread-block case achieves no DRAM access savings, and the 32 thread-block case does, fitting with our prediction. The 16 thread-block case actually performs better in terms of DRAM accesses than the 32 thread-block optimal case. However, looking at the normalized execution time in Figure 4.12b, we see that the 16 thread-block case has a large execution time increase. This is due to increased L1 instruction cache misses in the CPU core and overhead from the run time system, due to the

smaller pipe-width. This provides further motivation for predicting the optimal RAD value.

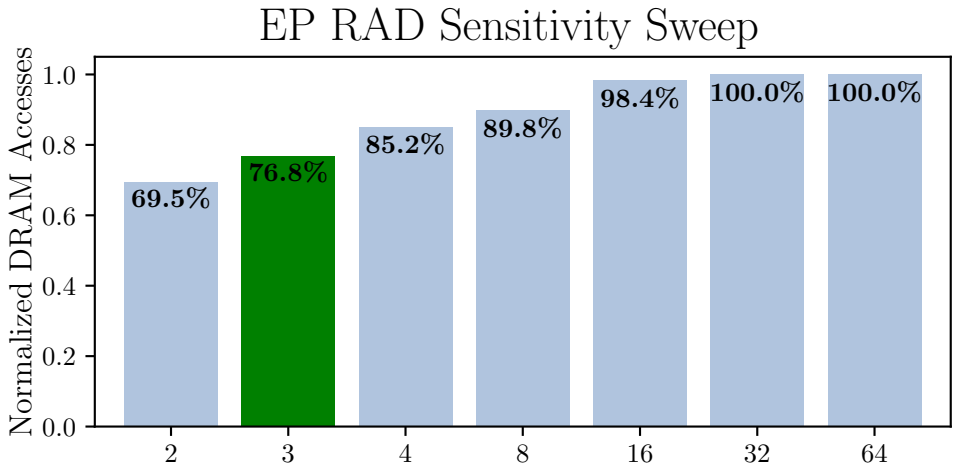
4.3.2.5 EP Sensitivity Study

Pattern	Stage Footprints					
	S0	S1	S2	TPB	FPT	RAD
CGC	4024	8024	16	256	12064	3

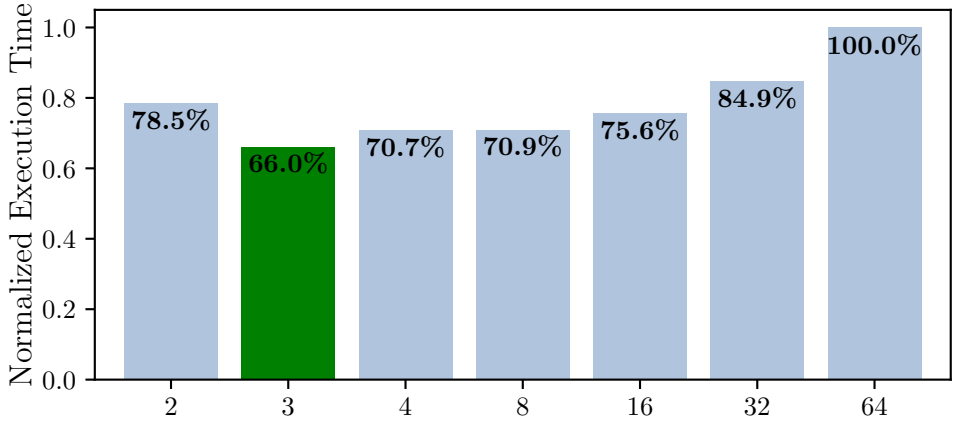
Table 4.7: EP Profile. Each stage's footprint is in bytes per thread (or bytes per CPU iteration). Using Equation 4.5, the threads-per-block (TPB) and footprint-per-thread (FPT) are used to calculate potentially optimal run-ahead distance (RAD) in terms of GPU thread-blocks using 4 MB LLC size in our configuration.

EP's profile is shown in Table 4.7. EP has three stages, a GPU stage sandwiched between two CPU stages, and has the largest FPT of any of our workloads at 12064 bytes-per-thread. We Estimate using equation 4.5 the optimal RAD value to be 2.7 thread-blocks, which since we cannot split thread-blocks into smaller parts, we round this to 3 thread-blocks.

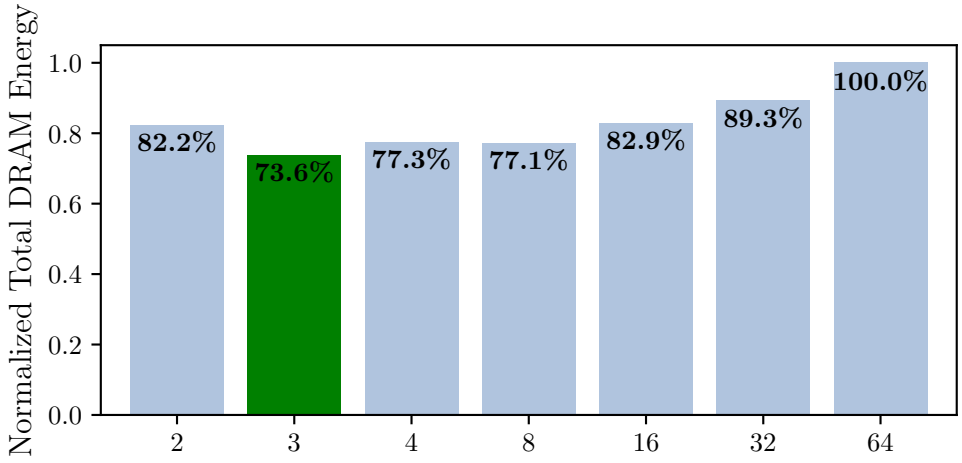
Figure 4.13 presents the results of the sensitivity study for EP. Indeed, we see that for the normalized DRAM energy (Figure 4.13c), EP performs best at 3 thread-blocks. Figure 4.13a, the normalized DRAM accesses, shows that reducing the number of thread-blocks even further reduces extraneous DRAM accesses some more. However,



(a) Normalized DRAM Accesses Sensitivity



(b) Normalized Execution Time Sensitivity



(c) Normalized DRAM Energy Sensitivity

Figure 4.13: EP Run-Ahead Distance (RAD) sensitivity study. X-axis shows RAD in GPU thread-blocks. Result of optimal RAD (Minimum Energy) shown in green.

Figure 4.13b - the normalized execution time - shows that at this RAD value the performance begins to suffer, and with it goes the energy usage as well. The performance degradation is due to blocks that belong to the CPU instruction cache being evicted from the (inclusive) L2, causing fetch stalls. This shows how with profiling and estimation, we can execute our technique with an optimal RAD value.

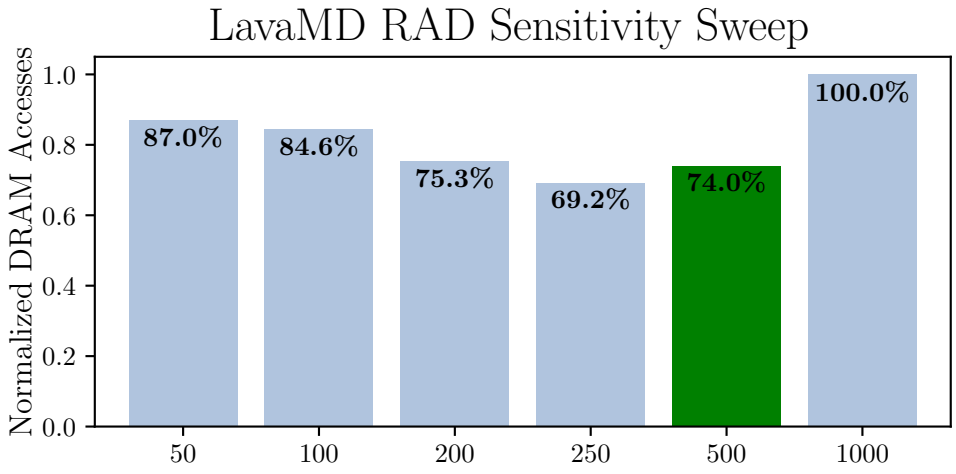
4.3.2.6 LavaMD Sensitivity Study

Pattern	Stage Footprints				
	S0	S1	TPB	FPT	RAD
CG	78.54	78.54	100	157.04	268

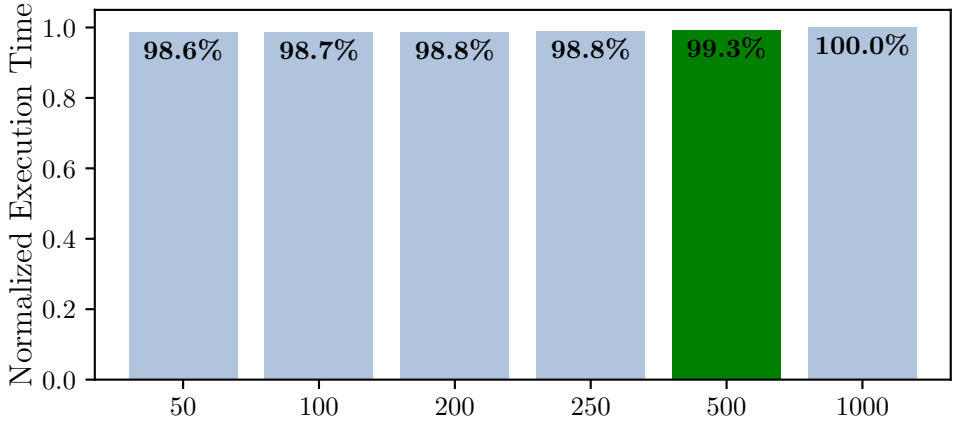
Table 4.8: LavaMD Profile. Each stage's footprint is in bytes per thread (or bytes per CPU iteration). Using Equation 4.5, the threads-per-block (TPB) and footprint-per-thread (FPT) are used to calculate potentially optimal run-ahead distance (RAD) in terms of GPU thread-blocks using 4 MB LLC size in our configuration.

The profile of LavaMD in Table 4.8 and the accompanying sensitivity study help explain the DRAM energy *increase* that we see in Section 4.3.1. First, using Equation 4.5 we calculate a potentially optimal RAD value of 268 thread-blocks.

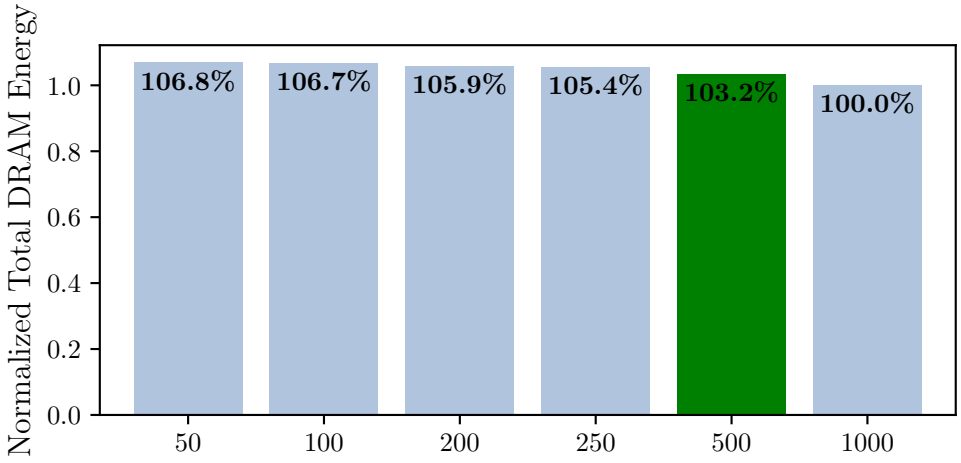
We see that in Figure 4.14a, which shows the normalized total DRAM accesses, LavaMD performs close to expectations. The observed optimum *in terms of DRAM accesses*, 250, is close to the esti-



(a) Normalized DRAM Accesses Sensitivity



(b) Normalized Execution Time Sensitivity



(c) Normalized DRAM Energy Sensitivity

Figure 4.14: LavaMD Run-Ahead Distance (RAD) sensitivity study. X-axis shows RAD in GPU thread-blocks. Result of optimal RAD (Min Energy) shown in green.

mate. In Figure 4.14, which presents the normalized execution time, we see that performance is basically flat, with fluctuations in execution time under 1%. Again, this is due to the large difference in execution time between the CPU producer and GPU consumer stages. The GPU kernel, when run in total, takes 52 times as long as the CPU stage, leaving little execution time to pipeline. And, since this producer-consumer relationship consists entirely of CG_WR dependence, the latency tolerant GPU's performance does not improve with reduced DRAM accesses.

Figure 4.14c reports the normalized DRAM energy for LavaMD. Here, the observed "optimum" is 500 at an energy *increase* of 3%. As we discussed in Section 4.3.1 LavaMD includes a stencil calculation in which there is significant reuse between GPU threads and CPU iterations. This causes a sharp increase in the DRAM's activation and precharge background energies that outweigh the energy saved from the reduction in DRAM accesses.

LavaMD shows that our technique is effective at reducing DRAM accesses, but that for some workloads DRAM energy management may

be more complex.

4.3.2.7 SmithWa Sensitivity Study

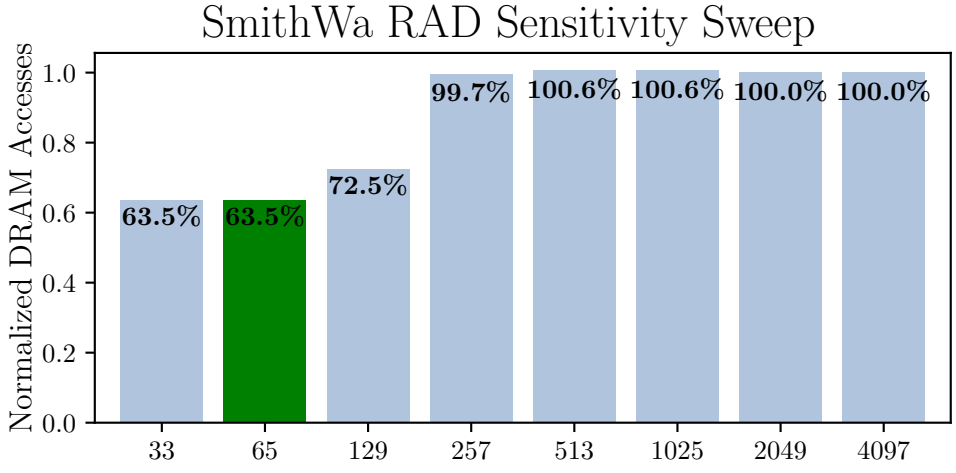
Pattern	Stage Footprints				
	S0	S1	TPB	FPT	RAD
GC	37	36	512	73	112

Table 4.9: SmithWa Profile. Each stage's footprint is in bytes per thread (or bytes per CPU iteration). Using Equation 4.5, the threads-per-block (TPB) and footprint-per-thread (FPT) are used to calculate potentially optimal run-ahead distance (RAD) in terms of GPU thread-blocks using 4 MB LLC size in our configuration.

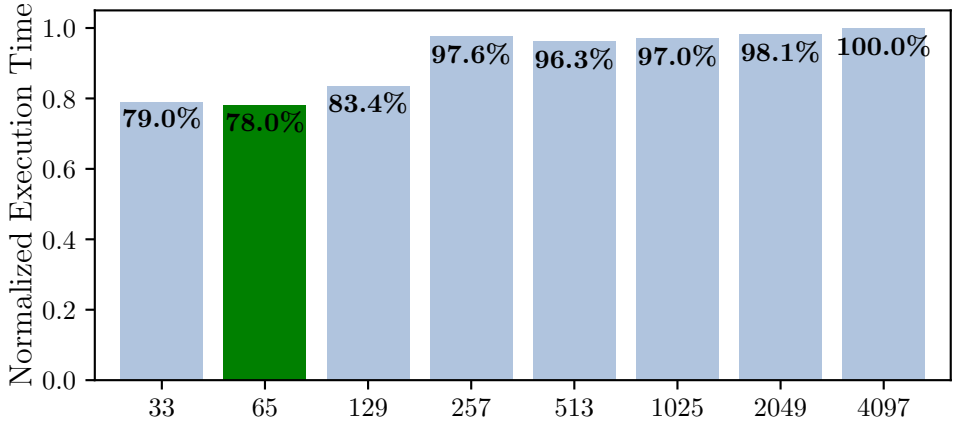
SmithWa's profile is reported in Table 4.9. It is another two stage GPU-to-CPU workload with a FPT of 73 Bytes. Using Equation 4.5, we estimate an optimal RAD value of 112 thread-blocks.

Figure 4.15 shows the results of the sensitivity study. Figure 4.15c reports the normalized DRAM energy usage for Smithwa. Of the RAD values we tested we observe that 65 thread-blocks, the largest RAD not exceeding our estimate, achieves the lowest total DRAM energy usage at 65% of the serial case (4097 thread-blocks). 33 thread-blocks achieves a similar energy footprint, and 129 thread-blocks achieves savings, though not as much as the 65 thread-blocks case.

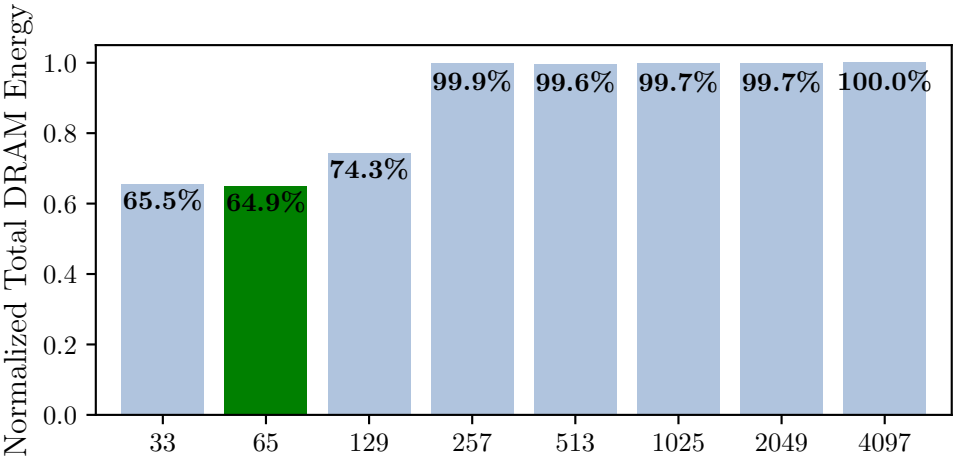
This is supported by Figure 4.15a which reports the normalized



(a) Normalized DRAM Accesses Sensitivity



(b) Normalized Execution Time Sensitivity



(c) Normalized DRAM Energy Sensitivity

Figure 4.15: SmithWa Run-Ahead Distance (RAD) sensitivity study. X-axis shows RAD in GPU thread-blocks. Result of optimal RAD (Minimum Energy) shown in green.

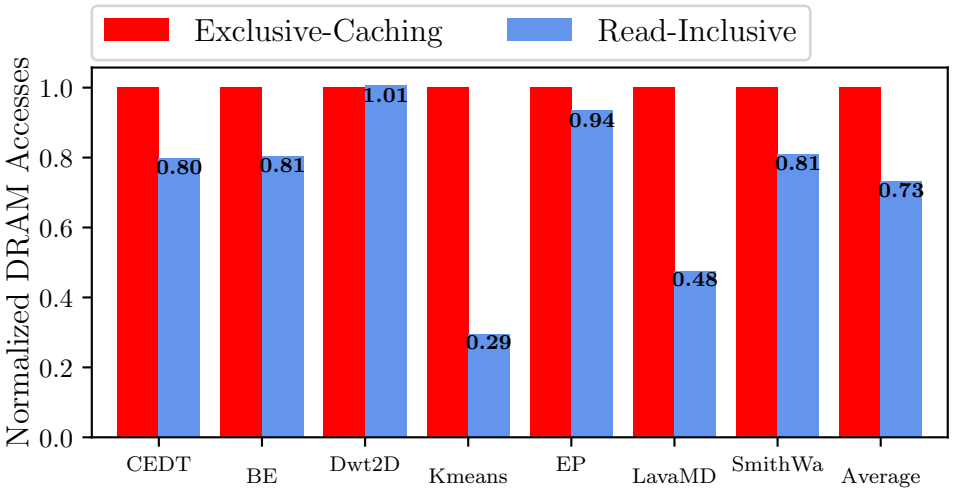
DRAM accesses for SmithWa. 33 and 65 thread-blocks, as expected, achieve the largest savings at 64% and 63% of the serial case, respectively. Meanwhile the 129 thread-blocks RAD bar achieves 72% of the serial case. Even though the cache-footprint at this RAD value, 4.6 MB, is larger than the size of the LLC, the CPU's L2 provides extra space. However, only a portion of the CPU's footprint can take advantage of this extra space.

SmithWa's dependency pattern as shown in Figure 4.8 consists of 22.2% GC_RW and 77.8% GC_WR. In order for the CPU's writeback L2 cache to provide extra buffer to supplement the LLC, the LLC must be exclusive of the L2. For the overall performance of our technique, it is important that the LLC has read-inclusivity. We will discuss the performance implications of read-inclusivity in depth in Section 4.3.3. However, in the context of the L2 providing extra room for our software pipeline, the 77.8% of the producer-consumer data that the CPU stage reads will be mirrored in the LLC, with the 22.2% of the data that the CPU writes sticking around in the CPU's L2 until it is written back.

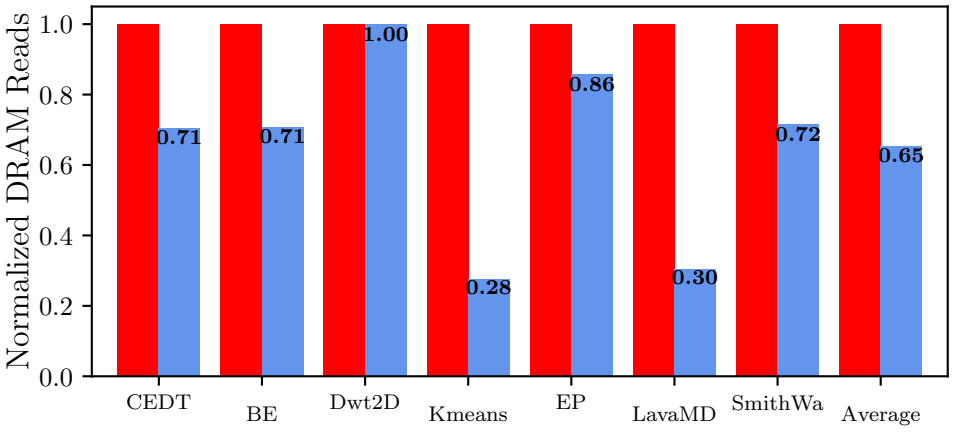
Finally, in Figure 4.15b, the we report the normalized execution time for SmithWa. The curve follows closely the curve of Figure 4.15a, the normalized DRAM accesses. SmithWa's consumer is a latency intolerant CPU, so reducing the accesses to DRAM for its shared data has a positive effect on performance. At the optimal RAD we achieve a 22% execution time reduction.

4.3.3 Read-Inclusivity

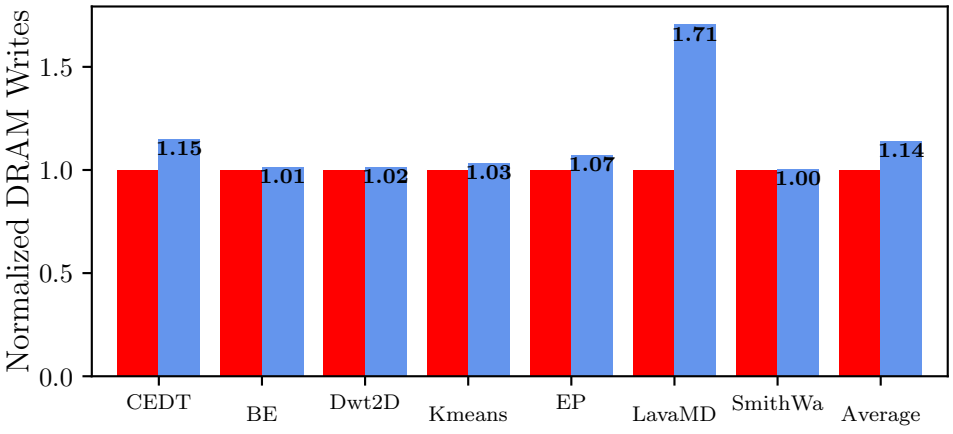
The LLC in our simulation does not cache Reads as they arrive from DRAM. Instead, data is filled into the LLC only when evicted from the CPU's L2 cache or when the GPU writes through to the LLC (*i.e.*the LLC is managed as a victim cache). For producer-consumer dependences where the producer is a Read, this presents a problem that can degrade effectiveness of our technique. For one, in our simulation the GPU's L2 cache does not write back clean lines to the LLC upon eviction. This means that data brought into the cache system by a GPU producer's read will not get written to the LLC, effectively rendering it lost and providing no benefit to the CPU consumer that follows it. The CPU L2's cache in our simulation evicts clean



(a) Normalized Total DRAM Accesses



(b) Normalized DRAM Reads



(c) Normalized DRAM Writes

Figure 4.16: DRAM Accesses, DRAM Reads, and DRAM Writes of the optimal Run-ahead Distance under a read-inclusive LLC caching policy normalized to an exclusive LLC caching policy (Red).

lines to the LLC. However, Reads to a line by a consumer following a CPU-Read producer that occur before that line has been written back will trigger a speculative read to DRAM in our simulation. Since coherence for clean lines is easy to enforce, managing the LLC with a "Read-Inclusivity" policy, *i.e.*filling the LLC with lines brought into the cache system by Reads upon arrival from DRAM, makes sense.

Figure 4.16a shows the savings in DRAM accesses we achieved with a Read-Inclusivity policy. This can be broken down into DRAM Reads (Figure 4.16b) and DRAM Writes (Figure 4.16c). The blue bars show the value of the optimal RAD presented in Section 4.3.1 under a read-inclusive policy normalized to the same RAD value with an exclusive-caching policy (the red bars).

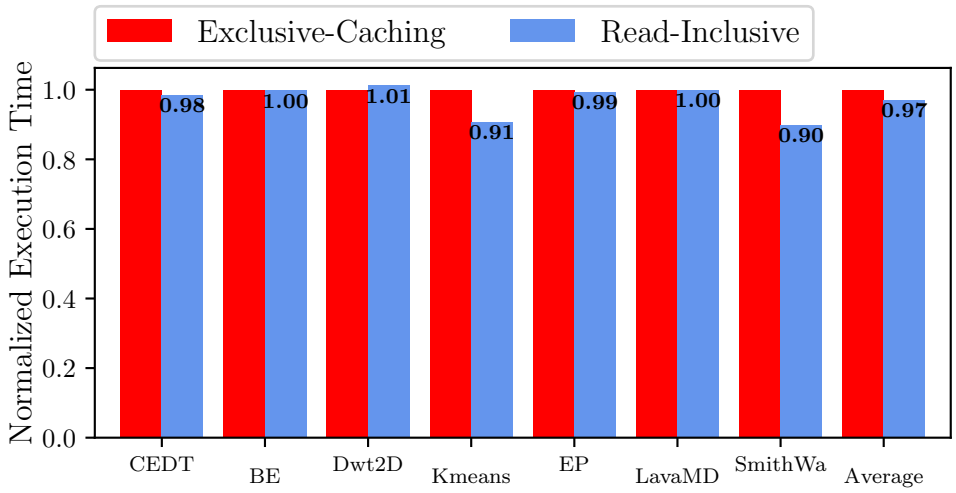
On average, read-inclusivity achieves reductions of 26.7% and 34.7% for total DRAM accesses and for DRAM reads, respectively. A natural consequence of filling the LLC with more data is that more blocks get evicted. Thus, read-inclusivity *increases* writes to DRAM by 14.2%.

For some benchmarks, read-inclusivity is integral to the temporal

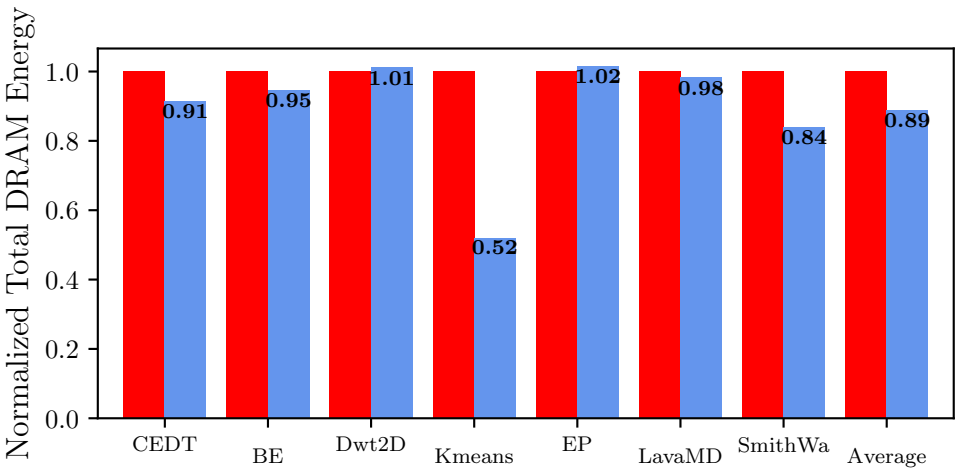
locality improvement our technique achieves for producer-consumer sharing. As shown in Figure 4.8 CEDT, Kmeans, and SmithWaterman all have significant read-first producer-consumer sharing patterns. They achieve 20.0%, 70.5%, and 19.0% reductions, respectively. These savings, and Kmeans' impressive drop in particular, show the importance of read-inclusivity for our technique.

BE, EP, and LavaMD all show significant savings as well. However, these savings are due to reuse of blocks by the GPU and not by sharing between stages (save for EP which has a small amount of read-first transactions, outweighed by the GPU's temporal locality). Finally, DWT2D achieves a small 0.7% increase in total DRAM accesses, having no producer-consumer temporal locality, nor typical temporal locality in its access pattern.

Figure 4.17a shows the ultimate effect that read inclusivity has on execution time and energy for our workloads. On average, read inclusivity reduces execution time by 3.0% Benchmarks with primarily CPU consumer patterns - Kmeans and SmithWa - achieve execution time reductions of 9.3% and 10.1%. A small fraction of EP



(a) Normalized Execution Time



(b) Normalized Total DRAM Energy

Figure 4.17: Execution Time and Total DRAM Energy of the optimal Run-ahead Distance under a read-inclusive LLC caching policy normalized to an exclusive LLC caching policy (Red).

and CEDT's accesses are CPU consumers and they achieve 0.7% and 1.6% reductions, respectively. BE and LavaMD, being GPU consumer pipelines, receive no execution time reduction despite significant DRAM access reductions as a consequence of the latency tolerant GPU. Dwt2D actually increases in execution time slightly, a 1.2% increase.

Figure 4.17b shows the DRAM energy benefits of Read Inclusivity. On average, by choosing this policy we achieve an 11.0% reduction in DRAM energy usage averaged across all the workloads. Kmeans achieves an impressive 48.2% savings, with CEDT and SmithWa following at 8.6% and 16.0%, respectively. BE and LavaMD get 5.3% and 1.7% savings, respectively, from the significant reuse they exhibit. Dwt2D again shows a slight 1.1% increase, and EP shows a 1.6% increase in energy usage despite its 0.6% DRAM access reduction due to primarily DRAM precharge background energy increasing.

4.4 Related Work

Hestness *et al.* find that pipelining benchmarks with GPU-kernel-synchronous characteristics can increase performance in a number of interesting ways. Indeed, they mention pipelining leads to coordinated use of cache capacity, saving off-chip memory accesses contributing to an increase in performance for some of their examined benchmarks. They do not however study the effect in detail, nor focus on controlling their pipeline width for redundant access elimination [45].

Work by Kim *et al.* recognizes that GPGPU workloads may consist of multiple dependent stages that include CPU, GPU kernels, I/O, and copies that constitute pipeline parallelism [55]. They introduces several optimizations in the hardware and virtual memory system to automatically schedule GPU cooperative thread arrays (thread-blocks in our work), based on their dependence relationships with other stages. Rather than study integrated heterogeneous microprocessors, it investigates these pipeline optimizations for discrete GPGPU platforms. In contrast, our work studies integrated chips, and focuses specifically on saving energy by reducing superfluous DRAM

accesses.

Kayi *et al.*, and Cheng *et al.* both dynamically detect producer-consumer sharing in multiprocessors and come up with coherence protocol optimizations to programs exhibiting producer-consumer sharing [56, 57]. They do not examine GPUs and the complexities they introduce to coherence and producer-consumer sharing.

Several benchmark suites have been developed in recent years to provide suitable programs to test heterogeneous chips. Previously researchers needed to adapt CPU and Traditional GPU benchmarks to glean insights about heterogeneous chips. These suits contain benchmarks which exhibit sharing, producer-consumer relationships, synchronization and more [13, 44, 50].

4.5 Conclusions

In this chapter we presented a novel locality transformation, Heterogeneous Pipeline Scheduling for Cache, to improve the DRAM access profile and in some cases execution time profile of workloads that contain producer-consumer data dependencies. Instead of scheduling

large heterogeneous producer-consumer data pipelines serially, which cause spills of shared data out of the on-chip cache and into DRAM, we schedule them in a software pipeline. By blocking our pipeline with the proper size, or run-ahead distance, we are able to keep shared data on chip. In this way we save energy from unneeded DRAM accesses, and in some cases improve performance.

We implement our technique on gem5 with a thread-block dispatcher based synchronization method. Implementation of this technique for specific workloads and systems will depend on the amount of shared cache available and the amount of data-per-thread that the heterogeneous workloads generate. Practical implementation for a cadre of larger data-per-thread applications or hardware with smaller shared caches may necessitate a synchronization scheme that is able to gainfully execute at finer granularities, with lower overhead. We found that for the workloads we evaluated, a GPU thread-block level of granularity allowed us to achieve performance improvements in the aggregate.

By a simple profiling of our workloads we estimated the opti-

mal run-ahead distance, and observed how these estimates performed with a run-ahead distance sensitivity study. We showed how different access dependencies (read-write pairings) and heterogeneity can influence the DRAM access profile, execution time savings, and DRAM energy savings. Such profiling could be performed by a compiler, providing performance hints to the run-time system controlling the software pipeline. Facilitating automatic optimizations for heterogeneous produce-consumer data sharing applications.

Chapter 5: Conclusion and Future Work

Recent trends in computing have elevated GPUs' importance greatly, with GPUs becoming ubiquitous in many platforms. Not only in massive high-performance GPGPU computing rigs, but also in lower power mobile and "edge" devices. Traditional programming paradigms associated with GPGPU computing, namely the serial scheduling of "embarrassingly parallel" kernels, continue to see success in discrete GPU systems. This thesis meanwhile, explores how GPGPU programming for integrated CPU-GPU devices in low-power systems requires a more nuanced approach to fully utilize the heterogeneous hardware and efficiently parallelize more types of codes of more varied complexity. We proposed two techniques that optimize GPGPU programs for integrated heterogeneous CPU-GPU microprocessors.

The gradual integration of CPUs and GPUs from discrete com-

puting entities, to parts of the same system-on-a-chip have imparted advantages in the communication of data between CPU code and GPU kernels. Because of the improved physical proximity of the two core types, data that travels between them is able to stay on chip in a coherent cache system, reducing both the performance cost of a transfer, and the energy cost associated with off-chip DRAM access.

First, in Chapter 3, by leveraging low-cost on-chip data communication to enable low-latency kernel launches, we are able to gainfully execute much smaller granularities of parallelism on the GPU. Our technique "Nested MIMD-SIMD Parallelization" extracts fine grain parallel "SIMD" loops nested within complex, irregular MIMD loops and executes them as kernels on the GPU. By launching multiple kernels from unique threads and overlapping execution, we are able to increase temporal and spatial utilization of the GPU, while simultaneously utilizing CPU compute resources in concert. We evaluate our technique on both a cycle-accurate simulator, gem5-gpu, and real hardware, showing significant speedups in each.

Second, in Chapter 4 we propose a locality transformation called

Pipelined CPU-GPU Scheduling for Caches, that intelligently schedules heterogeneous producer-consumer compute relationships, keeping their data inside the on-chip cache. By restricting producers to only running ahead of their consumer partners by a certain distance, and executing them as a concurrent software pipeline, we limit the cache footprint of each stage in the pipeline. Scheduling the software pipeline with this "Run-ahead distance" sized appropriately reduces the number of accesses to off-chip DRAM saving significant energy. The improved cache locality of these accesses proves a boon to the performance of CPU consumer stages in the pipeline, as the CPU is sensitive to access latency. This compounds with the natural performance improvement of overlapping computation in a software pipeline, and decreases DRAM energy further by decreasing background DRAM energy. We evaluate our technique with seven benchmarks on a cycle-accurate simulator, gem5.

5.1 Future Work

In both of the techniques that we propose and evaluate in this thesis, there is an element of software engineering necessary to realize performance gains. In other words we needed to do some rewriting of code by hand. Though much of our work was made possible by semi-automated processes like the low-latency kernel launch daemon in Section 3.5 and the software runtime system described in Section 4.2, we opted to forgo full automation of the coding process in favor of a deeper understanding of the performance of our techniques. An important direction therefore of future work is to automate this process, or make it significantly easier inside the compiler, possibly with the assistance of compiler directives such as in OpenMP [11]. Compounded with the automated compiler optimization, an lightweight runtime system layer could manage aspects like concurrent kernel launches, and pipeline operation.

The performance benefits of the locality transformation we propose in Chapter 4 depend on choosing a run-ahead distance that will sufficiently limit the overall cache footprint of the pipeline, as to keep

producer-consumer data on chip. In this research we achieve this by sweeping possible RAD values, and choosing the optimal RAD based on DRAM energy usage. We also propose a static prediction that can be calculated by profiling a workload and using Equation 4.5, something we do by hand. This profiling could be done by a compiler, and relevant data fed to the runtime system during execution. Another, further direction for choosing the best RAD could be to use performance counters to track LLC miss rates and other relevant statistics and adjusting the RAD during runtime.

Bibliography

[1] Anthony Danalis, Gabriel Marin, Collin McCurdy, Jeremy S. Meredith, Philip C. Roth, Kyle Spafford, Vinod Tipparaju, and Jeffrey S. Vetter. The Scalable HeterOgeneous Computing (SHOC) Benchmark Suite. In *Proceedings of the 3rd Workshop on General-Purpose Computation on Graphics Processing Units*, March 2010.

[2] Mark Harris. Unified memory in cuda 6, Nov 2013.

[3] Chris Gregg and Kim Hazelwood. Where is the Data? Why You Cannot Debate CPU vs. GPU Performance Without the Answer. In *Proceedings of the International Symposium on Performance Analysis of Systems and Software*, April 2011.

[4] Intel Corporation. Intel Sandy Bridge Microarchitecture.", Institution="Intel", Number="*http://www.intel.com.* Santa Clara, CA.

[5] N. Brookwood. AMD Fusion Family of APUs: Enabling a Superior, Immersive PC Experience. AMD White Paper. 2010.

[6] Apple Inc. A12 Bionic: The Smartest, Most Powerful Chip in a Smartphone.", Institution="Apple", Number="*https://www.apple.com/iphone-xs/a12-bionic.* Cupertino, CA.

[7] Jin Wang, Norm Rubin, Albert Sidelnik, and Sudhakar Yalamanchili. Dynamic Thread Block Launch: A Lightweight Execution Mechanism to Support Irregular Applications on GPUs. In *Proceedings of the International Symposium on Computer Architecture*, Portland, OR, June 2015.

[8] Marisabel Guevara, Chris Gregg, Kim Hazelwood, and Kevin Skadron. Enabling Task Parallelism in the CUDA Scheduler. In *Proceedings of the Workshop on Programming Models for Emerging Architectures held in conjunction with the Symposium on Parallel Architectures and Compilation Techniques*, September 2009.

[9] Lingyuan Wang, Miaoqing Huang, and Tarek El-Ghazawi. Towards Efficient GPU Sharing on Multicore Processors. *Performance Evaluation Review*, 40(2), September 2012.

[10] Daniel Gerzhoy, Xiaowu Sun, Michael Zuzak, and Donald Yeung. Nested mimd-simd parallelization for heterogeneous microprocessors. *ACM Transactions on Architecture and Code Optimization*, 16:1–27, 12 2019.

[11] The OpenMP API Specification for Parallel Programming. Intel Corporation. *http://www.openmp.org/wp/*. 2014.

[12] Shuai Che, Jeremy W. Sheaffer, Michael Boyer, Lukasz G. Szafaryn, Liang Wang, and Kevin Skadron. A Characterization of the Rodinia Benchmark Suite with Comparison to Contemporary CMP Workloads. In *Proceedings of the International Symposium on Workload Characterization*, December 2010.

[13] Y. Sun, X. Gong, A. K. Ziabari, L. Yu, X. Li, S. Mukherjee, C. Mccardwell, A. Villegas, and D. Kaeli. Hetero-mark, a benchmark suite for cpu-gpu collaborative computing. In *2016 IEEE International Symposium on Workload Characterization (IISWC)*, pages 1–10, 2016.

[14] John D Owens, Mike Houston, David Luebke, Simon Green, John E Stone, and James C Phillips. Gpu computing. *Proceedings of the IEEE*, 96(5):879–899, 2008.

[15] Y. Kim, H. Kwon, S. Doo, M. Ahn, Y. Kim, Y. Lee, D. Kang, S. Do, C. Lee, G. Cho, J. Park, J. Kim, K. Park, S. Oh, S. Lee, J. Yu, K. Yu, C. Jeon, S. Kim, H. Park, J. Lee, S. Cho, K. Park, Y. Kim, Y. Seo, C. Shin, C. Lee, S. Bang, Y. Park, S. Choi, B. Kim, G. Han, S. Bae, H. Kwon, J. Choi, Y. Sohn, K. Park, S. Jang, and G. Jin. A 16-gb, 18-gb/s/pin gddr6 dram with per-bit trainable single-ended dfe and pll-less clocking. *IEEE Journal of Solid-State Circuits*, 54(1):197–209, 2019.

[16] John A. Stratton, Christopher Rodrigues, I-Jui Sung, Nady Obeid, Li-Wen Chang, Nasser Anssari, Geng Daniel Liu, and Wen mei Hwu. The Parboil Technical Report. March 2012.

[17] OpenMP Source Code Repository. *http://www.pcg.ull.es/ompscr/*. 2004.

[18] SPEC OMP 2001. *https://www.spec.org/omp2001/*. 2001.

[19] Steven Muchnick. *Advanced Compiler Design and Implementation*. Morgan Kaufmann, San Francisco, CA, 1997.

[20] Michael Kruse and Hal Finkel. Loop Optimization Framework. Technical Report 1811.00632, arXiv, November 2018.

[21] SPEC OMP 2012. *https://www.spec.org/omp2012/*. 2012.

[22] Michael Kruse and Tobias Grosser. DeLICM: Scalar Dependence Removal at Zero Memory Cost. In *Proceedings of the International Symposium on Code Generation and Optimization*, February 2018.

[23] Jason Power, Joel Hestness, Mar S. Orr, Mark D. Hill, and David A. Wood. gem5-gpu: A Heterogeneous CPU-GPU Simulator. *Computer Architecture Letters*, 13(1), January 2014.

[24] Intel Core i7 - 6770HQ Processor. *https://ark.intel.com/products/93341/Intel-Core-i7-6770HQ-Processor-6M-Cache-up-to-3_50-GHz*.

[25] Daniel Lustig and Margaret Martonosi. Reducing GPU Offload Latency via Fine-Grained CPU-GPU Synchronization. In *Proceedings of the International Symposium on High Performance Computer Architecture*, June 2013.

[26] gem5 M5threads. *https://github.com/gem5/m5threads*. 2009.

[27] Neil Trevett. Opencl introduction. *Khronos Group*, 2013.

[28] Michael Mrozek and Zbigniew Zdanowicz. GPU Daemon: Road to Zero Cost Submission. In *Proceedings of the 4th International Workshop on OpenCL*, April 2016.

[29] Mayank Daga, Ashwin M. Aji, and Wu chun Feng. On the Efficacy of a Fused CPU+GPU Processor (or APU) for Parallel Computing. In *Proceedings of the Symposium on Application Accelerators in High-Performance Computing*, July 2011.

[30] Kyle Spafford, Jeremy S. Meredith, Seyong Lee, Dong Li, Philip C. Roth, and Jeffrey S. Vetter. The Tradeoffs of Fused Memory Hierarchies in Heterogeneous Computing Architectures. In *Proceedings of the ACM International Conference on Computing Frontiers*, May 2012.

[31] J. Dongarra and P. Luszczek. Introduction to the HPC Challenge Benchmark Suite. Technical report, University of Tennessee-Knoxville, 2005.

[32] Manish Arora, Siddhartha Nath, Subhra Mazumdar, Scott B. Baden, and Dean M. Tullsen. Redefining the Role of the CPU in the Era of CPU-GPU Integration. *IEEE MICRO*, November/December 2012.

[33] Standard Performance Evaluation Corporation. *http://www.spec.org/benchmarks.html*. 2015.

[34] Michael Zuzak and Donald Yeung. Exploiting Multi-Loop Parallelism on Heterogeneous Microprocessors. In *Proceedings of the 10th International Workshop on Programmability and Architectures for Heterogeneous Multicores*, January 2017.

[35] Rashid Kaleem, Rajkishore Barik, Tatiana Shpeisman, Brian T. Lewis, Chunling Hu, and Keshave Pingali. Adaptive Heterogeneous Scheduling for Integrated GPUs. In *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques*, August 2014.

[36] Vignesh T. Ravi and Gagan Agrawal. A Dynamic Scheduling Framework for Emerging Heterogeneous Systems. In *Proceedings of the 18th International Conference on High Performance Computing*, December 2011.

[37] Vignesh T. Ravi, Wenjing Ma, David Chiu, and Gagan Agrawal. Compiler and Runtime Support for Enabling Generalized Reduction Computations on Heterogeneous Parallel Configurations. In *Proceedings of the International Conference on Supercomuting*, June 2010.

[38] Florian Wende, Frank Cordes, and Thomas Steinke. On Improving the Performance of Multi-threaded CUDA Applications with Concurrent Kernel Execution by Kernel Reordering. In *Proceedings of the 2012 Symposium on Application Accelerators in High Performance Computing*, July 2012.

[39] Guray Ozen. *Compiler and Runtime Based Parallelization & Optimization for GPUs*. PhD thesis, Universitat Politecnica de Catalunya (UPC), November 2017.

[40] Yi Yang and Huiyang Zhou. CUDA-NP: Realizing Nested Thread-Level Parallelism in GPGPU Applications. In *Proceedings of the 19th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, February 2014.

[41] Franz Franchetti, Stefan Kral, Juergen Lorenz, and Christopher W. Ueberhuber. Efficient Utilization of SIMD Extensions. *Proceedings of the IEEE*, 93(2), February 2005.

[42] Vijay Nagarajan, Daniel J Sorin, Mark D Hill, and David A Wood. A primer on memory consistency and cache coherence. *Synthesis Lectures on Computer Architecture*, 15(1):1–294, 2020.

[43] Anthony Gutierrez, Sooraj Puthoor, Brad Beckmann, and Tuan Ta. The amd gem5 apu simulator: Modeling gpus using the machine isa, 2018.

[44] Juan Gómez-Luna, Izzat El Hajj, Victor Chang, Li-Wen Garcia-Flores, Simon Garcia de Gonzalo, Thomas Jablin, Antonio J Pena, and Wen-mei Hwu. Chai: Collaborative heterogeneous applications for integrated-architectures. In *Performance Analysis of Systems and Software (ISPASS), 2017 IEEE International Symposium on*. IEEE, 2017.

[45] Joel Hestness, Stephen Keckler, and David Wood. Gpu computing pipeline inefficiencies and optimization opportunities in heterogeneous cpu-gpu processors. pages 87–97, 10 2015.

[46] Jason Lowe-Power, Arkaprava Basu, Junli Gu, Sooraj Puthoor, Bradford Beckmann, Mark Hill, Steven Reinhardt, and David Wood. Heterogeneous system coherence for integrated cpu-gpu systems. pages 457–467, 12 2013.

[47] N. Agarwal, D. Nellans, E. Ebrahimi, T. F. Wenisch, J. Danskin, and S. W. Keckler. Selective gpu caches to eliminate cpu-gpu hw cache coherence. In *2016 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 494–506, 2016.

[48] N. P. Jouppi. Improving direct-mapped cache performance by the addition of a small fully-associative cache and prefetch buffers. In *[1990] Proceedings. The 17th Annual International Symposium on Computer Architecture*, pages 364–373, 1990.

[49] N. Aggarwal, J. F. Cantin, M. H. Lipasti, and J. E. Smith. Power-efficient dram speculation. In *2008 IEEE 14th International Symposium on High Performance Computer Architecture*, pages 317–328, 2008.

[50] M. D. Sinclair, J. Alsop, and S. V. Adve. Heterosync: A benchmark suite for fine-grained synchronization on tightly coupled gpus. In *2017 IEEE International Symposium on Workload Characterization (IISWC)*, pages 239–249, 2017.

[51] H. Wu, G. Diamos, J. Wang, S. Cadambi, S. Yalamanchili, and S. Chakradhar. Optimizing data warehousing applications for gpus using kernel fusion/fission. In *2012 IEEE 26th International Parallel and Distributed Processing Symposium Workshops PhD Forum*, pages 2433–2442, 2012.

[52] Jason Lowe-Power and Matt Sinclair. Re-gem5: Building sustainable research infrastructure, Sep 2019.

[53] Phil Rogers and A Fellow. Heterogeneous system architecture overview. In *Hot Chips Symposium*, pages 1–41, 2013.

[54] Karthik Chandrasekar, Christian Weis, Yonghui Li, Benny Akesson, Norbert Wehn, and Kees Goossens. Drampower: Open-source dram power & energy estimation tool. *URL: http://www. drampower. info*, 22, 2012.

[55] Gwangsun Kim, Jiyun Jeong, John Kim, and Mark Stephenson. Automatically exploiting implicit pipeline parallelism from multiple dependent kernels for gpus. In *Proceedings of the 2016 International Conference on Parallel Architectures and Compilation*, pages 341–352, 2016.

[56] L. Cheng, J. B. Carter, and D. Dai. An adaptive cache coherence protocol optimized for producer-consumer sharing. In *2007 IEEE 13th International Symposium on High Performance Computer Architecture*, pages 328–339, 2007.

[57] A. Kayi, O. Serres, and T. El-Ghazawi. Adaptive cache coherence mechanisms with producer–consumer sharing optimization for chip multiprocessors. *IEEE Transactions on Computers*, 64(2):316–328, 2015.