UNIVERSITY OF CALIFORNIA
SANTA CRUZ

OPTIMIZATIONS FOR ENERGY EFFICIENCY IN GPGPU ARCHITECTURES

A dissertation submitted in partial satisfaction of the
requirements for the degree of

DOCTOR OF PHILOSOPHY

in

COMPUTER ENGINEERING

by

Alamelu Sankaranarayanan

March 2016

The Dissertation of
Alamelu Sankaranarayanan is approved:

Professor Jose Renau, Co-Chair

Professor José Luis Briz, Co-Chair

Professor Matthew Guthaus

Professor Jishen Zhao

Tyrus Miller
Vice Provost and Dean of Graduate Studies

Table of Contents

List of Figures vi

List of Tables viii

Abstract ix

Acknowledgments xi

1 Introduction 1

2 Background 4
- 2.1 Evolution of GPUs . . . 6
- 2.2 The CUDA Programming Model . . . 7
- 2.3 Our Baseline GPGPU . . . 11

3 Simulation Methodology 13
- 3.1 The Need for Yet Another Simulator . . . 13
- 3.2 Functional Emulation of GPGPUs . . . 17
  - 3.2.1 Native Coexecution . . . 17
  - 3.2.2 Application Contamination . . . 18
  - 3.2.3 Trace Generation . . . 18
- 3.3 Modeling GPGPUs . . . 19
  - 3.3.1 Thread Management . . . 20
  - 3.3.2 Warp Scheduling and Context Management . . . 20
  - 3.3.3 Modeling Divergence . . . 20
  - 3.3.4 Spatial Sampling . . . 21
- 3.4 Performance Modeling . . . 22
- 3.5 Estimation of Energy and Power . . . 24
- 3.6 Summary . . . 24

4 Eliminating Redundant Memory Copies in a Heterogeneous CPU-GPU Architecture **25**
4.1 The Memory Transfer Problem . . . 26
4.2 Assessing the Need for Coherence . . . 29
4.3 Related Work . . . 30
4.3.1 Avoiding Memory Copies with Pinned Memory . . . 30
4.3.2 Avoiding Memory Copies with a Shared Virtual Space . . . 30
4.3.3 Integrated CPU-GPU Systems . . . 31
4.3.4 SoC Virtual Memory and Memory Copy Support . . . 32
4.4 Our Proposal: FuseTLB . . . 33
4.4.1 A Virtually Indexed and Virtually Tagged SM Memory Hierarchy . . . 34
4.4.2 The Need For a Shared L2 Cache in the GPU . . . 35
4.4.3 A Shared TLB between the CPU and the GPU . . . 36
4.4.4 Writeback Invalidating the SM-Caches . . . 36
4.4.5 Avoiding Coherence in the SMs . . . 37
4.4.6 Binary and OS Compatibility . . . 38
4.5 Experimental Setup . . . 39
4.6 Evaluation . . . 41
4.6.1 GPU Memory Hierarchy Energy . . . 42
4.6.2 Overall Performance . . . 46
4.6.3 Multiprogrammed Workloads . . . 50
4.6.4 TLB Characterization . . . 51
4.7 Summary . . . 53

5 An Energy Efficient GPGPU Memory Hierarchy with Tiny Incoherent Caches **55**
5.1 Common Approaches to Energy Efficiency . . . 56
5.2 Feasibility of Adding Filter Caches . . . 58
5.3 Our Proposal: Add a tinyCache per Lane . . . 59
5.3.1 Behavior of the tinyCache . . . 60
5.3.2 Maintaining Coherence . . . 62
5.3.3 Area Overhead . . . 64
5.4 Experimental Setup . . . 64
5.5 Evaluation . . . 66
5.5.1 Main Results . . . 66
5.5.2 Sizing the Tinycache . . . 72
5.6 Summary . . . 73

6 EESI: A Simple Architecture for GPGPU and CPU Workloads **75**
6.1 Execution of Divergent Applications on a GPGPU . . . 76
6.2 Proposals to Counteract the Effect of Divergence . . . 78
6.2.1 Mechanisms Based on Modifications to the Reconvergence Stack . . . 79
6.2.2 Software Mechanisms to Minimize Divergence . . . 81
6.2.3 Alternative Architectures . . . 81

6.3 The EESI Architecture 83
6.3.1 CUDA/OpenCL Compatibility 85
6.3.2 Support for Warping 86
6.3.3 Divergence Policies 87
6.3.4 Executing CPU Applications 88
6.4 Experimental Setup 89
6.4.1 Simulation Infrastructure 89
6.4.2 Benchmarks 91
6.5 Evaluation 91
6.5.1 Main Results 92
6.5.2 Role of Warping Mechanisms 94
6.5.3 EESI-T vs EESI-M 97
6.5.4 Sizing the TinyICache 98
6.5.5 Area Overhead Estimations 99
6.6 Summary 101

7 Conclusions **102**

Bibliography **104**

List of Figures

2.1 Evolution of GPUs 6
2.2 The CUDA programming model 8
2.3 Baseline architecture 11

3.1 High level architecture of GPSim 15
3.2 A detailed look at the GPSim emulator module 16
3.3 A contaminated CUDA binary 19

4.1 Memory transfer to GPU compute time ratio 27
4.2 The FuseTLB architecture 34
4.3 Memory hierarchy energy breakdown for a typical SoC (typSoC). 42
4.4 Comparing coherence traffic between CPUs and GPUs 43
4.5 Impact of avoiding memory copies and coherence on energy 44
4.6 Energy distribution across various blocks in the baseline memory hierarchy . . 45
4.7 Energy Distribution across various blocks without a STLB 45
4.8 Impact on energy with and without a STLB 46
4.9 Impact of FuseTLB on performance 47
4.10 Impact of FuseTLB on LLC misses 48
4.11 Impact of avoiding memory copies on IPC 49
4.12 Fairness of the FuseTLB architecture running both CPU and GPU workloads. . 51
4.13 Impact on fairness with shared and private TLBs 51
4.14 Sizing the TLB for GPGPU workloads 52
4.15 Sizing the TLB for CPU workloads 52
4.16 Comparing private and shared TLBs 53

5.1 Energy consumption of the on-chip memory hierarchy on a GPGPU 56
5.2 Proposed architecture with tinyCaches 59
5.3 Cache line states and transitions as seen by the tinyCache controller 60
5.4 Impact on the energy consumption after the addition of a tinyCache per lane . . 67
5.5 Impact on the IPC after the addition of a tinyCache per lane 68
5.6 Impact on the ED after the addition of a tinyCache per lane 69
5.7 Breakdown of memory accesses for caches in the on-chip memory hierarchy . . 70

5.8 Impact of caching global and shared references in the tinyCaches on the ED . . 71
5.9 Impact of the size of the tinyCache on IPC 73
5.10 Impact of the size of the tinyCache on ED 73

6.1 Execution of a divergent kernel on a SIMT architecture 77
6.2 The EESI architecture 83
6.3 EESI-T vs EESI-M 87
6.4 Performance of GPGPU and MIMD workloads on the EESI architecture 93
6.5 EPI for GPGPU and MIMD workloads on the EESI architecture 93
6.6 Impact of various warp switching policies on IPC and EPI 96
6.7 Performance without a TCI 98
6.8 Sizing the TCI for optimal IPC and EPI 99

List of Tables

2.1 GPGPU terminology and their equivalent terms in CUDA and OpenCL 10

3.1 Configuration parameters affecting the functional emulation of GPGPUs 21
3.2 Default simulation parameters in GPSim. 23

4.1 GPU workloads used in our evaluation 27
4.2 GPSim configuration to evaluate FuseTLB 39
4.3 List of architectual configurations evaluated 40
4.4 List of SpecRate and GPGPU workloads 41

5.1 GPSim simulation parameters to evaluate tinyCaches 65
5.2 GPU workloads used in our evaluation. 66

6.1 Proposals to mitigate the effect of divergence in GPGPU applications 78
6.2 GPSim simulator configuration used to evaluate the EESI architecture 90
6.3 Workloads used to evaluate EESI 91
6.4 Divergence handling mechanisms 92
6.5 Triggers for switching to the next warp. 95

Abstract

Optimizations for energy efficiency in GPGPU architectures

by

Alamelu Sankaranarayanan

It is commonplace for graphics processing units or GPUs today to render extremely complex 3D scenes and textures, in real time, both in the traditional and mobile computing spaces. The computational power required to do this makes them a valuable resource to exploit for general purpose computation. In order to map programs originally designed for sequential CPUs onto massively parallel GPU architectures, it would be necessary to justify the transition with huge performance benefits. Over the last couple of years, there have been numerous proposals to improve the performance of GPUs used for general purpose computing (GPGPUs), but without much consideration for energy efficiency.

In my dissertation, I evaluate the feasibility of GPGPUs from an energy perspective and propose some optimizations based on the unique programming model used by GPGPUs. First, I describe the simulation infrastructure, one of the few available to model GPGPUs today, both individually and as part of a heterogeneous system. Next, I propose a design using a shared translation lookaside buffer (TLB) to eliminate chronic memory copies between the CPU and GPU addressing spaces, making heterogeneous CPU-GPU designs energy efficient. Furthermore, to improve the energy efficiency of the on-chip memory hierarchy, I propose adding tiny incoherent caches per processing element, which can filter out frequent accesses to large shared and energy-inefficient cache structures. Finally, I evaluate a design which moves away from

the underlying SIMD architecture of GPUs towards a more MIMD-like architecture, enabling the execution of both CPU and GPGPU workloads without negatively affecting the energy efficiency availed by traditional workloads on GPGPUs.

Acknowledgments

I am one of the lucky few who had two excellent researchers, Professor Jose Renau and Professor José Luis Briz, as my advisors in grad school. They have been great technical mentors, and I am immensely grateful for their guidance, their patience and their belief in me. Thank you Professor Renau and Dr Briz.

I am also extremely grateful to the reading committee for taking the time out of their busy schedule, to read my work and offering feedback. Special thanks to Professor Guthaus for his meticulous final touches to my thesis.

I cannot imagine dealing with the unpredictable swells of student life without having the support of my lab mates. Sangeetha, Rigo, David, Ehsan, Gabriel, Elnaz, Rafael, Blake, Sina and Daphne, I will forever cherish the conversations we have had, both inside and outside the lab. I am looking forward to hearing great things from each and every one of you.

I would also like to thank my amazing circle of friends and family, who knew it was no good asking me when I would graduate, and both my sets of parents, who never stopped reminding me I had to!

And last, but not the least, I would like to say my sincerest thanks to Harish and Murphy. I know that without your love, sense of humor, encouragement and understanding, none of this would have been possible.

Chapter 1

Introduction

The ubiquitous nature of graphic processing units (GPUs) combined with their computational prowess and tremendous memory bandwidth, made them a attractive resource for system designers and programmers to exploit. The advent of programming languages and standards like CUDA [1] and OpenCL [60] gave the necessary initial thrust for their mainstream use alongside CPUs, for sections of the code that were data parallel and that mapped well on the regular SIMD-like hardware. With each generation, GPUs got more flexible, and powerful, and consequently, today there is a significant amount of research conducted by the computer architecture community to offload more tasks to the GPU by boosting the performance, the energy efficiency and ease of use of GPGPUs, *i.e.*, for general purpose computation on graphic processing units.

GPGPUs evolved from a very large number of highly specialized, non programmable fixed function units, to larger numbers of more programmable units, to an even larger number of even more programmable units with shared memory. This evolution, unlike that of the CPU, which focused on the performance of a single pipeline, can be attributed to the nature of the applications GPUs were originally designed for, *i.e.*, simple and regular data parallel applications, with limited communication between a very large pool of threads. Programming models like CUDA and OpenCL allow us to express and use the GPU hardware for more general purpose tasks, and extract high performance because of the massive multithreading that the architecture supports, but to make this possible, there are several complex and potentially energy hungry components including but not restricted to large multibanked cache structures and logic to support the massive multithreading.

As more compute units are integrated onto chips, and technology scaling pushes the operating frequency higher, the power consumption of GPUs has skyrocketed. To make general purpose computing truly feasible on GPGPUs, it is therefore imperative to maximize the hardware utilization and improve the energy efficiency. In this dissertation, I identify, evaluate and propose optimizations that exploit the unique nature of programming models like CUDA and OpenCL, and typical GPGPU applications, to improve the on-chip energy efficiency in GPGPUs.

GPGPUs and applications that run on GPGPUs differ tremendously both in design and behavior from traditional CPUs, and it is not trivial to extend existing CPU simulators to model and study GPGPUs. Chapter 2 gives a brief overview on GPGPU architecture and the programming model associated with it, and Chapter 3 elaborates on the simulation infrastructure

that I developed and use to model GPGPUs.

As applications get more diverse, it is increasingly important to improve the coupling between the CPU and the GPU. The biggest, and most obvious bottleneck both in terms of performance, and the energy efficiency, is the chronic exchange of data between the two memory addressing spaces. Chapter 4 elaborates on how the uniqueness of the programming model allows us to mitigate these memory copies.

As we move closer towards the individual processing elements within a GPGPU, from the memory subsystem, the first level shared data cache strikes us an obvious choice for optimization for energy efficiency. Once again, exploiting the intricacies of the programming model can allow tiny incoherent caches higher up in the memory hierarchy which can filter out a significant number of requests to the shared DL1 cache, and thereby reduce the consumed energy. This is discussed in more detail in Chapter 5.

There have been several proposals to maximize the utilization of cores available on the GPU. A sizable number of these try to allay the effect of control divergence, which impacts utilization severely. Since control flows are integral to general purpose applications, handling them efficiently is critical. In Chapter 6, we take a step back, and propose a novel MIMD-like architecture to handle highly divergent workloads without incurring any performance or energy efficiency penalty. Finally, I conclude my dissertation in Chapter 7.

Chapter 2

Background

With the help of microarchitectural techniques like out-of-order execution, branch prediction, speculation, superscalar execution, favorable transistor-speed scaling, strides in memory technology, etc., computer architects were able to extract at least a 2X improvement with each generation of microprocessors until the early 2000s [14]. However, with Moore's law [58], Pollack's law [7] and Dennard's law [25] closing in, we very quickly hit the power wall and resorted to multiple processor cores to extract performance. At this point, a dichotomy of approaches emerged: One approach expounded using a few high performance superscalar cores that were highly latency optimized for sequential code performance *i.e.*, the *multicore* approach. These cores are typically tightly knit with shared memory subsystems, allowing them to exploit task parallelism as much as possible, and then exchange data and/or synchronize with each other as needed. This is an intuitive extension to our sequential way of thinking of writing programs, and has been widely adopted [5, 24, 42].

The other approach, known as the *many-core* approach, proposed using a sea of smaller and simpler cores, relying on the execution of a massive number of threads to hide latency and maximize execution throughput. Using shared memory to exchange or synchronize data frequently would impose a significant communication penalty owing to the large number of cores, which meant that the cores needed to execute largely independently. The cores could all either execute the same instruction on multiple data, *i.e.*, follow the SIMD execution style, or could execute multiple instructions, and operate on multiple data, *i.e.*, follow the MIMD execution style. The SIMD paradigm is extremely efficient for data parallel applications, while MIMD is better suited to a program composed of independent tasks. In spite of being extremely efficient at what it did, the many-core approach had to deal with two main obstacles for mainstream acceptance, the first being able to extract enough parallelism from traditionally sequential programs to make use of the available cores, and then some more to hide the latency. Secondly, it was notoriously hard to build compilers, and even write programs for these architectures, even without the requirement of scalability [41, 74].

The key to extracting performance from either strategy lies in how easy it is for software developers to harness the available computational power, and since the traditionally sequential programs mapped easier to multicore systems, the former approach met with more success than the latter, for a majority of the next decade. Manycore architectures manifested themselves in a smaller scale as different vector coprocessors for specific applications that were inherently data parallel, or to support vector extensions to the ISA [8, 29, 65], but most prominently as graphic processing units or GPUs.

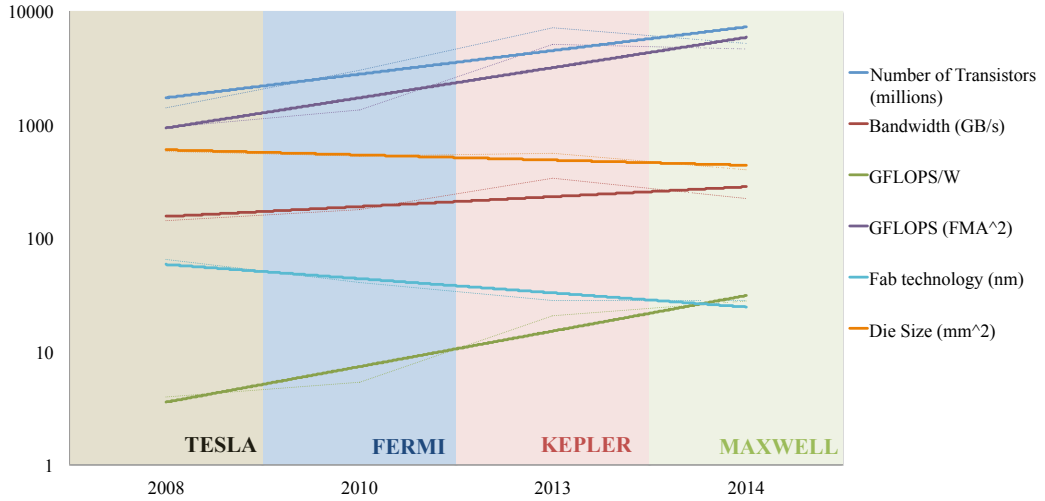


Figure 2.1: Evolution of GPUs

2.1 Evolution of GPUs

GPUs started out as a number of expensive fixed function pipelines that accepted vertices as input and output textures. The transition from 2D to 3D graphics and the rapid evolution of graphics algorithms soon demanded a programmable graphics pipeline, exposing the vertex and the fragment processor to graphics programmers. Graphics cards were soon being expected to render complex real time and 3D graphics, and GPU manufacturers started adding many more pipelines that were not only deeper with more capabilities and complexity, but also much cheaper owing to better and shrinking processor technology. As a result of all these factors, GPUs became more and more commonplace. At around the same time that uniprocessors hit the power wall (early 2000s), GPUs offered a few GFLOPS of compute power as we see in the evolution of GPUs highlighted in Figure 2.1.

It soon became obvious that GPUs, when not being used for graphics, could offer a

sizable boost to the performance of the main processor if it could offload suitable tasks to the GPU and take advantage of large amount of compute power available at its disposal. Early attempts to do this involved transforming data into GPU understandable vertices, issuing graphics commands in languages like OpenGL and Direct3D, and then reconverting the output back into a format that could be processed by the CPU. Needless to say, not only was this extremely tedious, but also it was not conducive to scaling or working across multiple generations of GPUs [63].

2.2 The CUDA Programming Model

To make the latent compute power of GPUs more accessible for use by non-graphics applications, there was a need for an alternate programming model that was

- Designed for highly parallel architectures at the very outset,
- Not tied in with the graphics pipeline (like OpenGL),
- Independent of the underlying graphics hardware, since GPUs evolved drastically, frequently, and more importantly, details of their implementation are not made public.

This led to a few significant attempts at abstracting highly parallel architectures like the Brooks paradigm, designed specifically for stream computing on graphics hardware [19], but it was the launch of the CUDA programming model by NVIDIA that catapulted GPUs firmly into the arena of general-purpose computation.

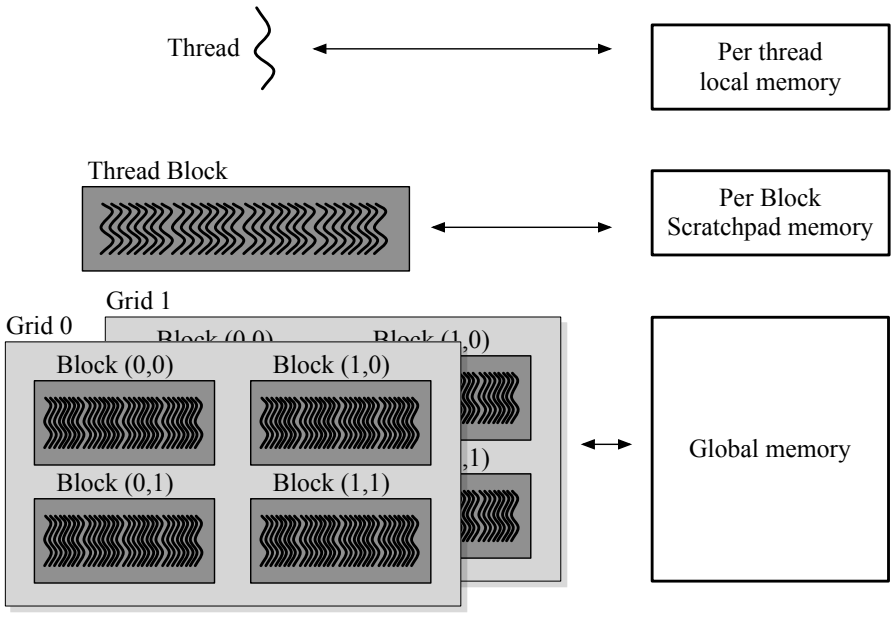


Figure 2.2: The CUDA programming model

Compute Unified Device Architecture or CUDA is a set of abstractions to leverage the compute hardware on NVIDIA GPUs. The three defining abstractions of CUDA are

- A hierarchy of threads,
- A hierarchy of memory shared at different levels, and
- Synchronization primitives like barriers.

These three abstractions are used to partition the program into smaller tasks that can be solved in parallel by blocks of threads, and these tasks, in turn, can be subdivided into sub-tasks that can be solved in parallel, and cooperate if needed using the shared memory and synchronization primitives. This abstraction does not need the hardware details to be exposed, and consequently is extremely scalable, as long as the underlying architecture is CUDA compatible.

A CUDA-compatible GPU (henceforth referred to in this thesis simply as a GPGPU) is imagined as a scalable array of multithreaded processors, known as *Streaming Multiprocessors* (SMs), each with a couple of processing elements known as *Scalar Processors* (SPs). A typical CUDA program, called a *kernel*, is a sequence of one or more highly parallel subroutines. A kernel can be visualized as a large set of parallel threads executing the same set of instructions. This large set of threads is divided into blocks of threads, each of which is assigned to an available SM. Since the model does not assume any particular number of SMs, the programmer has no idea when or where a block will be executed. The programming model does not allow any kind of direct synchronization between individual thread blocks, however, they may communicate by using atomic memory operations on *global memory*. This isolation between the thread blocks allows them to be scheduled in any order over a large number of cores.

Once a thread block is assigned to an SM, the SM creates smaller groups of threads known as warps. A warp of threads is executed concurrently on the SM with the threads starting at the same program address. A warp executes a common instruction at a time, so in case the threads diverge, the warp is executed serially. When the warp encounters a long latency instruction, the SM switches the warp out with another ready warp and strives to supply the SPs with a constant stream of ready instructions, as a result, maintaining high computational throughput. Threads within a thread block can cooperate using synchronization primitives like barriers and exchange information via *shared memory*.

The CUDA programming model allows access to different addressing spaces. Global Memory is a read and write memory whose contents are visible to all threads. On the first

Our terminology	CUDA	OpenCL
thread	thread	work item
(thread) block	thread block	work group
global memory	global memory	global memory
scratchpad	shared memory	local memory
constant memory	constant memory	constant memory
local memory	local memory	private memory
SM	Streaming Multiprocessor (SM)	Compute Unit
Lane	Scalar Processor (SP)	Processing Element (PE)
kernel	kernel	program

Table 2.1: GPGPU terminology and their equivalent terms in CUDA and OpenCL

generation of CUDA devices, global memory was the same as device memory but, after Fermi, global memory addresses could be cached closer within an SM. Shared memory is a high speed read/write memory whose contents are visible only by the threads within a block. It resides on the SM and is limited in size. The model also allows efficient reads from read-only texture and constant caches and has a private area for automatic variables. Apart from these, there is a super fast register file per SM, the contents of which are visible per thread only.

Following the success of CUDA to encourage more general-purpose computation on GPGPUs, a consortium was established by other GPU manufacturers that led to the development of the OpenCL standard as well as Microsoft's DirectCompute [16], an extension to the popular DirectX language. Most GPUs that offer compute as a part of their feature set today are OpenCL compliant. The OpenCL and the DirectCompute architectures closely match the CUDA architecture, and we evaluate a CUDA-compatible architecture and use CUDA benchmarks for the rest of the dissertation. We expect our observations and inferences to hold irrespective of the language our benchmarks are written in. For the sake of clarity, we have enlisted the terminology we use in the thesis and its CUDA and OpenCL equivalent in Table 2.1.

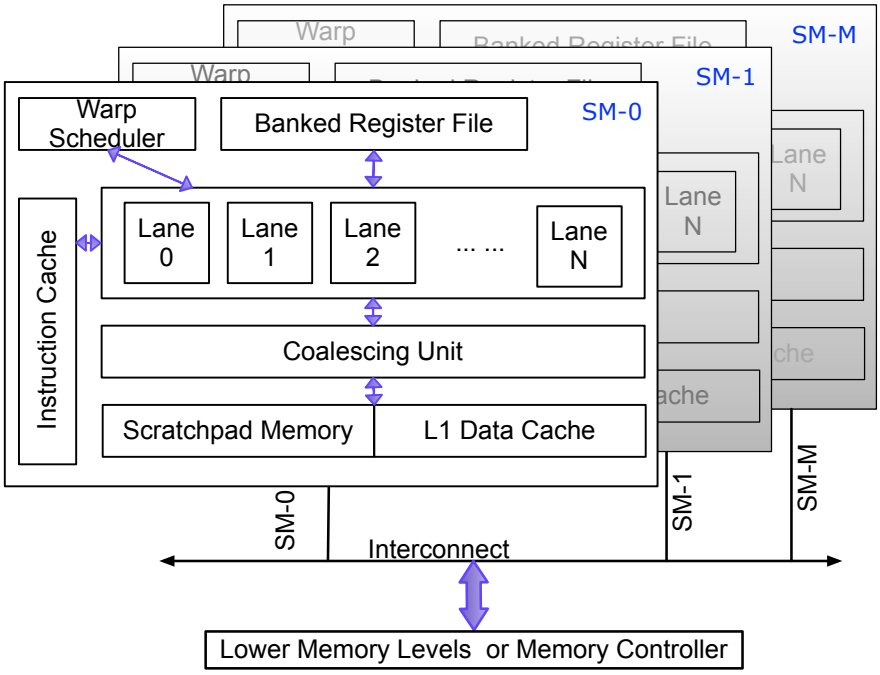


Figure 2.3: Baseline architecture

2.3 Our Baseline GPGPU

Figure 2.3 shows the architecture of our baseline GPGPU. Since architectural specifications of GPGPU hardware is not made public by the manufacturers, our baseline is built on our deductions from the publicly available details of modern Fermi-like [85] GPGPUs, the impositions of the programming model, and assumptions made by other researchers in the community [10, 31, 32, 34, 54, 81].

We assume that the Streaming Multiprocessor (SM) consists of 32 lanes, a register file that is shared by all the lanes, scratchpad memory, and a level-1 data cache (DL1). Even though the DL1 and the scratchpad memory are shown as two different blocks, they are a part of the same on-chip memory structure. The coalescing unit tries to combine memory requests

to the same cache line from different threads in a warp to a single memory request. There is an L2 cache that is shared by all multiprocessors.

There are also several other components in the SM that are not shown in this figure, like the instruction fetch unit, warp schedulers, the Special Function Units (SFUs), the texture and constant caches, etc. that are accounted for and modeled when relevant. More details about the modeling are available in Chapter 3.

Chapter 3

Simulation Methodology

In this chapter, I present GPSim which is a simulation infrastructure to model GPGPUs, built on top of an existing architectural simulator, ESESC [9].

3.1 The Need for Yet Another Simulator

The emergence of programming models like CUDA and OpenCL to harness the power of GPGPUs has aroused the curiosity of researchers and software developers alike, and there has been a flurry of activity in this area. With the lack of publicly available information about the architecture of GPGPUs, most of the early research either relied on analytical models [40] or in-house simulators [48] to estimate the impact of microarchitectural changes on GPGPU compute. These approaches typically lacked the ability to model the finer details, and there was an effort in the community to develop a more detailed and customizable simulator for GPGPUs. GPGPUSim [10] was an early simulator that offered support for a NVIDIA QUADRO FX GPGPU (and later on, for NVIDIA Fermi). This simulator offered options to

customize the GPGPU such as the number of shader cores, the number of registers, the size of the L1 cache, etc., but did not allow customization of the shader core or a configurable memory hierarchy seen in contemporary CPU simulators. This meant that simulating newer GPGPU designs and heterogeneous architectures would involve a significant coding effort if we relied on GPGPUSim.

GPSim was developed as a platform to obtain insight, enable research and potentially propose prudent refinements to the existing architecture of GPGPUs. It was designed with the following goals in mind:

- To provide a flexible and extensible framework to model GPGPUs
- To provide a framework that supports simulation of heterogeneous systems with both CPUs and GPUs as well as other futuristic many-core architectures
- To support simulation of a large number of cores in a reasonable amount of time
- To model the energy/power consumption

ESESC is a fast, cycle accurate, application level simulator with detailed performance, power and thermal models for modern CPUs. In addition to detailed models for in-order and out-of-order CPU cores, it also includes detailed shared memory models, network models, and the ability to fully configure a multiprocessor system to evaluate the performance of a multithreaded or multiprocess applications. GPSim uses this underlying framework and offers the same flexibility to configure next generation GPGPUs and evaluate a heterogeneous system with both CPUs and GPGPUs. Keeping in mind our goal to keep the simulator fast, GPSim

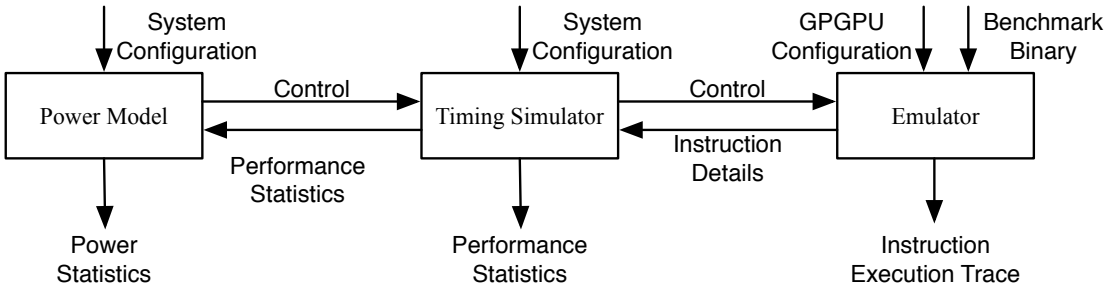


Figure 3.1: High level architecture of GPSim

employs native co-execution of benchmarks on real hardware and provides an option to enable spatial sampling which allows us to selectively execute only a few threads, in case of extremely regular applications. The timing model is tightly coupled with an integrated power model for CPUs, and GPSim extends ESESC to couple GPU cores to a power model for GPGPUs derived from GPUSimPow [54].

Figure 3.1 shows a high level overview of GPSim. The emulator module emulates the behavior of the target architecture and supplies a *trace* of instructions to the timing simulator which models the hardware structures in the processor (the pipeline, the load store queues, the execution units, etc.). This block interacts with the power model by feeding it relevant statistics like cache misses, number of load-store instructions, floating point operations, etc., which in turn can derive the power consumption by estimating the energy consumed for each action. Since the details of the hardware implementation of GPGPUs are not available publicly, we make reasonable assumptions about the underlying hardware with information available in the programming guide and/or patents. We discuss this and provide more details about GPSim in the following sections.

GPSim was one of the first simulators available to model GPGPUs and, until recently,

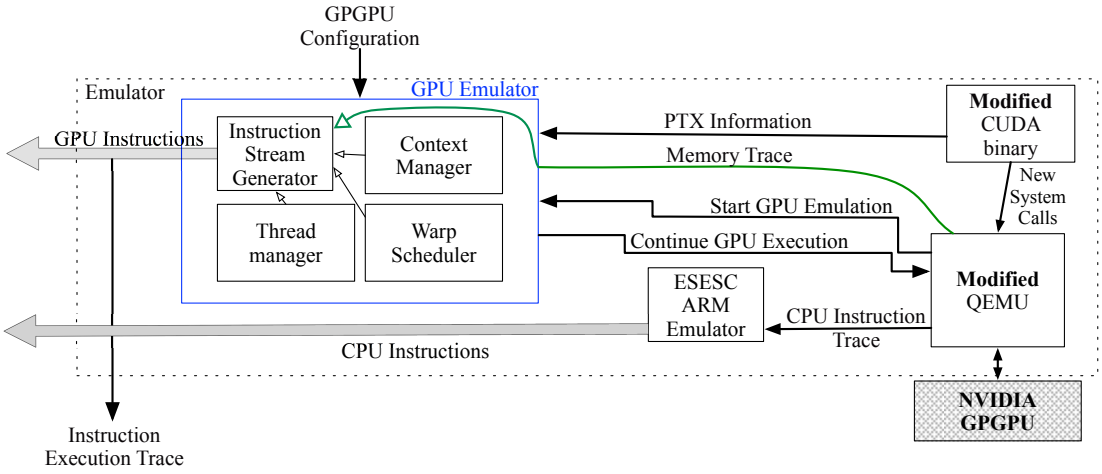


Figure 3.2: A detailed look at the GPSim emulator module

was the only one to support the simulation of heterogeneous architectures with both CPUs and GPUs. After GPSim, there have been a few other simulators that support the simulations of heterogeneous architectures with GPGPUs in them. GPGPUSim, since its early days, is now available with GPUWattch [53], a power model for GPGPUs. MacSim [45] and Mutli2Sim [81] are two other recent simulators that support simulation of heterogeneous architectures. Unlike GPSim that executes GPGPU applications natively, these rely either on Ocelot [28], LLVM [49], or on a native GPGPU emulator like GPGPUSim to generate a trace to be fed to the simulator and, hence, are inherently slower than GPSim.

GPSim is bundled and available for public use with ESESC at `https://github.com/masc-ucsc/esesc`

3.2 Functional Emulation of GPGPUs

As mentioned earlier, GPSim is built on top of ESESC and shares the infrastructure wherever possible to keep the design simple. ESESC uses a modified version of QEMU [11] which runs application binaries in user-mode and intercepts the instruction trace and passes it to ESESC. GPSim uses a similar approach as shown in Figure 3.2.

3.2.1 Native Coexecution

To determine the program flow, it is necessary to be able to disassemble the binary and mimic the behavior of the underlying architecture as correctly as possible. Unlike CPUs where the ISA represents the instruction actually executed by the processor, GPGPU manufacturers only expose a virtual ISA and not the native target architecture instructions. In the case of NVIDIA GPGPUs, this virtual ISA is called the *parallel thread execution* ISA (PTX). To derive the program flow from the PTX instructions, we need a disassembler and a detailed functional model of the underlying GPGPU. This is not only prone to a lot of approximation errors, it is also extremely slow, given that each GPGPU application usually has a couple thousand threads each. To circumvent this problem, GPSim uses native co-execution, *i.e.*, the GPGPU application is natively executed on a GPGPU in a lock-step fashion, while relaying the program flow and memory trace back to the emulator. Native co-execution provides tremendous speedups, and we have empirically observed a speed up of at least three orders of magnitude over the emulation speeds of the other simulators. Native co-execution also places the requirement of having a physical GPGPU as a part of the simulation setup.

3.2.2 Application Contamination

CUDA programs are not designed to work in a single step mode with CPUs. CPUs launch a kernel and specify the grid and block dimensions, and the CUDA device spawns as many threads as defined. Control is returned back to the CPU only after all the threads complete execution. This behavior does not work well with the idea of native co-execution because it makes the amount of state needed to be saved infeasible, given the number of threads. Therefore, to enable native co-execution, we *contaminate* the application to execute only a basic block of instructions at a time and return the control back to the CPU at the basic block boundary. As shown in Figure 3.3, before and after each kernel spawn, we need to load and save state (including the next destination basic block) and store the memory addresses after each load/store instruction as a part of the memory trace. Contamination of applications is done automatically by a preprocessing script, and it does not alter the program behavior in anyway. Despite the fact that the added instructions are mostly memory instructions, the performance penalty on native execution is negligible compared with simulation time.

3.2.3 Trace Generation

When the application is contaminated, the preprocessing script divides the PTX code into segments of code, each representing a single basic block. These segments are annotated with information used by the GPGPU emulator, *e.g.*, the basic block id, the number of registers used, divergence information, etc. When the contaminated application is executed on the GPGPU, it executes one basic block at a time and returns a data structure back to the CPU which tells the emulator the kernel executed, the basic block currently executed, the basic block

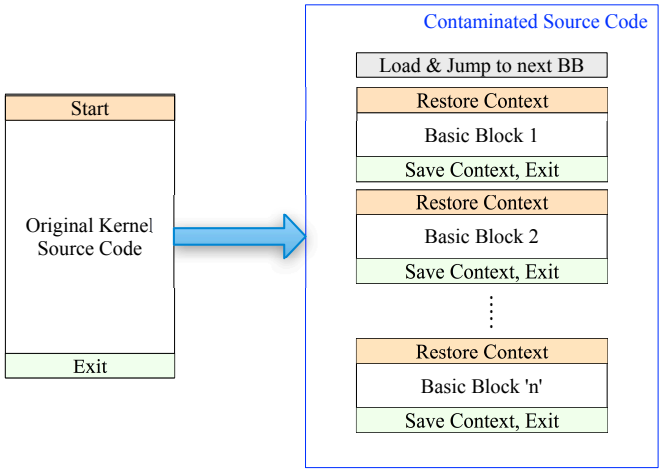


Figure 3.3: A contaminated CUDA binary

scheduled next, and the memory trace. Using this information and the annotations made by the preprocessing script after segmenting the code, the emulator pieces an execution trace together for each executed thread. The emulator then passes this trace to the timing model and sets the GPGPU up to execute the next basic block.

3.3 Modeling GPGPUs

The steps described above only provide the trace, but do not give any clue about how they would be executed on the core. To do this we need to functionally model the thread management on GPGPUs including the warp scheduling mechanisms, handling of synchronization primitives, etc.

3.3.1 Thread Management

The thread management block groups threads into thread blocks and assigns them to SMs for execution depending on the block and grid configuration that is input to the kernel as well as other architectural parameters that determine if the GPGPU has enough on-chip resources to sustain the threads in flight. It enables and disables threads that are to be natively executed on the GPGPU, *i.e.*, it ensures that only the threads in the current active warp are being natively executed on the GPGPU. It also keeps a record of how many threads have completed execution and when the GPU computation should stop.

3.3.2 Warp Scheduling and Context Management

Each SM on a GPGPU executes sets of threads known as warps, virtually as many threads as the number of lanes. Each time a warp hits a long latency instruction (for instance, a memory access which is a miss) or a synchronization primitive, it saves the context of the existing warp and moves on to the next ready warp, supplying the SM with a steady stream of ready-to-execute instructions. The warp scheduling policy is currently set to round robin, but there are a number of warp scheduling policies that can be implemented, many of which are listed in Table 6.1.

3.3.3 Modeling Divergence

When threads within a block diverge, it is believed that the divergence is handled serially, *i.e.*, threads taking a single path are executed first, followed by the next set of threads executing a single path, and so on. It is not clear whether or not these threads are allowed to

Parameter	Significance	Default Value
pes_per_sm	Number of lanes	32
warpsize	Number of threads in a warp	32
max_warps_sm	Maximum concurrent warps per SM	24
max_blocks_sm	Maximum concurrent blocks per SM	24
max_shmem_sm	Scratchpad memory available per SM	48K
max_regs_sm	Registers available per SM	65536
divergence_mech	Serial, Post Dom , SBI	Serial
unifiedCPUGPUmem	Memory copies between CPUs and GPUs	false

Table 3.1: Configuration parameters affecting the functional emulation of GPGPUs

re-converge back and, if they are, at what point. GPSim allows for multiple ways to handle divergence. At one extreme end, control divergence is handled serially and threads are not allowed to re-converge. At the other extreme end, we assume that each thread has the capacity to execute its own path. We also implement the post dominator reconvergence strategy where divergent threads can re-converge back at the reconvergence point [31].

3.3.4 Spatial Sampling

Almost all GPGPU applications have a very large number of threads and while native execution makes the emulation faster, the detailed timing model will still need to simulate all the threads. A significant number of GPGPU applications are largely regular with almost no divergence. Since the underlying architecture is also homogeneous, the behavior of blocks of regular threads on different SMs is largely the same. We exploit this fact and provide the option for Spatial Sampling of threads, *i.e.*, emulate all the threads (for correct program execution) but create a trace for only a select number of threads and pass it to the detailed timing model. Since the number of threads or blocks that we model can vary per benchmark, we also have an option

to turn this feature off completely.

3.4 Performance Modeling

ESESC supports detailed modeling of in-order and out-of-order processors. We add a third type of processor we call SMcore designed to model the behavior of a single SM. The SMcore is an extension of the simple in-order core model. It can process as many instructions as the number of lanes per cycle and no more than one instruction per thread per cycle. A single thread is affiliated with the same virtual lane for its lifetime. We assume that the register file is large enough to support the requirements of the in-flight threads; we do not model spilled registers. Divergence and synchronization primitives like barriers are handled by the emulator, and the result is an instruction stream that has instructions from the next ready warp, or *nops* to indicate that the lane has been underutilized. The number of integer, load/store and floating point execution units are configurable, and we use default values as mentioned in the programming guide. Branch prediction is not modeled for GPGPUs by default, but we have the option to include very simple branch predictors.

We take advantage of the flexible and configurable memory hierarchy that ESESC provides and setup a hierarchy like the one in GPGPUs by default. Each SM has a scratchpad cache for scratchpad accesses and a first level data cache. Based on the information in the programming guide, we model the DL1 as a multibanked cache structure and assume a default latency of 20 cycles [32]. Coalescing for the DL1 and scratchpad accesses is modeled using Miss Status Handling Registers (MSHRs). All the SMs share a common L2 cache and this L2

Default CPU configuration	
Number of cores	1 out-of-order core, 3.0 Ghz
Issue Width	4 Instructions
Re-order Buffer	256 Instructions
BTB	4K entries - 4-way
L1 TLB	64 entries / Fully Assoc. / 1 cycle
Core L1 cache	32KB / 4-way / 64B line / 4 cycles
Core L2 cache	512KB / 16-way / 64B line / 7 cycles
Default SM configuration	
Number of SMs	4 SMs, 1.7 Ghz
Number of SPs	32, in-order, per SM
Branch Predictor	None
Memory Speculation	None (Switch to the next ready thread)
Memory Coalescing	Enabled
Scratchpad	48KB / 8 banks / 16 cycles, per SM
SM L1 cache	64KB / 8-way /64B line/ 16 cycles, per SM
SM L2 cache	256KB / 16-way /64B line/ 7 cycles, per die
Shared Memory	
Shared TLB	512 entries / 4-way / 1 cycle
Shared LLC	8MB / 32-way / 128B line /14 cycles
Memory	18GBytes/s BW with 50ns access time

Table 3.2: Default simulation parameters in GPSim.

cache can either extend to the main memory or to a shared last level cache in case of heterogeneous systems. We have a basic bandwidth model to model the off chip memory accesses.

Table 3.2 summarizes some of the default configuration for our SM cores. We instantiate a GPGPU like we would instantiate a multicore with as many cores as SMs on the GPGPU and can also configure the number of lanes per SM. It is also possible to instantiate other CPU cores. The default configuration is one CPU core, along with 4 SMs, with 32 lanes each.

3.5 Estimation of Energy and Power

GPUSimPow [54] is a recently developed power model for GPGPUs that has been validated by power measurements on a GPGPU. We extend the power model in ESESC and provide an infrastructure to estimate the power consumption of GPGPUs based on GPUSimPow. ESESC internally uses CACTI [84] to estimate the energy for all the different blocks within a GPGPU, but it is also possible to ignore CACTI and specify externally computed energies via the configuration file for GPGPUs.

3.6 Summary

In this chapter, we provide a detailed look at our simulation infrastructure. We have enlisted a few parameters that reflect the our baseline architecture. In each of our proposals, we highlight and mention any additional modifications we may have made to the baseline. This simulator is now freely available as a part of ESESC along with a few pre-contaminated binaries.

Chapter 4

Eliminating Redundant Memory Copies in a Heterogeneous CPU-GPU Architecture

Traditional discrete GPUs, which are by design suited to handle huge streams of data, are being increasingly used in tandem with the CPU for general purpose applications with the aid of programming models like CUDA and OpenCL. These applications range from loosely coupled applications where the CPU offloads a large batch of data for the GPGPU to process to, more recently, tightly coupled applications operating systems abstractions like filesystems to GPGPUs [75]. Recent initiatives like Intel's Sandy bridge [92] and AMD's Fusion [17] implement a tightly coupled heterogeneous architecture on a single die. By doing so, they provide a powerful platform that integrates a modern CPU for efficient single threaded performance with a GPGPU for high throughput parallel processing [3, 4].

4.1 The Memory Transfer Problem

The OpenCL and CUDA programming models assume that the kernel is executed on a physically separate GPU which behaves like a coprocessor to the host running the CPU program. They also assume that both the device and the host maintain their own separate memory spaces (host and device memory) without hardware cache coherence between them. Thus, a typical CUDA or OpenCL application is composed of three distinct sequential stages: copying the input state from the CPU to the GPU, spawning kernels with large number of threads to compute on the GPU, and, finally, copying the result back from the GPU to the CPU address space. This copy between the host and the device is very expensive.

The need to move data back and forth between separate addressing spaces necessitates that for meaningful speedups the GPU compute time must be large enough to amortize the transfer time. This restricts the range of algorithms that can benefit from a GPU to those with a low transfer to compute ratio. In fact, simple applications with low data reuse like SAXPY or sparse matrix vector multiplication are slower when implemented for GPUs.

The transfer of data between the CPU and the GPU over the PCI Express bus is a well documented issue. The CUDA and OpenCL programming model allow us to amortize this cost over a very large computation, achieved by spawning a large number of threads. However many data parallel algorithms with a low compute to transfer ratio tend to perform better on the CPU than on the GPU or show a huge potential for improvement [36, 52]. Figure 4.1 shows the memory transfer to GPU compute ratio for several application in a single die environment. A PCI solution like a GTX580 would have an even worse ratio. Most applications that transfer

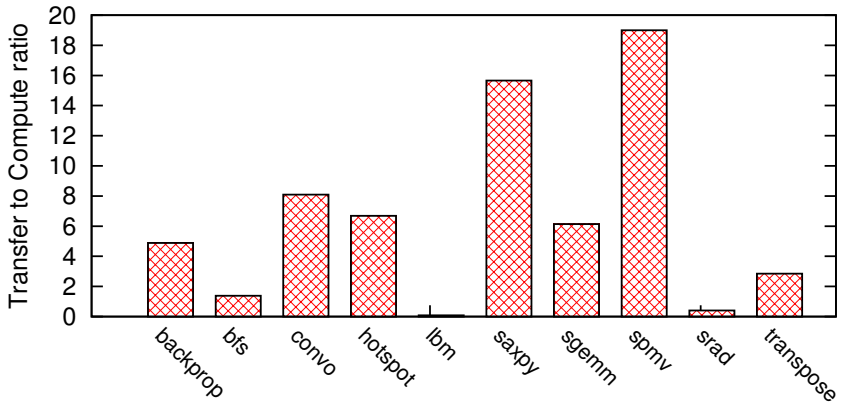


Figure 4.1: Memory transfer to GPU compute time ratio for several CUDA applications in a single die solution shows that memory transfer is key even for CPU-GPU multicores. The workloads considered here are described in Table 4.1.

Benchmark	**Description**	**Mem Transfer (MB)**	**Global Memory Access %**
Backprop	A machine learning algorithm used in a graph	17.75	60.32
BFS	A graph traversal algorithm	41.0	100
Convolution	An image processing algorithm	72.0	5.82
HotSpot	A tool to estimate the temperature for an architectural floorplan	12.0	30.95
LBM	Simulation of a Newtonian fluid using the discrete Boltzmann equation	555.46	100.0
SAXPY	A common subroutine from BLAS which performs $z = \alpha * x + y$	96.0	100.0
SGEMM	Matrix Multiplication	11.99	6.70
SPMV	A commonly used implementation for multiplication of sparse matrices	8.6	51.25
SRAD	Used to remove locally correlated noise in an image	8.0	40.44
Transpose	Compute the transpose of a matrix	55.125	100

Table 4.1: GPU workloads used in our evaluation

over the PCI express interconnect are deemed unfit to run on the GPU and, thus, a large category of applications cannot exploit the parallel throughput of the GPU.

We profiled several applications like SGEMM and SPMV from the Parboil benchmark suite [78] on a GTX580. In many applications, the memory transfer consumes 4X more time than the compute (*i.e.*, transfer to compute ratio or $\frac{\texttt{Memory Transfer Time}}{\texttt{GPU Compute Time}}$ is more than 4).

Most applications with the exception of LBM, SRAD, and BFS have over 2X more memory transfer time than GPU compute time. This means that it is possible to obtain more speedup by avoiding the memory copy than by improving the GPU compute time. For example, matrix TRANSPOSE and SAXPY have over a transfer to compute ratio of over 15. It is important to notice that the transfer to compute ratio does not include CPU time. If we exclude the benchmark IO and initialization, the CPU compute time represents less than 10% of the total time for the benchmarks analyzed.

Unifying the cache hierarchy will reduce a large chunk of the transfer time by eliminating the transfer over the PCI interconnect. However our experiments show that there is still a significant percentage of time that is spent on memory copies between the host and the device addressing spaces. We take the unification of the cache hierarchy a step further and propose removing the divide between the two addressing spaces by implementing a virtual shared memory system for a heterogeneous CPU-GPU multicore architecture on a single die based on a Translation Lookaside Buffer (TLB) level shared by both the CPU and GPU sides.

Placing the GPU in the same die as the multicore mitigates the memory transfer cost, but, as Figure 4.1 shows, this is still not enough and we need create a unified address space to avoid the memory transfers. Our proposal addresses this issue overcoming the drawbacks discussed in Section 4.3 for similar approaches and provides full binary compatibility to existing CUDA applications.

4.2 Assessing the Need for Coherence

OpenCL and CUDA models assume that CPU cores provide cache coherence between themselves while GPU cores do not. The new SoC integration opens up the question of whether all the cores (CPU and GPU) need to have coherence or can just continue working with two different address spaces.

If the CPU and GPU have two different address spaces, the CPU cores have coherence between themselves, but they are incoherent with the GPU cores. Since the programming models for GPUs have been designed assuming incoherent GPUs, the GPU cores also do not need hardware cache coherence. However, the problem with this approach is the costly memory copy needed between address spaces. Our experiments show that even in a single die SoC with a shared last level cache the performance overhead is over 30% for GPU workload suites like Parboil [78].

If we simply decide that the CPU and GPU share the same address space, both types of cores need to be coherent. The trend is to provide and require hardware cache coherence (EXOCHI [82], Bothnia [23], Intel Knights Corner, and AMD Fusion [77]). However, energy inefficiency proves to be a challenge with the hardware cache coherence approach. This is especially true for GPU applications with a high miss rate and sharing which triggers significant cache invalidations and writebacks. We observe that coherence represents over 11% of the memory traffic for GPU applications, while it is under 2% for traditional PARSEC applications. Thus, blindly enabling hardware cache coherence will incur significant energy costs for GPU applications.

4.3 Related Work

To improve the compute to memory ratio and make the argument of offloading general purpose tasks from the CPU to the GPU more compelling, there are a number of approaches that are popular in the community. We discuss these approaches and other proposals below.

4.3.1 Avoiding Memory Copies with Pinned Memory

The CUDA programming environment provides two mechanisms to make the data exchange between CPU and GPU more efficient: overlap the execution of one kernel with the memory transfer of another and a *zero-copy* strategy which uses pinned host memory. The first approach works best with the (rare) occurrence of two independent kernels that do not share data, and is only fully beneficial if we are able to completely overlap the transfer time for one kernel with the execution time of the other kernel. Zero-copy needs the host to pin down memory and is not sustainable with large or growing data sets. The AMD OpenCL Programming guide offers more advanced resources like zero-copy memory objects and zero-copy buffers. They can virtually eliminate memory transfers but still show different drawbacks like creating asymmetric memory behavior in the CPU and GPU sides. Many memory intensive algorithms use other approaches like moving the entire data-set to the GPU's memory [36], which is again not sustainable.

4.3.2 Avoiding Memory Copies with a Shared Virtual Space

Chinya *et al.* propose Bothnia [23], an extension to the Intel Integrated Graphics Driver, that avoids memory copies. It provides cache coherence and exception support and as-

sumes a hardware configuration with no shared memory controller, no last level cache (LLC) and no shared TLB (CPU and GPUs have only private TLBs). The CPU and GPU have different page tables and page table entries, and the Bothnia driver transcodes a given CPU virtual address translation into the corresponding GPU page table entry (PGTT) to keep the same virtual to physical mapping. Wang *et al.* propose EXOCHI [82], a new programming environment with a shared address space. It assumes the same architectural environment as in Bothnia, with different translation systems for the CPU and GPU sides (private TLBs) requiring *address translation remapping (ATR)*. Although EXOCHI does not require coherence, it requires programmer modifications. The paper points out that in the absence of hardware support for cache coherence, it is the responsibility of the programmer to use critical sections to protect other threads from reading or writing the data being processed by threads on the exo-sequencers.

4.3.3 Integrated CPU-GPU Systems

On-chip heterogeneous architectures combining CPUs and GPUs are a growing trend as exhibited by Intel's Sandy Bridge [92], Knights Corner, AMD's Fusion [17], and NVIDIA's Denver [13]. A radically different, but related, approach comes from Stratton *et al.* [79]. They convert CUDA code into standard C language, and, with the support of a runtime library, they manage to run the translated CUDA programs on a CPU multicore, preserving the CUDA semantics. The memory hierarchy design space available for the integration of CPU and GPU cores is leveraged in different ways. Zhu *et al.* [95] cite the example of a real-world application where assuming no overhead of the memory transfer between the CPU and GPU resulted in an extremely efficient implementation on the GPU.

The use of a shared L2 has been proposed in Woo and Lee [87], but only for running CPU workloads. Lee *et al.* [50] study the implications of sharing a L2 cache between a CPU and the GPU and leverage UCP [66] and RRIP [93] to mitigate the negative impact of GPU workload accesses on the CPU workload, but they do not use the LLC to improve data sharing between the CPU and the GPU like we do. An Intel patent [37] describes a shared cache to allow faster communication among heterogeneous cores through synchronization mechanisms, but it does not describe any address translation scheme.

4.3.4 SoC Virtual Memory and Memory Copy Support

Merely integrating a CPU and GPU on a die, sharing the same physical memory, does not lead to a dramatic performance increase for existing applications. Our proposal lies in the line of leveraging on-die integration and supporting a unified virtual memory space to eliminate memory copies between CPU and GPU, and it is therefore closely related to the following approaches.

Wong *et al.* propose Pangaea [86], a microarchitecture reorganization of both CPU and GPU to achieve tighter architectural integration within a heterogeneous CMP design, along with power and area efficiency. Pangaea keeps separate TLBs and supports coherence like in Bothnia. It assumes the heterogeneous Open MP model under EXOCHI [82] (which provides the necessary transcoding between the different PTEs) and the communication mechanism between the CPU and EU cores requires an ISA extension.

The Heterogeneous System Architecture (HSA) [70] is a set of specifications for tightly integrated CPUS and GPUs proposed by the cross vendor HSA foundation that allows a

unified main memory between the CPU and the GPU to eliminate the memory copies between them, implying a coherent memory system. In general, cache coherence is believed to be too much of an overkill in heterogeneous platforms such as combined CPU-GPU systems because of potential scalability problems and energy inefficiency [80, 94]. Modern AMD GPGPUs like the Kaveri [15] that are HSA compatible only provide optional coherence between the CPU and GPU using acquire/release memory instructions, which allows the programmer to define when exactly coherence is needed, and when it is not.

Hampton *et al.* [38] and Menon *et al.* [57] exploit the property of idempotence in kernels to implement support for exceptions in vector processors and GPUs, respectively. The focus of their papers, unlike ours, is on providing support for exceptions.

4.4 Our Proposal: FuseTLB

On the basis of our understanding of the programming models and the nature of typical GPGPU applications, we propose a simple design where we establish a virtual shared memory space between the CPU and the GPU by having purely virtual caches on the GPGPU side and sharing a TLB between them. We build our proposal on top of the baseline GPGPU (based on NVIDIA's Fermi [85]) as shown in Figure 2.3 and a modern CPU core (based on Intel's Sandy Bridge processor [92]). We call our design ***FuseTLB***. Figure 4.2 illustrates the architecture we propose, and we describe our proposal in detail below.

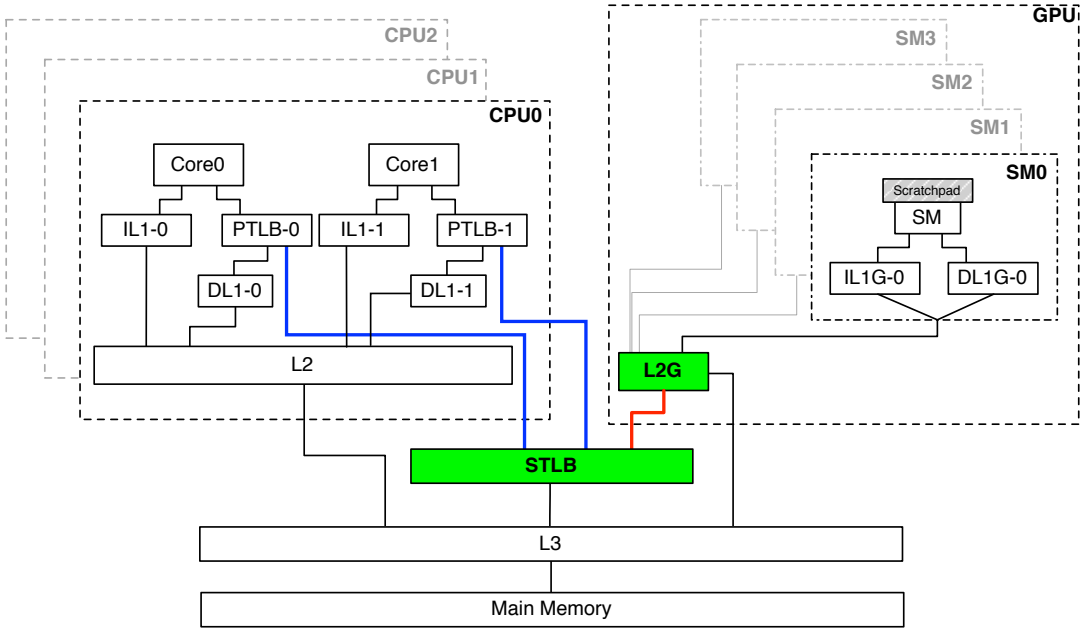


Figure 4.2: The FuseTLB architecture

.

4.4.1 A Virtually Indexed and Virtually Tagged SM Memory Hierarchy

The SM-L1 is a virtually indexed virtually tagged cache, private to each SM. This cache is split into two parts: a multibanked scratchpad without coalescing for scratchpad memory accesses and a single port cache with coalescing for global memory accesses, akin to NVIDIA's Fermi architecture. The scratchpad is important to model because for many applications it serves a significant fraction of the memory requests. This is relevant for our work because we also need to provide correct functionality and memory protection for the scratchpad.

Like the other SM caches, the scratchpad stores virtual addresses as well. As per the CUDA/OpenCL programming model, neither does the SM scratchpad survive kernel spawns, nor is it shared across blocks. We thus initialize the SM scratchpad before use, and, to protect the data from malicious applications, we discard its contents once the kernel completes execution.

To do this, we could use the present but unused valid bit. An access to the scratchpad without a valid bit returns zero. We could possibly trigger an exception for unauthorized accesses, but current GPUs have no means to handle exceptions. Even though we use pure virtual caches, we do not have privacy issues across kernel spawns because all the cache lines are invalidated from the SM-L1 cache once the kernel finishes.

We implement coalescing by merging many accesses that are sent to the same cache line in a single request. This is to reduce power and increase efficiency. In case multiple addresses are requested from the same port of the multibanked scratchpad, we serialize the requests.

4.4.2 The Need For a Shared L2 Cache in the GPU

SMs are designed to sustain many misses with thousands of in-flight threads. Typically, the SM-L1 cache misses are clustered to the same line. The modeled coalescing significantly reduces memory traffic, but it does not consider the observation that most SM-L1 misses across SMs are clustered. This can put tremendous pressure on shared resources and potentially make this design infeasible due to the inevitable energy consumption. This is why we extend the baseline architecture by adding a small non-inclusive virtual ***SM-L2*** cache shared by all the SM cores in the GPU.

NVIDIA devices support concurrent kernel execution starting from the Fermi architecture. Therefore, the SM-L2 cache needs a kernel-id to distinguish between accesses from different kernels. We can run blocks of the same kernel across diferent SMs as in NVIDIA devices, but each SM can only execute a given kernel at a given time, and therefore with n SMs

there will be at most *n* kernels executing simultaneously. Since the whole SM memory hierarchy must flush every cache line that belongs to the kernel just finished, there are at most *n* kernels cached in the memory simultaneously in the SM-L1 and L2 caches. As a result, $log_2(n)$ bits are needed for the kernel-id. In our system, we have four SMs which means just two additional bits per cache line tag.

4.4.3 A Shared TLB between the CPU and the GPU

Unlike other CPU-GPU on-die solutions, we also add a shared TLB (**STLB**) just before the LLC, that is shared by both the CPU and the GPU. Each CPU core has its own private TLB, but this is not the case for SMs. Since the SM-L1 and SM-L2 are virtually indexed and virtually tagged, we perform a TLB translation for the GPU at the STLB, just before the LLC is accessed. Thus, the STLB is effectively a second level TLB for the CPU cores but a first level TLB for the SM cores. It is only accessed when we miss on the SM-L2.

4.4.4 Writeback Invalidating the SM-Caches

As per the CUDA/OpenCL model, an access to a region of memory shared by GPU before the kernel finishes is undefined. Once the kernel finishes, the SM cache lines for that kernel are written back and invalidated in the SM-L1 and SM-L2. In CUDA/OpenCL, the only way to share data between thread blocks (workgroups) in a deterministic way is to use atomic instructions or the *Threadfence* directive, and we provide support for both. The CUDA/OpenCL programming model requires all the threads in a thread block to see the same consistent memory when they reach a barrier. Thus, atomic operations trigger a write back and invalidate on the

SM-L1 cache, but this is not needed for the SM-L2. There is no need to writeback or invalidate the cache when a thread block finishes. This is because the CUDA/OpenCL programming model does not permit blocks to share data. A buggy/malicious block could do it, but this is not a security concern because all these blocks belong to the same kernel.

4.4.5 Avoiding Coherence in the SMs

A typical multicore hierarchy with coherence can be adapted to the SM cores with just a few changes to make it functionally correct, but the features of the CUDA/OpenCL programming model open up opportunities for FuseTLB to improve the efficiency. Since a thread block is executed in a single SM, and the CUDA/OpenCL model does not have directives to share data across blocks, there is no need to have coherence between the SM-L1 caches.

If there is no kernel being executed on the SM, it means that the SM caches have already been flushed (since we writeback/invalidate when a kernel completes) and the CPU-cores are free to touch shared pages. Whenever a CPU core touches a page used by the SM cores, or whenever a shared GPU TLB entry is deallocated, it triggers a writeback invalidate in the SM memory hierarchy for the allocated virtual page. When an SM core touches a memory location from the CPU core, it does not need to writeback or invalidate the whole page in the core caches or DRAM. The reason is that the CPU memory hierarchy already has coherence. An SM memory request hit in the LLC will trigger a cache line invalidation in the core-L2 in the same way that invalidations are triggered between cores.

4.4.6 Binary and OS Compatibility

Our system fully preserves the semantics of the CUDA/OpenCL programming model. In this model, copying data between the CPU and GPU addressing spaces entails allocating memory for the kernel parameters in both the host and the kernel addressing spaces, obtaining a pointer to each parameter in each space. Let us assume for example that hA and dA respectively point to the same kernel input parameter in the host and the kernel (device) spaces. After the allocation we copy hA to dA and then spawn the kernel. All the GPU threads will be able to access dA. Our proposal assumes that CPU and GPU are integrated on chip and share a LLC, which avoids actual data transfers by mapping hA and dA to the same physical address. Consequently, data can be shared in the LLC. In a naive implementation, a write to dA would modify the original hA in the CPU space which is a deviation from the programming model. To avoid this, we could protect the page with copy-on-write (COW) in the SM TLB. On analyzing the applications we observe that none of the analyzed benchmarks need to have COW. To avoid redundant copies, the automatic COW is enabled per application. It is uncommon to clobber input buffers during a kernel computation, but there is no such restriction in the programming model and therefore it can happen.

Misses on the STLB necessitate a hardware page walk. Having a private TLB for the cores would also require a separate hardware page table walk on both the GPU and the CPU side. Since the GPU does not manage the TLB, we will need some support from the OS to handle these misses. The STLB triggers an exception when it misses, and it is handled much like one generated if the CPU TLB misses: a randomly picked core receives the request to

Default CPU configuration	
Number of cores	6 out-of-order core, 3.0 Ghz
Issue Width	4 Instructions
Re-order Buffer	256 Instructions
BTB	4K entries - 4-way
L1 TLB	64 entries / Fully Assoc. / 1 cycle
Core L1 cache	32KB / 4-way / 64B line / 4 cycles
Core L2 cache	512KB / 16-way / 64B line / 7 cycles
Default SM configuration	
Number of SMs	4 SMs, 1.7 Ghz
Number of SPs	32, in-order, per SM
Branch Predictor	None
Memory Speculation	None (Switch to the next ready thread)
Memory Coalescing	Enabled
Scratchpad	48KB / 8 banks / 16 cycles, per SM
SM L1 cache	64KB / 8-way /64B line/ 16 cycles, per SM
SM L2 cache	256KB / 16-way /64B line/ 7 cycles, per die
Shared Memory	
Shared TLB	512 entries / 4-way / 1 cycle
Shared LLC	8MB / 32-way / 128B line /14 cycles
Main Memory	18GBytes/s BW with 50ns access time

Table 4.2: GPSim configuration to evaluate FuseTLB

handle a miss from another address space (GPU application). The exception handler just needs to follow the usual routine, but use the GPU address space. This operation is not too complex since we only deal with a shared TLB, and not private TLBs per SM.

4.5 Experimental Setup

We evaluate our proposal using GPSim and ESESC. Our baseline architecture is an integrated CPU-GPU on the same die sharing a TLB and the last level cache. The relevant simulation configuration parameters are listed in Table 4.2. We use the CACTI [84] model to estimate the energy consumption of the memory structures.

Exp	Configuration
typSoC	Baseline architecture with a shared TLB, without virtual memory (memory copies), with coherence.
copySoC	Baseline architecture with a shared TLB, without virtual memory (memory copies), without coherence.
FuseTLB	Our proposal with a shared TLB, with virtual memory (no memory copies), without coherence.
noSTLB	Similar to the FuseTLB, except that instead of a STLB, we have two TLBs, one for all the CPUs and one for all the GPUs
privTLB	A naive implementation, where there is no shared or second level TLB, only private TLBs per core and SM, without coherence.

Table 4.3: We evaluate five different architectural configurations.

As mentioned in Section 4.1, applications designed for use on GPGPUs are highly compute intensive and try to avoid memory transfers between the host and the device as far as possible. Most applications in popular benchmark suites [2, 21, 78] also reflect this choice in design. Based on published results [52] and analyses [36], we picked benchmarks from all the aforementioned suites that are impacted by frequent and/or large memory exchanges in addition to a few typical and regular benchmarks. Table 4.1 lists and briefly describes the GPU workloads that we evaluated.

All of the GPU workloads were obtained from commonly used test suites [21, 78], and all except LBM were run with the largest dataset made available to completion. SAXPY uses 8MB arrays. Convolution was performed on a 3072x3072 sized 2D image. Transpose used a 2688x2688 square matrix as its input.

To evaluate FuseTLB, we ran several GPU workloads (listed in Table 4.1) using a few different configurations which are explained in Table 4.3. These configurations are described

Set	Spec Rate Apps	GPU Workloads
S1	mcf - art - soplex - perlbench	Backprop, SPMV, SAXPY, transpose, LBM, BFS, SRAD
S2	dealII - wupwise - libquantum - soplex	Backprop, SPMV, SAXPY, transpose, LBM, BFS, SRAD

Table 4.4: List of SpecRate and GPGPU workloads

in reference to the baseline architecture. To simulate the effect of a true heterogeneous core in a few experiments, we also ran a subset of the GPU workloads with two sets of 4 SPECrate applications each. We choose a few random mixes of GPU and CPU application workloads. All of them have one GPU application and 4 SPEC2006 applications per run. To capture the contention of the active CPU or SM cores, we ran the GPU workloads to completion. If a benchmark finished early, it was relaunched again, but the simulator would not gather the statistics for that particular benchmark anymore. When we evaluated GPU programs run along with multi-threaded CPU programs, we waited until either the GPU application finished or 1 billion instructions from the CPU were simulated, whichever occurred later.

4.6 Evaluation

The evaluation is divided in three main subsections. We start by showing the dynamic energy savings in the GPU memory hierarchy, then show the overall performance improvements and conclude providing insights about the TLB. We also provide a brief evaluation of the fairness of multi-programmed workloads.

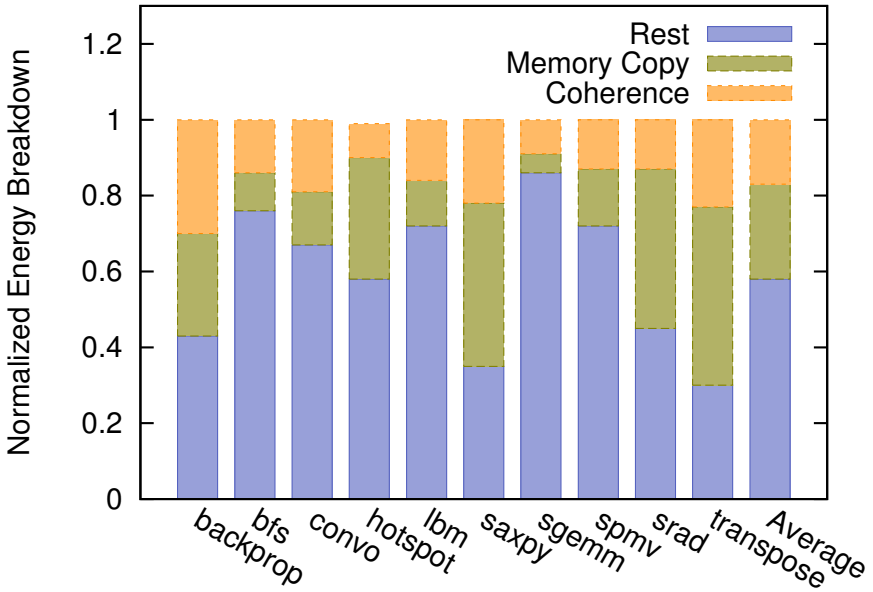


Figure 4.3: Memory hierarchy energy breakdown for a typical SoC (typSoC).

4.6.1 GPU Memory Hierarchy Energy

We implement a model based on CACTI [84] to extract energy per access for the GPU memory hierarchy. We model the scratchpad memories, the DL1s and shared L2, the TLB and lastly the LLC. Our energy model is independent of the execution time (we use a perfect clock gate and only consider the memory hierarchy). FuseTLB is compared against the baseline (copySoC). Additionally, two alternate memory hierarchies are evaluated. One in which CPU and GPU do not share a TLB, and instead use a private TLB (half the size) for each (noSTLB). The other scenario is one where we implement a more traditional TLB hierarchy in which each SM has its own private TLB (privTLB), making it comparable to the CPU cores.

We start by highlighting the contributions of two features we see in CPU-GPU SoCs today: coherence and memory copies across address spaces. Figure 4.3 is a stacked-plot of

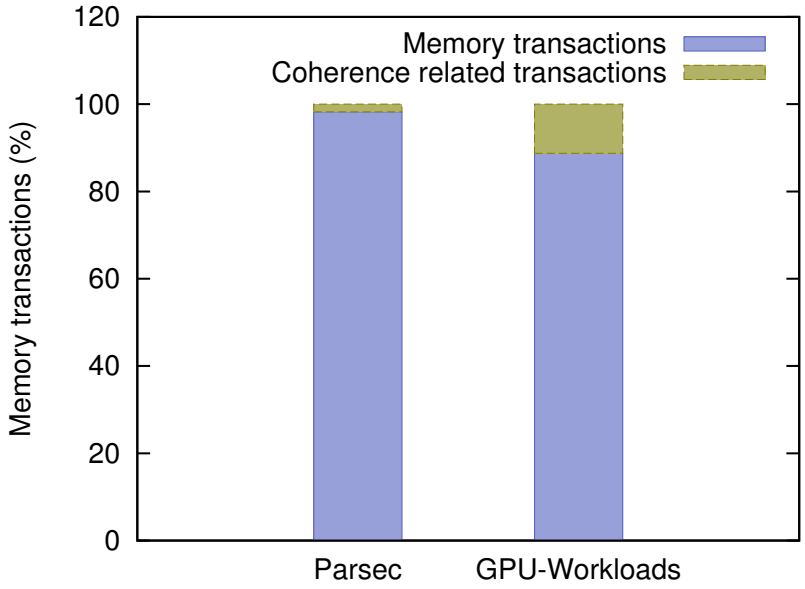


Figure 4.4: CPUs have lesser cache coherence traffic than GPGPU workloads.

the energy consumption seen in a CPU-GPU SoC which needs coherence and memory copies (typSoC), one that does not need coherence, but uses memory copies (copySoC), and finally, FuseTLB which uses neither. Hardware cache coherence represents around 10% of the memory hierarchy energy consumption. The memory copies between the CPU and the GPU represent between a small 5% for SGEMM to over 50% for transpose. The significant contribution that coherence and memory copies make to the overall energy consumption allows FuseTLB to decrease the energy consumed by and average of 38%, well below copySoC and typSoC.

Figure 4.4 provides an understanding of why the hardware cache coherence is especially bad for GPU applications. It shows that for PARSEC workloads, the cache coherence traffic represents less than 2% of all the memory transactions. Compare this with GPU applications, where, due to the higher miss rate, cache coherence traffic represents over 11% of all the memory traffic. This implies that cache coherence is more problematic for the GPU hierarchy

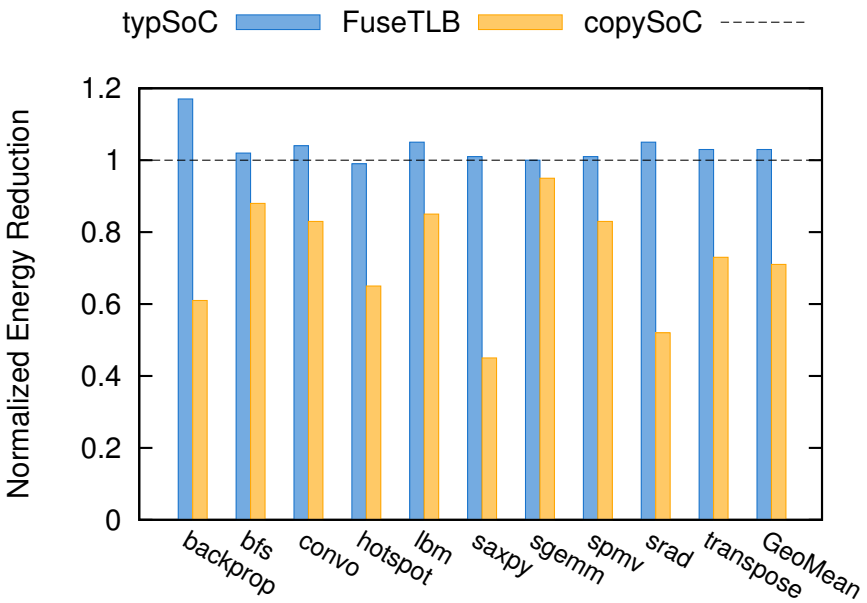


Figure 4.5: Avoiding memory copies and coherence is highly energy efficient.

than for the CPU hierarchy because of the application characteristics.

Figure 4.5 shows the effect of coherence and memory copies on the baseline architecture. Values are normalized to the baseline without coherence (copySoC), which is more energy efficient than typSoC, the option with coherence. Eliminating both coherence and memory copies (FuseTLB) achieves up to 27% energy reduction on an average.

Figure 4.6 and Figure 4.7 break down the energy consumed by the principal components of the memory hierachy in each experimental setup. The contribution of TLBs to energy consumption in this system hardly reaches 1% in the case we have private TLBs in all cores without any shared TLB.

Figure 4.8 compare FuseTLB with the designs without the STLB in terms of energy consumption, normalized to the baseline (no coherence in all the cases). FuseTLB is as or more

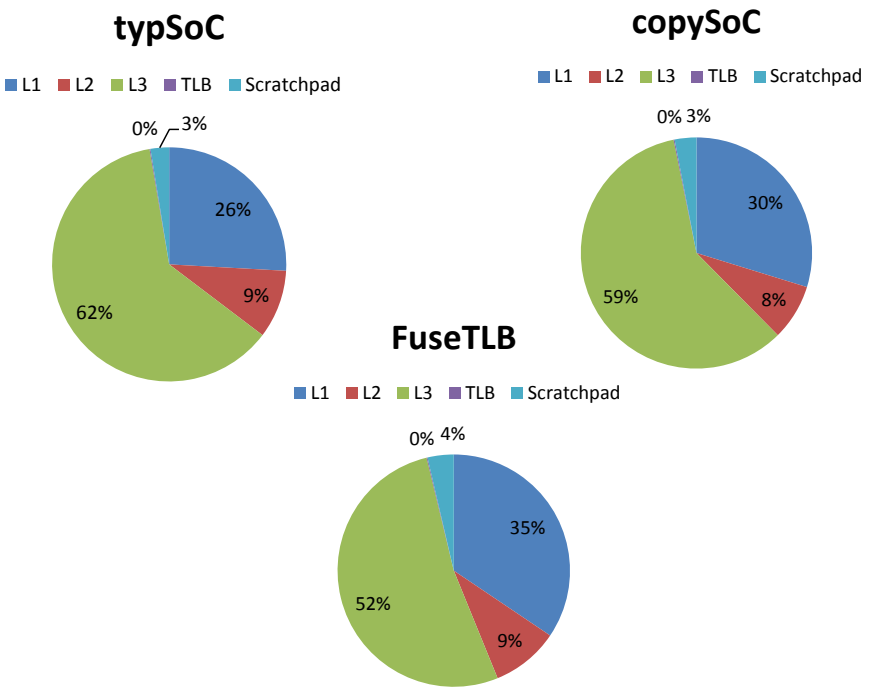


Figure 4.6: Energy distribution across various blocks in the baseline memory hierarchy

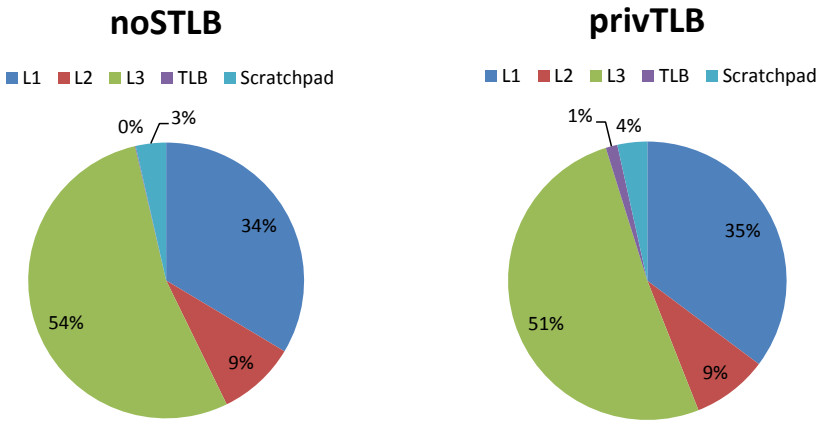


Figure 4.7: Energy distribution across various blocks in the two organizations without a STLB

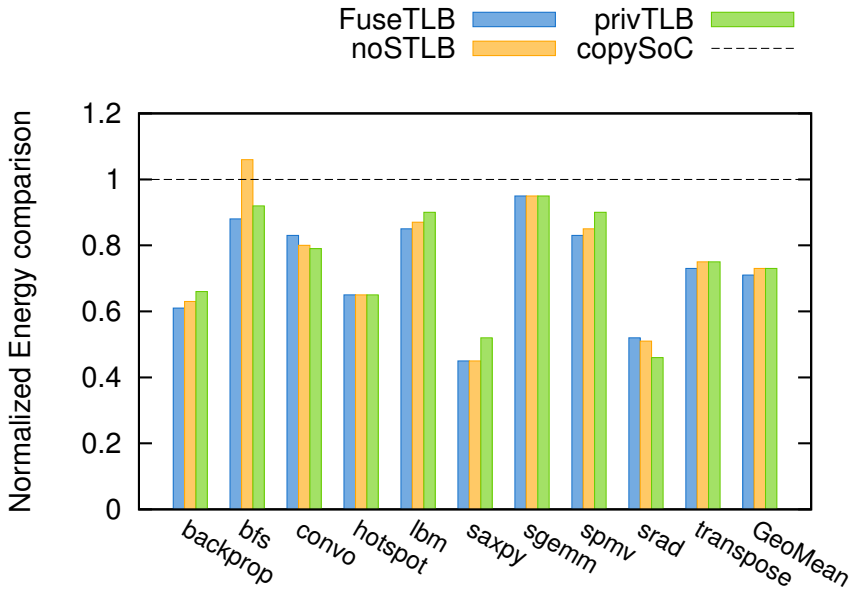


Figure 4.8: Comparison of energy consumption in FuseTLB (STLB) and non-STLB organizations, normalized to the baseline

energy efficient than the other TLB configurations.

In summary, avoiding memory copies reduces energy consumption independently of the TLB organization. On the average, both private and shared TLBs achieve over 25% energy savings, with the shared TLB solution achieving 27% energy savings.

4.6.2 Overall Performance

Figure 4.9 compares the breakdown of the speedup achieved with our proposed architecture against a CPU-GPU multicore that requires a memory copy between address spaces. It shows the FuseTLB configuration (proposed) performance normalized against copySoC configuration (baseline) shown in Table 4.3. On average, we achieve 100% speedup because we just cut in half the execution time. This considers the GPU and the CPU execution time. As

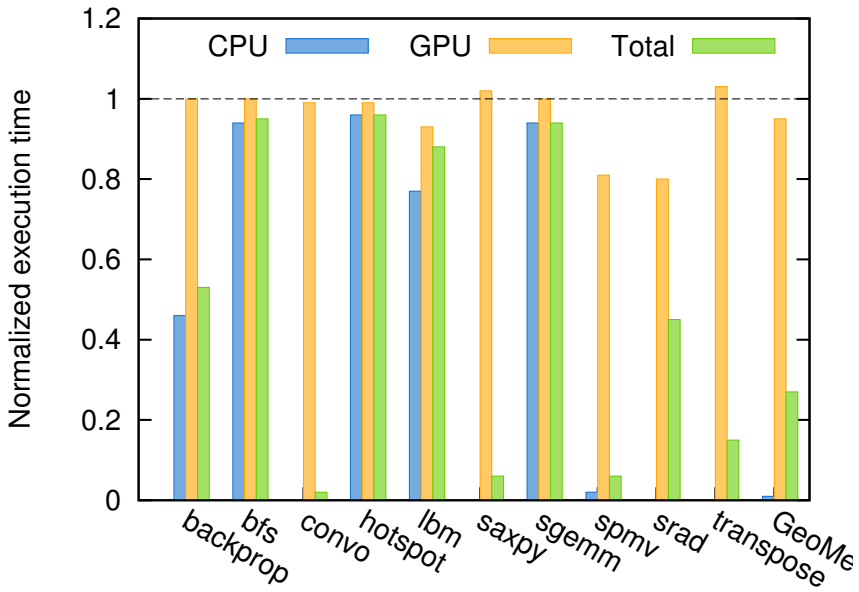


Figure 4.9: FuseTLB halves the execution time against an architecture that performs memory copies (copySoC).

explained in setup section, we exclude the application initialization and IO.

To provide more insights, Figure 4.9 provides the normalized speedup of the CPU and the GPU component of the application. Although we improve both CPU and GPU execution time, avoiding the memory copy has a significant impact on the CPU execution time. In some applications like convo, SAXPY, SRAD and transpose, the resulting CPU execution time without memory copy is negligible. Avoiding the memory copies not only improves the CPU performance, but also the GPU performance. The reason is that it avoids cache pollution and reduces the LLC miss rate. Interestingly, two of the best performing applications (SAXPY and transpose) have a slight increase in the GPU execution time. We attribute this to the prefetching effect of the memory copy on the GPU.

The elimination of the memory copy has a direct impact on the speedup not only

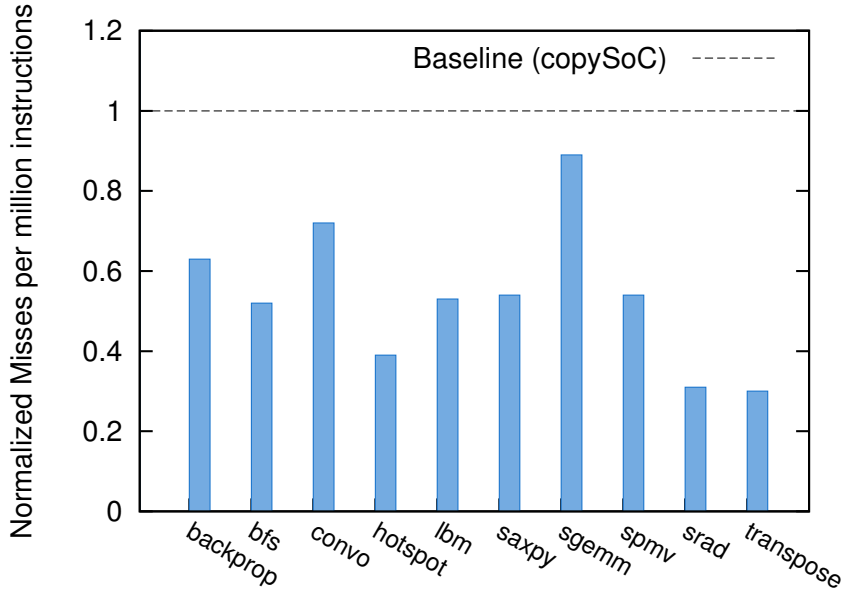


Figure 4.10: FuseTLB eliminates over half the LLC misses.

for applications where this can easily be expected (*i.e.*, those with a high transfer to compute ratio like SPMV or SAXPY (Figure 4.1)), but also for others like convo, SRAD, transpose and backprop. We attribute the speedups this set of workloads sees to the significant reduction in the number of last level cache (LLC/L3) misses. This reduction of almost 50% in the LLC miss rate can be observed in Figure 4.10 where we plot the number of misses in the LLC per million instructions observed in our proposed configuration (FuseTLB) normalized to the baseline configuration (copySoC).

Since our proposed solution eliminates the instructions needed for the memory copy, we also include Figure 4.11 that plots the IPC for the CPU and the GPU. It shows that the IPC increases accordingly for these applications. All data coming and going through memory copies in the baseline architecture can be now reused in the LLC. Data reuse is typical in many CUDA

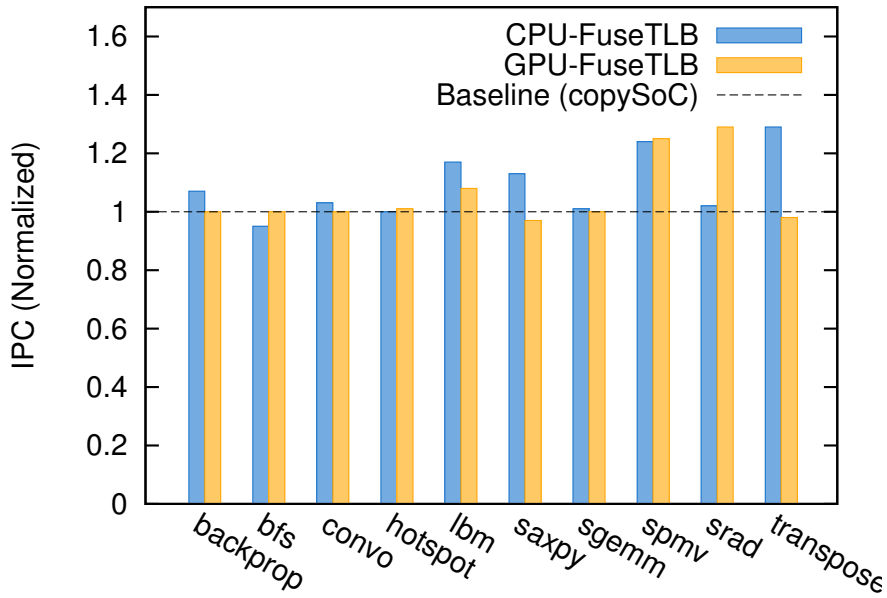


Figure 4.11: Both GPU and CPU have IPC improvements by avoiding memory copies.

benchmarks [36, 52].

The lower speedup logically corresponds to applications that have a high enough compute ratio or are highly divergent making them inherently slow, as is the case of BFS, hotspot and SGEMM. However, our proposal neither adds overhead nor does it make them slower. In fact, hotspot shows a significant reduction in the LLC miss rate.

FuseTLB achieves between 20 and 50 times speedups in applications like SAXPY, convo, and SPMV. These applications executed in a baseline architecture have much longer execution times when executed on the GPU rather than on the CPU. The proposed virtual shared memory for GPU-CPU multicores thus enables the GPU to excel in applications that currently have a transfer to compute ratio that make GPU execution expensive. Even if we exclude these

applications, we see a significant 49% speedup in the eight remaining applications.

4.6.3 Multiprogrammed Workloads

GPU workloads make typically a big pressure on the memory hierarchy, and we want to evaluate the impact of that behavior if a multiprogrammed CPU workload is sharing a die with FuseTLB as it could happen in an heterogeneous multicore. For that, a set of random combination of 4 SPEC benchmarks is run, both in isolation, and in conjunction with the GPU running an application. We compare the performance of each individual CPU application in the isolated multiprogram run against their performance in the runs coupled with a GPU application. This comparison highlights the impact of the GPU on the observed CPU performance.

As a metric, we evaluate both weighted speedup [76] and a harmonic mean of the speedup [55]. Equation 4.1 and Equation 4.2 formulate the metrics.

$$WeightedSpeedup = \frac{\sum^{n} \frac{IPC_{coupled}}{IPC_{isolated}}}{n} \tag{4.1}$$

$$HarmonicSpeedup = \frac{n}{\sum^{n} \frac{IPC_{isolated}}{IPC_{coupled}}} \tag{4.2}$$

Figure 4.12 shows the weighted and harmonic speedup. Both metrics agree on the throughput and fairness of FuseTLB. To compare the results against the baseline architecture, Figure 4.13 shows the box plot of the distribution of speedups across different architectures and for different metrics. The results indicates that FuseTLB has less impact on the CPU applications. This is because less traffic is generated due to no memory copy which reduces the pressure on the memory subsystem.

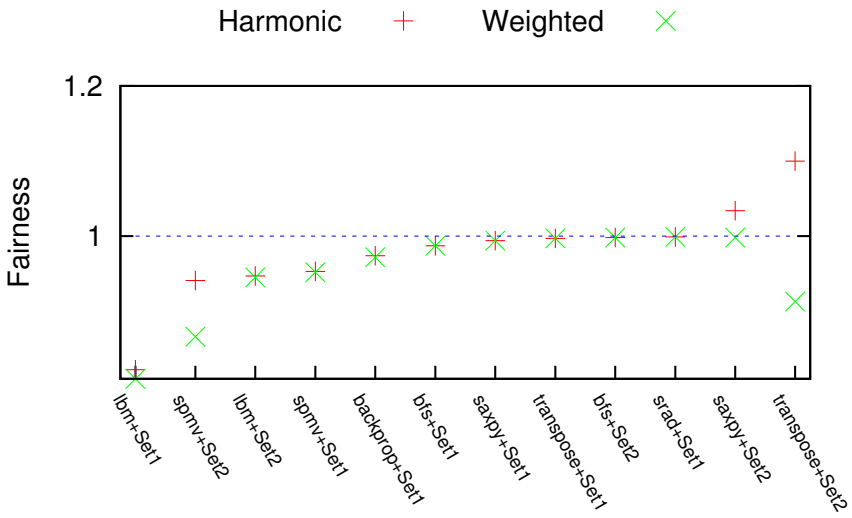


Figure 4.12: Fairness of the FuseTLB architecture running multiprogram applications on CPUs along with a selected benchmarks running on GPU

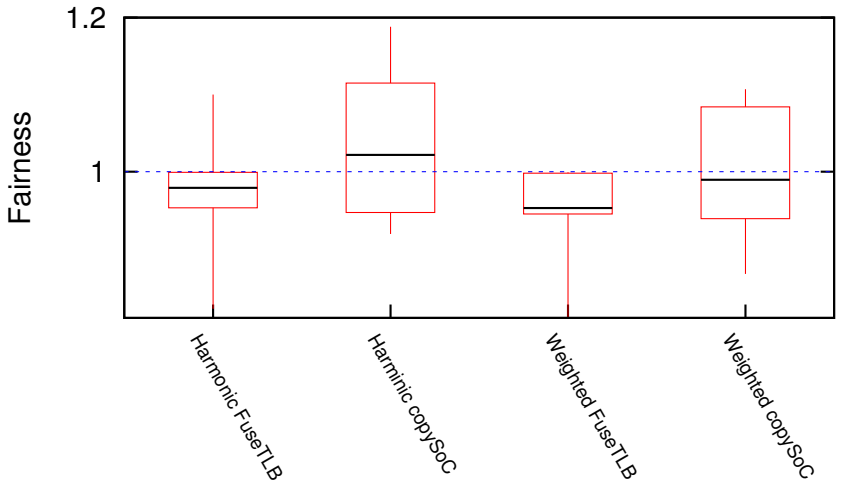


Figure 4.13: Shared and private TLBs have similar fairness.

4.6.4 TLB Characterization

To determine the size of the shared TLB, we varied the number of entries in the STLB and noted the impact on the IPC and the STLB miss rate. Figures 4.14 and 4.15 summarize our

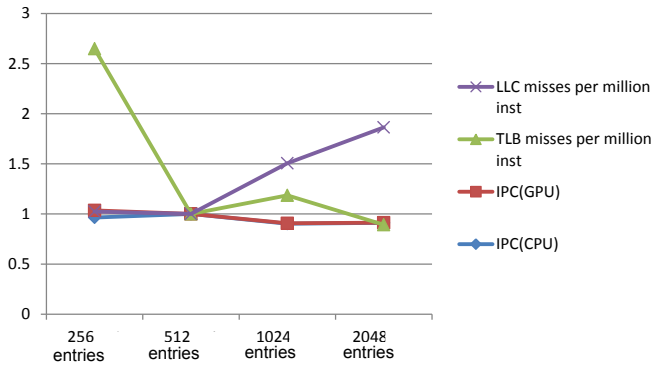


Figure 4.14: GPU workloads do not need many TLB entries.

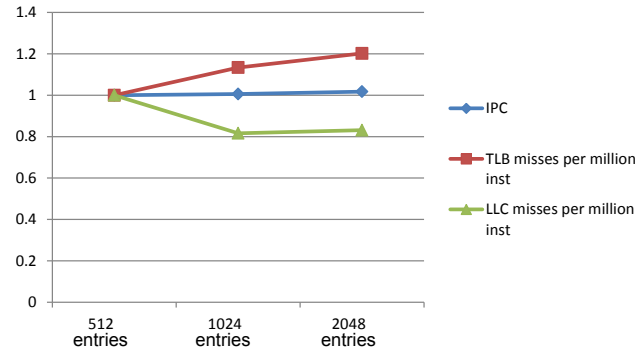


Figure 4.15: TLBs with more than 1024 entries are beneficial for Multi-programmed SPEC2006 rate applications.

results for a full set of GPU and multi-programmed SPEC2006 CPU workloads. Most systems have a TLB that has at least 512 entries, but we wanted to check if the GPU would be able to sustain performance with a smaller TLB with 256 entries. We eventually chose 512 entries to strike a balance between the performance and the energy. The main observation is that the GPU is more insensitive to smaller TLBs.

Figure 4.16 shows TLB misses per million instructions normalized to the baseline and compares the solution with STLB against private TLBs. It can be observed that misses are quite

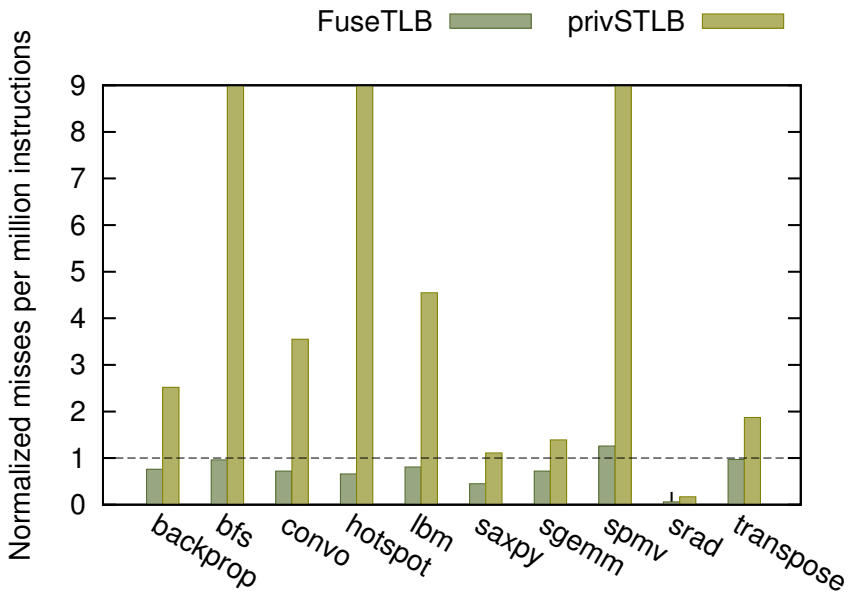


Figure 4.16: Private TLBs have more misses than shared TLBs.

high in the private TLB approach. Along with this high TLB miss ratio, the behavior of BFS has to do with another fact which is the elevated number of short (one byte) memory transfers that we have seen in our simulations. All these memory transfers become TLB accesses in our scheme with poor locality and a high TLB miss ratio.

The conclusion is that GPU applications are less sensitive to TLB misses than CPU applications, and the use of a shared TLB, aided by the prefetching and sharing of address space, reduces the TLB miss rate.

4.7 Summary

Integrating a CPU and a GPU on a single die opens up the opportunity to unify the address space and thereby eliminate costly memory transfers between the CPU and the GPU. This

presents several non-trivial design choices, and the trend seen in several new architectures from Intel and AMD is to propose a fully coherent memory system between the CPU and the GPU. We propose ***FuseTLB***, a simpler, energy efficient design to establish a virtual memory system for heterogeneous CPU-GPU architectures on a single die. The unique GPGPU programming models allow us to get away with simply a shared TLB between the CPU and the GPU cores to establish a protected shared memory space between them. We also propose a small, shared virtual cache between all the SMs to filter bursty requests from the SMs and reduce the pressure on shared resources, thus improving the energy efficiency. Our evaluation shows that the prefetching effects between the CPU and the GPU and the elimination of the memory transfers result in a speedup of over 49% and an additional 27% in energy savings. The addition of a shared L2 across all the GPU cores is also able to reduce the LLC miss rate by almost 50%. This CPU-only coherent memory hierarchy with incoherent GPUs is fully compatible with existing programing models like CUDA and OpenCL, and it provides 100% binary compatibility.

Chapter 5

An Energy Efficient GPGPU Memory Hierarchy with Tiny Incoherent Caches

Traditional CPUs are highly latency optimized: they have a few, very complex, high performance cores sharing a pool of memory and are extremely efficient at handling few threads. In contrast, throughput processors like GPGPUs employ hundreds of very simple cores to execute hundreds of thousands of lightweight threads to hide long memory latencies. It is this massive multithreading that helps GPGPUs achieve high performance. Several complex (and energy hungry) components are required to make this possible on the GPGPU, and two of the key structures are the highly banked scratchpad memory and on-chip data caches. These caches are shared across multiple lanes within a streaming multiprocessor on the GPGPU and play a big part in both the performance and the energy efficiency of the GPGPU. The first level data cache (DL1G) is a massive structure and, considering the support it needs to sustain requests from all the lanes and to coalesce memory, it is extremely energy inefficient. For the applica-

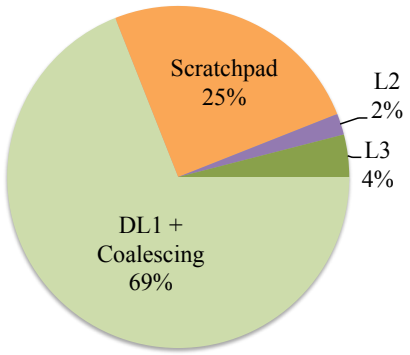


Figure 5.1: A breakdown of the energy consumption of the on-chip memory hierarchy in a typical GPGPU for the benchmarks listed in Table 6.3. The DL1G and the scratchpad memory account for most of the energy consumed.

tions we evaluated, we saw that in a typical GPGPU the DL1G accounts for almost 69% of the dynamic energy consumed by the on-chip memory hierarchy. With another 25% expended on the scratchpad memory, this amounts to a total of over 90% as seen in Figure 5.1.

In this chapter, we focus on the DL1G and the scratchpad memory and explore alternatives to make it more energy efficient.

5.1 Common Approaches to Energy Efficiency

Most of the research on efficiency in GPGPUs is steered toward improving their ability to sustain a high throughput (performance) in a discrete system, but with the newer generations of GPGPUs getting larger and the adoption of mobile GPGPUs, there is a growing emphasis on the energy efficiency of GPGPUs.

A common approach to energy efficiency is to make memory accesses to the memory hierarchy as efficient as possible. The goal here was to minimize the number of accesses to the

memory hierarchy and minimize the wastage of energy wherever possible. These ranged from judicious warp scheduling [31, 56, 62] to memory scheduling policies to exploit locality and boost performance [91]. Lee *et al.* study the effects of coalescing and different memory scheduling policies for a pre-Fermi memory hierarchy [51]. Gou *et al.* discuss the relation between the address pattern and warp locality [35] and use it to choose the access granularity based on the horizontal locality for a given memory instruction. They argue that the extra data requested would be used by neighboring warps in most cases, thus improving memory efficiency without the penalty of wasted memory bandwidth. Skadron *et al.* propose a software API in [22] that allows programmers to play with the data memory layout for a variety of common memory access patterns to extract locality and boost performance. Rhu *et al.* propose a locality aware memory hierarchy that is aware of the temporal locality in GPGPU applications [69], while Rogers *et al.* propose a scheduler that takes into account thread locality to limit the number of threads actively used per SP to take advantage of the intra- or inter-wavefront locality [72].

Gebhart *et al.* explore several register allocation algorithms and propose a compiler specifiable register file hierarchy that allows sharing of temporary register file resources among running threads, reducing the usage of this energy hogging resource [33], [32]. They also propose a unified scratch, register and primary cache that can be configured at runtime to minimize the access latencies [34]. Leng *et al.* propose DVFS on a SM-level to save more energy [53].

5.2 Feasibility of Adding Filter Caches

An orthogonal and seemingly straightforward solution is to add a small filter cache per lane to intercept frequent requests to the shared DL1G. The filter cache was first proposed for uniprocessors [46] and traded off a loss in performance for energy efficiency. The idea was extended to multicore processors using a victim cache at the filter cache level [64]. The problem is that if we have such a filter cache for each lane in the GPGPU, we will need to maintain coherence between all the lanes in an SM. A CUDA/OpenCL kernel usually spawns thousands of threads that cannot modify the same address without an explicit barrier, but they typically access the same cache line. This is a common strategy used by GPGPU programmers to maximize memory coalescing which directly impacts the performance (by reducing the memory bandwidth) of the application. This results in a sharing pattern with high false sharing, very different from one seen in typical multiprocessor applications, and makes it highly inefficient to have to maintain coherence between private filter caches per lane.

Instead of simple filter caches, we propose tinyCache, an incoherent, write-back cache *per lane* that leverages an adapted write-validate policy to maintain correctness. A tinyCache cache line can be merged back to the existing DL1G when the data is displaced without having to care about invalidations or updates which is made possible by the peculiarities of the CUDA/OpenCL programming model. We provide more details in following sections.

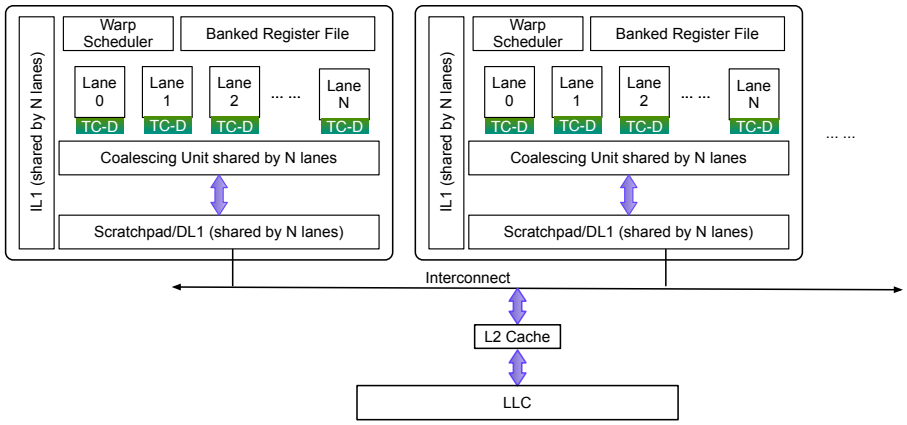


Figure 5.2: We propose adding tinyCaches per lane for energy efficiency These tinyCaches filter both scratchpad memory and global addresses.

5.3 Our Proposal: Add a tinyCache per Lane

As shown in Figure 5.2, the tinyCache is the first cache in the memory hierarchy for an individual lane, and all the global and scratchpad memory requests can be routed through it. The main idea behind this is to filter frequently accessed addresses and avoid a lookup from the lower levels. Since we are adding a level to the memory hierarchy, and since this cache lies in the critical path, it is extremely important to keep the access latency to this cache as small as possible. To maximize its ability to store frequently re-used data and exploit locality, we compare various configurations with a number of entries and different line sizes for the minimum energy delay product. Our sizing experiments, detailed in Section 5.5.2, helped us pick a tinyCache with 16 entries and 64B line size.

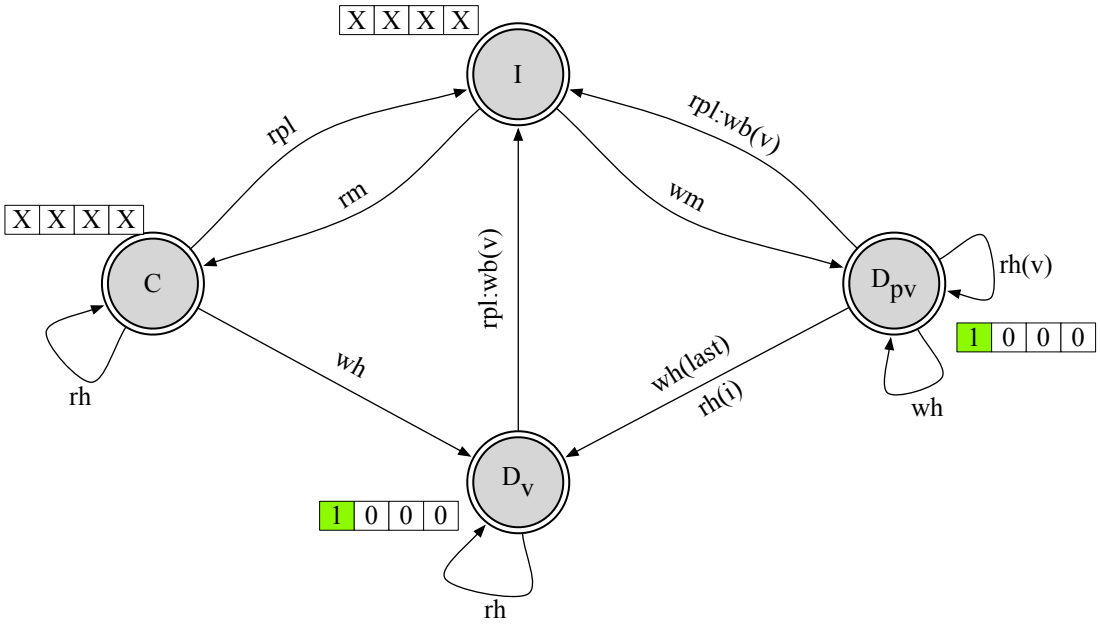


Figure 5.3: The tinyCache controller implements a write-validate write-on miss policy with write-back update. Each cache line has control bits per half-word (plus two state bits that are not shown). A shaded half-word means that the word was written on entering that state. In D_v, all the half-words are valid (both set and unset control bits denote *valid*), whereas in D_{pv} only the half-words marked *dirty* are valid (unset control bits indicate that the corresponding half-word is are *invalid*).

5.3.1 Behavior of the tinyCache

We adopt a write-validate write on-miss policy for the tinyCache [43]. Figure 5.3 shows the state diagram for a line in the tinyCache, detailing our implementation of the write-validate policy with write-back. We modify the role of the validity bits in the original protocol, and, depending on the state, they can act now as *don't care* bits, validity bits or dirty bits. We will henceforth refer to them as control bits. Our adaptation of the protocol requires four states which adds two bits per line, plus the control bits, to encode the line state (we omit the transient states here). A control bit per byte would guarantee correct operation with an overhead of 64

bits for a typical 64 Byte cache line. We analyzed typical applications and found that a word or half-word granularity suffices, so we selected the half-word granularity for our experiments.

All lines enter the invalid state (I) upon initialization of the cache. A read miss (rm) will make the controller fetch a clean memory block, setting the line in the clean (C) state (the control bits are *don't care* terms). A write hit on this clean line will switch the state to D_v, and the control bit(s) of the written half-word(s) will be set (meaning valid and dirty). The other control bits will be cleared (meaning valid and clean). Further write hits (wh) on a line in the D_v state will set the corresponding control bits. Note that any tag hit on a line in this state is also a half-word hit because all half-words are valid, irrespective of whether their control bit is set or cleared.

A write miss (wm) will allocate a cache line by selecting a victim, but will not fetch the missing block from memory to the cache, and the line will switch to the D_{pv} state. The control bits of all half-words will be cleared, except for those half-words that are written upon after the write miss. For the updated half-words, the control bit will be set, indicating that the half-word is both dirty and valid. Further read or write hits on valid (dirty) half-words in the line ($rh(v)$) imply no action. Write hits on an invalid half-word (i.e. tag hits, half-word misses) set the corresponding control bit. A write hit on the last invalid half-word in the line ($wh(last)$) will make the state change to the D_v state with no further action. A read hit on the line, addressing a half-word which is set invalid (i.e. a tag hit, word miss, noted $rh(i)$) will also switch the state to D_v, but the controller will fetch the block from memory and will merge it in the cache line with the half-words that have their control bit set.

The rpl event in the diagram stands for evictions. When a line is evicted I becomes

a transitory state leading to one of the three other states. The termination of a thread block triggers an invalidation of all the tinyCache lines in the SM where the block was executing. The rpl event on a line in the D_v or D_{pv} states triggers a write-back of the valid (dirty) half-words ($rpl : wb(v)$).

5.3.2 Maintaining Coherence

The biggest implication of having a cache per lane is the task of maintaining coherence across the lanes, since one or more lane may cache the same line from either the scratchpad memory or global memory. However, the CUDA/OpenCL programming model allows us to take certain liberties to establish coherence between these tinyCaches with a lower overhead. There are five scenarios to consider:

- **Some lanes write to different locations in scratchpad memory/global memory**: A typical programming pattern in CUDA applications is that consecutive threads reference consecutive memory elements and therefore consecutive lanes will share the same memory block in their tinyCaches, referencing only a part of it. This false sharing would wreak havoc on a coherence protocol when it comes to writings. The write-validate, write-back protocol described earlier let us obviate any invalidation or update across the tinyCaches. The CUDA/OpenCL programming model only ensures sequential consistency on writes by a single thread [6]. A thread reading a shared variable will not see a change made by another thread unless explicitly using a barrier, irrespective of whether or not these threads belong to the same warp or block, or whether they access the scratchpad memory or the global memory. Assume x(0) and x(1) are two array elements lying on the same

cache block. Thread 0 writes element x(0) and then reads element x(1), whereas thread 1, which is running concurrently, writes element x(1) and then reads x(0). The model does not allow any assumption on the values read from x(1) and x(2). On the other hand, both threads will have a copy of the same memory block in their tinyCache, in D_v state, with different control (dirty) bits set. When these blocks are replaced, the dirty half-words will be correctly written-back to memory. It is precisely this subtlety that we exploit to use incoherent caches.

- **Some lanes write to the same location in scratchpad memory/ global memory**: Since there is no ordering of threads, the programming model does not guarantee which thread will first write to the address [6]. Thus, we do not need to take further actions beyond the ones implicit in the write-validate policy with write-back as described above.

- **Atomic operations**: Atomic operations need to guarantee a consistent view of the memory. To avoid the coherence overhead, we do not cache these atomic addresses in the tinyCache.

- **Synchronization primitives (barriers)** : When we hit a barrier, we evict all the tinyCache entries.

- **A single byte write access cached by the tinyCache**: Since the cache line only has a control bit per half-word, we do not cache such addresses. Effectively, we trigger an eviction if the address was already cached or just bypass the tinyCache if the address was not cached.

Local memory is transparent to the programmer. It is a specially allocated area of the global memory used for spill code. Since there have been numerous techniques that improve the utilization and effective capacity of the register file and given the low frequency of its occurrence, we reserve the tinyCache only for the global and shared addresses.

5.3.3 Area Overhead

A control bit per half-word implies 32 control bits per line for a tinyCache with 64 B line size. The tag SRAM has two bits for the state plus 79-bit tag for a total of 81 bits per line. The tag is a part of the extended address that can indicate if a given reference is a global or a scratchpad memory reference as well as the block-id if needed. Thus, each tinyCache line needs 15B of meta data. Therefore each 16-entry tinyCache costs us just over 1 KB (1264 B), 19% of which is control overhead. In all, the tinyCaches account for about 9% of the area of the GPU on-chip memory hierarchy. Our estimation is based on GPUSimPow [54] including coalescing, shared memory, constant and texture cache, and the L2 cache, but excluding the LLC.

5.4 Experimental Setup

We compare the architecture we propose in Figure 5.2 with the baseline depicted in Figure 2.3. Following the trend of integrating GPUs and CPUs on a single die sharing caches [92], we incorporate an LLC in our baseline similar to [50, 90, 92]. Some relevant configuration parameters are listed in Table 5.1. Our proposal to add a tinyCache per lane, shown

Parameter	Configuration
SM cores	4 SMs, 1.5 Ghz
Number of lanes per SM	32, in-order
Memory Coalescing	Enabled
Maximum concurrent blocks per SM	8
Maximum concurrent warps per SM	24
Tinycache per lane	1KB / 8-way / 64B line /1 cycle
Scratchpad per SM	48KB / 8 banks / 18 cycles
L1 cache per SM	32KB / 8-way /128B line/ 18 cycles
L2 cache per die	256KB / 16-way /128B line/ 7 cycles
LLC per die	8MB / 32-way / 128B line /14 cycles
Memory	18GBytes/s BW with 50ns access time

Table 5.1: GPSim simulation parameters to evaluate tinyCaches

in Figure 5.2, is built on top of the baseline. As mentioned before, the tinyCaches were sized and parameters were chosen on the basis of the energy-delay product (ED) and the IPC.

We use CACTI [84] to estimate the latencies of the memory structures, except for the DL1G, whose complex architecture is not modeled well by CACTI. We rely on measurements and published Fermi latencies that estimate the DL1 latency to be around 18 cycles, which is consistent with related work [32, 92].

Our energy estimations include only the on-chip dynamic energy, and are based on the GPUSimPow power model and CACTI. We do not model the DRAM. As far as the tinyCaches go, we do not treat the line fills and coalescing requests like other regular requests; they are more expensive and accounted for separately. We expect the same leakage as with DL1 memory structures (without coalescing and crossbars) which should be less than 10% of the leakage of the total on-chip memory hierarchy.

The benchmarks we use to evaluate our proposal are from Rodinia [21] and Parboil [78] in addition to a few regular benchmarks from the CUDA SDK [2]. Table 5.2 lists and briefly describes the GPU workloads that we use in our evaluation. All benchmarks from

Benchmark	Description
Backprop	A machine learning algorithm used in a graph
BFS	A graph traversal algorithm
Convolution (convo)	An image processing algorithm
HotSpot	A tool to estimate the temperature for an architectural floorplan
SAXPY	A common subroutine which performs $z = \alpha * x + y$
SGEMM	Matrix Multiplication
SPMV	A commonly used implementation for multiplication of sparse matrices
SRAD	Used to remove locally correlated noise in an image
Transpose	Compute the transpose of a matrix

Table 5.2: GPU workloads used in our evaluation.

the standard benchmark suites were run to completion with the largest dataset made available. SAXPY uses 8MB arrays. Convolution was performed on a 3072×3072 sized 2D image. Transpose uses a 2688×2688 square matrix as its input.

5.5 Evaluation

Section 5.5.1 presents our main results and emphasizes the ability of the tinyCache to meaningfully filter out accesses to the DL1G, cutting down the energy. The sizing of the tinyCache is presented in Section 5.5.2.

5.5.1 Main Results

Different benchmarks exhibit different memory patterns, dominated by global references, scratchpad memory references or both. To maximize the potential of the limited entries in the tinyCache, each benchmark needs a specific set of references to be cached. We ran experiments where the tinyCache was allowed to cache only the global or only the scratchpad memory or both references and noted which configuration was the most effective in minimizing

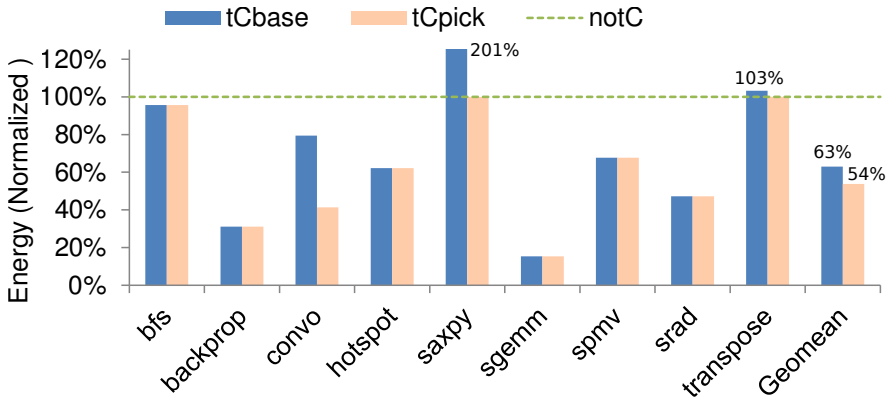


Figure 5.4: Addition of a tinyCache per lane reduces the dynamic energy consumption of the on-chip memory hierarchy by 37%.

the energy delay product for each benchmark.

Typically programmers are encouraged to make use of the scratchpad memory, and we see that caching these references does not affect the caching ability of the tinyCache for global references. We pick this configuration as our default policy and refer to it as ***tCbase***. To highlight the potential of tinyCaches for energy savings and their impact on performance we compare *tCbase* with a handpicked optimal configuration per application ***tCpick*** which could be a configuration that cached either one or both global and scratchpad memory references, offering maximum savings in the energy-delay product. Both are normalized to the baseline with no tinyCache ***notC***.

A well-designed tinyCache will be effective in filtering out frequent expensive requests to the DL1G and replacing these requests with energy-efficient accesses to the tinyCache. Figure 5.4 highlights that our base policy, *tCbase*, which caches both the global and scratchpad memory accesses, is able to achieve a significant reduction of around 37% in the total dynamic energy consumed by the on-chip memory hierarchy. Note that *tCpick* is just *tCbase* in many

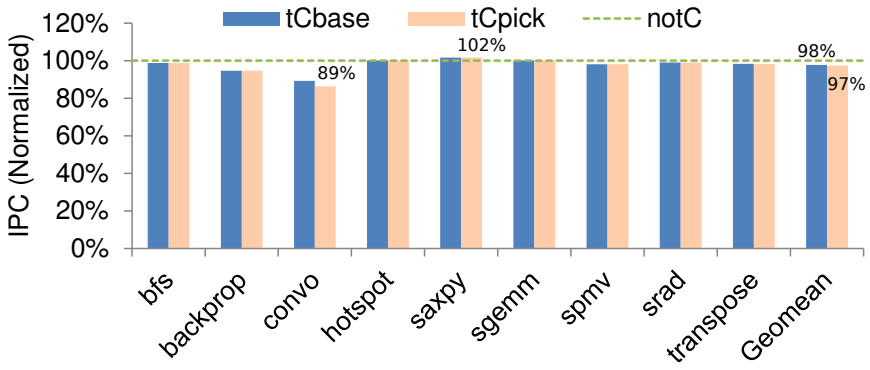


Figure 5.5: The IPC remains largely unchanged for most benchmarks, with a 2.3% reduction seen on average.

benchmarks except in the case of convo and SAXPY. SAXPY performs significantly worse with *tCbase* policy. SAXPY is a streaming application with no re-use of the data it fetches or intra-thread locality. In addition, each miss on the tinyCache entails an expensive line fill operation from the DL1G. Thus the presence of the tinyCache only adds overhead without filtering any traffic from DL1G. As a result, it is not beneficial to use a tinyCache with such applications. Since our emphasis for this paper is energy efficiency, *tCpick* for SAXPY is essentially the same as *notC* since by bypassing global we do not cache any references in the tinyCache. We could disable the tinyCache by exposing it through the API to the driver that spawns the kernels, similar to disabling L1 caching in the Fermi. We could also potentially detect streaming accesses in hardware and disable the tinyCache transparently; we leave this exploration to future work.

Figure 5.5 shows that the IPC per lane does not vary significantly across different policies. This is because of the large number of threads available on the GPGPU for interleaving which makes it possible to tolerate a wide range of memory latencies. We note that unlike the increase reported in SAXPY's energy numbers, the tinyCache does not hurt its performance. Transpose is a similar case. This application is right behind SAXPY when we consider the tiny-

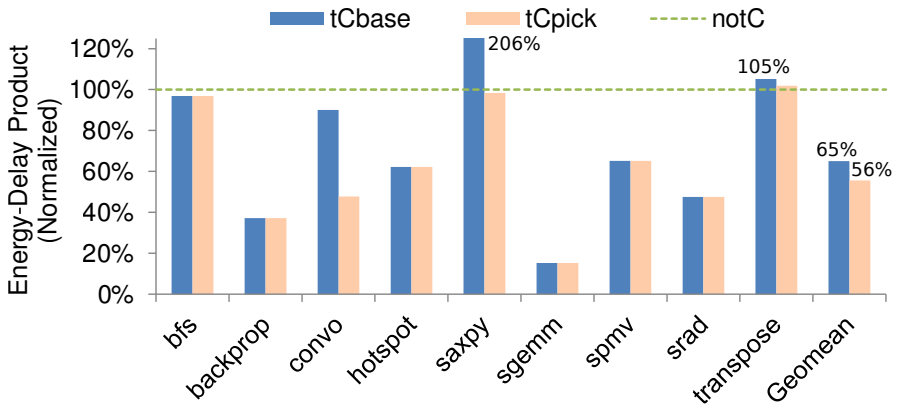


Figure 5.6: We see a 35% reduction in the ED after the addition of a tinyCache per lane

Cache miss ratio and yet the performance is not affected. This is because the latency added by the tinyCache is very small compared with the rest of the memory levels and is easily engulfed by the warping mechanism. We see a similar decrease in the IPC even if we opt for the *tCpick* policy.

The Energy Delay product (ED) in Figure 5.6 closely follows the IPC in Figure 5.4. The average reduction in the ED with respect to the baseline in Figure 5.6 highlights how tinyCaches achieve substantial energy savings for the moderate performance losses observed in Figure 5.5. Applications like backprop, SGEMM, SPMV and SRAD benefit from a tinyCache: they show a large decrease in energy consumption (68%, 85%, 32% and 52% respectively) for a negligible loss in performance. Convo yields good numbers if we bypass the scratchpad memory references and provide more space for the global references, but a streaming application like SAXPY does not benefit at all.

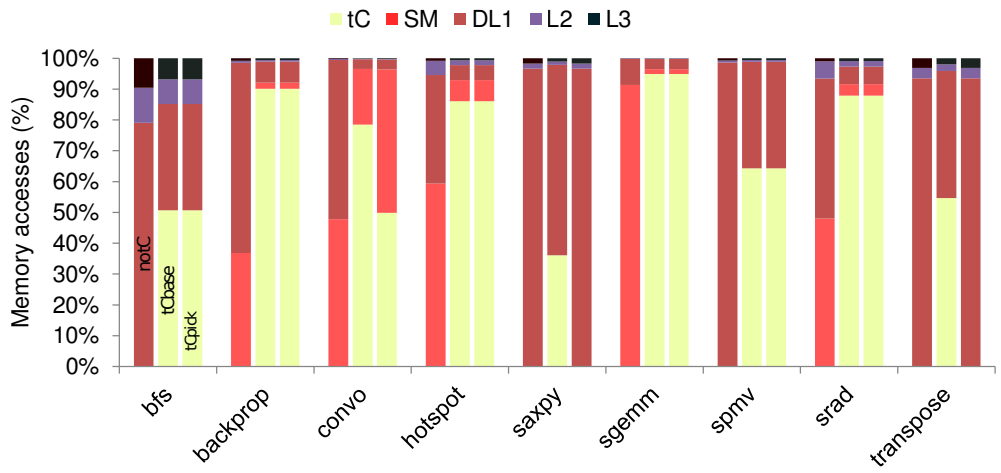


Figure 5.7: A breakdown of the memory accesses seen by each cache in the on-chip memory hierarchy for the baseline, the baseline tinyCache and the best pick per application. We see a 61.8% and 81% reduction to the number of accesses to the DL1G and scratchpad memory respectively.

Figure 5.7 shows the breakdown of memory accesses to each memory level, normalized to the baseline (no tinyCache). The leftmost bar in each group plots the baseline breakdown as a reference. The bars in the center and on the right stand for the base tinyCache (*tCbase*) and the best pick per application (*tCpick*) according to Figure 5.8. We observe that the number of accesses to DL1G drops by almost 62% on an average and by almost 81% for scratchpad memory references. Note that an access to the tinyCache is significantly less expensive compared to DL1G. This is because an access to the tinyCache does not require going through the coalescer and the address and data distribution network which are expensive parts in terms of energy consumption. The L2 cache shows a minor improvement as well. This is probably because the tinyCache is able to retain references that would otherwise have been displaced from the DL1G incurring a costly L2 reference.

We should note that memory accesses that would have coalesced in a typical GPGPU

memory hierarchy still have the same benefits in the tinyCache configuration except that they may have less data reuse in the tinyCache. Effectively, all the tinyCaches have the same address cache misses and these tinyCache misses are coalesced. Due to evictions at different times, we could potentially have a higher number of DRAM accesses. The worst case is evident in SAXPY, where the number of DRAM accesses nearly doubles with tinyCaches, increasing from a 1.7% global miss rate to 3%. Nevertheless, this is the worst benchmark and, on average, we maintain the same global miss rate of 1.5% across applications.

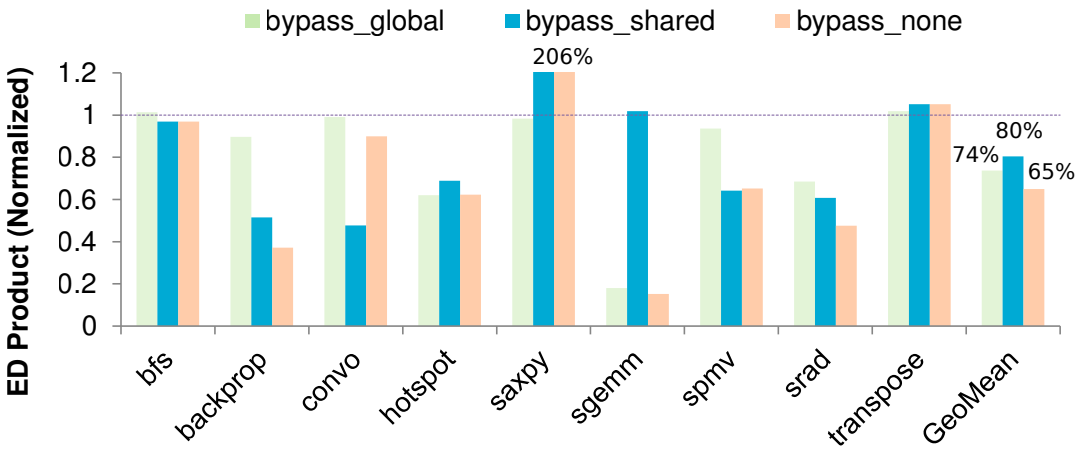


Figure 5.8: A tinyCache which caches both global and scratchpad memory references provides the most optimal savings in ED (and E) (**bypass_none**) as compared with a configuration caching only global references (**bypass_shared**) and one caching only scratchpad memory references (**bypass_global**).

We see that a tinyCache that caches both the global and scratchpad memory references is often the configuration that yields the best savings in Energy (E) and the ED product as seen in Figure 5.8. The long latencies associated with global references make them a good candidate for caching. However, this is not necessarily the optimal choice for all benchmarks. For example, an application like SGEMM which is highly scratchpad memory intensive prefers a configuration where we cache only the shared references and exploit the spatial locality exhib-

ited. On the other hand, convo, which involves both global and scratchpad memory references, prefers caching global references only. The reason behind is how consecutive threads reference neighboring array elements in convo. When these elements fall in different cache lines some thrashing may occur. We have observed that this effect strongly decreases in convo as we increase the size of the tinyCache.

5.5.2 Sizing the Tinycache

Determining the size of the tinyCache is a crucial step. In the architecture we propose, we add an extra level in the memory hierarchy –tinyCaches– between the lane and the DL1G to save energy by reducing the distance traversed by on-chip data transfers as close as thread locality allows, filtering requests that otherwise pass through the coalescing logic and go to the DL1G. This may sacrifice performance on behalf of the energy savings if the hit rate in this new level is low, because all the misses incur an additional delay. The tinyCache thus needs to be large enough to retain as many memory references as possible to maximize the hit rate, but conveniently small for energy efficiency. Since we know that memory access patterns are usually quite varied, a poorly designed cache might not be able to provide the benefit we seek. Our goal in picking a configuration was primarily energy efficiency, but not at the cost of a significant performance difference. We thus use ED as our primary metric to pick the tinyCache configuration.

We observe that the IPC per lane hardly fluctuates along the design space by changing the configuration (Figure 5.9) of the tinyCache. Figure 5.10 plots the normalized E and ED. We have chosen 16 entries and a line size of 64 B, as it is the smallest configuration with minimum

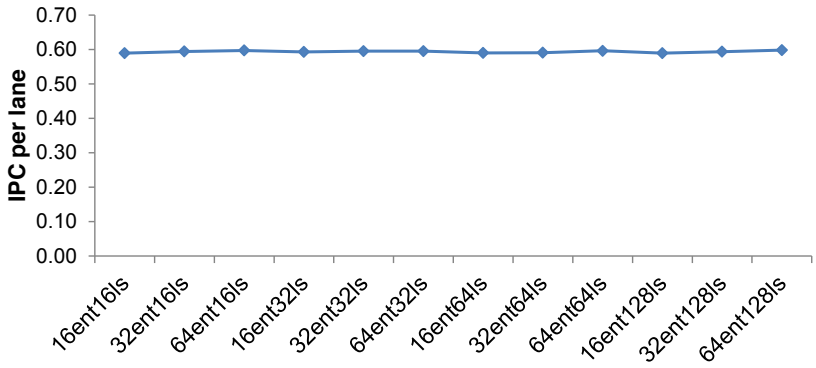


Figure 5.9: Impact of varying the line sizes and entries in the tinyCache on the IPC per lane

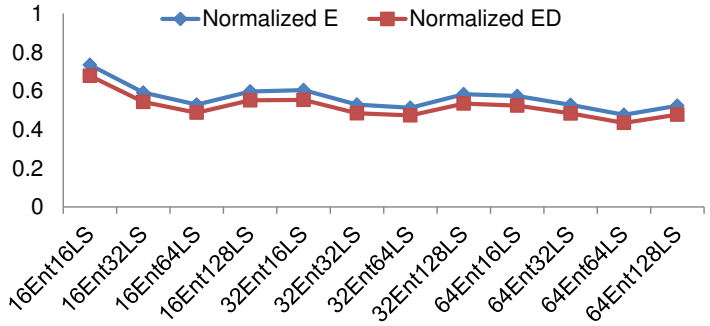


Figure 5.10: Impact of varying the line sizes and entries in the tinyCache on Energy (E) and Energy Delay (ED).

ED, to keep leakage low. In any case, as mentioned in Section 5.5.1, the overall leakage on the tinyCaches of the 32 lanes in this range is negligible compared to the rest of the system, and will likely be compensated by the reduction in temperature on DL1 and the coalescer.

5.6 Summary

In this chapter, we propose adding a tinyCache per lane in the GPGPU to filter out requests to the energy inefficient DL1 and shared memory to save energy. We do this by exploiting features of the unique programming model and avoid incurring the overhead associated

with coherence. We see a substantial saving of roughly 37% of the energy consumed by the on-chip cache hierarchy on average and a 35% reduction in the energy-delay product. While there are some memory access patterns that can benefit with larger tinyCache capacities or by caching more types of references, the difference is not large enough to justify a complex adaptive mechanism. The tinyCache also makes it possible for us to think beyond the data access patterns typically used while writing GPGPU applications and exploit locality in the tinyCache to gain further savings.

Chapter 6

EESI: A Simple Architecture for GPGPU and CPU Workloads

The key to maximizing performance in a GPGPU with underlying SIMD hardware is to maintain the execution throughput, *i.e.*, to keep a steady stream of instructions available for the lanes to execute. With the help of programming models that offer an abstraction of a complex hierarchy of threads, shared memory, and synchronization primitives, it becomes easier to map applications that are more general purpose onto the SIMD hardware. While the programming model allows each thread to maintain its own logical program counter, optimal performance demands minimal divergence across threads. This poses a limitation on the kinds of general purpose applications that can be ported on to the GPGPU, and, even today, we see that most of the applications that are ported to a GPGPU fall into the highly data parallel or extremely regular category of applications.

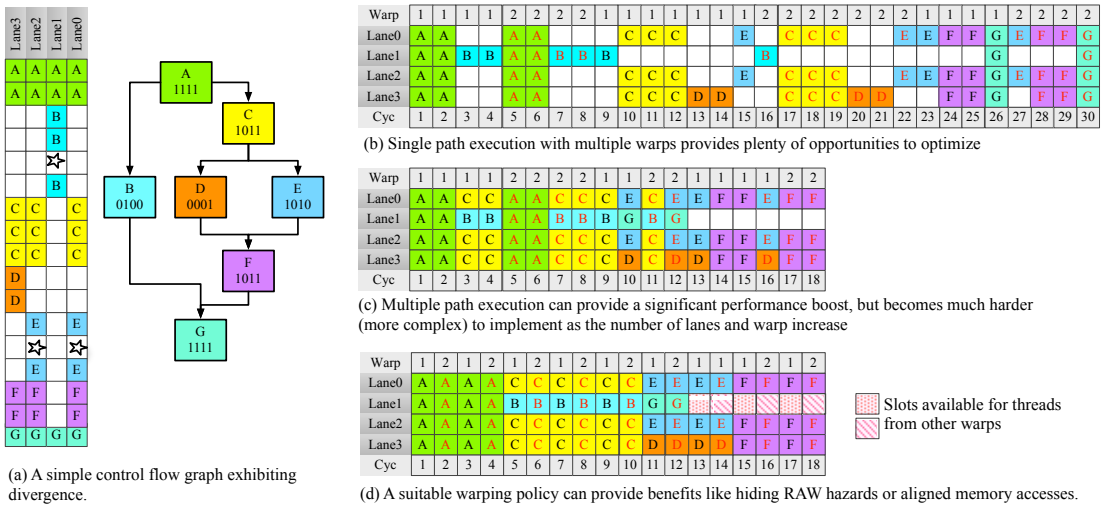
There have been numerous proposals to mitigate the effect of divergence, ranging

from supporting concurrent execution of a few divergent paths, to a newer architecture with a few semantic changes to the programming model. However, the one thing that is common to all these proposals is the complexity they add to the design as summarized in Table 6.1. This usually implies an area overhead and more energy consumption and, in spite of all this, still being unable to support truly divergent general purpose applications that are designed for MIMD machines. We discuss these proposals in greater detail in Section 6.2.

In this chapter, we present the ***Execute Everything Simply*** (***EESI***) architecture. By adding a tiny instruction cache, and an IF per lane in a GPGPU, and providing the requisite support, we allow the lanes to diverge freely and efficiently execute divergent applications. The SM morphs into a MIMD-like cluster of simple in-order cores, and, with this new design, we can support the execution of both traditional GPGPU and CPU (MIMD) workloads.

6.1 Execution of Divergent Applications on a GPGPU

Figure 6.1a shows the example of a simple control flow graph. In the Single Instruction Multiple Threads (SIMT) model [2], a single instruction queue feeds all the individual lanes (much like its SIMD cousin). Thus, all the lanes run the same instruction in a lock step fashion. When a branch instruction is encountered, the execution is typically serialized: threads taking one of the divergent paths are first executed followed by the threads taking the other path. GPGPU architectures typically manage branch divergence using two approaches. Intel architectures maintain a per-warp single PC (program counter) and a per-thread logical PC, masking out the results of those threads whose PC does not match the per-warp PC [26]. AMD and



(a) A simple control flow graph exhibiting divergence.

(b) Single path execution with multiple warps provides plenty of opportunities to optimize

(c) Multiple path execution can provide a significant performance boost, but becomes much harder (more complex) to implement as the number of lanes and warp increase

(d) A suitable warping policy can provide benefits like hiding RAW hazards or aligned memory accesses.

Figure 6.1: Execution of a divergent kernel on a SIMT architecture and opportunities to mitigate divergence in GPGPU applications

NVIDIA resort to a hardware mask associated with each branch which records which threads follow either the true or false path. All threads will actually execute but the output produced by the threads following the false path is masked out. Since branches can be nested and threads reconverge after a number of instructions in most branches, hardware masks are stored in a reconvergence stack [89].

Serial execution of threads can break the carefully coalesced memory accesses. To avoid this, the threads have to reconverge back before they start executing in lock-step fashion. Thus single path execution proves to be expensive, and the performance usually plummets. To circumvent this, programmers are encouraged to have minimal divergence in their code.

Proposal	Main hardware implications
DWF [31]	Warp update Register, PC-Warp LUT, Warp Pool, Warp allocator LUT
DWS [56]	Warp Split Table and changes in the scheduler
LWM [62]	Changes to RF, table of active masks, temporary active masks buffers
SBI [18]	Changes to RF and Scoreboard. 2 Contex Tables, Instruction buffer
DPS [68]	Duplicated scoreboard, changes to the scheduler, Dual-path stack
MPIPDOM [27]	Changes to scoreboard, extensions to Instruction Buffer and Reconvergence Table
TBC [30]	Warp buffers, Thread Compactor
CAPRI [67]	Prediction table, Warp Compaction Unit
BS [20]	None (software)
ID, BD [39]	None (software)
BH [73]	None if software, negligible if hardware
VT [47]	Complete GPGPU and ISA redesign
MSMD [83]	Complete GPGPU and ISA redesign
VWS [71]	Per lane instruction L1, new datapath, Warp Gang Unit

Table 6.1: Proposals to mitigate the effect of divergence in GPGPU applications

6.2 Proposals to Counteract the Effect of Divergence

There has been a substantial amount of research on techniques to improve the performance of divergent applications on GPGPUs. While some stick to single path execution model and try to group threads executing the same path dynamically, there have been others that propose multiple path execution that allows different lanes to concurrently execute different PCs as shown in Figure 6.1b. To do this, there are different types of hardware additions to the architecture such as control logic [71], instruction buffers and other forms of storage [30], additional tables [18] and interconnections like crossbars [83]. In addition to not being scalable (*i.e.*, each proposal can only support a predetermined number of execution paths), these designs add to the complexity of the already complex GPGPU and do not offer the flexibility of a MIMD-like design which supports truly divergent, general purpose code. Most of these proposals can be assigned to one of three groups listed below.

6.2.1 Mechanisms Based on Modifications to the Reconvergence Stack

In a seminal work, Fung *et al.* [31] proposed Dynamic Warp Formation (DWF) which dynamically regroups threads into new warps on the fly following the occurrence of diverging branch outcomes. This can increase conflicts on the register file which is tough to solve in some situations. They first apply immediate post-dominator reconvergence (IPDOM) [59]. They show that IPDOMs are not always the best reconvergence points so they also leverage threads from different warps that arrive at the same PC. Dynamic Warp Subdivision (DWS) [56] decomposes large warps into smaller warp-splits upon branch or memory divergence. Each warpsplit has its own divergence stack and is scheduled independently from the others. Like in conventional divergence, when a warp-split is running, it only uses as many lanes as threads in the warp-split; the others are idle. The focus is just to let other warp-splits in the same braid interleave when one of them misses in the cache.This requires code instrumentation to avoid over-subdivision, modification of the scheduler and of the per-warp scoreboard, and a warp-split table in addition to the conventional reconvergence stack. Diverged splits reconverge at the immediate post-dominator (IPDOM) of a subdividing branch but the IPDOMs of branches nested within the subdividing branch are ignored. To compensate, DWS uses heuristics that must carefully balance SIMD utilization with thread level parallelism (TLP).

Gebhart *et al.* also leverage the basic idea of DWS while considering the latency of references to large register files [32]. Alternatively, the Large Warp Microarchitecture (LWM) [62] uses fewer but larger warps, assuming that even in the presence of branch divergence there will likely be a large number of active threads in the large warp. Large warp masks identify active

threads which can be dynamically compacted together into fully populated sub-warps that better utilize the SIMD resources on the core. This calls for a two-level warp scheduling which splits all concurrently executing large warps into smaller fetch groups and prioritizes warps from a single fetch group until they reach a divergence point.

Brunie *et al.* suggest interleaving parallel instructions from different branches within the same warp (SBI) and/or different warps executing the same branch (SWI) [18]. Code and scheduling is subject to thread-frontier reconvergence constraints, instead of stack reconvergence [26]. To enforce reconvergence at the earliest point, they employ a synchronization instruction at each reconvergence point, which is treated as a synchronization barrier among warp-splits.

Dual Path Stack (DPS) [68] extends the reconvergence stack to push both taken and not-taken paths as a single entry. If we let both branches run in parallel, an architectural register accessed from both branches should be mapped to different physical registers to avoid false dependencies for which they design separate scoreboard units. The warp scheduler needs also to be redesigned. Like SBI, it shows poor benefit on benchmarks with multi-path or unstructured divergence.

The Scalable Multi-Path Microarchitecture (MPIPDOM) [27] is similar to DWS in that warps branching true and missing in the cache can switch to warps branching false and vice-versa, but it only applies to branch divergence. Instead of extending the reconvergence stack they record the reconvergence points in a reconvergence table which can spill to memory. A Splits Table (ST) keeps the state of the warp splits executing in parallel like the warp split table in DWS. A mask added to each field of the scoreboard records active threads in warps.

The instruction buffer is also modified to accommodate an entry per warp split.

Thread Block Compaction (TBC) [30] employs a block-wide instead of a warp-wide reconvergence stack allowing threads from different warps to be compacted into new warps depending on the branch outcome in a way close to LWM. It requires a warp buffer which is an instruction buffer with entries augmented with the thread IDs of the compacted warps. Compaction techniques like TBC or LWM require unnecessary synchronization which is alleviated in CAPRI [67] by dynamically identifying the compaction effectiveness of a branch, only stalling threads that are predicted to benefit from compaction.

6.2.2 Software Mechanisms to Minimize Divergence

Code transformations can also improve performance in the presence of branch divergence. Branch Splitting (BS) [20] splits parallelizable loops containing branches into multiple, single-path loops. Han and Abdelrahman [39] propose *iteration delaying* (ID) which targets branches enclosed in a loop and makes all the threads go through both taken and not-taken path in successive iterations. Unlike the Branch Herding proposal [73], they mask any output from threads taking a false path. They also propose *branch distribution* (BD), which reduces the length of branch paths by factoring out structurally similar code.

6.2.3 Alternative Architectures

There are a number of proposals that require a more drastic redesign of the microarchitecture. The Vector-Thread Architecture (VT) [47] was a seminal hybrid proposal of the vector and multithreaded model which allows to freely intermix vector and thread-fetches. Thread-

fetches enable vector processors (VP) threads to follow fully independent control paths. Instructions are packaged in atomic instruction blocks (AIBs). When running data-parallel code, a control processor broadcasts AIBs to all the slave VPs. In thread-parallel code, each VP directs its control flow by fetching its own AIBs. VPs hold their own register file with physical or logical registers depending on the implementation.

MSMD [83] is another non-standard GPU core capable of running SIMD fragments in MIMD manner. The SIMD fragments run as subsets of datapaths actively executing, while other subsets are idle because of narrower vectors than the available processing elements or because of divergence. It resorts to a few instruction buffers (IBs) shared among the lanes through an interconnection network. MSMD is very sensitive to the number of IBs because of buffer conflicts, and basic blocks are statically scheduled to the IBs which poses a number of problems. Syncing divergent datapaths requires syncing instructions but they do not provide an algorithm or heuristic that determines where to place them or which threads should synchronize. The Variable Warp Size architecture (VWS) [71] enables divergent applications to execute multiple control flow paths while keeping the lock-step SIMD model in non-divergent cases. There is an L1 instruction cache per SM. The datapath is split into eight slices capable of independently fetching, decoding and issuing instructions when required, but *warp ganging unit* forces these independent slices to operate in lock-step gangs when they run non divergent threads. Therefore, warps can dynamically vary in size with *ganged warps* operating in SIMD mode and *unganged warps* simultaneously running in MIMD mode on independent slices. Unlike our proposal, which can run MIMD and SIMD workloads in an energy efficient way, VWS can only execute SIMD workloads, by optimizing branch or memory divergence in them.

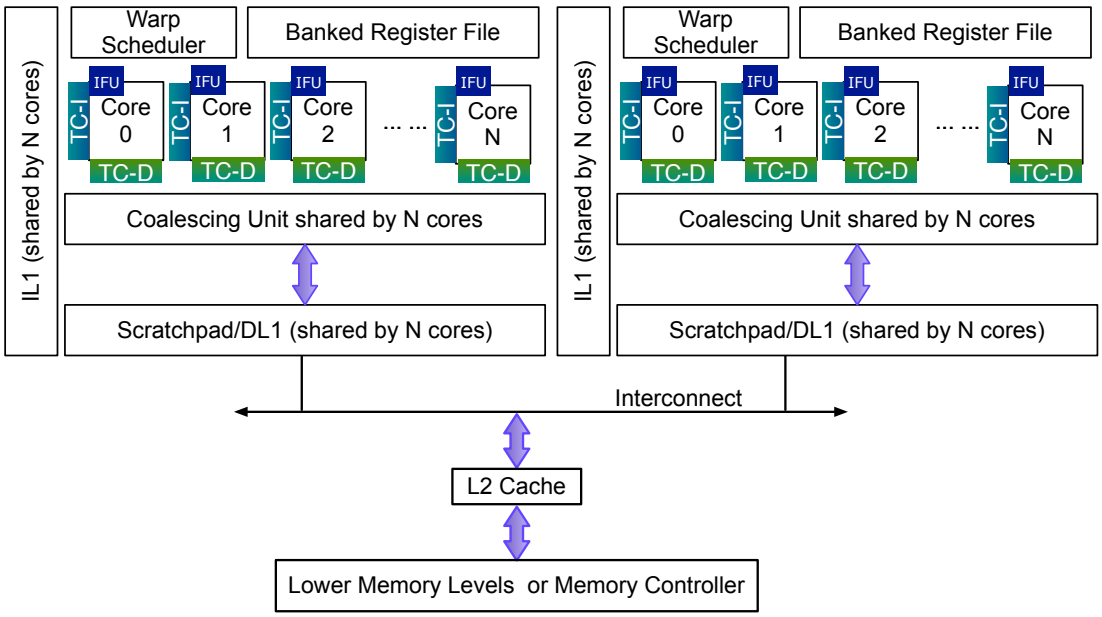


Figure 6.2: We propose EESI, a more MIMD-like design for energy efficient handling of divergent apps

An entirely different approach is Branch Herding (BH) [73] which forces all threads in a warp to take the same branch patch or to load the same memory address, increasing performance in error-tolerant GPU applications either by software (relying on the recent CUDA *__ballot* and __popc intrinsics) or by hardware at negligible cost.

6.3 The EESI Architecture

In an effort to support multiple divergent paths we first break away from the SIMT design of having a common instruction fetch and buffer shared across all the lanes. Instead, we begin with a design where each core has its own IF unit, a local instruction and data cache, and interfaces with the other cores through the memory hierarchy. To achieve this, we thus supplement each lane in our GPGPU with it own instruction fetch and buffers. Instead of

directly interfacing with the large IL1 cache available per SM, we add a *tiny instruction cache* or *tinyIcache (TC-I)* closer to each core. The tinyICache is small enough to provide an obvious benefit over not having to access the larger, slower and more energy inefficient IL1 each cycle, but at the same time large enough to withstand a large number of potentially divergent warps in flight. We evaluated this configuration over a large design space and propose a tinyIcache with 8 entries, each 64B wide. More details about the sizing are provided in section 6.5.4.

There is a lot of thought that application developers give to grouping the threads smartly so as to align memory accesses to the data cache. This is done to maximize the coalescing ability (and thus reduce the coalescing penalty) as well as to minimize pollution in the shared DL1 cache. Disturbing these memory access patterns could lead to more off-chip memory accesses and prove to be expensive (both in terms of performance and the energy consumed). If lanes are suddenly allowed to diverge, they could disrupt the carefully streamlined memory accesses and incur a large enough penalty. We propose the inclusion of tiny incoherent data caches per lane (TC-D) so as to minimize expensive accesses to the shared DL1G caches as proposed in Chapter 5. We adopt an advanced two-level register file as proposed by Gebhart *et al.* to provide both the performance and the energy efficiency that the design offers [32]. With its own register file, instruction fetch unit, instruction buffers, scoreboard, and local instruction and data cache, the design of the lane is now more like a generic MIMD core, and these generic MIMD cores can be logically grouped into SMs to parallel a traditional GPGPU.

6.3.1 CUDA/OpenCL Compatibility

To be compatible with CUDA and OpenCL, the EESI architecture must support the three fundamental abstractions of the programming model, *i.e.* the thread hierarchy, the memory hierarchy and the synchronization primitives. To execute a CUDA/OpenCL application, we adhere to the same hierarchy model for threads: threads are grouped into *blocks* of threads and executed one warpset after warpset. The execution of the threads on the hardware is invisible to the programmer, and, unlike the traditional GPGPU architectures where divergent threads are executed serially, the EESI architecture uses the individual IF units to allow simultaneous execution of divergent threads. Threads are allowed to diverge until they reach a synchronization primitive like a barrier or a memory fence, at which point, all threads within a block reconverge and halt execution until all the threads within a block reach the barrier.

The tinyCache is configured to cache both scratchpad and global memory references. The programming model allows all threads within a block to access the same scratchpad memory and it is possible that a single cache line with data from the scratchpad memory is present in the tinyCache of multiple cores. However the programming model also does not guarantee the order in which threads will be executed on the lanes, therefore if threads were needed to modify a single cache line, it would only be within the bounds of a barrier or memory fence. We use the same adopted write validate protocol proposed in Chapter 5 to maintain correctness and remove the requirement for coherence between the tinyCaches.

6.3.2 Support for Warping

Warping is one of the key features of traditional GPGPU architectures and is used extensively to hide latencies. Each time a trigger (usually a long latency memory instruction) is encountered, the warp scheduler assigns the next warpset for execution to the SM. All the lanes in the SM execute the same PC from threads within the same warpset and continue until the next warp switch trigger is encountered. To support fast and seamless switching from one warpset to the other, the warps that are allocated to an SM are never really switched out. The register file is large enough (or only as many warps are assigned to an SM) to save the context of all the warps. While warping is not necessary to ensure the correctness of execution, it is critical for the performance that we can extract from a GPGPU as seen in Figure 6.6a. We propose the addition of a simple warp scheduler which is a circular buffer that stores the PCs corresponding to threads in the same warpset for different cores.

To maximize the performance, it is critical to be able to schedule as many threads as possible on each SM, so that a steady flow of instructions are available to hide long latencies. This imposes a severe constraint on the number of registers available per thread, and this manifests as frequent and unavoidable RAW hazards. We propose fully leveraging the warp switching mechanism that is available to us by switching warps at each instruction. We study the implications of this proposal as well as other warp switch triggers like branches, or both memory and branch instructions and present our findings in Section 6.5.2.

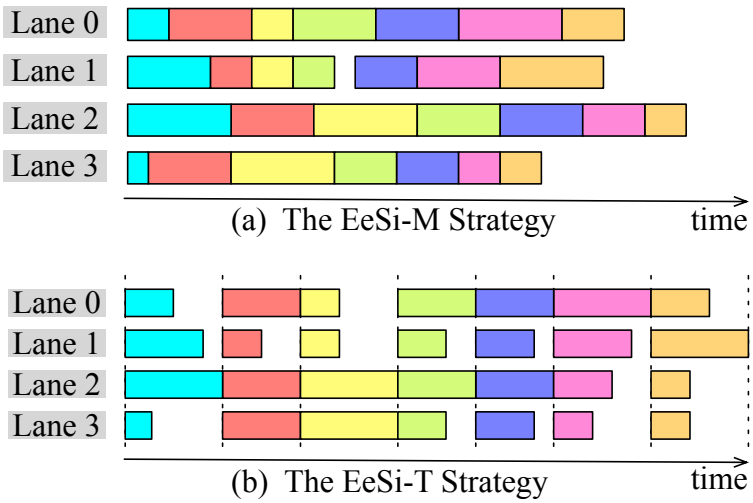


Figure 6.3: We evaluate two policies, the first policy (EESI-T) is more conservative. While different lanes can execute different PCs, they must all belong to threads in the same warpset. The second policy (EESI-M) is more free flowing, where different lanes can execute different PCs as well as different warps, and converge at a barrier / exit. Blocks of the same color, indicate that they belong to the same warp.

6.3.3 Divergence Policies

Since each core is equipped with its own tinyIcache and IF unit, it is indeed possible for all the cores to execute divergent threads at the same time. When all the instructions within a thread have been executed or if the thread encounters a trigger to switch to the next warp, the core can simply switch to the next thread assigned to it. This can continue as long as the threads that are executed belong to the warps that are currently scheduled for execution on the SM. This is analogous to different lanes in an SM in a traditional GPGPU executing different threads from different warpsets at the same time as shown in Figure 6.3a and exemplifies true MIMD behavior, which is not directly possible on a traditional GPGPU. We call this the **EESI-M** strategy.

Traditional GPGPU workloads have highly aligned memory accesses and exhibit a

fair amount of intra-warp locality. A MIMD oriented policy like EESI-M destroys these aligned access patterns and can impose a significant performance penalty. To avoid this, we propose a more conservative policy, **EESI-T**, where threads executing on different cores can execute divergent paths, but will switch to the next warpset in sync with the other cores, grouped together as an SM, as shown in Figure 6.3b. We compare their performance of these two policies in Section 6.5.3.

6.3.4 Executing CPU Applications

If we ignore the tinyCache and the tinyIcache, EESI is like a MIMD system with in-order cores and a large register file to handle multiple thread context. Since the register file has roughly 1K entries in contemporary GPGPUs like Fermi, we assume a RF of the same size which means that each core can behave like it supports 32 thread SMT contexts. This is too much for typical multithreaded applications, so to save energy we could power down all but 32 entries from the RF when running CPU or multithreaded applications.

The tinyIcache is virtually indexed and virtually checked and each time that there is a code modification, all the tinyIcaches need to be invalidated. The tinyIcaches have been sized to be efficient with GPGPU workloads (Section 6.5.4) which means that they are going to have a higher miss rate than a design optimized for MIMD workloads.

The tinyCache presents a more interesting problem. The tinyCaches are incoherent and not using them will create too much traffic to the shared L1 cache. EESI has a release consistency model. As such, we can allow the delay of all the globally performed stores until a memory ordering event like a memory fence is issued. The stores go to the tinyCache, and they

are written back on conflicts when a memory fence retires, and all the entries in the tinyCache are written back and invalidated. Thus, the tinyCache can be viewed as a larger store completion buffer inside the core not visible to the coherence traffic. Reads do not need trigger coherence events to other tinyCaches. Stores are considered globally complete when they are "displaced" from the tinyCache. The shared L1 cache has the coherent data. Since we assume a release consistency multithreaded program, a program with data races that breaks consistency is likely to show unexpected behavior, as we would observe in an aggressive out-of-order processor with a release consistency model.

Putting all the above together, we present the EESI architecture in Figure 6.2.

6.4 Experimental Setup

To evaluate how the EESI architecture fares for typical GPGPU applications, we compare it against a traditional GPGPU architecture supplemented with tinyCaches per lane as proposed in Chapter 5 and as seen in Figure 5.2.

6.4.1 Simulation Infrastructure

To simulate the EESI architecture, we configure the simulator to model simple in-order cores with dedicated IF units that are logically grouped and are viewed as lanes within an SM. The executing GPGPU benchmarks are oblivious to this mapping, and, from their point of view, they are executing on a traditional GPGPU. We disable this logical grouping and treat the in-order cores as cores in a multicore system when we execute CPU benchmarks. For

	Baseline GPGPU	EESI	Baseline MIMD
Core configuration	2 SMs with 32 in-order lanes each	64 inorder lanes/cores	
IF	Common IF and buffers per SM, shared by 32 lanes	IF and IB per lane/core	
Tiny cache Instructions	None	1KB / 8-way / 64B line / 1 cyc / incoherent	None
Branch Predictor	Not Taken Predictor per SM	Not Taken Predictor per Lane	Not Taken Predictor per core
Core Grouping	32 inorder lanes per SM, each SM allows a maximum of 24 concurrent warps and 8 concurrent blocks depending on the resources required per thread.		None
Register File	Shared, Banked RF with 32768 entries		64 Entries per core
Tiny cache Data	1KB/8-way/64B line /1 cyc/incoherent		None
Scratchpad	48KB/8 banks/18 cycles, shared by 32 lanes		None
DL1 cache	32KB/8-way/64B line/ 18 cyc/ incoherent, shared by 32 lanes		16KB/4-way/64B line/4 cyc per core/coherent
IL1 cache	32KB/8-way/64B line/ 4 cyc, shared by 32 lanes		16KB/2-way/64B line/ 4 cyc per core
L2 cache	256KB/16-way/64B line/ 7 cyc, for Data and Inst		
LLC	2MB/32-way/64B line/14 cyc		6MB/32-way/64B line/14 cyc
Main Memory	18GBytes/s BW with 50ns access time		
Core Frequency	1.5Ghz		

Table 6.2: GPSim simulator configuration used to evaluate the EESI architecture

comparisons with more aggressive divergence handling mechanisms in a traditional GPGPU, we extended GPSim to support the immediate post-dominator reconvergence policy [31] and another recent proposal that allows simultaneous execution of multiple divergent paths [18].

Our energy estimations are based on the power model in GPUSimPow [54] and CACTI [61]. Our estimations include only the on-chip dynamic energy; we do not include DRAM. To evaluate our CPU workloads, we use the same core parameters that we used for EESI, but supplement them with a much more substantial instruction and data cache hierarchy.

CPU Workloads	Description
Blackscholes	Calculate prices of a portfolio of options.
Bodytrack	Track the pose of a human with multiple cameras
Ferret	Content based similarity detection algorithm
FFT	FFT algorithm
FluidAnimate	Simulate an incompressible fluid
RADIX	Integer radix sort
Swaptions	Price a portfolio of swaptions
x264	Lossy video encoder

Table 6.3: Workloads used in our evaluation

Table 6.2 summarizes the architectural parameters we have used in our simulations.

6.4.2 Benchmarks

Apart from GPGPU workloads used to evaluate our prior work (Table 5.2), we also chose benchmarks from both the SPLASH [88] and the Parsec [12] benchmark suite for our CPU workload evaluations, all of which used the largest input set available, and ran until completion.

6.5 Evaluation

In this section we quantify the performance of EESI against more traditional GPGPUs (*i.e.*, more SIMD-like) and conventional CPU (*i.e.*, more MIMD-like) architectures. First, we present the performance of GPGPU and CPU workloads on the EESI architecture. Next, we evaluate the impact of different warp switch triggers on the performance and then discuss the implication of breaking away from the more conservative EESI warping mechanism (EESI-T) towards a more free flowing approach (EESI-M). Finally, we justify the need for a tiny instruction cache and offer some insights on sizing it as per our need.

Alias	Description
GPU-SE	Serial divergence handling mechanism, without immediate post-dominator re-convergence
GPU-PD	Serial divergence, with immediate post-dominator re-convergence [31]
GPU-SBI	Simultaneous Branch interleaving as proposed in [18]
EESI	Allow concurrent execution of multiple divergence paths, conservative, as shown in Figure 6.3b

Table 6.4: Divergence handling mechanisms

6.5.1 Main Results

Figure 6.4 summarizes the performance of our GPGPU and CPU benchmarks on the EESI architecture. Here EESI refers to the more conservative EESI-T strategy, which switches warps at every instruction. For GPGPU benchmarks, we compare against our baseline GPGPU equipped with three divergence handling mechanisms listed in Table 6.4. The implications of choosing the right warp switch trigger is discussed in Section 6.5.2. To keep the comparison fair, the baseline GPGPU also switches warps at each and every instruction.

A traditional GPGPU has a common instruction cache and could get away with supporting fewer maximum requests to the cache. However since we propose switching warps at every instruction, we raise the limit to sustain a large number of maximum requests for the baseline without which we see a 7% reduction in the execution time for the baseline.

GPGPU workloads cannot be executed on a traditional CPU, which is why there is no bar corresponding to the MIMD execution for these. The performance of our CPU benchmarks is compared against a more traditional CPU architecture described in Table 6.2. Since CPU workloads cannot be executed on traditional GPGPUs, so there are no bars corresponding to GPU-SE, GPU-PD and GPU-SBI for these benchmarks.

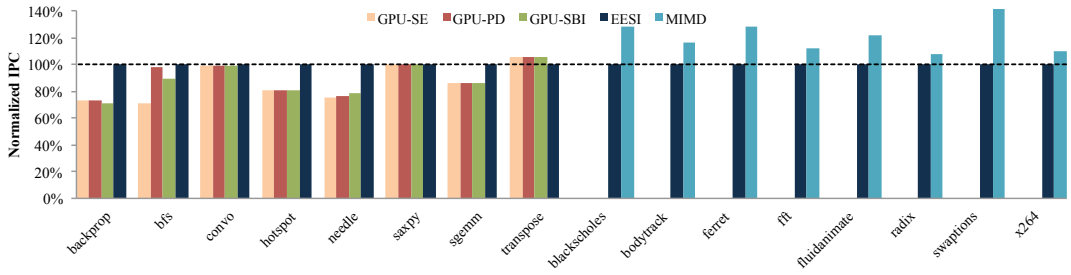


Figure 6.4: The EESI architecture supports the execution of traditional GPGPU workloads, with an average speedup of 13%, as wells as traditional MIMD (CPU) workloads, with a 20% hit in performance on average.

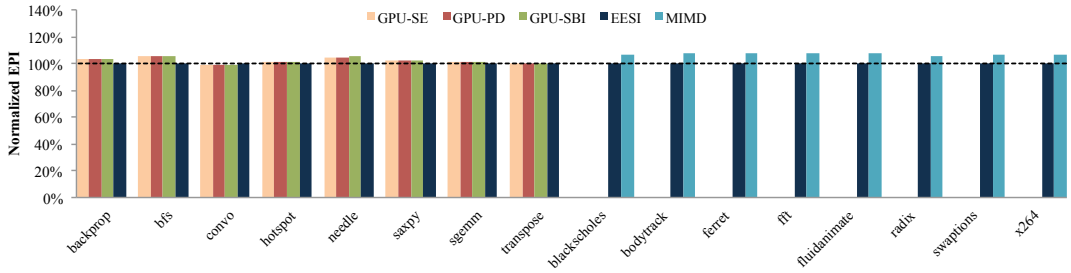


Figure 6.5: There is no significant change in the EPI for GPGPU workloads, but it shows a slight reduction (<10%) for traditional MIMD workloads.

To maximize performance on traditional GPGPU architectures, most GPGPU workloads try to keep the behavior as regular as possible, along with highly synchronized and aligned memory accesses. They are designed to maximize the occupancy on the GPGPU, by having as many threads in-flight as possible. The workloads we examine show these characteristics to varying degrees.

Backprop, needle, hotspot and SGEMM exhibit enough divergence in the application for EESI to exploit, and show almost a 25% improvement in the performance, on average. Even though BFS is inherently divergent, almost all the execution is done by one or two threads in a warp, and it shows very poor occupancy which is why we don't see as big a performance

improvement with EESI. Convo and SAXPY are extremely regular applications with no divergence and show no difference in execution time. All these applications show very regular memory access pattern, and prefer the more conservative EESI strategy to fully exploit intra-warp locality of memory references. This is discussed in more detail in Section 6.5.3.

Transpose is a memory intensive application that is susceptible to memory divergence and ideally prefers no warping or warping on branches (which are few and far between). Warping at every instruction seems to skew the memory access pattern, so EESI performs slightly worse than the traditional GPGPU architecture.

The presence of the tiny data and instruction caches results in a higher miss rate and average memory latency per instruction which is uniformly the reason for the deteriorated performance for the CPU benchmarks.

Figure 6.5 shows the energy per instruction (EPI). The larger caches are also the reason that the average EPI for a traditional MIMD is higher than EESI. The difference between the EPI for EESI and traditional GPGPU architectures is negligible.

6.5.2 Role of Warping Mechanisms

Warping is one of the key techniques used to hide latencies in GPGPUs and traditionally the trigger for switching to the next warp is a long latency memory operation. In this section, we discuss the implication of choosing different triggers for switching to the next warp.

We evaluated different configurations where we used the warp switch triggers listed in Table 6.5 to switch to the next warp. Since most of our applications are fairly regular and have a fair number of memory operations, most applications seemed to prefer switching at every

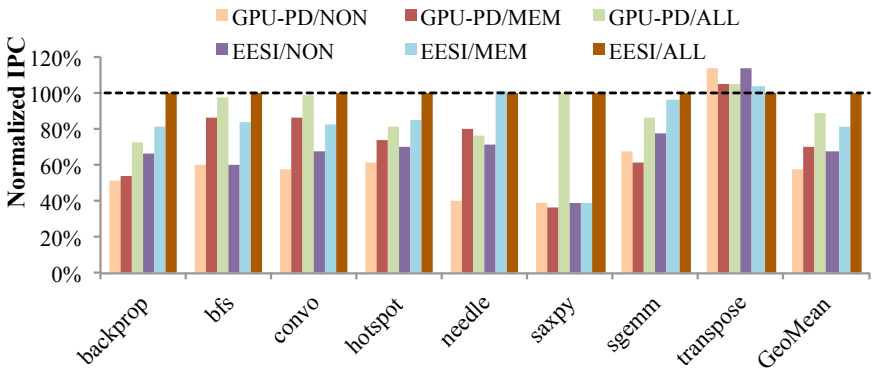
Alias	Description
NON	Warp only at a barrier or when the thread exits.
MEM	Warp only on long latency memory instructions.
BRA	Warp only when you encounter a branch.
MBR	Warp at both memory and branch instructions.
ALL	Warp after each and every instruction.

Table 6.5: Triggers for switching to the next warp.

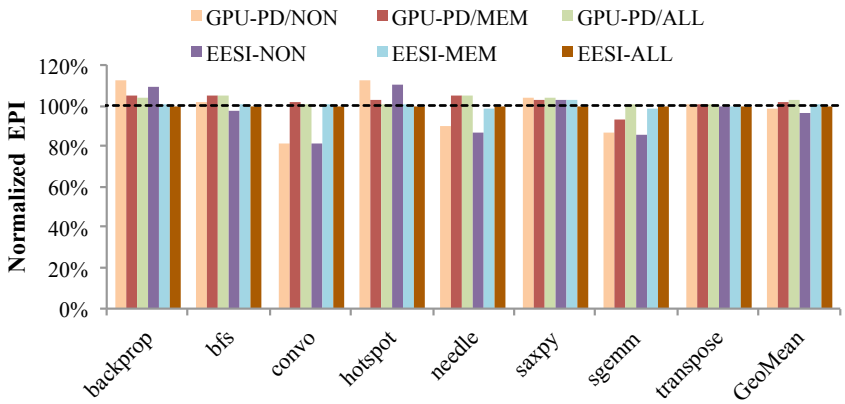
memory instruction (MEM) or after every instruction (ALL). We compare the performance of our baseline GPGPU and the EESI architecture and present the results ins Figure 6.6a. Other applications show negligible change in the average EPI, irrespective of the warp switching policy.

As expected, we see a sizeable loss (25%) in performance on average when we compare both the baseline and EESI without warping and with warping at every memory instruction ((GPU-PD/NON vs GPU-PD/MEM) and (EESI/NON vs EESI/MEM)). Applications like BFS, convo, hotspot and needle show the maximum sensitivity to switching at memory instructions in the baseline architecture. However, letting lanes diverge on the EESI architecture also increases the sensitivity of backprop to memory instructions. Benchmarks like SAXPY which show no reuse patterns and do not have enough threads to saturate the GPU are indifferent to the warp switch. Transpose shows a slight deterioration due to its sensitivity to the memory access pattern.

Switching warps at every instruction allows us to hide not only the memory latencies when they show up, but also RAW hazards that are very common given the limited resources available per lane per thread. Both the baseline (GPU-PD/NON vs GPU-PD/ALL) and EESI (EESI/NON vs EESI/ALL) show roughly a 50% improvement in performance, and even



(a) Normalized Execution Time



(b) Normalized EPI

Figure 6.6: Switching warps at each instruction offers a sizeable improvement in performance even for the baseline GPGPU architecture. The EESI architecture switching warps at each instruction, provides >30% increase (on average) over a traditional GPGPU architecture (GPU-PD) switching warps at a memory instruction (MEM). A few individual applications are indeed more energy efficient with conservative (or no) warping policies, but on the whole switching between warping policies alone does not offer a reduction in the EPI

a streaming application like SAXPY can double its performance.

EESI, combined with switching at every instruction (EESI/ALL), provides a 30% improvement in performance over a traditional GPGPU warping at every memory instruction (GPU-PD/MEM).

Figure 6.6 shows the impact of the above triggers on the EPI. Applications like convo, needle and SGEMM, which make heavy use of scratchpad memory, cache these accesses in the tiny data cache. When we disable warping, the tiny data cache is large enough to service scratchpad memory requests instead of fetching them from the more energy hungry scratchpad memory and this greatly improves the energy efficiency. We lose this advantage if we have a large number of warps and threads in flight.

It is important to note that disabling warping opens up the possibility for us to dramatically size down energy hungry resources like the register file or make a larger number of registers available to each thread, thereby increasing the performance. It thus might be possible to tune the EESI architecture to be more energy efficient without much loss in performance and eliminate the complexity of warping. We leave this exploration as future work.

6.5.3 EESI-T vs EESI-M

As discussed in Section 6.3, a truly divergent (*i.e.*, general purpose) application where every thread in every lane follows a different path could benefit even more if we allowed threads to break warp boundaries and execute in a more free-flowing fashion. However, the benchmarks we surveyed are optimized for traditional GPGPU architectures, and this would take a huge hit if we allow this and skew the carefully aligned memory accesses. Our experiments confirm the same, and the GPGPU benchmarks we study show almost a 30% reduction in the execution time if we opt for the more free flowing execution of EESI-M. This makes it a much less attractive option for traditional GPGPU workloads.

We are not aware of any existing GPGPU benchmarks that can exploit the true MIMD

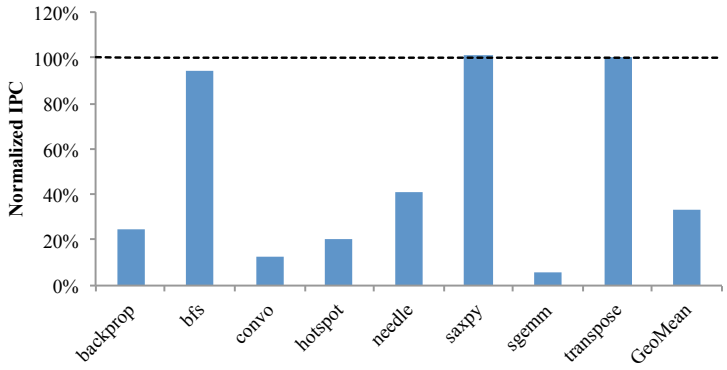


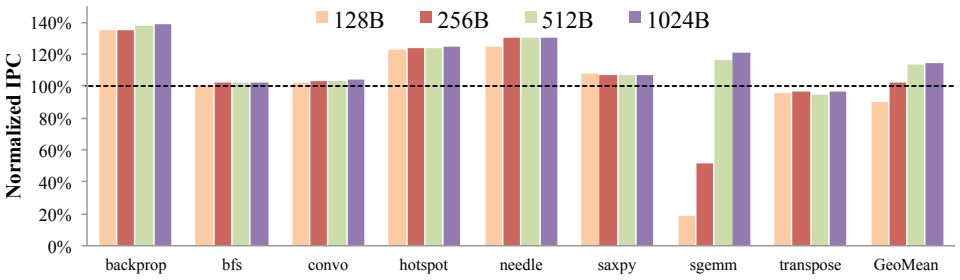
Figure 6.7: The absence of a TCI significantly affects the performance, causing almost a 60% drop in the performance.

behavior that the EESI-M can sustain. We believe that supporting EESI-M opens up avenues to port more general purpose applications like request response based server side applications, etc.

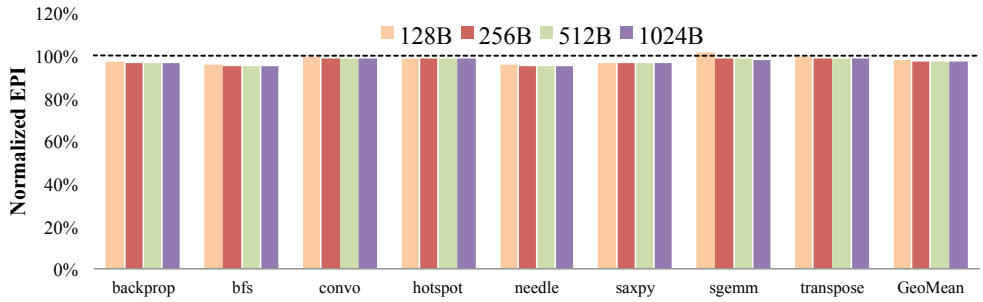
6.5.4 Sizing the TinyICache

The addition of a tiny instruction cache per lane is crucial to the increased performance and reduced EPI that the EESI architecture offers. Figure 6.7 shows that the shared instruction cache can become the bottle neck even for relatively small GPGPU kernels that warp at every instruction. The absence of the tiny instruction cache can reduce the execution time by more than 60% on average and increase the EPI by more than 20%. Both the plots are normalized to GPU-PD/ALL.

None of the GPGPU applications other than SGEMM, strongly prefer a larger tinyICache. CPU workloads show a clear preference for larger tinyICaches, and we choose a tinyICache that has 8 entries with 64B each. Changing the line size had no impact for either GPGPU



(a) Normalized Execution Time



(b) Normalized EPI

Figure 6.8: Varying the size of the tinyIcache impacted the performance of SGEMM, however it did not significantly impact the average EPI. We picked a tinyICache with 8 entries with 64B in each entry.

or CPU workloads.

6.5.5 Area Overhead Estimations

We try to estimate the area overhead of EESI by comparing the decode area with that of the register file.

First, we synthesized an Imagination MIPS M14K in-order core that has 4 pipeline stages, and that can execute the full MIPS R5 ISA. The Master Pipeline Control (MPC) block that includes all the decoding and control for the MIPS core has 8500 gates. To put it in per-

spective, this is equivalent in size to 5 times the MIPS register file with 32 entries. Thus, for a register file with 1024 entries per core, as in EESI, the area of the decode and control block, can be approximated to be about 15.6% of the RF area.

We also studied the OpenSPARC core which is an in-order core with 4 SMT contexts. The decode has approximately the same SPARC register file which with support for sliding windows is roughly equivalent to 288 register entries. The decode for the OpenSPARC also includes the instruction TLBs and the icount policies for the SMT. Since the EESI tinyICache is virtually indexed, the TLB does not need to be replicated. After excluding the TLB, the decode is almost 50% of the 288 entry register file, approximately the same area as that of a 144 entry register file. Thus, approximating in the same way as we did above, this tells us that the decode logic should occupy roughly 15% of the area of the register file.

Both of our estimations come from different inorder cores, but point to a similar area overhead of roughly 14% to 15% of area of the per core the register file. According to Khailany *et al.*, a 15% area for the register file translates to roughly 3% area overhead for the Imagine processor [44]. This estimation is consistent with the 2% area overhead estimation [31] by Fung *et al.*

We use CACTI [61] to estimate that the tinyIcache is roughly 7% the area of the RF. We add this to the overhead of the decode, and, based on the above, we conclude that the area overhead is between 4% and 5% per core for the EESI architecture. We believe that this is a conservative estimation of the overhead, since we do not have the need for complicated control and decode needed by some of the other proposals to support multipath execution [18, 71, 83] and attribute no area overhead to them.

6.6 Summary

In this chapter we present EESI, a novel MIMD-like architecture, designed with the following goals in mind.

- To support multiple divergent paths with a simple design.
- To offer support for both traditional GPGPU and CPU workloads.
- To sustain the energy efficiency of a more SIMD-like architecture for traditional GPGPU workloads.

By adding a tiny instruction cache and IF per lane we allow the lanes to diverge and efficiently execute divergent applications. This morphs an SM into a MIMD-like cluster of simple inorder cores. EESI leverages tiny incoherent data caches per lane to reduce overheads and coordinates switching between warps in different lanes (EESI-T) to take advantage of the coalesced memory accesses available in GPGPU programs. Our evaluations show a 13% performance benefit available by warping after each instruction for traditional GPGPU workloads and with no impact on the average EPI when compared to a traditional GPGPU. The downsized cores and the GPGPU optimized memory hierarchy also allow execution of traditional CPU workloads, albeit with a 20% hit in performance. Replicating the IF and adding a tiny instruction cache per lane does add an area overhead of roughly 5%, but we feel that this is a justifiable, especially if it means that we can execute more general purpose applications on the GPGPU.

Chapter 7

Conclusions

The early GPGPU arrived on the scene with the promise of being the perfect complement to the CPU to tackle more than just the embarrassingly data parallel category of applications that were usually assigned to SIMD machines. The newly designed programming models of the time allowed a larger set of regular applications that were executed sub-optimally on the CPU to harness the throughput of the GPGPU and promised performance gains ranging anywhere between 10X and 1000X. This led to a very sudden spike in the popularity of the GPGPU, and it soon became one of the hottest topics of research in the architecture community.

GPSim is a platform to simulate the behavior and estimate the performance of both traditional and futuristic GPGPUs. It was one of the first simulators to allow the simulation of a truly heterogeneous architecture with both CPUs and GPUs. It is the only GPGPU simulator that uses native coexecution of GPGPU applications and provides a significant boost in the simulation speeds. GPSim was used to study the GPGPU architecture and make the following new proposals.

With tighter coupling between CPUs and GPGPUs, transferring data between the two addressing spaces became the bottleneck, both in terms of performance and in terms of energy efficiency. FuseTLB is a simple energy efficient design that exploits the unique programming model for GPGPUs to unify the addressing spaces and create a virtual shared memory between CPUs and GPUs on the same die, without coherence.

An immersive study of potential energy optimizations within a GPGPU led to the proposal of the addition of tiny, incoherent, per-lane data caches to the GPGPU. This addition lets us filter out expensive accesses to the shared data structures and offers a substantial saving of roughly 37% of the energy consumed by the on-chip cache hierarchy on average.

In spite of the GPGPU coming a long way both in terms of capability and capacity, their applicability and scope is constrained by the limited support for divergent applications. EESI is a novel MIMD-like design to better enable divergence and allow the execution of both GPGPU (barely divergent) and traditional CPU (fully divergent) workloads. A key feature of the EESI architecture is the simplicity of design. It proposes the removal of all specialized hardware and logic that are added to the GPGPUs to support divergence, and instead adds tiny instruction and data caches per lane, decoupling the instruction fetch and breaking the requirement of lock step execution of all the lanes. Despite the replication of the instruction fetch and the tiny instruction and data caches, the design is just as energy efficient as a regular GPGPU is for SIMD workloads. This is crucial to help the GPGPU step outside the high performance computing box, and find acceptance in both mainstream and mobile applications.

Bibliography

[1] CUDA C Best Practices Guide. http://docs.nvidia.com/cuda/cuda-c-best-practices-guide/.

[2] CUDA SDK.

[3] CUDA zone. https://developer.nvidia.com/cuda-zone.

[4] GPGPU.org. http://www.gpgpu.org.

[5] Intel 2600K sandy bridge processor. http://ark.intel.com/products/52214/Intel-Core-i7-2600K-Processor-(8M-Cache-up-to-3_80-GHz).

[6] NVIDIA CUDA C Programming Manual. https://docs.nvidia.com/cuda/cuda-c-programming-guide/.

[7] Pollack's rule. https://en.wikipedia.org/wiki/Pollack%27s_Rule.

[8] Architecture instruction set extensions programming reference. *Intel Corporation, Feb*, 2012.

[9] Ehsan K Ardestani and Jose Renau. ESESC: A Fast Multicore Simulator Using Time-Based Sampling. In *International Symposium on High Performance Computer Architecture*, HPCA'19, 2013.

[10] Ali Bakhoda, George L. Yuan, Wilson W. L. Fung, Henry Wong, and Tor M. Aamodt. Analyzing CUDA Workloads Using a Detailed GPU Simulator. In *IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS 2009)*, pages 163–174, April 2009.

[11] Fabrice Bellard. QEMU, a fast and portable dynamic translator. In *Proceedings of the annual conference on USENIX Annual Technical Conference*. USENIX, 2005.

[12] Christian Bienia, Sanjeev Kumar, Jaswinder Pal Singh, and Kai Li. The PARSEC benchmark suite: characterization and architectural implications. In *PACT '08: Proceedings of the 17th international conference on Parallel architectures and compilation techniques*, pages 72–81, New York, NY, USA, 2008. ACM.

[13] Darrell Boggs, Gary Brown, Nathan Tuck, and KS Venkatraman. Denver: NVIDIA's first 64-bit ARM processor. *IEEE Micro*, (2):46–55, 2015.

[14] Shekhar Borkar and Andrew A. Chien. The future of microprocessors. *Commun. ACM*, 54(5):67–77, May. 2011.

[15] Dan Bouvier and Ben Sander. Applying AMDs Kaveri APU for heterogeneous computing. In *Hot Chips: A Symposium on High Performance Chips (HC26)*, 2014.

[16] Chas Boyd, Xin Huang, Cody Pritchard, et al. DirectX 11 DirectCompute: A teraflop for everyone. In *Microsoft Game Technology Conference. London, UK:[sn]*, 2010.

[17] Alexander Branover, Denis Foley, and Maurice Steinman. AMD Fusion APU: Llano. *IEEE Micro*, (2):28–37, 2012.

[18] Nicolas Brunie, Sylvain Collange, and Gregory Diamos. Simultaneous branch and warp interweaving for sustained GPU performance. In *Computer Architecture (ISCA), 2012 39th Annual International Symposium on*, pages 49–60, June 2012.

[19] Ian Buck, Tim Foley, Daniel Horn, Jeremy Sugerman, Kayvon Fatahalian, Mike Houston, and Pat Hanrahan. Brook for GPUs: Stream Computing on Graphics Hardware. In *ACM SIGGRAPH 2004 Papers*, SIGGRAPH '04, pages 777–786, New York, NY, USA, 2004. ACM.

[20] Snaider Carrillo, Jakob Siegel, and Xiaoming Li. A control-structure splitting optimization for gpgpu. In *Proceedings of the 6th ACM Conference on Computing Frontiers*, CF '09, pages 147–150, New York, NY, USA, 2009. ACM.

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

[22] Shuai Che, Jeremy W Sheaffer, and Kevin Skadron. Dymaxion: optimizing memory access patterns for heterogeneous systems. In *Proceedings of 2011 International Confer-*

ence for High Performance Computing, Networking, Storage and Analysis, SC '11, pages 13:1–13:11, New York, NY, USA, 2011. ACM.

[23] Gautham N. Chinya, Jamison D. Collins, Perry H. Wang, Hong Jiang, Guei-Yuan Lueh, Thomas A. Piazza, and Hong Wang. Bothnia: a dual-personality extension to the Intel integrated graphics driver. *SIGOPS Oper. Syst. Rev.*, 45(1):11–20, Feb. 2011.

[24] Intel Corportation. Intel Core 2 Extreme Processor X6800 and Intel Core 2 Duo Desktop Processor E6000 and E4000, Specification Update Doc. No.313279-024, Feb 2008.

[25] Robert H Dennard, VL Rideout, E Bassous, and AR LeBlanc. Design of ion-implanted MOSFET's with very small physical dimensions. *Solid-State Circuits, IEEE Journal of*, 9(5):256–268, 1974.

[26] Gregory Diamos, Benjamin Ashbaugh, Subramaniam Maiyuran, Andrew Kerr, Haicheng Wu, and Sudhakar Yalamanchili. SIMD re-convergence at thread frontiers. In *Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-44, pages 477–488, New York, NY, USA, 2011. ACM.

[27] Ahmed ElTantawy, Jessica Wenjie Ma, Mike O'Connor, and Tor M Aamodt. A scalable multi-path microarchitecture for efficient GPU control flow. In *High Performance Computer Architecture (HPCA), 2014 IEEE 20th International Symposium on*, pages 248–259, Feb 2014.

[28] Naila Farooqui, Andrew Kerr, Gregory Diamos, S. Yalamanchili, and K. Schwan. A framework for dynamically instrumenting GPU compute applications within GPU ocelot.

In *Proceedings of the Fourth Workshop on General Purpose Processing on Graphics Processing Units*, GPGPU-4, pages 9:1–9:9, New York, NY, USA, 2011. ACM.

[29] Nadeem Firasta, Mark Buxton, Paula Jinbo, Kaveh Nasri, and Shihjong Kuo. Intel AVX: New frontiers in performance improvements and energy efficiency. *Intel white paper*, 2008.

[30] Wilson WL Fung and Tor M Aamodt. Thread block compaction for efficient SIMT control flow. In *High Performance Computer Architecture (HPCA), 2011 IEEE 17th International Symposium on*, pages 25–36, Feb 2011.

[31] Wilson WL Fung, Ivan Sham, George Yuan, and Tor M Aamodt. Dynamic Warp Formation and Scheduling for Efficient GPU Control Flow. In *IEEE/ACM International Symposium on Microarchitecture (MICRO-40)*, pages 407–420, 2007.

[32] Mark Gebhart, Daniel R. Johnson, David Tarjan, Stephen W. Keckler, William J. Dally, Erik Lindholm, and Kevin Skadron. Energy-efficient mechanisms for managing thread context in throughput processors. In *Proceedings of the 38th annual international symposium on Computer architecture*, ISCA '11, pages 235–246, New York, NY, USA, 2011. ACM.

[33] Mark Gebhart, Stephen W Keckler, and William J Dally. A compile-time managed multi-level register file hierarchy. In *Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-44 '11, pages 465–476, New York, NY, USA, 2011. ACM.

[34] Mark Gebhart, Stephen W Keckler, Brucek Khailany, Ronny Krashinsky, and William J Dally. Unifying primary cache, scratch, and register file memories in a throughput processor. In *Proceedings of the 45th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-45 '12, 2012.

[35] Chunyang Gou and Georgi Gaydadjiev. Exploiting SPMD horizontal locality. *IEEE Computer Architecture Letters*, 10(1):20–23, 2011.

[36] Chris Gregg and Kim Hazelwood. Where is the data? Why you cannot debate GPU vs. CPU performance without the answer. In *International Symposium on Performance Analysis of Systems and Software*, ISPASS, Austin, TX, April 2011.

[37] Frank T Hady, Mason B Cabot, John Beck, and Mark B Rosenbluth. Heterogeneous processors sharing a common cache, 06 2009.

[38] Mark Hampton and Krste Asanović. Implementing virtual memory in a vector processor with software restart markers. In *In ICS*, pages 135–144, 2006.

[39] Tianyi David Han and Tarek S. Abdelrahman. Reducing branch divergence in GPU programs. In *Proceedings of the Fourth Workshop on General Purpose Processing on Graphics Processing Units*, GPGPU-4, pages 3:1–3:8, New York, NY, USA, 2011. ACM.

[40] Sunpyo Hong and Hyesoon Kim. An analytical model for a GPU architecture with memory-level and thread-level parallelism awareness. In *Proceedings of the 36th annual international symposium on Computer architecture*, ISCA '09, pages 152–163, New York, NY, USA, 2009. ACM.

[41] Charles R Johns and Daniel A Brokenshire. Introduction to the cell broadband engine architecture. *IBM Journal of Research and Development*, 51(5):503–519, 2007.

[42] Tim Johnson and Umesh Nawathe. An 8-core, 64-thread, 64-bit power efficient SPARC SoC (Niagara2). In *Proceedings of the 2007 international symposium on Physical design*, pages 2–2. ACM, 2007.

[43] Norman P. Jouppi. Cache write policies and performance. In *Proceedings of the 20th annual international symposium on computer architecture*, ISCA '93, pages 191–201, New York, NY, USA, 1993. ACM.

[44] Brucek Khailany, William J Dally, Andrew Chang, Ujval J Kapasi, Jinyung Namkoong, and Brian Towles. VLSI design and verification of the Imagine processor. In *Computer Design: VLSI in Computers and Processors, 2002. Proceedings. 2002 IEEE International Conference on*, pages 289–294. IEEE, 2002.

[45] Hyesoon Kim, Jaekyu Lee, Nagesh B Lakshminarayana, Jaewoong Sim, Jieun Lim, and Tri Pho. Macsim: A CPU-GPU heterogeneous simulation framework user guide. *Georgia Institute of Technology*, 2012.

[46] Johnson Kin, Munish Gupta, and William H. Mangione-Smith. The filter cache: an energy efficient memory structure. In *Proceedings of the 30th annual ACM/IEEE international symposium on Microarchitecture*, MICRO 30, pages 184–193, Washington, DC, USA, 1997. IEEE Computer Society.

[47] Ronny Krashinsky, Christopher Batten, Mark Hampton, Steve Gerding, Brian Pharris,

Jared Casper, and Krste Asanovic. The vector-thread architecture. *IEEE Micro*, 24(6):84–90, 2004.

[48] Nagesh B Lakshminarayana and Hyesoon Kim. Effect of Instruction Fetch and Memory Scheduling on GPU Performance. *Workshop on Language, Compiler, and Architecture Support for GPGPU, in conjunction with HPCA/PPoPP 2010*, 2010.

[49] Chris Lattner and Vikram Adve. LLVM: A compilation framework for lifelong program analysis & transformation. In *Code Generation and Optimization, 2004. CGO 2004. International Symposium on*, pages 75–86. IEEE, 2004.

[50] Jaekyu Lee and Hyesoon Kim. TAP: A TLP-aware cache management policy for a CPU-GPU heterogeneous architecture. In *High Performance Computer Architecture (HPCA), 2012 IEEE 18th International Symposium on*, pages 1 –12, Feb. 2012.

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

[52] Victor W. Lee, Changkyu Kim, Jatin Chhugani, Michael Deisher, Daehyun Kim, Anthony D. Nguyen, Nadathur Satish, Mikhail Smelyanskiy, Srinivas Chennupaty, Per Hammarlund, Ronak Singhal, and Pradeep Dubey. Debunking the 100x GPU vs. CPU myth: an evaluation of throughput computing on CPU and GPU. *SIGARCH Comput. Archit. News*, 38(3):451–460, Jun. 2010.

[53] Jingwen Leng, Tayler Hetherington, Ahmed ElTantawy, Syed Gilani, Nam Sung Kim, Tor M Aamodt, and Vijay Janapa Reddi. GPUWattch: enabling energy optimizations in GPGPUs. *ACM SIGARCH Computer Architecture News*, 41(3):487–498, 2013.

[54] Jerome Lucas, Sunil Lal, Michael Andersch, Mauricio Alvarez-Mesa, and Ben Juurlink. How a Single Chip Causes Massive Power Bills GPUSimPow: A GPGPU Power Simulator. In *IEEE International Symposium on Performance Analysis of Systems and Software*, 2013.

[55] Kun Luo, Jayanth Gummaraju, and Manoj Franklin. Balancing Throughput and Fairness in SMT Processors. In *International Symposium on Performance Analysis of Systems and Software*, pages 164–171, Tucson, Arizona, Nov. 2001.

[56] Jiayuan Meng, David Tarjan, and Kevin Skadron. Dynamic warp subdivision for integrated branch and memory divergence tolerance. In *Proceedings of the 37th annual international symposium on Computer architecture*, ISCA '10, pages 235–246, New York, NY, USA, 2010. ACM.

[57] Jaikrishnan Menon, Marc De Kruijf, and Karthikeyan Sankaralingam. iGPU: Exception Support and Speculative Execution on GPUs. In *Proceedings of 39th International Symposium on Computer Architecture "'(ISCA)"'*, 2012.

[58] Gordon E Moore et al. Progress in digital integrated electronics. In *Electron Devices Meeting*, volume 21, pages 11–13, 1975.

[59] Steven S. Muchnick. *Advanced Compiler Design and Implementation*. Morgan Kaufman Publishers, 1997.

[60] Aaftab Munshi et al. The OpenCL specification. *Khronos OpenCL Working Group*, 1:l1–15, 2009.

[61] Naveen Muralimanohar, Rajeev Balasubramonian, and Norman P Jouppi. CACTI 6.0: A Tool to Model Large Caches. Technical Report HPL-2009-85, April 2009.

[62] Veynu Narasiman, Michael Shebanow, Chang Joo Lee, Rustam Miftakhutdinov, Onur Mutlu, and Yale N Patt. Improving GPU performance via large warps and two-level warp scheduling. In *Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-44 '11, pages 308–317, New York, NY, USA, 2011. ACM.

[63] Hubert Nguyen. *GPU Gems 3*. Addison-Wesley Professional, first edition, 2007.

[64] Young Jin Park, Hong Jun Choi, Cheol Hong Kim, and Jong-Myon Kim. Energy-aware filter cache architecture for multicore processors. In *Electronic Design, Test and Application, 2010. DELTA '10. Fifth IEEE International Symposium on*, pages 58 –62, Jan. 2010.

[65] Alex Peleg and Uri Weiser. MMX technology extension to the Intel architecture. *Micro, IEEE*, 16(4):42–50, 1996.

[66] Moinuddin K. Qureshi and Yale N. Patt. Utility-based cache partitioning: A low-overhead, high-performance, runtime mechanism to partition shared caches. In *Proceedings of*

the 39th Annual IEEE/ACM International Symposium on Microarchitecture, MICRO 39, pages 423–432, Washington, DC, USA, 2006. IEEE Computer Society.

[67] Minsoo Rhu and Mattan Erez. CAPRI: Prediction of compaction-adequacy for handling control-divergence in GPGPU architectures. In *Proceedings of the 39th Annual International Symposium on Computer Architecture*, ISCA '12, pages 61–71, Washington, DC, USA, 2012. IEEE Computer Society.

[68] Minsoo Rhu and Mattan Erez. The dual-path execution model for efficient GPU control flow. In *High Performance Computer Architecture (HPCA2013), 2013 IEEE 19th International Symposium on*, pages 591–602, Feb 2013.

[69] Minsoo Rhu, Michael Sullivan, Jingwen Leng, and Mattan Erez. A locality-aware memory hierarchy for energy-efficient GPU architectures. In *Proceedings of the 46th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-46, pages 86–98, New York, NY, USA, 2013. ACM.

[70] Phil Rogers and AC Fellow. Heterogeneous system architecture overview. In *Hot Chips*, volume 25, 2013.

[71] Timothy G Rogers, Daniel R Johnson, Mike O'Connor, and Stephen W Keckler. A variable warp size architecture. In *Computer Architecture (ISCA), 2015 ACM/IEEE 42nd Annual International Symposium on*, pages 489–501, June 2015.

[72] Timothy G. Rogers, Mike O'Connor, and Tor M. Aamodt. Cache-conscious wavefront

scheduling. In *Proceedings of the 45th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-45 '12, New York, NY, USA, 2012. ACM.

[73] John Sartori and Ravindra Kumar. Branch and data herding: Reducing control and memory divergence for error-tolerant GPU applications. *Multimedia, IEEE Transactions on*, 15(2):279–290, 2013.

[74] Harsh Sharangpani and Ken Arora. Itanium Processor Microarchitecture. *IEEE Micro*, 20(5):24–43, Sept. 2000.

[75] Mark Silberstein, Bryan Ford, and Emmett Witchel. GPUfs: The case for operating system services on GPUs. *Commun. ACM*, 57(12):68–79, Nov. 2014.

[76] Allan Snavely and Dean M Tullsen. Symbiotic job scheduling for a simultaneous multithreading processor. In *International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 234–244, Cambridge, Massachusetts, Nov. 2000.

[77] Kyle L. 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 9th conference on Computing Frontiers*, CF '12, pages 103–112, New York, NY, USA, 2012. ACM.

[78] John A Stratton, Christopher Rodrigues, I-Jui Sung, Nady Obeid, Li-Wen Chang, Nasser Anssari, Geng Daniel Liu, and Wen-mei W Hwu. Parboil: A revised benchmark suite

for scientific and commercial throughput computing. *Center for Reliable and High-Performance Computing*, 2012.

[79] John A Stratton, Sam S Stone, and W Hwu Wen-mei. MCUDA: An efficient implementation of CUDA kernels for multi-core CPUs. volume 5335 of *LNCS*, pages 16–30. Springer-Verlag, Berlin, Heidelberg, 2008.

[80] David Tarjan and Kevin Skadron. The sharing tracker: Using ideas from cache coherence hardware to reduce off-chip memory traffic with non-coherent caches. In *Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '10, pages 1–10, Washington, DC, USA, 2010. IEEE Computer Society.

[81] Rafael Ubal, Byunghyun Jang, Perhaad Mistry, Dana Schaa, and David Kaeli. Multi2sim: a simulation framework for CPU-GPU computing. In *Proceedings of the 21st international conference on Parallel architectures and compilation techniques*, PACT '12, pages 335–344, New York, NY, USA, 2012. ACM.

[82] Perry H. Wang, Jamison D. Collins, Gautham N. Chinya, Hong Jiang, Xinmin Tian, Milind Girkar, Nick Y. Yang, Guei-Yuan Lueh, and Hong Wang. EXOCHI: architecture and programming environment for a heterogeneous multi-core multithreaded system. In *Proceedings of the 2007 ACM SIGPLAN conference on Programming language design and implementation*, PLDI '07, pages 156–166, New York, NY, USA, 2007. ACM.

[83] Yaohua Wang, Shuming Chen, Jianghua Wan, Jiayuan Meng, Kai Zhang, Wei Liu,

and Xi Ning. A multiple SIMD, multiple data (msmd) architecture: Parallel execution of dynamic and static SIMD fragments. In *High Performance Computer Architecture (HPCA2013), 2013 IEEE 19th International Symposium on*, pages 603–614, Feb 2013.

[84] Steven JE Wilton and Norman P Jouppi. CACTI: An Enhanced Cache Access and Cycle Time Model. *IEEE Journal of Solid-State Circuits*, 31(5):677–688, May 1996.

[85] Craig M Wittenbrink, Emmett Kilgariff, and Arjun Prabhu. Fermi GF100 GPU architecture. *IEEE Micro*, (2):50–59, 2011.

[86] Henry Wong, Anne Bracy, Ethan Schuchman, Tor M. Aamodt, Jamison D. Collins, Perry H. Wang, Gautham Chinya, Ankur Khandelwal Groen, Hong Jiang, and Hong Wang. Pangaea: a tightly-coupled IA32 heterogeneous chip multiprocessor. In *Proceedings of the 17th international conference on Parallel architectures and compilation techniques*, PACT '08, pages 52–61, New York, NY, USA, 2008. ACM.

[87] Dong Hyuk Woo and Hsien-Hsin S. Lee. Compass: a programmable data prefetcher using idle GPU shaders. In *Proceedings of the fifteenth edition of ASPLOS on Architectural support for programming languages and operating systems*, ASPLOS '10, pages 297–310, New York, NY, USA, 2010. ACM.

[88] Steven Cameron Woo, Moriyoshi Ohara, Evan Torrie, Jaswinder Pal Singh, and Anoop Gupta. The SPLASH-2 Programs: Characterization and MethodologicalConsiderations. In *International Symposium on Computer Architecture*, pages 24–36, Santa Margherita Ligure, Italy, Jun. 1995.

[89] Sven Woop, Jörg Schmittler, and Philipp Slusallek. Rpu: A programmable ray processing unit for realtime ray tracing. *ACM Trans. Graph.*, 24(3):434–444, Jul. 2005.

[90] Yi Yang, Ping Xiang, Mike Mantor, and Huiyang Zhou. CPU-assisted GPGPU on fused CPU-GPU architectures. In *Proceedings of the 2012 IEEE 18th International Symposium on High-Performance Computer Architecture*, HPCA '12, pages 1–12, Washington, DC, USA, 2012. IEEE Computer Society.

[91] George L Yuan, Ali Bakhoda, and Tor M Aamodt. Complexity Effective Memory Access Scheduling for Many-Core Accelerator Architectures. In *IEEE/ACM International Symposium on Microarchitecture (MICRO-42)*, pages 34–44, Dec. 2009.

[92] Marcelo Yuffe, Ernest Knoll, Moty Mehalel, Joseph Shor, and Tsvika Kurts. A fully integrated multi-CPU, GPU and memory controller 32nm processor. In *Solid-State Circuits Conference Digest of Technical Papers (ISSCC), 2011 IEEE International*, pages 264 – 266, Feb. 2011.

[93] Ping Zhou, Bo Zhao, Jun Yang, and Youtao Zhang. A durable and energy efficient main memory using phase change memory technology. In *Proceedings of the 36th annual international symposium on Computer architecture*, ISCA '09, pages 14–23, New York, NY, USA, 2009. ACM.

[94] Xiaocheng Zhou, Hu Chen, Sai Luo, Ying Gao, Shoumeng Yan, Wei Liu, Brian Lewis, and Bratin Saha. A case for software managed coherence in many-core processors. In *Poster on 2nd USENIX Workshop on Hot Topics in Parallelism HotPar10s*, 2010.

[95] Yuhao Zhu, Yangdong Deng, and Yubei Chen. Hermes: an integrated CPU/GPU microarchitecture for IP routing. In *Proceedings of the 48th Design Automation Conference*, DAC '11, pages 1044–1049, New York, NY, USA, 2011. ACM.