NORTHWESTERN
UNIVERSITY

Computer Science Department

Technical Report
Number: NU-CS-2023-14

November, 2023

Uncovering Latent Hardware/Software Parallelism

Vijay Kandiah

Abstract

With the breakdown of Dennard Scaling, modern heterogeneous systems necessitate parallelism at both the hardware and software layers to meet today's demands for performance and energy efficiency. However, today's processors do not fully utilize the available parallel resources, leaving a lot of the system performance and energy efficiency unrealized. I postulate that the performance and energy efficiency of modern systems can be improved by leveraging information across system abstraction layers to uncover latent parallelism in hardware and software. Achieving peak throughput on modern CPUs often requires high CPU single-instruction, multiple-data (SIMD) unit utilization. To maximize the use of these SIMD/vector units in a user-friendly manner, I present Parsimony, a single-program, multiple-data (SPMD) programming model that exposes SIMD-level parallelism within general-purpose languages like C++. To further improve system throughput, we must also improve memory performance. To this end, I introduce Hybrid Consistency (HC), a hardware design that blends strong and weak memory consistency models by performing fine grained memory reordering to uncover memory level parallelism. Besides improving the performance of modern systems, we must improve their energy efficiency to meet today's performance targets while staying within a practical power budget. I demonstrate that GPU energy efficiency can be improved by uncovering parallelism in hardware computation structures with ST2 GPU. To evaluate ST2 GPU, we need accurate

performance and power models of modern GPUs. While GPU performance modeling has progressed in great strides, the community lacks an accurate power model for modern GPUs. To address this decade-long gap, I present AccelWattch, a robust power modeling framework for modern GPUs.

Keywords

Parallel Computing, Computer Architecture, Compilers, Memory Consistency, Energy Efficiency, Graphics Processing Units, Single-instruction Multiple-data

NORTHWESTERN UNIVERSITY

Uncovering Latent Hardware/Software Parallelism

A DISSERTATION

SUBMITTED TO THE GRADUATE SCHOOL
IN PARTIAL FULFILLMENT OF THE REQUIREMENTS

for the degree

DOCTOR OF PHILOSOPHY

Computer Engineering

By

Vijay Kandiah

EVANSTON, ILLINOIS

November 2023

Vijay Kandiah

vijayk@u.northwestern.edu

ORCID iD: 0000-0002-6853-9964

ABSTRACT

With the breakdown of Dennard Scaling, modern heterogeneous systems necessitate parallelism at both the hardware and software layers to meet today's demands for performance and energy efficiency. However, today's processors do not fully utilize the available parallel resources, leaving a lot of the system performance and energy efficiency unrealized. I postulate that the performance and energy efficiency of modern systems can be improved by leveraging information across system abstraction layers to uncover latent parallelism in hardware and software. Achieving peak throughput on modern CPUs often requires high CPU single-instruction, multiple-data (SIMD) unit utilization. To maximize the use of these SIMD/vector units in a user-friendly manner, I present Parsimony, a single-program, multiple-data (SPMD) programming model that exposes SIMD-level parallelism within general-purpose languages like C++. To further improve system throughput, we must also improve memory performance. To this end, I introduce Hybrid Consistency (HC), a hardware design that blends strong and weak memory consistency models by performing fine grained memory reordering to uncover memory level parallelism. Besides improving the performance of modern systems, we must improve their energy efficiency to meet today's performance targets while staying within a practical power budget. I demonstrate that GPU energy efficiency can be improved by uncovering parallelism in hardware computation structures with ST2 GPU. To evaluate ST2 GPU, we need accurate performance and power models of modern GPUs. While GPU performance modeling has progressed in great strides, the community lacks an accurate power model for modern GPUs. To address this decade-long gap, I present AccelWattch, a robust power modeling framework for modern GPUs.

ACKNOWLEDGEMENTS

My Ph.D. journey has been an incredible experience thanks to the support and mentorship I have received from a number of people over the last six years.

First and foremost, I would like to express my deepest gratitude to my advisor, Professor Nikos Hardavellas, for introducing me to academic research, for teaching me how to be a better researcher, and for his unwavering support and mentorship throughout my time at Northwestern. My decision to pursue a Ph.D. was undoubtedly motivated by his support and encouragement while I was trying my hand at research during my first summer at Northwestern as a Master's student. This work would not exist without his help.

I would also like to thank my dissertation committee members, Professor Simone Campanoni, Professor Russ Joseph, and Dr. David Nellans, for their insightful feedback and support in helping structure my dissertation into its current format. I thank my industry collaborators Dr. Oreste Villa, Dr. Daniel Lustig, and Dr. David Nellans for their invaluable guidance and mentorship in bringing the Parsimony project to fruition.

Additionally, I would like to express my gratitude to all my colleagues and collaborators at Northwestern: Professor Peter Dinda, Ali Murat Gok, Georgios Tziantzioulis, Enrico Armenio Deiana, Haiyang Han, Brian Suchy, Mike Wilkins, Atmn Patel, Connor Selna, and everyone else who contributed to this work and offered companionship and support throughout my Ph.D. journey. Furthermore, I would like to thank my collaborators from the University of British Columbia and Purdue University for their important contributions to the AccelWattch project.

Lastly, I would like to sincerely thank my parents, my sister, and my girlfriend for their unconditional love, support, and sacrifice, all of which made this journey possible.

THESIS STATEMENT

Modern hardware has evolved to become highly parallel. Today's processors and software do not fully utilize these parallel resources, leaving a lot of the system performance and energy efficiency unrealized. I postulate that leveraging information across system abstraction layers can bridge the gap between what is possible and what is realized today.

TABLE OF CONTENTS

[1]This section is based on our CGO'23 paper about Parsimony [25].

[2]This section is based on our (to be submitted) ISCA'24 paper about HC [68].

[3]This section is based on our DAC'21 paper about ST^2 GPU [105].

[4]This section is based on our MICRO'21 paper about AccelWattch [121].

LIST OF FIGURES

LIST OF TABLES

CHAPTER 1
INTRODUCTION

The first 25 years of microprocessors saw an exponential growth in single-thread performance as the number of onboard transistors grew exponentially along with the microprocessor clock frequencies. As transistor sizes shrunk, the reduction in transistor power offered by Dennard Scaling [1] allowed chip manufacturers to continue drastically increasing core clock frequencies without significantly increasing the overall power consumption of the chip. However, Dennard scaling broke down in 2005 [2]. Chip manufacturers could no longer keep the power envelope constant from generation to generation and simultaneously achieve potential performance improvements. The primary reason for this breakdown is that as transistor sizes go down, the static power losses as a fraction of the supplied power increase even more rapidly. Static power dissipates as heat, causing the chip to warm up. This increase in temperature exponentially increases static power, entering a positive feedback loop that threatens thermal runaway. Thus, as seen in Figure 1.1, the breakdown of Dennard scaling created a "power wall" that forced processor clock rates to stop increasing any further, peaking at around 3 – 4 GHz, and processor power consumption peaked in the range of a few hundred watts. Hence, chip manufacturers were no longer able to attain the likes of 2X improvement in processor performance roughly every 18 months through clock frequency scaling.

Upon the breakdown of Dennard scaling, to continue increasing processor performance even further without raising power consumption beyond the limits of practical cooling technology, the industry has turned towards extensive parallelism. Modern systems introduce compute parallelism in various flavors: with increase in the number of processor cores by introducing multicore processors, with wider single-instruction, multiple-data(SIMD)/vector units such as x86 AVX-512 [3],

and with heterogeneous computing using co-processors and/or accelerators such as Graphics Processing Units (GPUs). However, sequential programs written in standard sequential programming languages still dominate today's important domains [4]. Adapting these programs to take advantage of parallel computing systems often requires considerable programmer effort. Hence, there is now a necessity for system software to have user-friendly paradigms that allow programmers to leverage the various dimensions of parallelism offered by modern heterogeneous systems, without having to worry about the complex hardware implementation details. However, current user-friendly approaches targeting this goal, such as the widely popular OpenMP [5] paradigm, generally maintain serial loop semantics which inhibits the full utilization of the parallel compute resources available in hardware, as explained in Chapter 2.1.3.

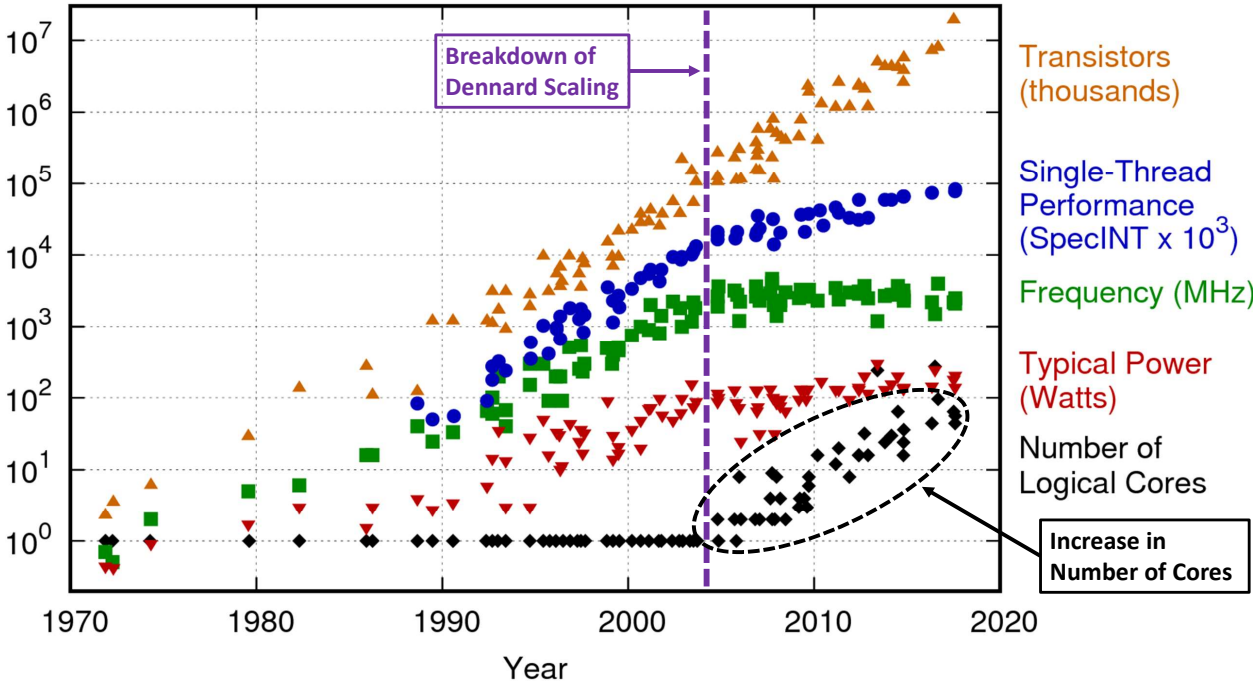


Figure 1.1: 42 Years of Microprocessor Trend Data [6].

One avenue of parallelism in modern systems is at the SIMD/vector unit level. Maximizing the utilization of these SIMD/vector compute units is often necessary to achieve the advertised

peak computational performance on modern CPUs. For instance, in Intel's Cascade Lake microarchitecture [7], leveraging the vector units allows up to 68 32-bit integer operations to be simultaneously executed within each core. On the other hand, without using any vector units, a maximum of only 10 32-bit integer operations can be simultaneously executed within each core. Although new ISA extensions such as x86 AVX-512 [3], ARM SVE [8], and RISC-V "V" extension [9] continue to introduce instructions with richer computational power, targeting these ISAs still remains a major challenge for developers and toolchain providers. Single-program, multiple-data (SPMD) programming models such as ispc [10] have been proven to be an effective way to use high-level programming languages to target these vector ISAs. Unfortunately, many such SPMD frameworks have evolved to have either overly restrictive language specifications or under specified programming models as explained in Chapter 2.1. This has impeded the wide-scale adoption of SPMD-style programming to leverage the SIMD-level parallelism offered by modern CPUs. To overcome such limitations of prior SPMD frameworks and facilitate SPMD-style programming of SIMD/vector units, we introduce Parsimony (PARallel SIMd), a well-specified SPMD programming approach and compiler flow that efficiently targets a CPU's SIMD/vector units while remaining compatible with standard programming models, languages, and compiler toolchains. Parsimony's programming model semantics enable a standalone compiler IR-to-IR (Intermediate Representation) optimization pass that can "program" SIMD/vector units independently of other compiler passes. In other words, the standalone Parsimony compiler pass translates SPMD-annotated function(s) into architecture-independent vector IR. Hence, the standard compiler back-end for each architecture can optimize Parsimony generated vector IR for the target ISA as it sees fit. We show that our prototype implementation of Parsimony in LLVM [11] achieves performance parity with state-of-the-art SPMD frameworks (i.e., ispc) and custom handwritten AVX-512 code, without requiring the use of a specialized programming language or compiler.

Generating SIMD/vector instructions alone is not enough to maximize system throughput. Unfortunately, the performance of memory is still the limiting factor for performance of several important applications [12]. Furthermore, this has been the case for several decades [13]. Rodrigues *et al.* [14] found that High Performance Computing (HPC) programs are often dominated by memory instructions (45% of all executed instructions) and integer instructions (29.5% of all executed instructions). Additionally, a majority of these integer instructions were found to be used to calculate memory addresses. As such, to maximize system throughput, we also need to maximize the performance of the memory system. We can improve memory performance in two ways, by improving the latency of memory operations, and by improving memory throughput. This dissertation focuses on leveraging memory-level parallelism to improve memory throughput. Improving the latency of memory accesses to a particular structure usually involves advancements in device physics and is beyond the scope of this dissertation.

Parallelism in memory can be improved by relaxing memory ordering constraints to allow memory accesses to execute out-of-order. Relatively strong Memory Consistency Models (MCMs) such as x86-TSO [15] enforce needlessly restrictive ordering constraints that serialize memory accesses going to different locations at the load-store queue. On the other hand, weaker memory models such as IBM Power [16]–[18] lay undue burden on the programmer or software layer to specify ordering with memory fences/barriers for program correctness. We can bridge this performance-programmability gap by enforcing strong MCM ordering only when absolutely necessary; i.e., by allowing memory accesses to execute out-of-order when their reordering does not affect observable program behavior. I capitalize on this observation and propose Hybrid Consistency (HC), an efficient hardware design that blends strong and weak MCMs by enabling a fine grained non-speculative reordering of memory operations at the load and store buffers. HC allows the programmer to reason about the program with strong MCM guarantees, thus requiring

less programmer burden relative to weaker MCMs. Under the hood, HC selectively relaxes ordering constraints for memory operations whenever safe to do so, thus extracting memory-level parallelism to improve system performance.

In addition to improving system performance, chip designers also need to improve energy efficiency to meet performance targets while staying within a reasonable power budget. GPUs are becoming increasingly popular for accelerating both general-purpose and HPC applications. There are 152 GPU-accelerated systems in the most recent TOP500 HPC list [19], and 70% of the top-50 HPC applications are GPU-accelerated [20]. Similarly, GPUs have become the dominant platform for machine learning and AI acceleration [21]. To meet ever-increasing performance targets, designers cram increasingly more cores per GPU chip, leading to a commensurate rise in power consumption. However, the power budget of modern GPUs is already reaching the limits of practical cooling technology. For example, both NVIDIA's Volta GV100 architecture [22] and the previous-generation Pascal GP100 [23] are limited by the same 250 W thermal design power, even though GV100 contains 43% more CUDA cores. To continue increasing the core count at a constant power budget, the cores must become more energy efficient. One way to improve the energy efficiency of GPUs is to leverage the parallelism inside computation structures such as arithmetic units to make these CUDA cores more energy efficient. We observe that the computed values of consecutive arithmetic computations from the same code location in real-world GPU applications are often highly correlated. We leverage this important but overlooked program behavior to propose Spatio-Temporal Shared-Thread (ST^2) adders, a power-efficient speculative adder design that utilizes the spatio-temporal history of arithmetic operations in a GPU kernel to perform additions. We show that ST^2 adders guarantee correct results while saving 70% of the nominal adder power. Furthermore, we estimate that ST^2 GPU, our proposed GPU architecture that uses ST^2 adders, saves 21% of the GPU chip energy with practically no performance and area overheads.

To evaluate hardware advancements such as ST^2 GPU, GPU architects require robust tools that will enable them to quickly and accurately model both the performance and power consumption of modern GPUs. However, while GPU performance modeling has progressed in great strides [24], GPU power modeling has lagged behind. We address the lack of cycle-level power modeling tools for modern GPUs by introducing AccelWattch, a GPU power model that is configurable, capable of cycle-level calculations in emulation and trace-driven environments, and supports DVFS. We validate AccelWattch on a NVIDIA Volta GPU, and show that it achieves strong correlation against hardware power measurements. Additionally, we demonstrate that AccelWattch can enable reliable design space exploration. Finally, we use a version of AccelWattch to perform the evaluation of ST^2 GPU.

The rest of this dissertation is structured as follows. Chapter 2 describes the Parsimony programming model. Chapter 3 details Hybrid Consistency (HC). Chapter 4 presents ST^2 GPU and AccelWattch. Finally, Chapter 5 provides a summary of my contributions, discusses directions for future work, and concludes this dissertation.

CHAPTER 2
PROGRAMMER-FRIENDLY HARDWARE-ACCELERATED SIMD/VECTOR PARALLELISM

2.1 Parsimony: Enabling SIMD/Vector Programming in Standard Compiler Flows[1]

2.1.1 Introduction

Achieving high computational performance on modern CPUs often requires making effective use of those CPUs' SIMD or vector units. Although single-thread performance scaling has slowed in recent years, single-instruction, multiple-data (SIMD) and vector ISAs continue to be an area of active innovation [26]–[28]. SIMD/vector registers are getting wider, with 512b registers already in widespread use. New ISA extensions such as x86 AVX-512 [29], ARM SVE [30], and the RISC-V "V" extension [31] continue to introduce instructions with richer computational power. For many workloads, these innovations can translate directly into improved throughput; however, targeting these new ISAs remains a major challenge for developers and toolchain providers alike.

Programming approaches targeting CPU SIMD/vector units fall broadly into three categories today. The simplest approach for programmers is to enable auto-vectorization of serial code. This works well for some applications [32] but can partially or completely fail to vectorize in other cases [33]. Moreover, the serial semantics of loops do not allow users to express synchronization points across loop iterations. This restriction makes it impossible to express operations such as "shuffles", which are often performance-critical to parallel workloads. A second approach is explicit SIMD/vector programming. This approach takes many forms including employing inline

[1]This section is based on our CGO'23 paper about Parsimony [25].

assembly, using low-level C intrinsics, or relying on pre-packaged SIMD-optimized libraries such as Enoki [34] or SLEEF [35]. Forcing developers to write low level code that explicitly maps the SIMD-amenable portions of their problem onto differing hardware ISAs is tedious, error-prone, and burdensome. The third approach is using a single-program, multiple data (SPMD) programming model that assumes a fixed number of threads or program instances executing in parallel. SPMD programming models such as `ispc` [10] have already proven effective at extracting good performance from CPU SIMD/vector units while retaining a user-friendly interface.

Unfortunately, current SPMD frameworks have made programming model decisions that make it difficult to express certain classes of algorithms and hard to integrate their compilation logic into existing compiler flows. For example, although `ispc` [10] delivers great performance, it requires writing programs in a custom "C-like" programming language as well as using a specialized standalone compiler infrastructure (derived from LLVM [11]); this increases the burden of adopting it into large projects. Another example, still from `ispc`, is the size of the thread "gang"[2] which is specified using a compiler flag. This approach is far from ideal. For instance, in a 512b SIMD architecture, a gang size of 16 would be ideal for 32b values, but inefficient for 8b values. A gang size of 64 would be ideal for 8b values but add tremendous register pressure with 32b values. Having to select a single gang size for the entire compilation unit makes performance tuning extremely tedious or impossible. Similarly, while threads in an `ispc` gang execute in synchronous fashion, later innovations in GPU SPMD programming models such as CUDA [36] have deprecated such "warp-synchronous" programming approaches in order to improve the soundness of the threading model [22].

As such, the goal of Parsimony is to introduce a well-defined SPMD programming model and compiler flow that efficiently targets a CPU's SIMD/vector units while remaining compatible with

[2] A "gang" in `ispc` is a group of concurrent program instances.

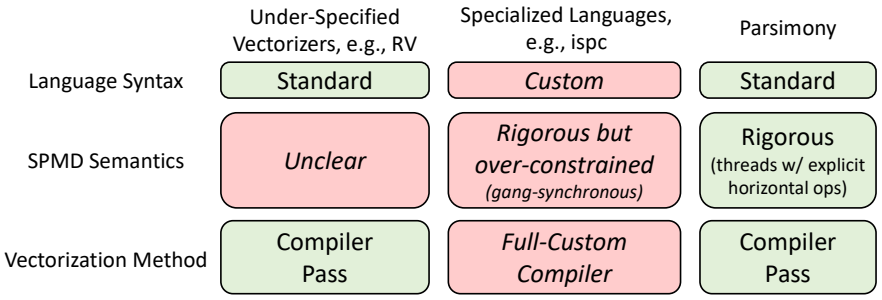


Figure 2.1: Existing SPMD vectorizers are effective but have shortcomings. Whole-Function Vectorization [37] and Region Vectorizer (RV) [38] do not clearly specify their intended semantics. Others (e.g. `ispc`) are overly-restrictive and hard to integrate into large projects. Parsimony targets well-defined SPMD semantics compatible with standard compilers, while achieving similar performance targets.

standard programming models, languages, and compiler toolchains. As shown in Figure 2.1, Parsimony's design starts at the language semantics level and is designed to be compatible with any number of front-end language syntax choices. A Parsimony-compatible language must only introduce the ability to conceptually instantiate a programmer-specified set of threads—using the term "thread" in the semantic sense, not necessarily as a true operating system (OS) thread. Inter-thread communication is permitted, but only when obeying standard inter-thread communication rules. Thus, Parsimony must expose efficient "horizontal synchronization" operations to facilitate synchronization within gangs. Due to this primarily single-threaded model, the code can pass through any standard optimization flow in the compiler. Vectorization instead occurs through a standalone IR-to-IR transformation pass that translates the SPMD-annotated function(s) into architecture-independent vector IR. Each architecture's standard back-end can then optimize the translated IR for the target ISA as it sees fit.

Overall, the contributions of Parsimony are as follows:

1. We present Parsimony, a well-specified programming model and compiler framework designed to remain fully compatible with standard language semantics and compiler flows.

2. We demonstrate a prototype implementation of Parsimony in LLVM, with performance results showing that our SPMD variant performs as well as state-of-the-art SPMD frameworks (i.e., `ispc`) and custom AVX-512 code, without requiring the use of a specialized programming language or compiler.

3. Based on our experience building Parsimony, we identify places where improvements/extensions to LLVM's IR would facilitate better integration of SPMD flows, and we provide takeaways for how languages and language extensions like C++ and OpenMP can integrate the Parsimony approach to SPMD for improved performance and programmer productivity.

4. We publicly release our Parsimony compiler framework and benchmarks to facilitate further research.

2.1.2 Background

SIMD ISA extensions employ a fixed-width SIMD register file and an instruction set that operates on fixed-width operands, e.g., 128b, 256b, or 512b in the case of x86 AVX-512 [29]. Conversely, "vector" ISA extensions such as ARM SVE [30] and RISC-V "V" [31] employ a vector-length-agnostic (VLA) instruction set that allows for implementations with different vector widths to support the same ISA. For example, ARM SVE supports hardware vector width implementations that can vary between 128b and 2048b in 128b increments. This enables pre-compiled code to run seamlessly across the supported vector widths without requiring recompilation or multiple program versions targeting differing hardware. In this section, unless otherwise specified, we use the terms "SIMD" and "vector" interchangeably as the differences between these approaches are important only if programmers are using low-level intrinsics, and are generally not significant if being targeted by a SPMD-style program.

The currently mainstream techniques to leverage SIMD and vector instruction sets and extract SIMD-level parallelism on CPUs are briefly discussed below.

Auto-Vectorization: In classical loop auto-vectorization, the compiler attempts to transform a region of serial code (usually a loop) into a block of vector instructions [39]–[41]. To do this, it relies on algorithms such as alias analysis as well as target-dependent heuristics to determine whether vectorization would be both legal and profitable. While this approach requires little to no additional programmer effort, auto-vectorization is opportunistic and is generally limited by the level of sophistication of the compiler's analysis abilities. As such, it tends to produce highly variable performance characteristics across systems. Language extensions such as C++ `std::execution::unseq` [42] or OpenMP `#pragma omp simd` [5] aim to improve the efficiency of auto-vectorization by providing user annotations or hints to the compiler, e.g., to ignore cases where the compiler cannot prove there are no loop-carried dependencies; however, many of the same fundamental challenges remain.

Vectorization remains an active area of research. Outer-loop vectorization [43] focuses on loops that are not the innermost in a hierarchy. This introduces additional challenges, as it raises the probability that there will be divergent control flow among outer loop instances. SLP vectorization [44], [45] is another auto-vectorization technique that combines similar independent scalar instructions to form vector instructions. Hence it offers more flexibility than loop vectorization because it does not just target parallelism across loop iterations. Additionally, auto-vectorization is performed on serial loops, and serial loops do not allow programmers to express horizontal communication between iterations.

Low-Level Intrinsics: SIMD/vector intrinsics are small functions that map nearly 1:1 to assembly instructions for a particular ISA. Manually inserting SIMD/vector intrinsics requires significant low-level programmer effort and is inherently non-portable. Nevertheless, due to the lim-

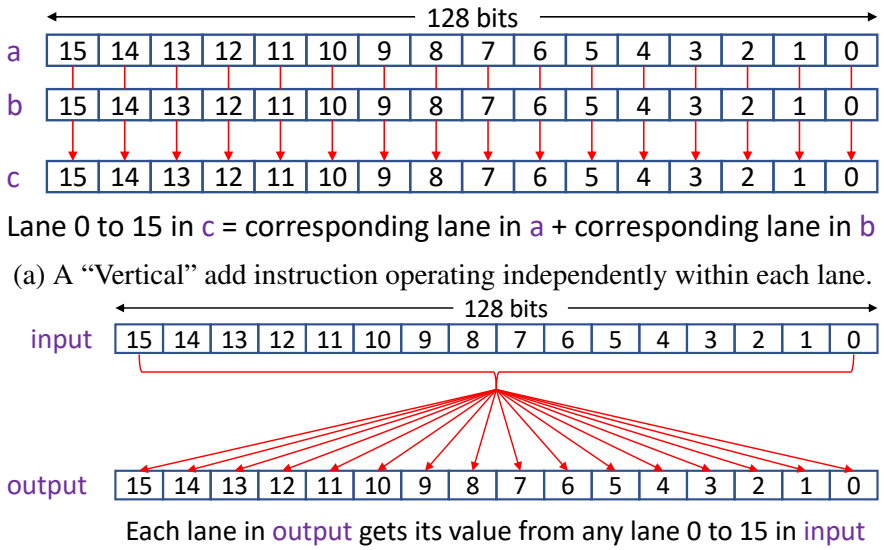
itations of the other programming approaches described in this section, libraries aiming for peak performance often contain extensive use of intrinsics in spite of the engineering costs [46].

SIMD Libraries: Libraries like enoki [34] and SLEEF [35] shift code portability into a library-supported layer for different architectures. While using SIMD libraries does make the source code more readable than the lower level intrinsics approach, similarly high performance can be achieved only if the source code can be expressed in terms of the limited set of exposed APIs that are specified by these libraries.

Dedicated SPMD Languages: SPMD-on-SIMD programming models such as `ispc` [10] generate multiple conceptual program instances that operate on different data from scalar C-like code. Each instance of the program is then mapped to a different SIMD lane to extract parallelism. While such programming models are very appealing to efficiently utilize the CPU SIMD/vector units, their use of non-trivial new keywords like "varying" and their reliance on non-standard compilation toolchains has made their widespread adoption to be practically difficult.

2.1.2 Mapping SPMD Programs to SIMD/Vector Units

When viewed through the lens of a SPMD program, both SIMD and vector ISA extensions enable traditional data-level parallel operations through "vertical" operations in which multiple logical "lanes" all operate independently. The number of lanes that can concurrently execute in hardware is thus a function of the SIMD/vector width and the data width of the operand being operated upon. For example, Figure 2.2a shows a SIMD add instruction performing a vertical addition of two 8-bit values across 16 lanes of two input 128b registers. In contrast, "horizontal" instructions operate across the lanes. For instance, a shuffle instruction exchanges values across a single SIMD/vector input register as shown in Figure 2.2b. Modern SIMD/vector ISAs also include complex instructions that are neither purely vertical nor purely horizontal. For example, AVX-512



(a) A "Vertical" add instruction operating independently within each lane.

(b) A "Horizontal" shuffle operation permitting any-to-any lane output.

Figure 2.2: SIMD/Vector operations can occur both per-lane and across lanes in high performance ISAs.

includes instructions that perform a vertical operation in combination with a horizontal operation (e.g., *vpsadbw* [29]). Such instructions are harder for compilers to target, but recent work [47] has improved the situation.

An enabling feature of modern SIMD and vector ISAs that allows for efficient SPMD programming is the support for masked execution with per-lane predication of execution output. The predication mask registers have one bit per lane and masked-off lanes will not modify their subportion of the output register used by the SIMD instruction. This fine-grained predication is critical when mapping programs onto a single thread executing SIMD instructions even though the SPMD threads diverge along different control flow paths.

2.1.3 Motivating Improved SPMD Semantics

In the examples below, we analyze variants of a simple program that copies data from each position in an array into an adjacent array position. This program is not as innocent as it may seem; it

```
// OpenMP version
template<typename T, unsigned N>
void foo(T* a) {
    #pragma omp simd
    for (unsigned i = 0; i < N; i++) {
        T tmp = a[i];
        // data race! cannot synchronize
        a[i+1] = tmp;
    }
}
```

Listing 2.1: OpenMP maintains serial execution semantics.

highlights several interesting subtleties that can arise in SPMD programming model decisions and exposes compiler implementation issues that may appear.

Listing 2.1 presents a version of the program written in C++ with OpenMP. As required by most `#pragma` implementations, the program semantics can be fully understood by ignoring the `#pragma`: e.g., each loop iteration reads the value of `a[i]` and writes it to `a[i+1]`, where the latter is then read during the next loop iteration. In OpenMP the use of the `#pragma` allows the compiler to legally ignore loop-carried dependencies that can be difficult to analyze (though in this case the dependency is obvious). With the fall-back capability of generating single-threaded code despite the program's `#pragma` SIMD directive, many auto-vectorizing compilers will choose to output functionally correct single-threaded execution and fail to vectorize this simple piece of code. Neither OpenMP nor similar constructs as in C++ `std::execution::unseq` provide an explicit way for programmers to specify that a loop-carried dependency should be ignored, allowing vectorization of the load and the store, i.e., to have all loop iterations first perform the load before any parallel execution path performs the store. Pragmatically specifying this type of synchronization requires explicitly breaking the single loop into portions, and while straightforward in this program, it becomes complex or impossible in large regions.

An `ispc` version of the same program is shown in Listing 2.2. Due to `ispc's` gang-

```
// ispc version (limited support for templates)
void foo(uniform int a[]) {
    foreach(uniform i : 0 ... N) {
        int tmp = a[i];
        // implicitly gang-synchronous!
        // correct only if N <= compile time gang size
        a[i+1] = tmp;
    }
}
```

Listing 2.2: `ispc` code is "gang-synchronous".

```
// Parsimony version:
template<typename T, unsigned N>
void foo(T* a) {
    #psim gang_size(N) {
        uint64_t i = psim_get_lane_num();
        T tmp = a[i];
        psim_gang_sync();  // explicit!
        a[i+1] = tmp;
    }
}
```

Listing 2.3: Parsimony makes gang size and horizontal synchronization explicit.

synchronous execution model, `ispc` *requires* all threads in the gang to execute the load before any thread executes its store. However, in `ispc`, the gang size is a compilation flag that is tightly coupled with the ISA SIMD width of the target machine. Programmers can access it through the `programCount` variable, but not set it. Therefore, the correctness of this code changes depending on the relationship between gang size and `N`. This is less than ideal from a programming model perspective.

Listing 2.3 now demonstrates how the running example would be written using Parsimony, using `#psim` syntax as one example of how to demarcate an explicit SPMD parallel region. Described in more detail later in Section 2.1.4, Parsimony compatible code explicitly instantiates a programmer-specified number of independent threads that can also be grouped into gangs. The gang size need not match the hardware's SIMD width; the compiler back-end can map any gang

```
// Parsimony version:
template<typename T, unsigned N>
void foo(atomic<T>* a) {
    #psim gang_size(N) {
        uint64_t i = psim_get_lane_num();
        a[i].fetch_add(1, memory_order_relaxed);
        a[i+1].fetch_add(1, memory_order_relaxed);
    }
}
```

Listing 2.4: Example showing how "gang-synchronous" behavior can break compiler optimizations legal for single-threaded code.

size onto any target ISA. Because the number of threads is specified at the program level, a developer can reason about program correctness strictly based on the programming model. There is no requirement to know the compiler options being specified nor the SIMD/vector width of future hardware the program will be executed on. As with modern GPUs [36], but differing from `ispc`, Parsimony eschews a gang-synchronous programming model and instead requires the programmer to explicitly synchronize across a gang when needed. This makes it easier to incorporate standard sequential semantic compiler passes and facilitates the possible adoption of more flexible forward progress guarantees in the future.

Listing 2.4 presents a different example showing how a gang-synchronous programming model can introduce semantics incompatible with standard compiler optimizations in languages such as C++. Because the example operations are atomics, there are no concerns about data races regardless of the actual execution order in hardware. A typical single-thread compiler optimization pass can tell that the atomics are performed to two adjacent non-aliasing addresses. Therefore, it would be legal for the compiler to reorder the atomics arbitrarily. However, in a gang-synchronous model, all threads in the gang are required to perform the first atomic before any thread in the gang performs the second atomic. Therefore, the second atomic in each thread must read the result written by the first atomic from the adjacent thread (except at the boundary condition). To pre-

serve this semantic, the compiler cannot reorder the atomics. Hence, optimization passes capable of reordering memory operations in cases such as this have to be explicitly disabled, modified, or specialized, making it difficult to integrate "gang-synchronous" SPMD models with modern vectorizing compiler flows.

The examples above show three important takeaways that motivate Parsimony's design. First, the semantics of a SPMD programming model should be well-defined in a way that the programmer can reason about at the language level, i.e., the semantics should not depend on any compile-time flags. Second, designing the semantics independently from the syntax allows the SPMD semantics to be integrated into widely used languages, facilitating adoption. Finally, the SPMD semantics should strive to be compatible with standard single-thread semantics to facilitate integration into standard compiler flows. The next subsection explains Parsimony's programming model and how it meets all of these goals.

2.1.4 Parsimony Programming Model

Parsimony is a general-purpose SPMD programming model designed to integrate cleanly into any programming language that supports threading and shared memory semantics. For explanatory purposes and in our implementation, we use standard C++ as the target language; however, the same principles extend to other languages.

In Parsimony's SPMD programming model, which is depicted in Figure 2.3, a SPMD region is a region of code in which a fixed number of conceptually independent threads are created. The SPMD region executes within the parent thread. This means the threads are conceptually "forked" at the start of the region and "rejoined" at the end of the region, where the parent thread continues its execution. However, no threads are actually forked in an operating system sense; the "fork" and "join" describes the threads' behavior within the language semantics. Within each thread, standard

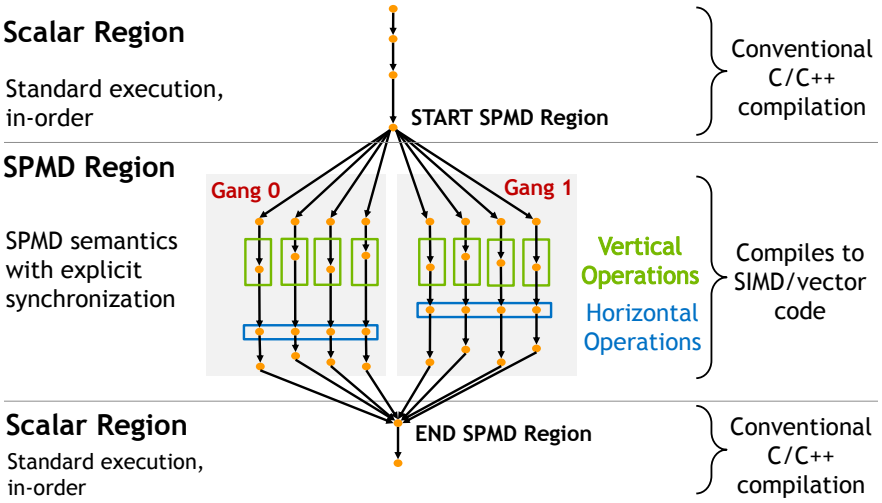


Figure 2.3: The Parsimony SPMD programming model.

intra-thread sequencing rules apply.

Threads are also grouped into *gangs* of a fixed size, determined by the programmer as part of the syntax declaring the parallel region. This allows different SPMD regions in a program to operate on different gang sizes, which is useful when differing functions operate on data structures having different element sizes. This differs from `ispc`'s approach, which specifies the size of the gang using a target-dependent compiler flag. Also unlike `ispc`, Parsimony threads are not "gang-synchronous"; there is no implicit synchronization between threads at every sequence point (i.e., before or after each statement). As mentioned earlier, this choice provides more optimization and scheduling flexibility to the compiler.

In Parsimony, synchronization between threads is instead performed using explicit horizontal operations. In contrast to auto-vectorization of loops or other language constructs such as `std::execution::unseq`, which is the current C++ standard recommendation for code targeting SIMD units [42], Parsimony threads may also communicate through memory using standard inter-thread memory ordering rules. As long as data races between threads are avoided, communication through memory is well-defined behavior.

```
void foo(uint32_t* a, uint32_t* b) {
  #psim gang_size(16) num_spmd_threads(N) {
    size_t i = psim_get_lane_num();
    if (a[i] + i < b[i]) {
      a[i] += 1;
    }
    b[i] = psim_shuffle_sync<uint32_t>(a[i], i + 4);
  }
}
```

Listing 2.5: Parsimony syntax, as embedded in C++.

Parsimony guarantees concurrency among threads in each gang; if one thread in a gang has started, other threads in the gang are also guaranteed to start. The fairness guarantee is that if all threads in the gang individually make forward progress, then all threads will eventually make forward progress. This rule is necessary to ensure that horizontal operations (that occur across threads) behave correctly. However, Parsimony does not provide *global* forward progress guarantees, e.g., if one thread in a gang is waiting on a spinloop that will be signaled by another thread in the same gang, then the stalling thread will block the progress of the entire gang. Additionally, there is also no guarantee of concurrency or forward progress among different gangs. These restrictions are tighter than those in place on modern GPUs [22] which support a single-instruction multiple-thread (SIMT) programming model. GPUs may have hardware-assisted independent thread scheduling [22], whereas Parsimony relies on a more restricted forward progress model to ensure that there is no need for a software implementation of concepts such as SIMT convergence stacks [48], [49] or launching of multiple OS threads to enable thread preemption.

Listing 2.5 shows the syntax we have used to prototype Parsimony and employed in the examples in this work. These syntax choices are not fundamental and could be adapted as needed for different languages/frameworks. As shown, a SPMD region is identified with the `#psim` construct and prefixed with syntax indicating the gang size (`gang_size`) as well as the number of total threads (`num_spmd_threads`) or gangs (`num_spmd_gangs`). This gang size can

take any compile-time constant value; there is no dependency on the hardware vector width. The last gang may be partially full depending on whether the number of threads is a multiple of the gang size. The user can obtain the unique thread number within the SPMD region using `psim_get_thread_num()`, the gang number with `psim_get_gang_num()`, and the lane number within the gang with `psim_get_lane_num()`.

To allow further compiler optimization, the user can call the routines `psim_is_tail_gang()` and `psim_is_head_gang()` to explicitly identify the first and the last gang in the region. This is unique and important because the first and last gang are typically used to perform operations on the boundary of data structures. Hence, more expensive boundary condition checks are often performed there and a programmer may not want to burden all threads with performing those. The compiler can use this information to automatically extract the first and last gang into a copy of the function that is separate from the rest, so that the boundary condition checks can be optimized away from the non-boundary gang execution. Parsimony provides no guarantee of ordering among gangs, so depending on the compiler implementation they can be executed sequentially, out of order, and/or in parallel.

The body of the SPMD region automatically captures variables from outside the region by reference, as needed; the SPMD region itself takes no explicit arguments. SPMD threads may contain any arbitrary language constructs, including arbitrary control flow or memory access patterns, subject to standard language semantics. Parsimony also provides a set of APIs for operations not typically exposed in standard language APIs, such as saturating math operations and horizontal shuffle and data exchange operations.

As mentioned earlier, the choices described above were made to facilitate the use of a standalone IR-to-IR vectorization pass that can be integrated easily into standard language toolchains. The next section describes the details of how to implement such a pass.

2.1.5 Parsimony Compiler Implementation

We now describe how Parsimony SPMD semantics integrate into a typical compiler flow and our prototype that manifests these concepts. We use LLVM for our implementation, though these concepts should generalize to other compilers.

The overall flow for Parsimony compilation works as follows, with each step described in further detail below. First, the program is compiled from source into the compiler's intermediate representation (IR) by the compiler front-end. The front-end is modified only in two ways: to support SPMD semantics within the source language as needed, and to disable any early-stage auto-vectorization that might occur by default. Second, the new Parsimony IR-to-IR vectorization pass is added to the middle-end optimization process. This new pass vectorizes the SPMD regions and is the core of the Parsimony design. Finally, the IR is translated to machine-specific assembly using the unmodified compiler back-end.

2.1.5 Front-End

The job of the compiler front-end within Parsimony is to produce a list of SPMD regions to be vectorized by the middle-end. We assume that the vectorizer operates at the level of whole functions, and as such, the front-end must extract SPMD regions from serial code into standalone SPMD-annotated functions. The vectorized function can later be re-inlined by the back-end in order to avoid the overhead of an extra function call. The SPMD annotation attached to the function must record relevant metadata such as the gang size and the total number of threads executing that region as specified by the Parsimony programming model.

Our prototype implements SPMD function extraction by piggybacking on Clang support for the extraction of `#pragma omp parallel` code regions. OpenMP parallel regions are implemented in Clang by outlining the parallel region into a standalone function, implicitly capturing any

```
// Original source code, before extraction
void foo(int* a) {
    // code before...
    #psim gang_size(G) num_spmd_threads(N) {
        // SPMD region code
    }
    // code after...
}

////////////////////////////////////////////////////////

// After extraction
void foo(int* a) {
    // code before...
    for (unsigned i = 0; i < N; i += G) {
        if (i + G <= N) {
            foo_extracted_full(/* captured vars */);
        } else {
            foo_extracted_partial(/* captured vars */);
        }
    }
    // code after...
}

// SIMD annotation: gang size G
inline void foo_extracted_full(/* captured vars */) {
    // SPMD region code
}

// SIMD annotation: gang size G
inline void foo_extracted_partial(/* captured vars */) {
    if (thread_id < N) {
        // SPMD region code
    }
}
```

Listing 2.6: An abstracted representation of the SPMD region extraction process performed by the front-end.

needed variables being referenced. After function extraction, the Parsimony front-end re-intercepts the OpenMP thread fork API and replaces it with a loop around a call to the Parsimony-vectorized function(s). This loop, which iterates over all of the gangs in the region, is specialized based on whether the total number of threads is known to be an exact multiple of the gang size and whether there are calls to APIs such as `psim_is_head_gang()` or `psim_is_tail_gang()`. Listing 2.6 shows a stylized example of the front-end flow.

2.1.5 Middle-End Vectorizer

The Parsimony vectorization phase is responsible for vectorizing SPMD-annotated functions generated by the front-end. This phase follows a flow similar to many existing vectorizers [10], [37], [38] but is tailored specifically to Parsimony's flavor of SPMD semantics. It is important to note that existing vectorizers often rely on being placed at a particular point within a bespoke sequence of optimization passes [50], whereas Parsimony's vectorization pass can be placed anywhere in the optimization pipeline. Parsimony's middle-end flow starts with the analysis of the scalar code, followed by transformation into vector code, as described below.

Control Flow and Mask Calculation: An SPMD annotation indicates that the function must be translated into a version in which G independent threads execute the function in SIMD fashion, where G is the gang size. The vectorized function's control flow must account for the possibility that the conceptually independent threads can diverge along different control flow paths. Capturing this divergent behavior requires the SIMD thread to reach all control flow branches executed by any of the conceptual SPMD threads. Threads that are not currently actively executing any particular control flow path need to be masked off so as to not disturb the values in those threads' lanes of the vector values.

Parsimony uses the following process to calculate its vectorization masks. First, it uses pre-

existing LLVM support for "structurizing" the control flow graph into a state where all forward control flow consists only of "if-then" patterns[3] [51]. Then, similar to prior work [37], two masks are prepared for each basic block: an entry mask and an active mask. In loop headers, the entry mask represents the mask of threads that entered the loop, and hence those which must also collectively exit the loop once all threads have finished iterating. In other basic blocks, both masks are identical. The active mask for each basic block is calculated as the logical-AND of the predecessor's entry mask and (if applicable) the condition on the branch at the end of the predecessor block. Loops also receive a dedicated mask for each exit; threads incrementally update these masks as they exit the loop. Once all threads have reconverged at the loop exit, the exit masks are used to steer subsequent control flow.

Shape Analysis: Shape analysis is a blanket term for various techniques described in literature as stride, affine, uniform, convergence, or divergence analysis [52]–[55]. Shape analysis attempts to track patterns in the value in all SPMD threads' copies of a single variable. For example, if the compiler can prove that a particular variable will always have identical contents in all SPMD threads, then it is *uniform*. If the compiler can prove that a particular variable will always be equal to some base value common to all threads plus a per-thread offset that is some fixed multiple of the thread number, then it is *strided*.

Shape analysis is critical to the performance of vectorized code in several ways. First, uniform values can be stored in scalar registers and be operated on by scalar instructions which can improve latency, throughput, and/or register pressure in many CPU architectures. Second, branches for which the condition is a uniform value can also be translated into scalar branches, rather than relying on masking the successor blocks, thus decreasing execution of fully masked dead code paths. Finally, shape analysis is crucial to the selection of efficient memory access instructions.

[3]This pass assumes the control flow is structured. For unstructured control flow, partial linearization [38] could be used.

The naive vectorization of a load and store instruction where each SPMD thread may be accessing unrelated memory addresses generate a SIMD gather or scatter operation, respectively. SIMD gathers and scatters are very slow on most modern CPUs—often no faster than performing each individual serialized scalar accesses. However, if the shape of the addresses accessed can be proven to be either uniform or strided, the compiler can generate highly efficient scalar or packed SIMD operations, respectively.

Parsimony classifies all value shapes into one of two categories: *indexed* or *varying*. Indexed values can be represented as a fixed common base value that may or may not be known at compile time, plus a per-thread offset that must be known at compile time. The common base values are maintained as scalar values in the IR, but the offsets are stored as metadata within the compiler. Varying values are those which are not indexed; these are stored as vector values in the IR. Note that both uniform and strided values are subsets of indexed values; the broader indexed category allows for more shape patterns to be captured, thus enabling more optimization.

Parsimony's shape analysis iterates on a per-instruction basis. Constants and function arguments are marked uniform. Calls to Parsimony APIs have operation-dependent shapes. For example `psim_get_lane_num()` is indexed with stride 1, while `psim_get_num_threads()` is uniform. The shape of each instruction is calculated by applying the semantics of the instruction to the shapes of its operands and then, if possible, interpreting the result as a new indexed value. If this is not possible, the output shape is marked as varying. If an instruction's input operand is not immediately available, e.g., due to a circular dependency within a loop, then the calculation proceeds speculatively but optimistically; the process then advances iteratively, recalculating any speculated shapes, until the result converges.

For example, consider an integer add or multiply instruction applied to two indexed operands with values $(a_{base} + a_i)$ and $(b_{base} + b_i)$, respectively, where a_{base} and b_{base} are the common base

values and a_i and b_i are the offsets for lane i. Addition produces

$$(a_{base} + a_i) + (b_{base} + b_i) = (a_{base} + b_{base}) + (a_i + b_i),$$

which can easily be interpreted as a new indexed value. Multiplication produces

$$(a_{base} \times b_{base}) + (a_i \times b_{base}) + (b_i \times a_{base}) + (a_i \times b_i).$$

This value can only be interpreted as indexed if a_{base} and b_{base} are known at compile time [53]. Otherwise, the two middle addends are neither common across all lanes nor per-lane values that are known at compile time.

For many instructions, the ability to classify a shape as indexed depends on certain facts about the input operands. For example, for a logical-AND operation, the outcome

$$(a_{base} + a_i)\&(b_{base} + b_i) = (a_{base}\&b_{base}) + (a_i + b_{base})$$

holds if b is a uniform negative power of two and a is an even multiple of $-b$, but may not hold otherwise. To enable this, some vectorizers also track metadata about properties such as variable alignment manually [10].

Parsimony performs shape analysis with the help of the z3 SMT solver [56] in two phases. In an offline phase, a large set of conditional shape transformations (such as shown above for logical-AND) are verified for correctness. At compilation time, known facts about IR values are tracked as z3 model constraints and a particular shape transform is applied only after verifying that its preconditions are satisfied by the operands. Although verifying the transformations can be slow, checking the preconditions takes just fractions of a second, so this online checking imposes

negligible compile-time overhead. This two-phase validation of transformations allows any new proposed transformation to be rigorously, yet easily, verified before being deployed in Parsimony.

Instruction Transformation: Transformation is the step where each instruction in the original scalar function is converted into the form it will take in the vectorized function. Most instructions will be vectorized, but some may remain scalar, e.g., if operating only on indexed values. We describe the handling of various instruction types below.

Arithmetic instructions are converted into vector form if their output shape is varying. For example, an instruction

```
%2 = mul nsw i32 %0, %1
```

operating on varying values `%0` and `%1` and producing varying value `%2` will be transformed into

```
%2 = mul nsw <G x i32> %0, %1
```

where `G` is the gang size. Arithmetic instructions operating on and producing only indexed values remain scalar, as only their common base value is stored at runtime.

In `alloca` instructions (stack allocation), the original size is multiplied by the gang size and pointer types are adjusted accordingly. A more optimized implementation could also (where possible) swizzle the data layout from array-of-structs into struct-of-arrays to avoid unnecessary gather/scatter operations on stack-allocated values [10].

Memory instructions are converted into a number of forms dependent on the shape of their address operands. Loads from a uniform address remain as regular scalar loads into uniform values. Stores to a uniform address are racy, unless only one thread is active; Parsimony chooses to emit a compile time warning then chooses one active thread to perform the scalar store. Loads from, or stores to, an address which is indexed with offset stride equal to the size S of the scalar type being accessed are converted into *packed* vector loads or stores of $G \times S$ consecutive bytes, respectively.

These packed operations are typically an order of magnitude more efficient than gather/scatter on all CPUs we have tested with Parsimony. Loads and stores of indexed values with other forms of stride may be converted into a packed load/store plus shuffle operation(s) if the indices remain within a particular bound (in our implementation, $4\times$ the gang size), as the accesses plus the extra shuffle(s) are still faster than performing gather/scatters. However, loads or stores of varying values must be converted into gather/scatter operations. All vector memory accesses are masked by the thread block's active mask to ensure that inactive lanes do not clobber data in memory or perform out-of-bounds accesses.

Branch instructions with varying values used as the condition are transformed into non-conditional branches to the originally-taken branch. This ensures that all paths through the CFG, potentially taken by any thread, will be properly evaluated. This can be further optimized by explicitly checking at runtime if any thread takes the branch and following the not-taken branch if none do. Prior projects have chosen to do this both implicitly [38] or explicitly via keywords such as `ispc`'s `cif` [10]. Branch instructions with uniform condition values remain as conditional branches.

The behavior of function call transformations depends on the callee. Calls to Parsimony intrinsic functions are implemented to match the semantics of that function. In many cases, e.g., for `psim_get_thread_num()`, the function can be replaced by a scalar or vector constant. Calls to functions with known vector interfaces can be made directly, adjusting for API peculiarities as needed (e.g., only some gang sizes may be available). Annotations analogous to `#pragma omp declare simd` [5] could be used to indicate that any standalone function should be vectorized and exported. Our prototype currently supports interaction with vector functions in the SLEEF math library [35], but we envision generalizing this in the future. Calls to scalar functions that cannot be inlined are transformed into a serial loop of scalar calls by each active thread individually. Note that this is another way in which the lack of gang-synchronous execution requirements makes

Parsimony code easier to compose, as separately-compiled scalar functions cannot be transformed to execute in gang-synchronous fashion.

ϕ nodes that have varying output values and are the join point for two forward edges must be converted into `select` operations. This operation picks the contents of each lane in the output vector value individually, based on the active mask of whichever predecessor is the 'then' block in the 'if-then' pattern that the entire CFG was earlier adapted into. This step is the key to ensuring that live values are not clobbered by unmasked arithmetic instructions executed by active and inactive lanes in the CFG predecessors. Other ϕ nodes can be transformed just as regular arithmetic instructions are.

2.1.5 Back-End

Once the vectorization pass has completed, the result can be passed to any number of other optimization passes and then to the unmodified compiler back-end. As part of this process, the IR will in most cases be further simplified. The back-end is also responsible for unrolling each vector instruction if the IR instruction's vector width (i.e., usually the gang size) does not match the width of the instructions available on the target. For example, with a gang size of 32 and a target ISA with 512b vector registers, an integer add IR operation on 32b `int`s (32×32b=1024b) would reduce down to two 512b SIMD assembly add instructions. The back-end is free to schedule these instructions however it chooses, subject to not breaking the semantics of horizontal operations.

Our Parsimony prototype focuses on x86 and AVX-512, an ISA with fixed-width vector support. We explored support for ARM SVE, a vector length-agnostic ISA, but LLVM support for such VLA ISAs in general is significantly less mature than for AVX-512, so we leave a full evaluation on SVE as future work.

2.1.6 Evaluation Methodology

We evaluate the Parsimony prototype on two benchmark suites. First, we ported the `ispc` benchmark suite to Parsimony enabled C++. Comparing to `ispc` directly allows us to quantify if `ispc's` more restrictive SPMD model enables better, worse, or similar performance to the more general Parsimony SPMD model. We adapted the `ispc` versions into Parsimony maintaining exactly the same algorithms. Second, we ported 72 benchmarks from the Simd Library [57], a popular high-performance image processing and machine learning library. This suite contains multiple versions of each benchmark, including serial and hand-coded versions specifically optimized for SIMD/vector ISA back-ends using manually-tuned low-level intrinsics. Due to pragmatic limitations, such as the Simd Library making heavy use of templates and custom C++ datatypes, we were unable to port these benchmarks to `ispc`—demonstrating the need for maintaining language and compiler level compatibility in SPMD programming systems.

Our IR-to-IR Parsimony pass is based on LLVM 15.0.1 [51] and our auto-vectorization comparisons were performed with LLVM's default vectorization (loop + SLP) pipeline. We also compared against various research and production auto-vectorizers, but elide the results because the broad trends were similar to LLVM's auto-vectorization, despite some variations in individual benchmarks. We compiled the `ispc` code with the latest release version of `ispc` (v1.18.0) [58] with default compilation flags. For all results, we report averages collected over five workload executions on a Intel® Xeon® Gold 6258R CPU with AVX-512 support compiled with Clang options `-O3 -march=native -mprefer-vector-width=512`. All experiments are single-threaded from the OS's point of view because Parsimony's SPMD design focuses on efficient SIMD/vector execution within a core.

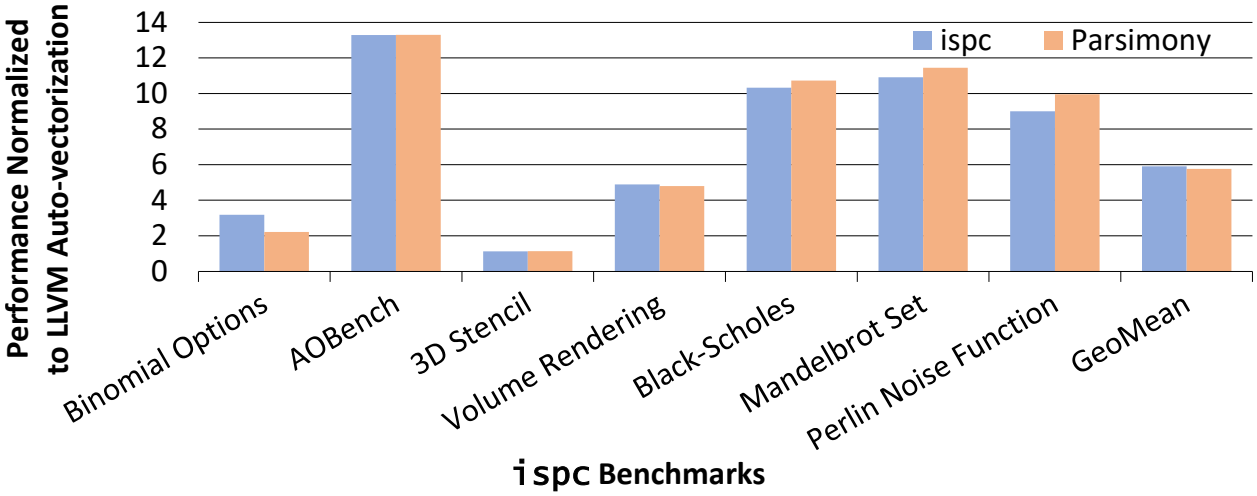


Figure 2.4: Parsimony and `ispc` performance compared to LLVM Auto-vectorization.

2.1.7 Experimental Results

Figure 2.4 shows the performance of Parsimony and `ispc` on 7 `ispc` benchmarks provided in the original `ispc` paper, normalized to the baseline LLVM 15.0.1 auto-vectorized serial implementation. Parsimony and `ispc` achieve a geomean speedup of $5.9\times$ and $6\times$ relative to auto-vectorization respectively. Parsimony closely matches `ispc's` performance on all benchmarks except Binomial Options, for which Parsimony achieves $0.71\times$ of `ispc's` performance. We were able to narrow this performance gap down to `ispc's` use of its built-in SIMD math library function `pow`. Our Parsimony prototype uses the SLEEF [35] math library for math functions such as `pow`, and SLEEF's implementation of `pow` for x86 AVX-512 is $2.6\times$ slower. This performance difference is not inherent to the `ispc` or Parsimony SPMD design choices. This demonstrates that gang-synchronous and non gang-synchronous SPMD designs can achieve nearly identical performance on modern architectures, *therefore we conclude that there is no performance penalty for choosing our easier-to-adopt non-synchronous SPMD semantics.*

To demonstrate the robustness of the Parsimony approach, Figure 2.5 shows the performance of Parsimony SPMD implementations, auto-vectorized serial C++ implementations, and hand-written

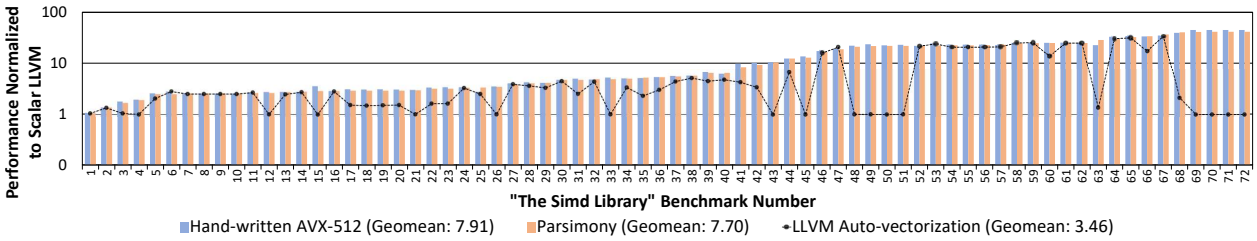


Figure 2.5: Speedup over LLVM scalar compilation, i.e., with vectorization disabled, on 72 Simd Library benchmarks.

AVX-512 implementations of 72 Simd Library benchmarks normalized to un-vectorized scalar implementations. LLVM's Auto-vectorization yields a geomean $3.46\times$ speedup over LLVM's scalar baseline, while Parsimony yields a geomean $7.7\times$ speedup; this results in a geomean $2.23\times$ speedup for Parsimony over auto-vectorization. Furthermore, the handwritten AVX-512 intrinsics implementations perform negligibly better than Parsimony, as Parsimony achieves a geomean $0.97\times$ performance relative to handwritten implementations. *From these results, we conclude that Parsimony's flavor of SPMD semantics is capable of delivering near-peak SIMD performance without requiring programmers to resort to architecture-specific low-level intrinsic programming.* Moreover, Parsimony manages to achieve high performance while having a $7\times$ average code reduction relative to handwritten implementations while ensuring code portability with good compiler and language compatibility.

2.1.8 Discussion

For pragmatic reasons Parsimony uses a small number of architecture-specific IR constructs during instruction transformation. These operations are exposed in multiple SIMD and vector ISAs, although not always in the same way. We envision that important, common operations such as "multiply and return upper half" will be included as general-purpose compiler IR constructs in the future in order to further decouple vectorization from architectural constraints. For existing in-

structions that are neither purely horizontal or vertical, we explored exporting language level APIs with higher-level portable abstractions as part of this work. For instance, we abstracted the AVX-512 `vpsadbw` instruction, which accumulates the sum of absolute differences of 8b values in sets of eight lanes from the input register into a single 16b value shared by 8 lanes, using an opaque data structure added to the Parsimony programming API that could have multiple back-end implementations. For other instructions which may truly be unique to a particular ISA, developing a clean general-purpose exposure for them up through the programming model would be an interesting area of future work.

Parsimony's SPMD programming model differs from other contemporary parallel language approaches in several important ways. The C++ standard uses `std::execution::unseq` to describe loops in which different instances are not related by the "sequenced before" relationship that otherwise orders operations within the same thread. Unfortunately, concurrent accesses by multiple unsequenced evaluations to the same address are considered racy and hence have undefined behavior. Similarly `std::execution::par` allows spawning of threads to execute instances, but describes instances as indeterminately-sequenced, implying that there is no concurrency between iterations assigned to the same thread, which prevents important horizontal operations with high-performance ISA support from being used. This could be resolved by introducing a `std::execution::spmd` execution policy relying on Parsimony SPMD semantics, as well as by introducing horizontal operations and other relevant SPMD APIs. Likewise, OpenMP `#pragma omp simd` and OpenACC pragmas generally maintain serial semantics and hence also do not permit horizontal operations. These languages could similarly introduce keywords or annotations for interpreting loops as to be executed using Parsimony SPMD semantics, but these would likely no longer use `#pragma` notation, as pragmas are generally meant to be safe to ignore.

2.1.9 Related Work

Compiler auto-vectorization has a long history [39], [59]–[62]; loop vectorization and superword-level parallelism (SLP) vectorization are well researched classical compiler optimizations that are enabled in many compilers today. Traditional loop vectorization has seen numerous advances such as outer-loop vectorization [43] and vectorization, for interleaved [41] and misaligned [63] data access patterns. SLP vectorization [44], [45], [64] has been developed as an alternative more flexible approach to loop vectorization. In contrast to these, Parsimony does not need to extract SIMD/vector parallelism from source code for vectorization thanks to its explicitly parallel SPMD semantics.

SPMD programming models with data parallel languages such as `ispc` [10] and ones with C++ SIMD extensions such as Sierra [65] have well-defined SPMD semantics but are more restrictive than Parsimony's proposed semantics. Prior work has also studied the use of GPU-focused SPMD programming models to target CPU SIMD units [66], [67]. Compiler passes such as the Whole Function Vectorizer (WFV) [37] support vectorization of arbitrary functions using SPMD-like semantics and Moll and Hack [38] extend this to support arbitrary unstructured control flows. However, unlike Parsimony, these passes do not provide precisely defined semantics.

2.1.10 Conclusion

In this work we demonstrate that having rigorous SPMD threading, memory, and forward progress semantics can facilitate the adoption of SPMD into widely-used general-purpose programming languages and toolchains. Our Parsimony compiler prototype shows that C++ code written using these principles can match the performance of code written using custom SIMD-targeted languages and AVX-512 assembly intrinsics. From this we conclude that by leveraging the right set of SPMD semantics, SPMD programming within mainstream languages is a more effective method

of programming a CPU's SIMD/vector units rather than relying on custom languages or low-level intrinsic programming.

CHAPTER 3
UNLOCKING MEMORY PARALLELISM THROUGH FLEXIBLE MEMORY REORDERING

3.1 HC: Fine grained Dynamic Blending of Memory Consistency Models[1]

3.1.1 Introduction

As shared-memory multiprocessors become increasingly more parallel with rapidly growing processing core counts, their performance scalability has become increasingly more susceptible to the speed of data communication through their intricate memory systems. Memory performance is the limiting factor for the performance of several important applications [12], [14]; 63% of the normalized CPI stack averaged across 15 multithreaded workloads from the PARSEC[69] and NAS[70] application suites shown in Figure 3.1 corresponds to memory component accesses, i.e., accesses to L1, L2, and L3 data caches, and dram. Thus, to improve the performance of modern systems, we need to improve their memory performance.

Shared-memory systems guarantee the correctness of inter-core communication by relying on memory consistency models (MCMs) that stipulate ordering guarantees for the execution of memory operations with respect to their program order. The MCMs present in today's systems range from relatively strong models such as the popular TSO [15] present in x86-64 systems to relatively weak models such as the ones present in ARM [71] and IBM Power [16] systems. Strong memory models present an intuitive programming interface to the user by preserving the program order of memory accesses. This is done by conservatively serializing the execution of most memory ac-

[1]This section is based on our (to be submitted) ISCA'24 paper about HC [68].

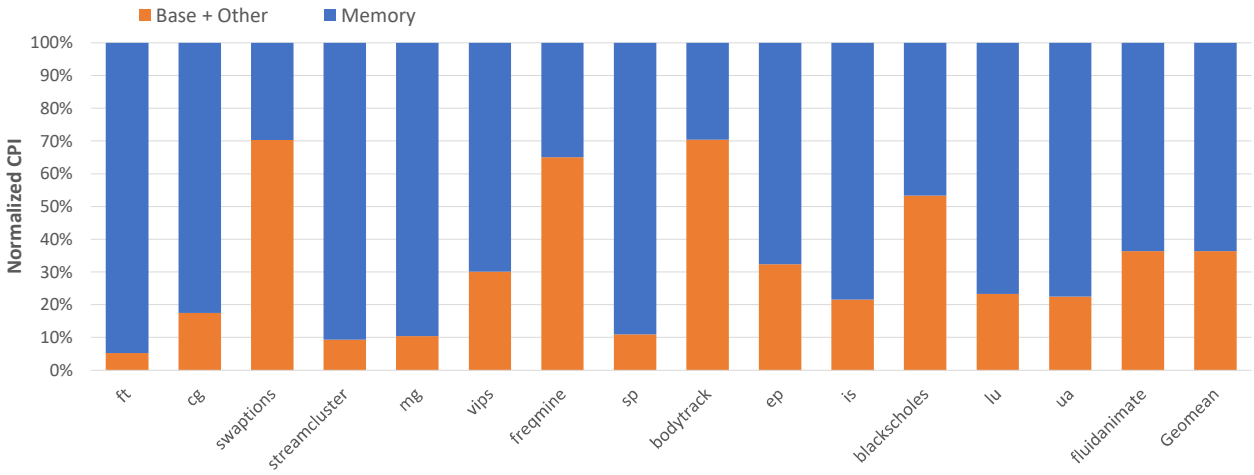


Figure 3.1: Normalized CPI stack breakdown of multithreaded applications from PARSEC[69] and NAS[70] suites. To improve system performance, we need to improve memory performance.

cesses, which can cause significant performance penalties due to restrictions on the reordering of memory accesses. On the other hand, weak memory models relax most, if not all, memory ordering constraints to allow the reordering of memory accesses to hide their long latencies and improve performance over strong models. However, such weak models require the user or software layer to insert memory ordering directives, such as memory fences and barriers, to enforce ordering in hardware whenever necessary to maintain program correctness. Thus, weak models trade off some of the programmability offered by stronger models in favor of improved performance.

Closing the performance-programmability gap between strong and weak MCMs has been widely researched [72]–[82]. Previous proposals to blend strong consistency models with weak consistency models enable reordering of memory accesses while still adhering to the strong model's ordering guarantees at the programmer interface. Some prior work employ speculative memory access reordering [72], [75]–[79], [81], [82]. However, such speculation-based techniques require extensive bookkeeping and sophisticated recovery mechanisms to handle misspeculations. Other proposals avoid speculation by employing memory access classification mechanisms that operate

at run-time complemented with compiler support [73], or are static compiler-only solutions that rely on programming language properties [74] to mark accesses to private and shared read-only memory locations as "reorder-safe". The key insight with this category of approaches is that they enforce MCM ordering only when necessary. Private and shared read-only accesses do not need to be executed in program order to guarantee strong MCM ordering at the programming interface. Thus, private and shared read-only accesses that are identified by the classification mechanism as reorder-safe are allowed to be executed out of program order from the processor's load and store buffers to achieve high memory-level parallelism without the need for speculative execution. Although compiler-based classification approaches are lightweight, they rely on static-time alias analysis to identify reorder-safe memory accesses and are conservative in nature. Approaches based on guarantees offered by programming languages offer a high coverage of reorder-safe accesses, but they cannot be applied to existing frameworks written in other languages. Unlike such proposals, techniques that perform the classification dynamically at runtime work well for code written in any language.

OS-based dynamic classification techniques that target memory access reordering at the load and store buffers, such as End-to-End SC [73] incur negligible area and energy overhead by relying on existing OS structures, such as the translation lookaside buffer (TLB) and the page table (PT), to mark entire pages as private or shared read-only. Performing the classification at the page granularity results in a significant amount of missed reordering opportunities due to a high degree of false sharing within pages. In Figure 3.2, 6 out of 15 applications show more than 40% (up to 91% in *fluidanimate*) false sharing when the classification is performed at the page granularity. This leads to a substantial decrease in the fraction of memory accesses that can be identified as reorder-safe from 55% to 23% (geomean across all 15 applications) if we perform the classification at the granularity of pages rather than cache lines. In addition to suffering from false sharing, prior

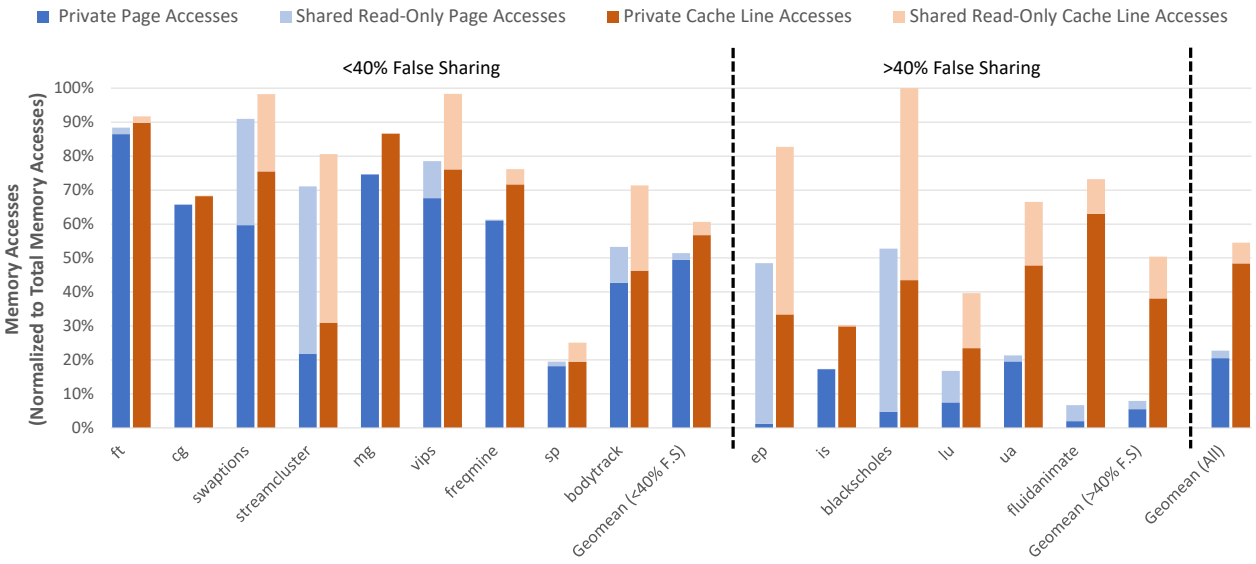


Figure 3.2: Classification of Memory Accesses as Private or Shared Read-Only. Access classification at the cache line granularity identifies double the number of reorder-safe accesses as page granularity classification.

OS-based mechanisms that target memory access reordering perform a non-temporal classification; i.e., once a memory location moves from being private to a shared state upon the first access by a second thread, it remains in this shared state for the remainder of the application's time, preventing any further reordering. Allowing re-classification of memory locations may increase the opportunities to reorder accesses to temporarily private locations.

This paper introduces Hybrid Consistency (HC), a lightweight hardware extension that enables flexible memory access reordering while still allowing users to rely on strong ordering guarantees to reason about their programs. Due to its prevalence in modern Intel and AMD multiprocessors, we pick TSO as the strong consistency model whose ordering guarantees that we relax with HC. Nevertheless, our HC design could also be applied to relax ordering in multiprocessors that implement sequential consistency [83]. In contrast to prior OS-based dynamic classification techniques such as End-to-End SC's dynamic variant (SC-dynamic), HC's access classification is performed

at a finer granularity whilst keeping the complexity of hardware changes minimal. Our key insights towards increasing coverage of reorder-safe accesses over prior work whilst maintaining a low-complexity implementation include the observation that we can capture most of the benefits of performing the classification at the granularity of cache lines instead of pages by grouping cache lines together into memory regions and tracking only a limited number of memory region owner threads per page. Additionally, HC performs eager re-classifications of memory regions to increase memory reordering for multithreaded applications with fork-join models and migratory data. Apart from extending the private TLBs with HC state bits per memory region, we propose to logically split the monolithic load and store buffers into weak-ordering and strong-ordering buffers by using a single reorder-safe bit per load and store buffer entry. Thus, accesses that go to logically weak-ordered buffers are allowed to be executed out-of-order, whereas accesses that go to the logically strong-ordered buffer maintain TSO ordering with respect to other accesses in the logically strong-ordered buffer. Experimental results show that HC outperforms the current state-of-the-art for memory access reordering with OS-based dynamic classification, End-to-End SC by a geomean 24% (up to 114%) across a set of 15 multithreaded PARSEC and NAS applications, with minimal increase in area and energy overheads. This is because HC can recognize 73% of all memory accesses as reorder-safe, while End-to-End SC only recognizes 39% of the same as reorder-safe.

In summary, the contributions of this work are as follows:

- We observe that practically all performance benefits of memory accesses reordering that can be achieved by performing dynamic classification at the granularity of cache lines over pages can be observed with a lightweight design that does not need to track memory at the cache line granularity.

- We show that performing a temporality-aware memory access classification by allowing

eager re-classifications can improve performance by up to 55% over non-temporal access classification designs.

- Armed with the above, we present HC, a hardware design that allows flexible memory access reordering at a finer granularity than previous approaches with a minimal increase in area and energy overheads. HC maintains the TSO ordering guarantees at the programmer interface while allowing safe reordering of 73% of all memory accesses.

- We evaluated HC on 15 multithreaded workloads to show that it improves system performance by 24% (up to 114%) over the current state-of-the-art design for memory access reordering with dynamic access classification.

3.1.2 Background

3.1.2 Dynamic memory reordering with End-to-End SC

A previous approach to dynamic memory reordering at the load and store buffers, End-to-End SC [73], performs a page granularity dynamic memory access classification with minimal area and energy overheads by relying on existing OS structures. End-to-End SC classifies entire pages as untouched, private read-only, private read-write, shared read-only, or shared read-write. Upon the first access to a page by any thread, the page transitions from the untouched state to one of the two private states with the accessor thread assigned as the owner of that page. Upon any subsequent access to that page by any thread that is not the owner, the page transitions to one of the shared states. This transition involves a OS TLB shootdown request to invalidate the TLB entry corresponding to the PTE present in the processing core to which the owner thread is mapped to. All memory accesses to pages in any of the private states or the shared read-only state are allowed to execute out-of-order from the load and store buffers, while the rest are not. As illustrated

in Figure 3.2, classification at the granularity of pages leads to a considerable number of misclassifications of accesses as unsafe to reorder. Furthermore, End-to-End SC does not reclassify a page as private once the page transitions to a shared state, thus missing out on further opportunities to reorder accesses.

End-to-End SC's low-complexity hardware design requires only two extra bits per TLB Entry and Page Table Entry; one write bit and one shared/private bit; to monitor the classification state. It also needs an additional field in the PTE to keep track of the ID of the owner thread. Additionally, End-to-End SC implements two separate store buffers; one for safe out-of-order stores and one for unsafe in-order stores. While End-to-End SC also performs a static compile-time analysis to complement its dynamic scheme, the additional performance improvement from adding its compile-time classification is only 0.5% over its dynamic scheme.

Our design is inspired by End-to-End SC's utilization of existing OS structures to perform a dynamic classification. While End-to-End SC is proposed as a design for ensuring Sequential Consistency, we consider a hardware TSO implementation that employs the OS/hardware-based dynamic scheme of End-to-End SC (SC-dynamic) as the current state-of-the-art TSO implementation. We compare the performance of our HC design with this TSO implementation.

3.1.2 Low Latency TLB Shootdowns

Virtual to physical address translation is performance critical for multiprocessors because it is performed on every memory access. Therefore, the processing cores in modern multiprocessors employ multiple private TLB structures, such as the L1 data TLB (DTLB), the L1 instruction TLB (ITLB), and a unified L2 TLB (STLB) to cache Page Table Entries (PTEs). These private TLBs of each core must be kept coherent with the Operating System's Page Table (PT) to uphold a unified view of virtual memory across all cores in the system. To update an entry in these private TLBs,

modern systems call upon the OS to invalidate the existing TLB entry to be able to replace it with the updated one from the page table. This process, called a TLB shootdown [84], involves Inter-Processor Interrupts (IPI) originating from the initiator core performing any modifications to the page table. The OS IPI handler of the core making the PTE modification (i.e., the initiator core) sends an IPI to the TLBs of all cores that might cache a copy of this PTE (i.e., the victim cores). Naive TLB shootdowns are generally expensive with an overhead of about 6600 cycles [85] due to the context switch and execution latencies for invoking and executing the OS handler respectively, and the IPI latencies. Additionally, the OS conservatively estimates the set of victim cores to send TLB invalidation requests to, resulting in false positives in the set of victim cores and unnecessarily interrupted cores.

Mechanisms such as DiDi [85] have been proposed to reduce the latency of TLB shootdowns. DiDi employs a shared, inclusive second-level TLB structure with an associated directory and a per core Pending TLB Invalidation (PTLBI) buffer to eliminate the need for costly IPIs, thus reducing the performance impact of TLB shootdowns by an order of magnitude. DiDi's 4096 entry shared, inclusive TLB structure avoids false positives in the set of victim cores that receive TLB invalidation requests. Moreover, DiDi enables invalidation of TLB entries on victim cores without interrupting their instruction stream by using per-core PTLBI buffers to inject memory barriers into the Load/Store Queue (LSQ) of the victim cores. DiDi's PTLBI invalidates the stale TLB entry at each victim core upon completion of the inserted memory barrier and sends an acknowledgment back to its centralized directory. Thus, by avoiding costly context switch overheads, DiDi is able to complete TLB invalidations on victim cores in a few hundred cycles, which is up to ten times faster than traditional TLB shootdowns.

In this work, we employ DiDi to update the classification state of HC memory regions in TLB entries with *TLBUpdate* messages during the HC state transitions described in Section 3.1.4.3.

We include the area and energy overheads of DiDi's hardware structures in our discussion of HC's overheads in Section 3.1.6.

3.1.3 HC Design Exploration

As shown in Figure 3.2, the memory access classification performed at the cache line granularity can identify more than twice as many reorder-safe accesses than when performed at the page granularity. This corresponds to an increase in opportunities to reorder memory accesses, which in turn leads to an improvement in system performance, as we demonstrate in Section 3.1.6. However, the TLB area overhead to naively keep track of the classification state for every cache line would be 64 times greater than doing so at page granularity, because there are 64 64B cache lines in a 4KB page. For simplicity, we assume that the number of OS threads does not exceed the number of processing cores. Thus, we would need *log(n)* bits to naively keep track of the owner thread ID for each private cache line within the TLB entry. For a 16-core multiprocessor with 4-bit thread ID, this adds up to 256 additional bits to keep track of the owner thread IDs. If we need 2 additional bits to monitor the classification state for each cache line in each TLB entry, we would require a total of 128 additional bits per TLB entry. This amounts to an additional 384 bits per TLB entry for the naive design, which would increase the TLB entry's size to over four times that of a typical TLB entry accounting for both data and tag information. Thus, the naive design is clearly impractical to implement in real processors.

We explore the design space to create a design that enables the performance benefits of classifying memory accesses at the granularity of cache lines instead of pages, while still being practical to implement. We first perform a sweep of the classification granularity, going from the size of cache lines (64B) to the size of pages (4KB), to find the largest memory region size that achieves most of the performance benefits of a cache-line classification. To conduct this granularity sweep,

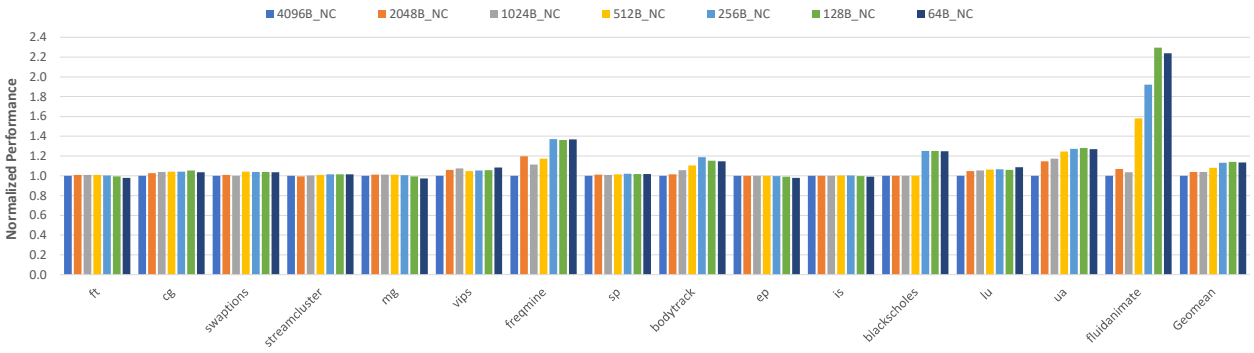


Figure 3.3: Performance impact of memory access classification granularity. Classifying at 256B granularity yields the same performance as classifying at cache line granularity.

we fabricate an oracle classification design that transmits the classification of a memory access, i.e., whether the access is reorder-safe or reorder-unsafe, to the load and store buffers without any costs involved to perform the actual classification. The load and store buffers allow reorder-safe memory accesses to perform out of order, while reorder-unsafe accesses maintain TSO ordering with respect to other reorder-unsafe accesses. Figure 3.3 illustrates the performance of this no-cost (NC) design, with the access classification performed at a progressively finer granularity of memory regions, ranging from page size (*4096B_NC*) to cache line size (*64B_NC*), normalized to *4096B_NC*. We observe from Figure 3.3 that classifying at 256B granularity (*256B_NC*) yields the same 13% geomean performance improvement over page granularity as classification at cache line granularity (*64B_NC*). The only application for which *64B_NC* does noticeably better than *256B_NC* is *fluidanimate*, yet *256B_NC* still manages to achieve 85% of the performance improvements of *64B_NC* over classifying at page granularity. If each TLB entry requires 2 additional bits to monitor of classification state in a page-level scheme, a design that performs a 256B level classification would require only 32 additional bits per TLB entry, as opposed to a cache line-level scheme, which would require 128 additional bits per TLB entry. Thus, we choose 256B as the optimal classification granularity for HC as it provides the same performance as the cache-line

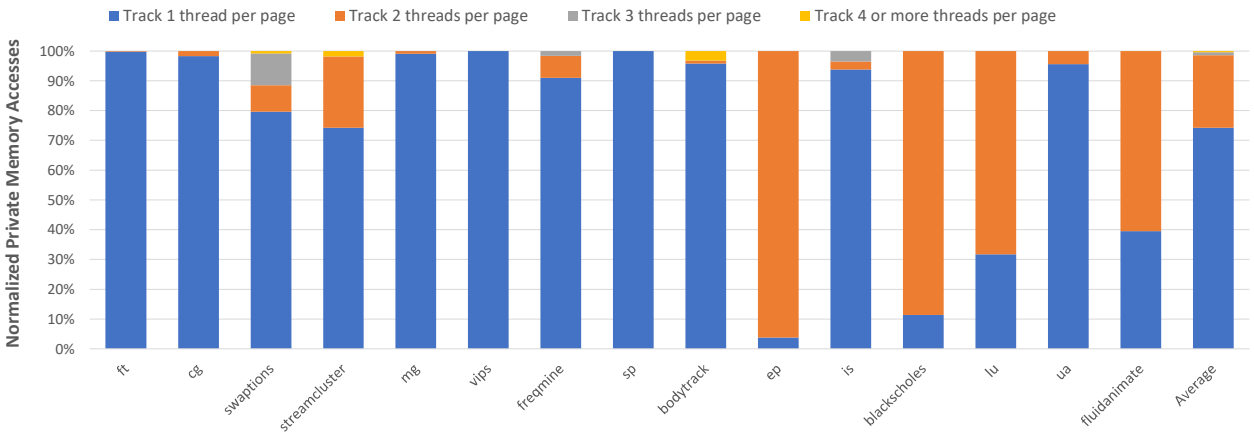


Figure 3.4: Breakdown of accesses to private HC regions by number of unique threads to keep track of per page. Keeping track of two unique threads per page is sufficient to capture all private accesses.

granularity design, whilst needing four times fewer bits per TLB entry to monitor region classification states.

Next, we consider the area overhead at the Page Table level. For every one of the 16 (for 4KB sized pages) 256B-sized memory regions within a PTE, which we will call as HC regions or simply as regions henceforth, we would need two bits to monitor the classification state. Additionally, we would need to keep track of the owner thread ID for each private HC region. Thus, for n processor cores, we need a total of $log(n) + 2$ bits for every HC region within a PTE. This amounts to $32\ bits + 16*log(n)$ additional bits per PTE which adds up to 96 additional bits per PTE for a 16-core multiprocessor, which is not quite scalable with increasing multiprocessor core counts. We postulate that the owners of private HC regions across a page are going to be the same or almost the same. Thus, we look at coalescing HC region owners to be able to keep track of fewer unique threads per page. Figure 3.4 shows the fraction of memory accesses that go to private HC regions (private memory accesses) broken down by the number of unique threads to keep track of per page. It is evident from Figure 3.4 that keeping track of two unique threads per PTE is sufficient

to identify 99% of private memory accesses. *swaptions* is the only application for which tracking more than 2 threads per PTE is necessary to capture noticeably more private memory accesses, but keeping track of only two threads already captures 88% of them. Thus, we choose to allocate space to keep track of two unique threads per PTE and make each private HC region within the PTE point to one of these two owner threads. If a PTE has more than two unique owners for private HC regions within it, we treat all private HC regions that are intended to be owned by the third or more owner threads as shared HC regions instead of private.

The design we have envisioned so far only allows for a single transition of a HC region from an unclassified state to be private to an owner thread upon the first access to it. Now, upon the first subsequent write access to the same region by another thread, the HC region will transition to a shared read-write state. All accesses to this HC region henceforth until the end of the application's runtime will be classified as reorder-unsafe and thus respect TSO ordering. However, allowing the re-classification of this HC region to be private to a new owner thread may increase reordering opportunities. Consider an example where a master thread initializes a set of HC regions. These HC regions will now be classified as private to the master thread. Next, the master thread directs each worker thread to access and modify a unique portion of the set of initialized HC regions. All initialized HC regions in the current design would be transitioned to shared read-write and all subsequent accesses by worker threads to these HC regions will be reorder-unsafe. Ideally, these HC regions should be re-classified to be private to their respective worker thread to allow for subsequent accesses to be classified as reorder-safe which allows them to execute out-of-order and improve system performance. Similarly, re-classifications are important when applications have migratory data behavior, i.e., when a single thread accesses and modifies a HC region for a long window of execution time before passing the ownership on to another thread and so on. Hence, to increase our coverage of private memory accesses, we consider adding re-classifications to HC.

Our HC design along with state transitions to allow for re-classifications is described in detail in Section 3.1.4. Once a HC region is classified as private to an owner thread, we perform eager re-classifications of the HC region to be private to a different thread upon any subsequent access by the other thread. We set a limit, PTC_{thr}, on the number of private transitions, known as Private Transition Count (PTC), that can take place for a HC region before it permanently transitions to a shared state. In our initial design without any re-classifications, PTC_{thr} is one because this allows the first transition of a HC region from unclassified state to be private to the first accessor thread. Subsequent accesses by other threads will cause the HC region to be permanently transitioned to a shared state. Quite clearly, increasing PTC_{thr} will either be beneficial in applications with behaviors similar to the examples above, or will not do much to improve performance if the HC region is just passed around by all threads. The more reuse a thread sees on a HC region, the more it benefits from relaxing ordering through re-classification of the region to be private to it. However, this comes with a cost, as there is an overhead associated with these re-classifications (private to private transitions). This cost includes invoking the OS handler, sending a *TLBUpdate* message to the processing core that maps to the owner thread, and waiting for an acknowledgment signal back from the core. We detail the cost of re-classifications and other HC state transitions in Section 3.1.4.3. Including all these transition costs in our HC design, Figure 3.5 shows the performance impact of increasing PTC_{thr} from 1 (*HC1*) to 32 (*HC32*) over that of a design with no re-classifications (*HC1*). We observe from Figure 3.5 that most of the performance benefit comes from allowing one re-classification, i.e., PTC_{thr} set to 2 (*HC2*). The re-classification costs involved when increasing PTC_{thr} beyond two balances out, or in some cases such as mg outweighs, the benefits obtained from allowing more re-classifications. Overall, (*HC2*) obtains a geomean 7% and a maximum of 55% (in ep) performance improvement over the design with no re-classifications, *HC1*. Hence, we choose a scheme that allows for one eager re-classification of HC regions as our

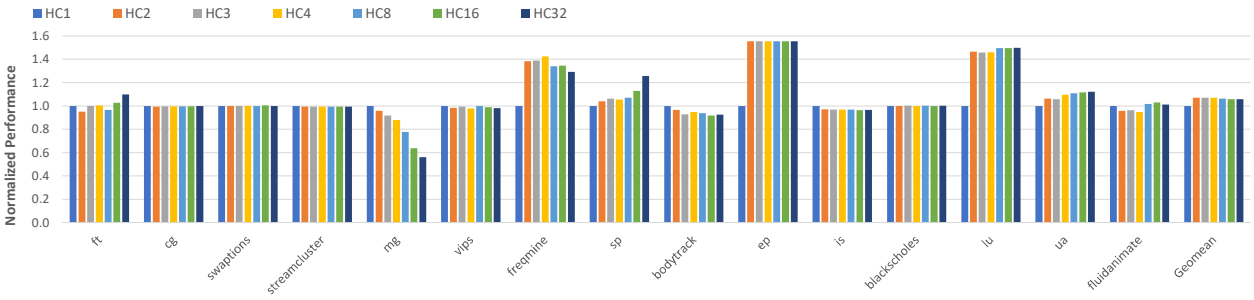


Figure 3.5: Performance impact of increasing HC private transition count(PTC) threshold. The maximum performance benefit of eager re-classifications comes from allowing just one re-classification (*HC2*).

final HC design. Note that we apply the same private transition threshold, PTC_{thr}, of 2 to all HC regions across all applications. However, each HC region in each application might benefit from having an independent PTC_{thr}. We leave the extension of our re-classification scheme to enable adaptive PTC_{thr} per HC region as an interesting future work direction. PTC_{thr} could possibly be adjusted dynamically for each HC region based on some application runtime metric.

3.1.4 OS/Hardware co-design for HC

3.1.4 Extending the Page Table and TLB

We capitalize on the observations made in the previous section to come up with a HC design that is practical to implement while enabling all of the performance benefits of a cache line granularity classification scheme. Figure 3.6 shows the extensions we make to the Page Table and the TLB.

We extend each PTE with additional fields (highlighted in blue in Figure 3.6) to allow the OS to keep track of each HC region's classification state (at the OS level) and PTC to enable re-classifications. As discussed in the previous section, we keep track of two unique owner thread IDs per page and have a 1-bit index, *Owner idx*, for each private HC region to index to either one of these owner threads. We can get away with using only a 1-bit PTC field in the PTE even though

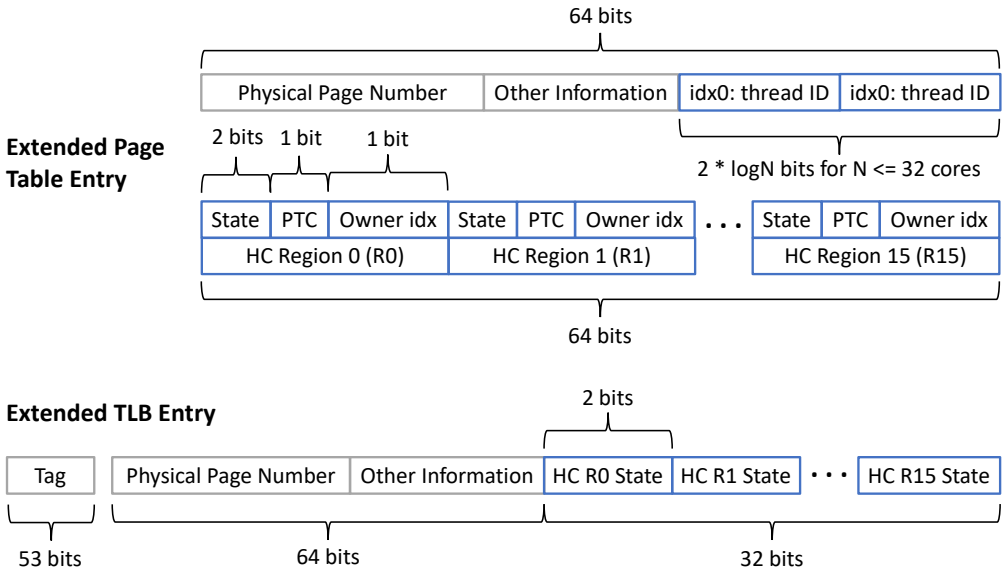


Figure 3.6: Extensions to the OS Page Table Entry and the TLB entry to support HC. Changes are highlighted in blue.

we allow two private transitions by only setting this PTC field for additional private transitions. In other words, an initial transition of a HC region from the unclassified state to be private to the first accessor thread will not set this PTC field whereas a subsequent private transition will set it. Thus, we infer that the logical PTC count is 1 for this HC region after the initial transition to private even with the PTC bit still unset. We detail the state transitions of HC regions in Section 3.1.4.3. In total, each HC region requires 4 bits to account for the classification state, PTC, and *Owner idx*. This amounts to 64 additional bits per PTE as there are 16 256B-sized HC regions each within each 4KB page. We require *2*log(N)* bits per PTE to keep track of the two owner thread IDs where N is the number of processing cores. There are 10 available (AVL) bits in a typical 64-bit PTE [86]. We propose to use these AVL bits to store the two owner thread IDs inside the PTE. Overall, for multiprocessors with up to 32 physical cores, each PTE will require 64 additional bits, i.e. an additional cache line, to monitor HC region state information. For simplicity, we assume that the number of threads in the application does not exceed the number of physical cores. In the

case that the number of threads for an application exceeds the number of cores, we hypothesize a design that tracks the processing core IDs instead of thread IDs by obtaining the mapping between the two from the Thread-Control-Block (TCB) that is maintained by the OS for each application thread.

A typical TLB entry in a 64 entry 4-way DTLB for a x86-64 processor that support Intel 5 level paging [87] and 57-bit virtual addresses uses 41 bits of the VPN and a 12-bit address-spaced identifier (ASID) as the TLB tag. The 64-bit PTE is stored in the corresponding data field of the TLB entry. Thus, a typical TLB entry can be considered to be 117 bits wide accounting for both tag and data fields. We extend each TLB entry with an additional 32-bit field that keeps track of the HC Region classification state at the TLB level. Each of the 16 HC regions within a TLB entry uses 2 bits to monitor its classification state. By monitoring the HC classification state at the TLB level, we can determine whether memory accesses that hit in the TLB are reorder-safe or reorder-unsafe without having to invoke the OS handler for each access to check the corresponding PTE. Note that we do not need to keep track of owner thread IDs and PTC at the TLB level because our state transitions described in Section 3.1.4.3 will invoke the OS handler upon TLB accesses when necessary to look at the corresponding PTEs.

For a 16-core multiprocessor with 4-bit thread indices, the naive cache line granularity classification design discussed at the start of Section 3.1.3 would keep track of the 2-bit HC classification state, the 1-bit PTC, and the 4-bit owner thread index for each cache line at the TLB level, which amounts to an additional 448 bits per TLB entry. In comparison to that, our design choices so far manage to reduce the area requirements per TLB entry by a factor of 14X to 32 bits. Similarly, at the Page Table level, the naive design would require an additional 448 bits per PTE. Our design choices manage to reduce this area cost by a factor of 7X to 64 additional bits per PTE. It is important to note that these reductions in TLB and Page Table area requirements are attained without

losing any performance benefits over the naive cache line granularity design.

3.1.4 Memory Pipeline Design

We logically split the monolithic load and store buffers into weak-ordered and strong-ordered buffers by extending their entries with a single reorder-safe bit. Entries in the load or store buffers with this reorder-safe bit set can execute out-of-order, while those without the bit set must respect TSO ordering with each other. We propose a memory pipeline design that initiates the address translation request to the TLB upon the issue of the memory access from the Reorder buffer (ROB). Typical DTLBs have a single cycle access time. Upon DTLB hits, if the access is to a private or shared read-only HC region, we insert the access into the logically weak-ordered buffer. Upon DTLB misses, we conservatively assume the access to be reorder-unsafe and insert it into logically strong-ordered buffer. This is done to avoid stalling the memory pipeline waiting for a DTLB miss to be served. An alternative memory pipeline design can possibly start the TLB access in parallel with inserting the access out of the ROB into the logically strong-ordered buffer. Once the TLB access gets served, it can set the reorder-safe bit if the access is to a private or shared read-only HC region to move the access to the logically weak-ordered buffer. The results in Figure 3.8 show that, for our evaluation suite of applications, there is no performance penalty for choosing the former design discussed above; i.e., the design that waits one cycle for the DTLB access to complete before inserting the load or store into its respective buffer.

3.1.4 HC State Transitions

Figure 3.7 shows the state transitions of HC regions at both the Page Table level and the TLB level. The initial state of all HC regions of a page at the PT level upon page allocation by the OS page fault handler is set to *Unclassified* with PTC set to 0 and no owner thread assigned to

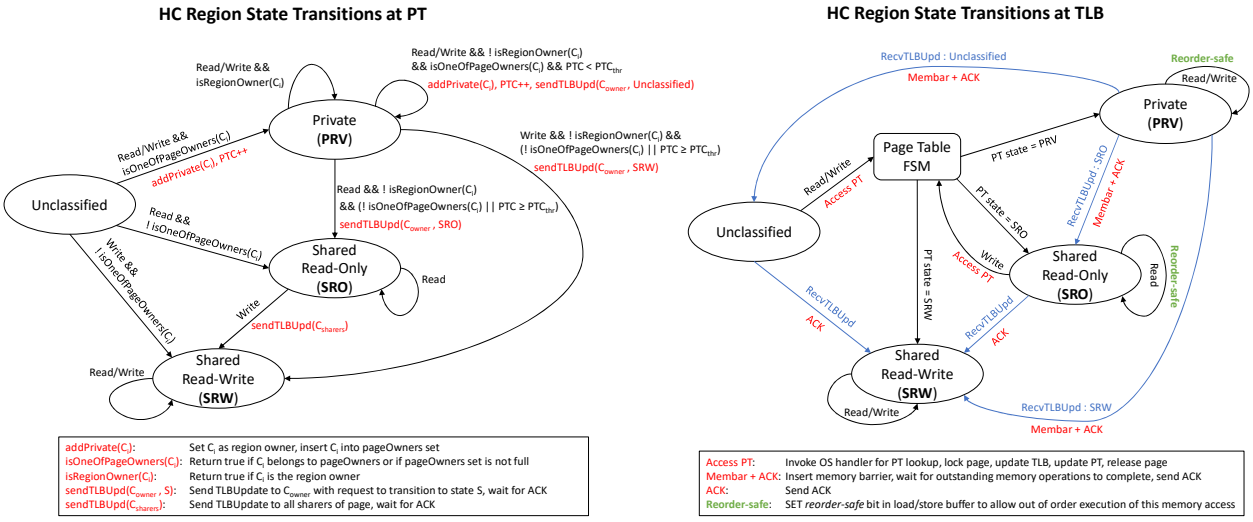


Figure 3.7: State transitions of a HC region at the OS Page Table level and at the TLB level.

the region. The first read or write to a HC region in the application will trigger a TLB miss. The OS TLB miss handler serving this miss will check the corresponding PTE and find that the region is in the *Unclassified* state. The OS handler checks if this first read or write to a HC region is done by a thread that is already part of the *pageOwners* set (two unique owner threads per PTE) of the corresponding PTE or if the *pageOwners* set of the PTE is not full yet; i.e., the OS performs the *isOneOfPageOwners* check described in Figure 3.7 for the accessor thread. If this *isOneOfPageOwners* check returns true, the HC region at the PT level will transition to the *Private (PRV)* state with the accessor thread marked as the *RegionOwner* of this region. The logical PTC for this HC region is also incremented by 1 due to this initial transition to *PRV* state. If the *isOneOfPageOwners* check returns false, the region will transition to the *Shared Read-Only (SRO)* state or the *Shared Read-Write (SRO)* based on whether the access is a read or a write respectively. HC region in the *PRV* state will remain in this state as long as all subsequent accesses to this region are made by the *RegionOwner*. Upon a first subsequent access to this HC region by a thread that is not the *RegionOwner* but for one that the *isOneOfPageOwners* check returns true, we

allow re-classifications of this region to be *PRV* with the new thread marked as the *RegionOwner*. This *PRV* to *PRV* transition, i.e., the re-classification of the region, only takes place if the PTC of the HC region is less than PTC_{thr}. The PTC_{thr} for HC design is two, thus allowing one *PRV* to *PRV* transition after the initial transition to *PRV* from *Unclassified*. There is a cost involved with performing this *PRV* to *PRV* transition. The OS handler sends a *TLBUpdate* message to the initial *RegionOwner*. This action is marked in red in Figure 3.7 as *sendTLBUpd*. We leverage DiDi [85] to send *TLBUpdate* messages as explained in Section 3.1.2.2. This *TLBUpdate* message requests the processing core that is mapped to the initial *RegionOwner* thread to update the HC region state in it's corresponding TLB entry to *Unclassified*. The HC region state transitions at the TLB level are shown on the right in Figure 3.7. Upon receiving the acknowledgment signal, *ACK* from the initial *RegionOwner*, the OS handler assigns the new thread as the *RegionOwner* and the region's PTC is incremented by 1. When a thread that is not the *RegionOwner* issues a subsequent access to this region, and the *isOneOfPageOwners* check returns false or if the region's PTC is already equal to *PTC-thr*, the HC region will transition to either *SRO* or *SRW* based on the type of access. This transition incurs the same cost as performing a *PRV* to *PRV* transition by sending a *TLBUpdate* message to update the TLB entry at the initial *RegionOwner*. HC regions in *SRO* transition to *SRW* upon any write access to the region. This *SRO* to *SRW* transition involves sending a *TLBUpdate* message to all sharers of the page directing them to update HC region state in their respective TLB entries to *SRW*. The accurate list of sharers of each page is maintained by DiDi [85], as discussed in Section 3.1.2.2. A HC region in the *SRO* state will remain in this state as long as all subsequent accesses to this region are reads. All HC regions in the *SRW* state will remain in this state until the end of application runtime. Thus *SRW* is our terminal state.

At the TLB level, any access that hits in the TLB and goes to a HC region in the *Unclassified* state will incur a Page Table lookup by invoking the OS handler to check the corresponding PTE.

This may incur transitions at the PTE following the PT level state transitions described above. The HC region state determined at the PTE will then be installed by the OS handler at the corresponding TLB entry. For TLB misses, the OS TLB miss handler will follow the same process as above by checking the PTE to determine the HC region state to be installed at the TLB entry. A HC region in the *PRV* or *SRW* state at the TLB will remain in the same state upon any reads or writes to it. Similarly, a HC region in the *SRO* state will remain in the same state upon any reads to it. However, any write to a region in the *SRO* state will trigger a PT lookup to inform the OS PT about a write to a region in *SRO*. Following the PTE state transition for any write to a region in *SRO*, the region will transition to *SRW* at both the PT and TLB levels. A HC region present in a TLB entry in any of the four states may receive a *TLBUpdate* message from a OS handler thread executing on a different core. This action is marked in red in Figure 3.7 as *recvTLBUpd* and the *TLBUpdate* message includes a target state to transition the HC region to. We handle these HC region state updates without interrupting the instruction stream by leveraging DiDi's TLB shootdown mechanism to insert a memory barrier and update the TLB entry upon completion of the barrier. Moreoever, we only insert memory barriers upon receipt of a *TLBUpdate* message to transition a region that is in *PRV* state to any other state. We need to insert a memory barrier in this case to preserve TSO guarantees as we might have out-of-order writes to this HC region that are already in-flight from the store buffer. These out-of-order writes need to be completed before transitioning the state of this HC region. Reorder-safe accesses are not allowed to be reordered across a memory barrier at the load and store buffers. Upon completion of this memory barrier, DiDi's PTLBI control unit will update the TLB entry with the correct HC region state and send an acknowledgment back through DiDi's structures to the OS handler thread that sent the *TLBUpdate* message. We do not need to insert a barrier for *TLBUpdate* message that transitions a HC region from *Unclassified* or *SRO* to the terminal state *SRW* because writes to a region in the *Unclassified* state already follow TSO

Table 3.1: Modeled System Characteristics

Parameter	Value
Number of Cores	28 (14 per socket)
Processor Frequency	3.3 GHz
Reorder Buffer Size	192 Entries
Load Buffer Size	72 Entries
Store Buffer Size	42 Entries
L1 ITLB Config	Private, 128 Entries, 4-way, 1cycle
L1 DTLB Config	Private, 64 Entries, 4-way, 1 cycle
L2 STLB Config	Shared, 4096 Entries, 2-way, 30 cycle
Page Size	4KB
Cache Block Size	64B
L1I and L1D Cache Config	Private, 32KB, 8-way, 6 cycles
L2 Cache Config	Private, 256KB, 8-way, 12 cycles
L3 Cache Config	Shared, 2.5MB, 20-way, 71 cycles
DRAM Latency	100 cycles

ordering, and there would only have been reads to a region in *SRO* state. All reads and writes to HC regions in the *PRV* state and all reads to HC regions in the *SRO* state at the TLB level are marked as reorder-safe in the load and store buffers, thus allowing them to be executed out-of-order.

3.1.5 Evaluation Methodology

To evaluate the impact of our design choices for HC and to compare the performance of our final HC design with the current state-of-the-art TSO implementation, we implement both HC and End-to-End SC within a modified version of the Sniper multicore simulator [88], [89]. The baseline system that we model in Sniper resembles a two-socket machine with Intel Xeon CPU E5-2695 v3 (Haswell) processors. Each socket has 14 processing cores with a bidirectional ring interconnect. We replace the baseline 4-way 512 entry private STLB with DiDi's 2-way 4096 entry shared second-level TLB to facilitate low-latency TLB updates to HC regions. This modification has a

negligible effect on system performance because the applications on which we evaluate HC have a geomean DTLB hit rate of 99.1%. We validate the memory access latencies of the modeled system against real hardware by microbenchmarking. The baseline TSO implementation we model in sniper is the dynamic scheme of End-to-End SC (SC-dynamic). We disable Simultaneous Multithreading (SMT) and thread migration in Sniper so that each application thread is assigned to a unique processing core and remains there until the end of its execution period. Table 3.1 lists the relevant characteristics of our modeled system. We measure the area and energy overheads of our TLB extensions using CACTI [90].

Our evaluation suite consists of 15 multithreaded applications from PARSEC [69] and NAS [70] benchmark suites. We use the "*simlarge*" input set for PARSEC applications and the "*A*" workload size for NAS applications. We use OpenMP or pthreads versions of these applications and run 16-thread configurations of these multithreaded applications for all our experiments, except for the scalability study shown in Figure 3.10. We simulate the execution from start to completion for all but four (*streamcluster, lu, sp, ua*) applications in our modified version of Sniper with each application's Region of Interest (ROI) simulated in *detailed* mode. We only simulated the first 10 billion instructions in the ROI of the four applications (*streamcluster, lu, sp, ua*) in order to keep simulation times practical (under 3 days per application run).

3.1.6 Experimental Results

Figure 3.8 shows the performance impact of our design choices normalized to the performance of our no-cost oracle classification design that operates at the page granularity (*4096B_NC*) but follows the state transitions described in Section 3.1.4.3. *256B_NC* uses a 256B granularity classification instead of a page granularity classification while still maintaining a no-cost oracle design. By performing the classification at a finer granularity, *256B_NC* yields a geomean performance

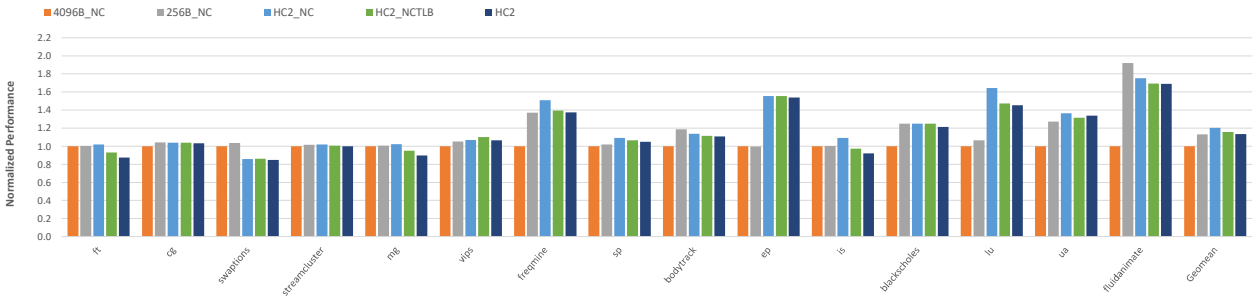


Figure 3.8: Performance impact of HC design choices. *HC2* outperforms a no-cost oracle page-based classification design, *4096B_NC* by a geomean 13% across our applications.

improvement of 11% over *4096B_NC*. *256B_NC* achieves a maximum 92% performance improvement over *4096B_NC* in *fluidanimate*. This is because *fluidanimate* has the highest degree of false sharing of cache lines (91%) when the classification is performed at the page granularity, as shown in Figure 3.2. Thus, by moving to 256B granularity, *256B_NC* reduces the number of mis-classifications of accesses as reorder-unsafe to improve reordering opportunities, which in turn improves system performance by up to 92%.

Beyond performing the classification at the 256B granularity, *HC2_NC* includes our design choice of keeping track of only two region owners per PTE and allowing one re-classification of a HC region, i.e., one *PRV* to *PRV* transition after the initial *Unclassified* to *PRV* transition. Note that *HC2_NC* is still a no-cost oracle implementation. *HC2_NC* attains a geomean performance improvement of 20% over *4096B_NC*. We see a maximum performance improvement of 55% in *ep* and *lu* for *HC2_NC* over *256B_NC*, which corresponds to 55% and 64% improvement over *4096B_NC* for *ep* and *lu* respectively. *swaptions* is the only application for which *HC2_NC* experiences a 15% drop in performance when compared to *4096B_NC*. This performance decrease is due to our choice of keeping track of only two region owners per PTE. We need to keep track of 3 region owners per PTE to capture 100% of private accesses in *swaptions* as shown in Figure 3.4.

All design iterations in Figure 3.8 discussed so far do not model any costs of an actual imple-

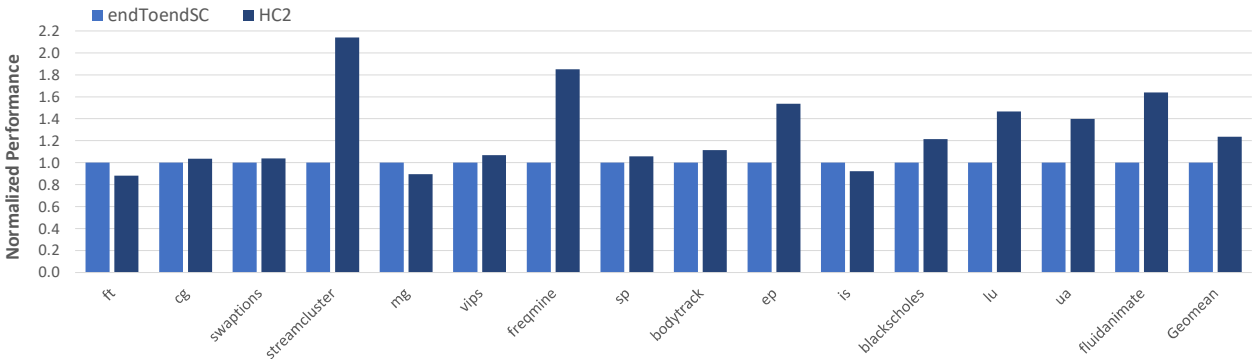


Figure 3.9: Performance of *HC2* normalized to the performance of *endToendSC*. *HC2* outperforms *endToendSC* by a geomean 24% across our applications.

mentation. The next design point, *HC2_NCTLB*, models the transition costs described in Section 3.1.4.3 but assumes that there is no cost to access the DTLB before inserting loads and stores into their respective buffers. Thus, *HC2_NCTLB* is still an oracle classification design. *HC2_NCTLB* attains a geomean performance improvement of 15% over *4096B_NC* even with accounting for the costs to perform HC state transitions.

Finally, our HC design that models all transition costs as well as the DTLB access latency before inserting loads and stores into their respective buffers, *HC2*, achieves a geomean performance improvement of 13% over *4096B_NC*. *HC2* observes a maximum 68% performance improvement over *4096B_NC* in *fluidanimate*. This primarily comes from classification at a finer granularity as discussed above. Additionally, *HC2* observes a performance improvement of 53% and 45% in *ep* and *lu* over *4096B_NC* respectively. Unlike in *fluidanimate*, these performance improvements in *ep* and *lu* do not come from classification at a finer granularity. Rather, they primarily come from enabling eager re-classifications as discussed above. We consider *HC2* as our final HC design that models all our design choices and their associated costs.

Figure 3.9 shows the performance of the current state-of-the-art TSO implementation, *endToendSC*, and *HC2*, normalized to the performance of *endToendSC*. Overall, *HC2* acheives a ge-

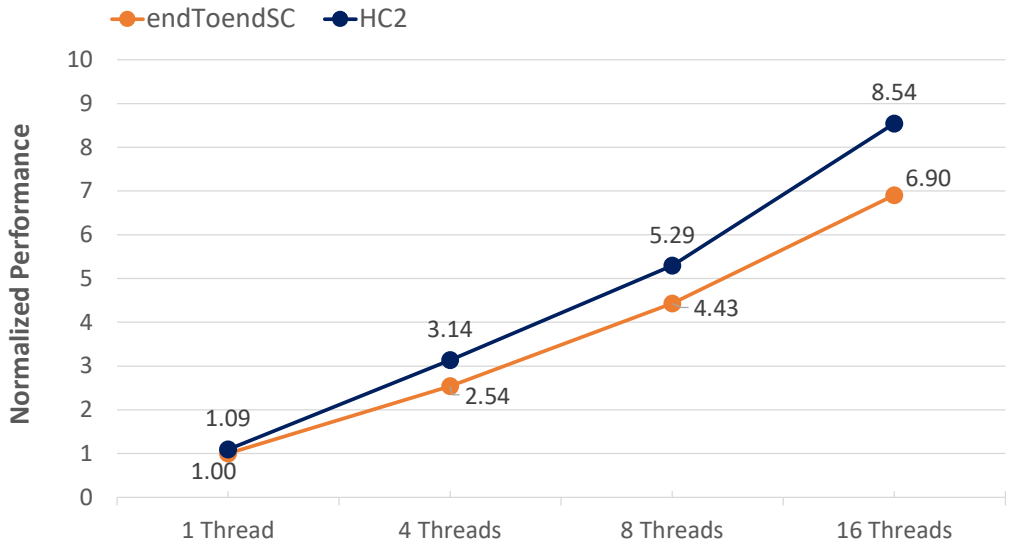


Figure 3.10: Performance scalability of *endToendSC* and *HC2*. *HC2* maintains a consistent 20-24% performance lead over *endToendSC* while increasing application thread counts.

omean 24% performance speedup over *endToendSC*. Unlike *endToendSC*, which does not transition a page present in its private read-write state to its shared read-only state, *HC2* allows a region to transition from *PRV* to *SRO* even when there have been private writes to this region. This optimization allows *HC2* to attain a performance speedup of 113% over *endToendSC* for *streamcluster*. Moreover, by performing the access classification at a finer granularity than pages and by allowing eager re-classifications, *HC2* identifies a geomean 73% of all memory accesses as reorder-safe while *endToendSC* is only able to identify 39% of the same as reorder-safe. Due to this, *HC2* outperforms *endToendSC* by over 10% in more than half of our evaluation suite (8 out of 15 applications). Furthermore, *HC2* achieves more than 40% performance speedup over *endToendSC* for *streamcluster*, *freqmine*, *ep*, *lu*, *ua*, and *fluidanimate*.

Figure 3.10 shows the geomean performance of *endToendSC* and *HC2* across our application suite with increasing application thread counts normalized to the geomean performance of single-threaded *endToendSC*. *HC2* with 16 threads shows a speedup of 8.54X over single-threaded

endToendSC, while *endToendSC* with 16 threads only achieves a 6.90X speedup over the same. *HC2* outperforms *endToendSC* and maintains its 20% to 24% performance lead over *endToendSC* while increasing thread counts to 4, 8, and 16. Thus, *HC2*'s maintains a consistent performance improvement over *endToendSC* while scaling application thread counts.

Based on our CACTI [90] model, the HC TLB extensions require an additional 32 bits per TLB entry, increasing the die area of a typical 4-way 64 entry DTLB by 32%, from 0.015 mm^2 to 0.019 mm^2. Additionally, the energy per DTLB access increases negligibly by 2%. As discussed in Section 3.1.3, a naive HC design that performs the classification at the cache line granularity would achieve the same performance as *HC2* but would require an additional 448 bits per DTLB entry. This would increase the DTLB die area by 196% to 0.044 mm^2. The energy per DTLB access would also increase by 50%. This is clearly impractical to implement. It is important to note that page granularity classification schemes such as *endToendSC* require only 2 bits per TLB entry and incur negligible area and energy overheads. In comparison to *endToendSC*, HC trades off a 32% increase in DTLB area and a negligible increase in energy in favor of improving performance by up to 113%. HC relies on DiDi for its *TLBUpdate* messages during its state transitions that are described in Section 3.1.4.3. DiDi's shared second-level TLB structure [85] has a total size of around 4KB and would require a die area of 0.145 mm^2. Additionally, this structure has a power consumption of 13.5mW. Overall, HC's area overhead including DTLB extensions and DiDi's structures is around 0.677 mm^2 for the system we model in Sniper. This amounts to less than a 0.001% increase in die area for a two-socket system with Intel Xeon CPU E5-2695 v3 (Haswell) processors.

3.1.7 Related Work

There is a substantial amount of prior work on efficient hardware designs with speculation support to provide strong MCM gurantees [72], [75]–[79], [81], [82]. In-window speculation [72] allows loads to speculatively execute out-of-order with a moderately complex design that keeps track of the speculative state only within the ROB window. More sophisticated out-of-window speculation approaches [75], [77]–[79], [81], [82] achieve higher performance than in-window speculation by keeping track of more speculative state information and employing complex misspeculation detection and recovery mechanisms. However, these approaches are too complex to be realized in today's processors, unlike HC.

There is recent interest in non-speculative memory access reordering at the load and store buffers [91], [92]. Ros and Kaxiras [91] propose to non-speculatively coalesce stores in the store buffer by performing them in atomic groups by forcing a new lexicographical order for them. While the stores in atomic groups follow a global order, HC's reorder-safe stores can perform in any order with respect to any other reorder-safe or reorder-unsafe store. Ros and Kaxiras [92] also recently proposed a non-speculative load reordering mechanism that leverages the coherence protocol and intentionally delays conflicting stores in the store buffer to hide any load reordering done by another core from being observed by the core issuing the conflicting store. This allows them to do out-of-order commits for reordered loads without needing to wait for the reordered loads to become non-speculative. However, unlike HC, their approach modifies the coherence protocol, which increases implementation complexity. Ros and Kaxiras [93] later show that non-speculative load reordering can also be accomplished without having a load queue entirely. However, unlike HC, this approach also makes important changes in the coherence protocol.

Some non-speculative approaches that provide strong MCM gurantees [73], [74] maintain low complexity by employing hardware-software co-designs that conservatively reorder memory ac-

cesses by performing a data-based classification [73] or by leveraging programming model guarantees. [74].

End-to-End SC [73], as discussed in Section 3.1.2, employs both a dynamic OS-based classification and a static compile-time classification that complements its dynamic scheme. Our work is inspired by End-to-End SC's OS-based dynamic classification scheme. In Section 3.1.6, we show that by classifying at a finer granularity and by allowing eager re-classifications, HC outperforms End-to-End SC's dynamic classification scheme for a TSO implementation by a geomean 24%. End-to-End SC's compile-time scheme only adds a 0.5% performance improvement on top of it's dynamic scheme. Moreover, End-to-End SC's design incorporates two store buffers, one for out-of-order stores, and one for in-order stores. Implementing two store buffers adds complexity and could lead to under-utilization and stalls in one of the two buffers.

ROOW [74] exploits the Data-Race-Free (DRF) semantics of the code with a compiler implementation that performs region-based classification by delineating DRF code regions from synchronization regions (sync regions). Their compiler sets a 1-bit DRF flag using a dedicated instruction to mark the start of DRF regions. Stores that are inserted into the store buffer while this DRF flag is set are allowed to execute out-of-order while stores in the sync regions execute in-order. ROOW also performs compile-time alias analysis to conservatively prevent reordering across DRF and sync regions when there could be accesses that go the same memory location in both regions. HC's load and store buffer design is inspired by ROOW's monolithic buffer that switches between out-of-order and in-order execution for its DRF and sync regions respectively. Thus ROOW marks entire regions as safe or unsafe to reorder with a single bit in the monolithic store buffer. In contrast to ROOW, HC delineates 256B memory regions dynamically to allow a finer grain mixing of reorder-safe and reorder-unsafe memory accesses. We believe a design that employs both ROOW and HC working together can further improve system performance by

allowing ROOW's region-based classification to complement HC's dynamic classification.

Classification of memory accesses or memory regions has been widely researched for optimizing the cache coherence protocol [94]–[104]. Compile-time approaches [97], [101] are conservative in nature due to their reliance on static-time alias analyisis to classify memory accesses. Approaches based on programming language properties [96], [103] are very accurate but cannot be applied generically to all existing codes. Unlike HC, Coherence-directory based classification mechanisms [98], [100] are not applicable to reorder accesses at the load and store buffer level because their classification is performed later in the memory pipeline, usually at the Last Level Cache (LLC). Similar to HC, TLB-based classification mechanisms [95], [102], [104] account for temporarily-private memory regions. However, unlike HC, they still perform their classification at the page granularity which incurs a high degree of false sharing while identifying reorder-safe memory access as shown in Figure 3.2 which affects system performance as shown in Figure 3.8.

3.1.8 Conclusions

Closing the performance gap between strong and weak memory consistency models is important to improve the performance of modern multiprocessors. We can close this gap by enforcing strong MCM ordering only when necessary; by allowing memory accesses to execute out-of-order from the processor's load and store buffers when their reordering does not affect observable program behavior. In this work, we present Hybrid Consistency (HC), an efficient hardware design that blends strong and weak MCMs by enabling a fine grained mixing of reorder-safe and reorder-unsafe accesses at the load and store buffers. HC relies on existing OS structures for a low-complexity design with minimal area overhead and negligible energy overhead. We show that HC's dynamic temporality-aware classification technique attains all the performance benefits of memory reordering with a cache line granularity classification without having to monitor the classification state

for individual cache lines at the TLB and Page Table level. Our design choices enable HC to detect 73% of all memory accesses as reorder-safe, which is significantly higher than the 39% identified by the previous OS-based dynamic classification approach for memory reordering. Finally, we show that HC outperforms the prior approach by a geomean 24% across a suite of 15 multithreaded applications.

CHAPTER 4

ENABLING IN-COMPUTE PARALLELISM FOR GPU ENERGY EFFICIENCY

In chapters 2 and 3, I targeted improving system performance. Besides performance, chip designers also need to make energy-efficient chips in order to meet the ever-growing performance targets while staying within a reasonable power budget. In this chapter I target directly improving energy-efficiency with ST^2 GPU in Section 4.1. To evaluate ST^2 GPU and other such hardware advancements, architects need robust tools that allows them to quickly and accurately model both the power consumption and performance of modern systems. As such, in Section 4.2, I present AccelWattch, a cycle-accurate power model for modern GPUs. I use AccelWattch to evaluate ST^2 GPU in Section 4.1.6.

4.1 ST^2 GPU: An Energy-Efficient GPU Design with Spatio-Temporal Shared-Thread Speculative Adders[1]

4.1.1 Introduction

Graphics Processing Units (GPUs) are becoming increasingly popular for accelerating both general-purpose and high-performance computing applications. Currently, there are 152 GPU-accelerated systems in the TOP500 HPC list [19] and 70% of the top-50 HPC applications are GPU-accelerated [20]. As the appeal of GPUs grows, so does the demand for higher performance. To meet the ever-increasing performance targets, designers cram increasingly more cores per GPU chip, leading to a commensurate rise in power consumption. However, the power budget of modern GPUs is

[1]This section is based on our DAC'21 paper about ST^2 GPU [105].

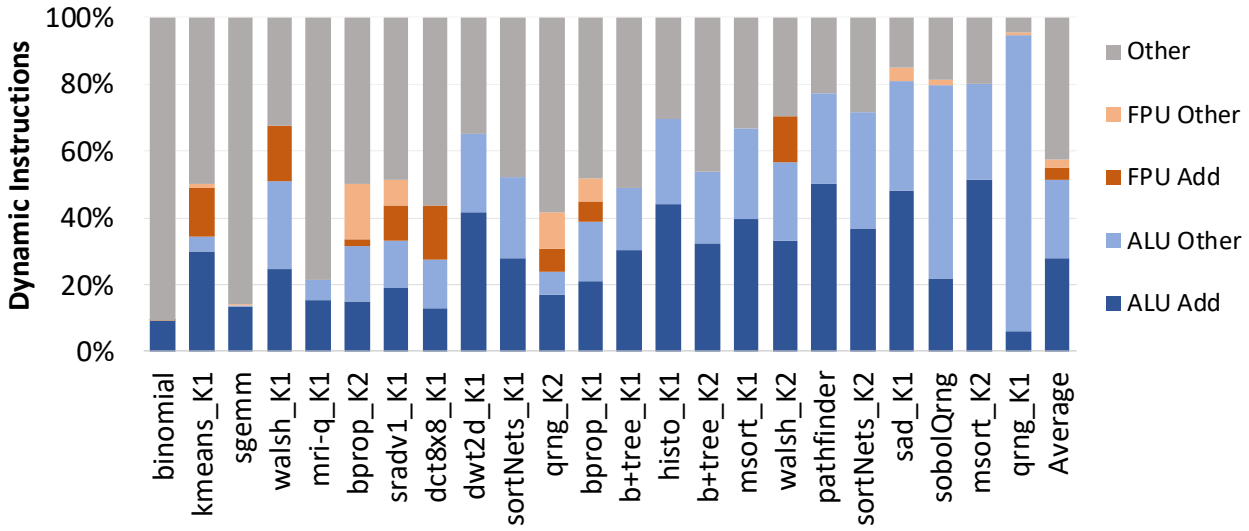


Figure 4.1: ALU and FPU operations are prevalent in GPU kernels.

already reaching the limits of practical cooling technology. For example, both NVIDIA's Volta GV100 architecture [22] and the previous-generation Pascal GP100 are limited by the same 250 W thermal design power, even though GV100 contains 43% more CUDA cores. In order to continue increasing the core count at a constant power budget, the cores must become more energy efficient. Owing to their sheer number on a chip, add/subtract execution units such as integer arithmetic and logic units (ALUs) and floating-point units (FPUs) are collectively among the most power-hungry hardware components. Often, ALUs and FPUs are exercised intensely by workloads: 21 out of 23 kernels from Rodinia [106], NVIDIA CUDA Samples [107], and Parboil [108] running on an NVIDIA TITAN V Volta exhibit high arithmetic intensity, i.e., more than 20% of the executed dynamic instructions are ALU and FPU instructions (Figure 4.1). In this work, we directly target a reduction in ALU and FPU power consumption by introducing a new power-efficient adder design, and an associated GPU architecture.

Our work is inspired by the observation that real-world GPU applications exhibit an important but overlooked behavior: the computed values of consecutive operations from the same line of

code are often highly correlated. (i.e., the values computed by the same instruction as it repeatedly executes, tend to be of similar magnitude). We capitalize on this observation and propose Spatio-Temporal Shared-Thread (ST^2) adders, a power-efficient speculative adder design that utilizes the spatio-temporal history of arithmetic operations in a GPU kernel to perform additions. While the adder executes speculatively, mispredictions are immediately detected upon the nominal end of the adder's execution and corrected in subsequent cycles. Thus, ST^2 adders guarantee correctness. At the same time, they save 70% of the adder power and achieve 27% higher prediction accuracy over the current state-of-the-art VaLHALLA [109] design.

We incorporate ST^2 adders into a new GPU architecture, ST^2 GPU. ST^2 GPU modifies the pipeline of an NVIDIA Volta GV100 to accommodate the variable-delay adders, and facilitates access to history tables by piggy-backing on the GPU's operand collector. The ST^2 GPU design achieves a 21% chip energy reduction across 23 kernels from Rodinia [106], NVIDIA CUDA Samples [107], and Parboil [108], with practically no performance and area overheads.

This work makes the following contributions:

- We observe, explain and quantify spatio-temporal value correlation on real-world GPU applications.
- We propose ST^2 adders, a speculative adder design that exploits spatio-temporal value correlation to perform carry speculation and reduce power consumption. ST^2 adders guarantee correctness and outperform state-of-the-art designs.
- We perform a design-space exploration of carry speculation units on GPUs along the spatial axis (PC correlation), temporal axis (history depth), and history sharing among threads, and arrive at a practical, high-performance carry speculation unit for GPUs.
- We propose ST^2 GPU, an architecture that integrates ST^2 adders and carry speculation units

into the warp pipeline, and show it achieves significant power savings with negligible overheads.

4.1.2 Background

4.1.2 Volta Architecture and Execution Model

Our GPU architecture model is inspired by the NVIDIA Volta GV100 GPU architecture [22], and particularly the TITAN V Volta. The TITAN V Volta has 80 Streaming Multiprocessors (SMs) each with 64 32-bit integer units (ALUs), 64 32-bit floating-point units (FPUs), 32 64-bit double-precision units (DPUs), 4 special function units (SFUs) for complex operations (e.g., log, square root), and 8 tensor cores for matrix arithmetic. Our design targets the adders within the ALUs, FPUs and DPUs.

GPUs execute programs known as "kernels", which typically comprise thousands of threads. Upon launching a kernel, each thread gets its own GPU-wide unique *global* thread ID. Threads do not execute instructions independently; rather, sets of 32 threads (warps) execute the same instruction on different data. Each thread in a warp is identified by its *local* thread ID, i.e., a number between 0–31. In the rest of the section, we refer to these *global* and *local* thread IDs.

GPU kernels are offloaded to the the GPU device through the use of CUDA, a parallel computing platform and application programming interface model. The CUDA programming environment provides a parallel thread execution (PTX) [110] instruction set architecture (ISA), which is an intermediate ISA that exposes the GPU as a data-parallel computing device. PTX programs are translated at install time to the target hardware ISA that executes natively on the GPU.

4.1.2 Speculative Adders

To reduce power consumption, speculative adders divide a regular adder's full bit range into smaller bit ranges ("slices") and run them in parallel. As slices are smaller, they can execute at a fraction of the nominal clock period. Speculative adders exploit the unused clock period to scale down each slice's supply voltage to the lowest setting that allows the slice to still fit within the same cycle time [111], gaining quadratic power savings. However, running the slices in parallel breaks the carry-propagation chain. Speculative adders overcome this obstacle by speculating on the carry-in of each slice.

Approximate speculative adders [112]–[115] do not possess error correction mechanisms and wrong results are supplied whenever a carry-in is mispredicted. In contrast, variable latency speculative adders [109], [116] detect mispredictions at the end of the nominal execution and occupy additional execution cycles to recompute with the corrected carry-in if an error occurred. Thus, they always provide the correct result, but incur an overhead whenever a misprediction occurs. Our design is inspired by VaLHALLA [109], a recently-proposed variable-latency adder that is shown to outperform prior speculative adder designs. VaLHALLA provides a static prediction for all slices' carry-ins based on the correlation between the length of the carry propagation chain and the input operands.

4.1.3 Spatio-Temporal Value Correlation in GPUs

A characteristic of applications is that code execution repeats, both within threads of computation (e.g., in loops) and across threads (e.g., the same kernel running on separate threads). Thus, the same instructions, at the same PC, repeat, one iteration after another and one kernel thread after another. As these "hot" instructions operate in succession they transform data. While the data values produced by different instructions often bear limited correlation with each other, instructions

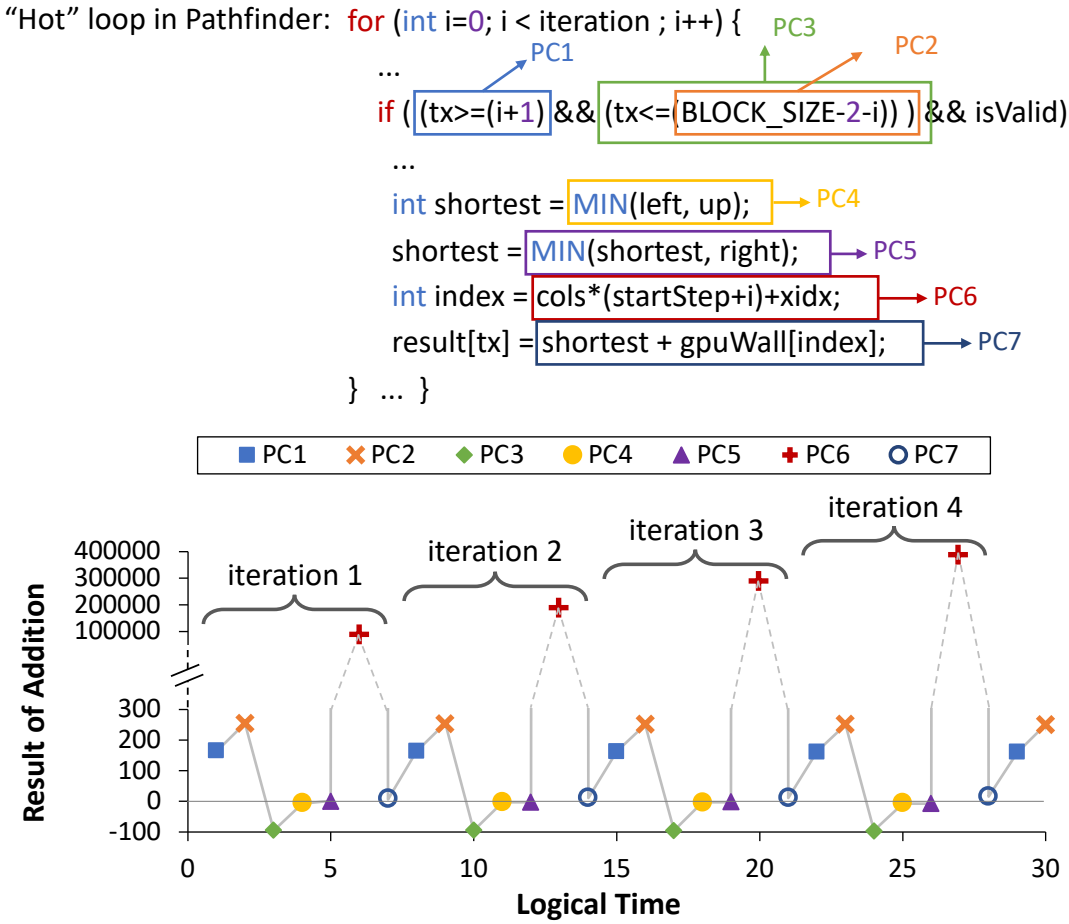


Figure 4.2: Value evolution of addition results from the Pathfinder kernel.

at the same PC often operate on arguments of similar magnitude and produce values similar to the ones the same instruction produced in the previous invocation. For example, the same instruction that increments the iterator of a loop will keep executing, repeatedly producing a sequence of nearby values (e.g., 1, 2, 3). Another instruction in the body of the loop may operate on other data and produce new values. As that instruction repeats, it produces values that tend to be within similar magnitudes across a short window of time, gradually evolving rather than wildly fluctuating across the integer or floating-point range. In short, code repetition gives rise to value correlation.

Figure 4.2 shows a real-world example of value correlation. The code snippet on Figure 4.2

is the hot loop of the kernel in *pathfinder* from the Rodinia benchmark suite (Section 4.1.5.1). We highlight the loop's addition operations and mark them with their logical PC, ranging from PC1 to PC7. As these additions execute and operate on the application's data, they produce new values. Figure 4.2 (bottom) shows the evolution of these values in logical time (i.e., in the order of instruction execution). Control and data dependencies force the instructions at PC1, PC2, PC3, PC4, PC5 and PC7 to execute in this exact order, while PC6 is ordered between PC3 and PC7.

As Figure 4.2 shows, when observed as a whole the values generated by these additions as they execute in order vary greatly. The exist values in the 100s (PC1, PC2), around 0 (PC4, PC5, PC7), and even tens of thousands (PC6) or negative (PC3). While there is some correlation in the magnitude of the results from different instructions that execute consecutively (e.g., both PC4 and PC5 produce values close to zero), this correlation is weak and is often broken by instructions producing wildly different results. However, the values produced by the *same* instruction (i.e., at the same PC) across iterations are of *similar magnitude* and strongly correlated.

This value correlation translates to correlation in the carry chains. Operations on small positive numbers yield short carry chains, (e.g., PC1, PC2, and PC7 which produce carry chains that do not propagate beyond the first 8 bits), while instructions producing larger values produce longer carry chains (e.g., PC6's results may produce a carry that propagates through the first 16 bits). Additions producing negative results (e.g., PC3) may produce carry chains that propagate all the way to bit 63. We observe that while the carry chain length is weakly correlated across different instructions, it is strongly correlated across subsequent executions (temporal correlation) of the same instruction (spatial correlation).

We quantify this spatio-temporal value correlation in our workload suite in Figure 4.3. We envision additions performed not in a monolithic 64-bit adder, but rather by stringing together 8-bit adder slices, each fed with the carry out of the previous 8-bit slice. We compare the carry-

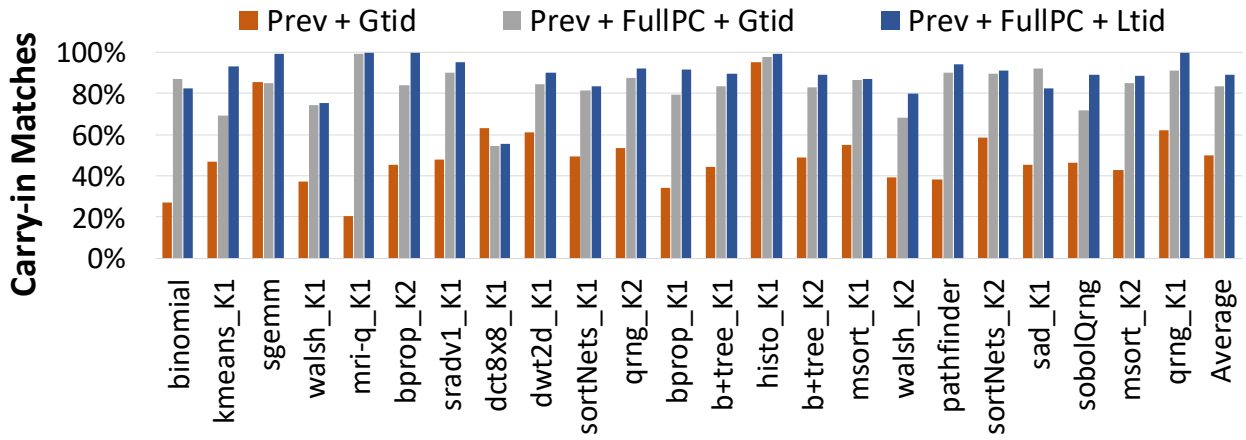


Figure 4.3: 8-bit slice carry-in correlation across the temporal & spatial axes.

outs/carry-ins between adder slices as instructions execute. When we compare the carry-ins between consecutive additions executed within each thread (same global threadID), regardless of the PC, only 50% match on average (Prev+Gtid). Thus, there is practically no correlation along the temporal axis alone. However, when we compare separately the slice carry-ins from consecutive executions of the *same* PC within each thread, we find matches in 83% of the cases on average (Prev+FullPC+Gtid). Thus, while temporal correlation alone is limited, the spatio-temporal correlation is strong. In addition, as all threads in a GPU kernel's block execute the same program, they can learn from each other. When we compare not against the previous execution of an instruction at the same PC by the same thread (same global threadID), but across all threads in the same warp lane (between 0 and 31), then matches are found in 89% of the cases (Prev+FullPC+Ltid), showing that sharing history among threads can enhance the speed of finding correlations. In the following section we capitalize on these observations to design the ST^2 adders.

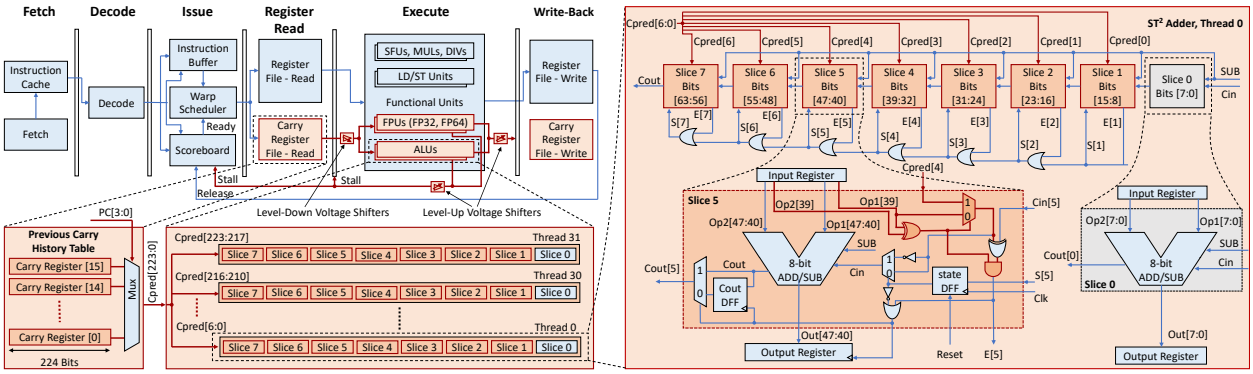


Figure 4.4: Adder slice design. Slices 1-7 are similar. Changes over VaLHALLA are highlighted in red.

4.1.4 ST^2 Design and Space Exploration

4.1.4 ST^2 Adder Slice Design

The ST^2 adder is inspired by VaLHALLA [109] and significantly improves upon it. Figure 4.4 depicts the design of the ST^2 adder, with slices 0 and 5 shown in detail. While The NVIDIA TITAN V Volt GPU has only 32-bit adders, here we show the design of ST^2 for the general case of a 64-bit adder. For simplicity of explanation, let's assume that the nominal latency of an ADD operation is 1 cycle.

At the beginning of an ADD operation, ST^2 makes a prediction of the carry-in for each slice and performs the ADD computation. The prediction is communicated to the adder through signals $Cpred[0] - Cpred[6]$ from the Carry Register File to each adder slice in Figure 4.4 (Section 4.1.4.2 explains how ST^2 makes these predictions). At the end of the nominal execution cycle, each slice compares the prediction it received ($Cpred[4]$ for slice 5) with the carry-out generated by the previous slice ($Cin[5]$ for slice 5). If they do not match then a misprediction has occurred. In that case the slice consumes an additional cycle to re-compute the ADD operation with the inverse carry-in of the previous cycle ($\neg Cpred[4]$ for slice 5). Thus, the execution of an ADD may take one or two cycles, depending on whether there was a misprediction.

If slice i mispredicts, then all carry-outs generated by slices $i+1, ..., 7$ are suspect of being incorrect. Thus, upon a misprediction, an error signal is generated ($E[5]$ for slice 5) that propagates to all higher-order slices (signals $S[i]$) and informs them that they may have received an erroneous carry from their previous slices. Each of the affected slices will then proceed with a second cycle of computation, using the inverse carry than the one assumed in the previous cycle. A 1-bit State DFF register keeps track of whether the slice is performing the first or the second cycle of computation. At the beginning of an ADD operation, all State DFFs are reset to 0. At the end of the first cycle, each slice's State DFF is updated by OR-ing the error signals of the current and all previous slices, and then stays at that value until a new operation is assigned to the adder. Thus, the State DFF remembers whether the predicted carry is to be trusted or not. At the end of this second cycle all correct carry-ins are known, and each ST^2 slice decides to either keep the results already in its output register (i.e., the first cycle's computation was the correct one) or overwrite them with the result of the second cycle. This operation is similar to a Carry Select Adder (CSLA) [117]. Unlike CSLA, though, which always performs computations with both carry-ins for all slices, ST^2 performs additional slice computations only when a misprediction occurs, and only on the subset of slices that cannot trust their prediction. Thus, ST^2 avoids unnecessary computations and exhibits significant power savings over CSLA.

4.1.4 ST^2 Carry Speculation Mechanism and Comparison to VaLHALLA

ST^2 improves upon VaLHALLA by offering improved carry speculation. Specifically, it employs speculation only when necessary, provides per-thread history-based predictions, promotes thread-history sharing, and is adapted to GPU pipelines. We arrive at the ST^2 architecture by performing a design space exploration, shown in Figure 4.5. As the figure shows, static carry prediction (e.g., always predict 0—*staticZero*) suffers from high error rates (*staticOne* is even worse). VaLHALLA

reduces the misprediction rate through dynamic speculation. However, dynamic speculation is not always necessary. If the most significant bits (MSbs) of the two input operands of the previous slice (Op1[39] and Op2[39] for slice 5) are both zeros, then the carry-in will surely be zero; if they are both ones, then the carry-in will surely be one. Such static predictions are guaranteed to be correct. ST^2 capitalizes on this observation by having each slice *peek* at the MSbs of the previous slice to make a static prediction, and relies on dynamic speculation (and risks errors) only when static predictions are not possible. VaLHALLA always performs dynamic speculation even in these cases. Retrofitting VaLHALLA with *Peek* further reduces its misprediction rate by 18%.

The second limitation of VaLHALLA is that it predicts a single 1-bit carry for the entire adder, which is broadcasted to all slices. Providing the same prediction to all slices increases the misprediction rate and causes additional execution cycles. Instead, ST^2 makes separate carry-in predictions for each slice by observing that consecutive arithmetic operations are correlated (Section 4.1.3). Correlation increases the likelihood that carry chains have similar lengths among operations executed close in time. Thus, ST^2 remembers, in a *previous carry* history table, the carry-outs produced by each slice for an ADD at time i, and uses them as per-slice predictions for the carry-ins at time $i + 1$. The corresponding *Prev+Peek* design puts everything together and achieves a 26% reduction in miss rate over VaLHALLA.

The achieved 20% misprediction rate may still be relatively high, though, as each misprediction means the ALU will consume additional cycles, raising the probability of structural hazards. *Prev+Peek* fails to achieve very low misprediction rates because it allows all instructions to alias with each other. While consecutive executions of the same instruction may be highly correlated and produce similar output values (Section 4.1.3), executions of different instructions are less likely to correlate. To disambiguate predictions, ST^2 employs a number of PC bits as part of the index into the previous carry history table by using the lowest k bits of the PC as index to the previous carry

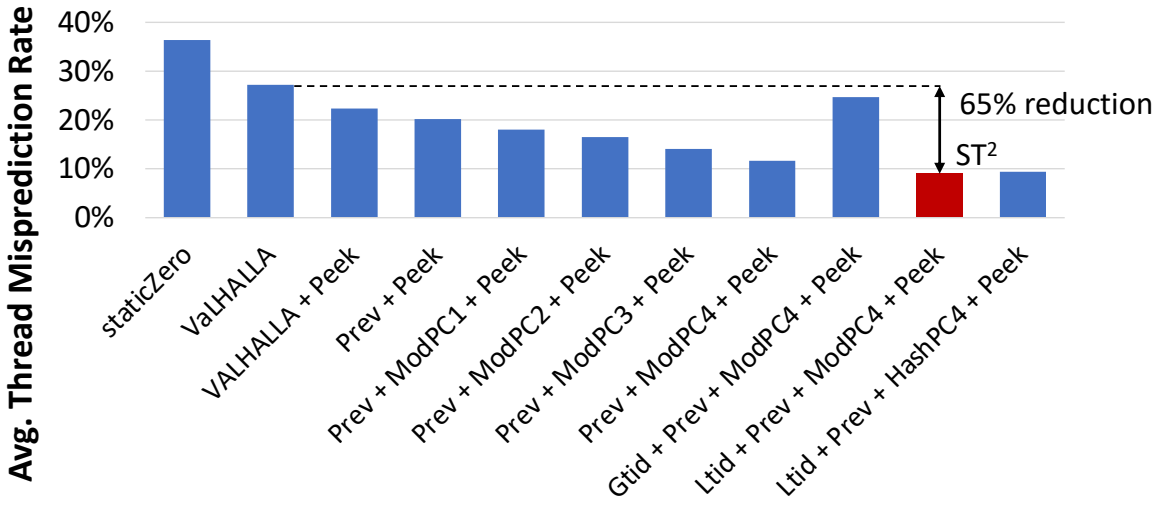


Figure 4.5: Design space exploration for ST^2 carry speculation mechanism.

history table (*ModPC*k). Figure 4.5 shows that as k increases, the misprediction rate falls. At 4 PC bits the misprediction rate is just 12%, a full 57% lower than VaLHALLA. Increasing k further provides diminishing returns.

We speculate that the main limitation of the latest design we arrived at is thread aliasing. While the history from different instructions is disambiguated, all threads in *Prev+ModPC4+Peek* share the same history, which may cause undesirable destructive interference. Moreover, there may be significant thread contention, as all threads that execute the same (or aliasing) instructions may simultaneously attempt to update the same entry in the carry history table. To address these problems, we add to *Prev+ModPC4+Peek* the ability to disambiguate threads by adding the global thread ID as part of the carry history table index. The resulting *Gtid+Prev+ModPC4+Peek* design completely disambiguates threads. As we see in Figure 4.5, however, this design fares significantly worse than most other designs. Thus, sharing history among threads may be beneficial, indicating that interference may be constructive as well. By having threads operate on similar data, they can act as "prefetchers" of the correct carry-ins into the history table. Our data indirectly support the

hypothesis that constructive interference among threads may be more prevalent.

Armed with this new intuition, we modify the design to incorporate the local thread ID instead of the global ID in the carry speculation table index, thereby allowing threads to share predictions across warps. The resulting design, *Ltid+Prev+ModPC4+Peek*, achieves a small 9% misprediction error (Figure 4.5), which is 65% lower than VaLHALLA's. Volta GV100 supports up to 2048 threads/SM [22], thus *Gtid+Prev+ModPC4+Peek* may require a 15-bit history table index (11 global thread ID bits + 4 PC bits) and a commensurately large history table. In contrast, *Ltid+Prev+ModPC4+Peek* exhibits minimal contention that can be practically addressed with random arbitration, as only the few warps executing the register write-back pipeline stage at the exact same cycle in the same SM's computational cluster may conflict with each other, and only when threads within these warps mispredict. The speculative adder design we pick for ST^2 is *Ltid+Prev+ModPC4+Peek*. More complex PC-based indexing (e.g., XOR-hash of 4-bit PC chunks) provide no additional benefits.

The final ST^2 design is in stark contrast to VaLHALLA. VaLHALLA predicts the same carry-in for all slices, while ST^2 predicts an independent carry-in per slice based on instruction history (the *Prev* mechanism). VaLHALLA performs a prediction on every ADD, while ST^2 predicts only when necessary (the *Peek* mechanism). VaLHALLA predictions are performed for each adder, while ST^2 allows history sharing across threads. These improvements result in significantly higher prediction accuracy for ST^2 over VaLHALLA.

It is important to note that configurations to the left of *Ltid+Prev+ModPC4+Peek* in Figure 4.5 would be unimplementable due to numerous hardware threads requiring simultaneous read/write accesses to the same carry history table entry. This design space exploration shows that our ST^2 adder design which uses *Ltid+Prev+ModPC4+Peek* exhibits lower misprediction rates than even these optimistic approaches (including VaLHALLA) shown in Figure 4.5 which ignore contention

in accesses to the same history table entry by multiple hardware threads in an SM.

4.1.4 ST^2 GPU Microarchitecture

Figure 4.4 shows a model of a modern GPU warp pipeline with the proposed modifications to support ST^2. As ST^2 operates at lower-than-nominal voltage, level shifters are required when crossing voltage domains. A Carry Register File (CRF), placed next to the regular register file, holds the per-slice carry-outs produced by previous add operations in a history table. The CRF is read along with operands from the register file in the register read pipeline stage. The speculated carry-ins from CRF are sent to the warp's Functional Units (FUs)—i.e., adder or FMA units in ALUs, FPUs or DPUs, depending on the operation—during the execute stage together with the operands, and are utilized by ST^2 adders to perform the computation. The CRF is structured as a 16×224-bit register file. A CRF read uses PC[3:0] as an index to retrieve 224 bits. These correspond to 7 carry bit predictions (one for each of $slice_1, ..., slice_7$) of each of the warp's 32 threads. Upon completion of the operation, threads with mispredictions update their corresponding bits in the CRF with the new carry-outs, to be used as predictions in subsequent operations. The CRF is updated at the write-back pipeline stage along with the register write-back, similarly to a register file update.

When an FU detects a misprediction, the operation is repeated with the inverse carry-ins to recover from the error (Section 4.1.4.2). Upon a misprediction, the FU generates a stall signal that propagates to the scoreboard to prevent the issue of another instruction on the still-occupied functional unit, and stalls the pipeline register to prevent instructions already scheduled from getting to the execute stage.

ST^2 GPU employs ST^2 adders not only in integer ALUs, but in FPUs and DPUs as well when performing mantissa operations. Mantissas are 23 or 52 bits for FP32 and FP64, so these units

use 3 or 7 slices, respectively. We refrain from employing speculative adders for exponent operations (exponents are only 8-11 bits wide and speculation does not provide any benefit) or in other complex units such as multipliers.

4.1.5 ST^2 GPU Evaluation Methodology

We use GPGPU-Sim 3.x [118] in PTX simulation mode, which is calibrated against an NVIDIA TITAN V Volta and shown to have high correlation with hardware performance measurements [119]. We use this version as the baseline and modify it to incorporate the ST^2 GPU architecture. Our simulator models ALUs, FPUs and DPUs as seperate components that perform adds, subtracts, and a host of other simpler operations. Multipliers are modeled as separate units.

4.1.5 Workloads

Our evaluation suite consists of 23 kernels selected from 18 workloads from NVIDIA CUDA Samples [107] (cudaTensorCoreGemm, BinomialOptions, fastWalshTransform, dct8x8, sortingNetworks, quasirandomGenerator, histogram, mergesort, and SobolQRNG), Rodinia [106] (kmeans, backprop, sradv1, dwt2d, b+tree, and pathfinder), and Parboil [108] (sgemm, mri-q, and sad). All workloads were compiled with NVCC V9.1.85 with support for Volta using the *arch=sm_70* compiler option. We excluded workloads that could not compile for GPGPU-Sim (e.g., due to unimplemented instructions like *warp.sync*) or were impractical to simulate ($>$ 2 days per run). Additionally, we excluded a few short-running kernels for which we were unable to collect reliable hardware power measurements from our power modeling workflow (it probes the hardware at 50–100 Hz, and as a consequence we could not validate our power model for these workloads against hardware measurements to ensure its accuracy). We use the largest available input configuration for all workload runs.

4.1.5 Circuit Design

We model all adder designs in Verilog. We synthesize all designs with the same optimization parameters using Synopsys Design Compiler (H-2013.03-SP5-4) and Synopsys IC Compiler (vI-2013.12-SP5), using the Synopsys SAED 90 nm library. We simulate the netlists using Synopsys VCS-MX (I-2014.03-2) in analog mode and Synopsys HSpice (K-2015.06-1) to analyze their energy and delay characteristics. The reference adder is the default adder synthesized by Synopsys Design Compiler. It is a state-of-the-art, industrial-strength design directly imported from the Synopsys DesignWare Library [120], and synthesized using the recommended default optimization settings to obtain an overall balanced design. We determine the minimum execution delay of the reference adder when nominal voltage is supplied, and use it to define the nominal clock period. Then, we identify the voltage at which we can scale the slices while still fitting within the nominal clock period. From this characterization we extract the reference adder and slice power consumption we use in our modeling. While the circuit modeling is performed with a 90 nm cell library, we estimate that the relative energy differences across adder designs will persist when we scale the designs to the 12 nm FinFET process that NVIDIA Titan V Volta uses.

We perform a design space exploration to identify the optimal ST^2 slice bitwidth. We synthesize sub-adders of different bitwidths, feed them with random vectors as inputs, and evaluate their power consumption on the same random input sequence. We identify 8-bit slices as the best design option for ST^2, as they allow the supply voltage to scale to 60% of the reference voltage, leading to 75–87% potential energy savings per adder. We model the energy and area footprint of ST^2's carry speculation unit independently.

4.1.5 Power Modeling

To evaluate power consumption, we develop a power model and extensively validate it before collecting power results for ST^2 GPU over the baseline. We use an earlier version of AccelWattch [121] (Section 4.2) that was calibrated using a set of micro-benchmarks and an NVIDIA TITAN V Volta GPU. We develop a suite of 123 micro-benchmarks that isolate and stress specific GPU hardware components. We run these kernels on silicon to collect hardware power measurements using the NVIDIA Management Library at 50–100 Hz. We then use a least-square-error solver to calibrate the AccelWattch power scaling factors per component.

At a high level, the power model used for evaluating ST^2 GPU is represented by:

$$P_{\text{total}} = P_{\text{const}} + (N_{\text{idleSM}} \times P_{\text{idleSM}}) + \sum_{i=1}^{N} (P_{\text{i}} \times Scale_{\text{i}}) \tag{4.1}$$

The constant power P_{const} includes power from components such as GPU board fans, power regulators, peripheral circuitry, and leakage. P_{idleSM} models static power per idle SM, and is multiplied by the number of idle SMs, N_{idleSM}. We model the dynamic power of each component i by multiplying the component's scaling factor $Scale_i$ estimated by the solver, with the component power we obtain from our AccelWattch simulations, P_i. To estimate the total modeled system power, we sum the dynamic power of each component with the chip's constant power and the total idle SM power. P_{const} and P_{idleSM} are also estimated by our solver across all microbenchmarks.

The NVIDIA Volta GV100 does not have divider units. Instead, divisions are performed in hardware as an algorithm that uses other instructions (e.g., FMAs, shifts, etc). However, the PTX ISA includes division instructions, which subsequently are modeled by GPGPU-Sim. Thus, we also separately model the power consumption of division operations; this power does not correspond to a single hardware component, but rather to the collective instructions that execute on hardware to calculate the result of a division operation.

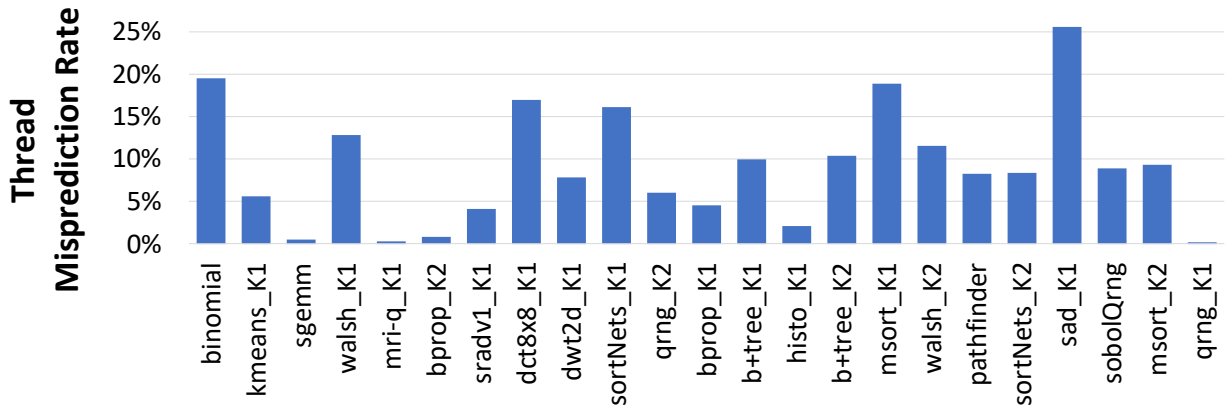


Figure 4.6: Thread misprediction rate for ST^2 adders.

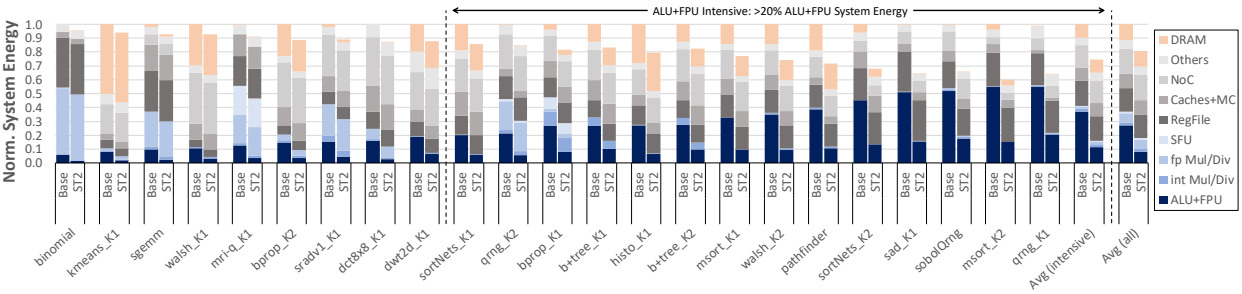


Figure 4.7: Normalized system energy for the baseline and ST^2 GPU architectures.

We evaluate the accuracy of our power model by applying it on our benchmark suite of 23 kernels. The model was trained on the microbenchmark stressors only, thus our benchmark suite constitutes a proper validation set. We find that our power model attains an average absolute relative error of $10.5\% \pm 3.8\%$ (95% confidence interval), giving it a strong Pearson r coefficient of 0.8.

4.1.6 Evaluation

We evaluate our design by running the kernels in our benchmark suite on a simulated ST^2 GPU architecture using our in-house versions of GPGPU-Sim and AccelWattch that incorporate our adder and warp pipeline designs and the power model. ST^2 GPU attains a 9% average thread misprediction rate across all kernels (Figure 4.6).

We calculate that on average across all kernels, a single thread misprediction causes 1.94 slices to re-compute their results (up to 2.73). Thus, mispredictions do not cause an excessive energy penalty.

Figure 4.7 shows the energy breakdown in our suite when running on a simulated TITAN V Volta baseline and on ST^2 GPU. On average the baseline spends 27% of the total system energy on ALUs/FPUs, which corresponds to 30% of the chip energy (excluding DRAM). Kernels such as *qrng_K1* spend as much as 57% of the system energy on ALUs/FPUs. We observe that several of our kernels have high arithmetic intensity, with 14 out of 23 workloads each spending more than 20% of the system energy on ALU/FPU units. These workloads spend on average 31% of the total system energy on ALUs/FPUs, corresponding to an average of 40% of the total chip energy. Thus, minimizing the ALU/FPU energy consumption holds significant promise in increasing the energy efficiency of GPUs.

Figure 4.7 shows that across our kernel suite, ST^2 saves on average 19% of the system energy, which corresponds to 21% average chip energy savings (excluding DRAM). The energy savings for kernels with high arithmetic intensity are even higher: ST^2 GPU saves on average 26% of the system energy (up to 40% for *msort_K2*), which corresponds to 28% chip energy savings (up to 42%).

These energy savings come with practically no performance overhead. A mispredicted carry-in for even one adder slice in a single thread in a warp would stall the entire warp until the correct output is obtained. While the penalty may seem high, the ST^2 speculation mechanism exhibits very low misprediction rates (Section 4.1.4.2), and GPUs are tolerant of such additional latencies. Our experiments show that ST^2 GPU practically provides the same performance as the baseline: the execution time is within 0.36% of the baseline on average. The worst performance impact among our 23 kernels is suffered by *dwt2d_K1*, which shows a still small 3.5% slowdown.

The voltage level shifters required for ST^2 adders to interface the new power domain have negligible area and power overheads. Level shifters in a 45 nm technology can be made at 2.8 μm^2 [122]. We estimate that, using level shifters for each adder's input operands and outputs on the chip, even without scaling from 45 nm to 12 nm FinFETs, these level shifters in total occupy less than 5.5 mm^2, which for a NVIDIA Titan V Volta is 0.68% of the 815 mm^2 chip area. [123] shows that level shifters for 16 nm FinFETs consume 1.38 fJ per transition and 307 nW in static power. Assuming these level shifters in our design, their total static power consumption for a Titan V Volta chip without any scaling to 12 nm FinFETs is only 0.6 W. Under the worst case estimation that every single bit of every instruction that goes through an adder unit flips, thus consuming the maximum amount of energy in the level shifters, the total dynamic power consumption of the level shifters, averaged across all kernels in our suite, is a mere 470 μW. This overestimated level shifter overhead amounts to just a 0.5% penalty in our average system energy savings bringing it down to 18.5%. Finally, the worst-case delay per falling or rising transition for a 500 mV to 790 mV crossing is only 20.8 ps; in our analysis we also consider the additional delay imposed by the level shifters at the inputs and outputs of our ST^2 adders.

Finally, the ST^2 GPU area overhead is negligible, as we illustrate by considering a hypothetical NVIDIA TITAN V Volta with ST^2 GPU. Every SM has a 448-byte CRF (16×224 bits), thus the entire chip requires just 35 kB of total additional area. Moreover, each slice (except 0) has 2 bits for the state and Cout DFFs (Figure 4.4). Thus, each ALU adder requires an additional 14 bits, and FP32 and FP64 adders need 4 and 12 bits, respectively, for the mantissa adders. Overall, the space requirements for the DFFs amount to an additional 15 kB per chip. This brings the ST^2 overhead to a total of 50 kB per chip, which is a mere 0.09% of the on-chip caches and register files.

4.1.7 Related Work

Approximate speculative adders [112]–[115] split execution into multiple slices that run in parallel with predicted carry-ins. However, they do not employ error correction and supply wrong results whenever a carry-in is mispredicted. VLSA [116] speculates on carry-ins, but detects mispredictions and occupies additional cycles to recompute with the corrected carry-in if an error occurs. CASA [115] provides a static prediction for all slices' carry-ins based on the correlation between the input operands. VaLHALLA [109] extends CASA to a variable-latency speculative adder that speculates carry-ins for all operations. In contrast, ST^2 employs speculation only when needed, and introduces novel concepts such as per-thread history-based predictions and thread-history sharing, and is adapted to GPUs.

4.1.8 Conclusions

Just like most modern chips, GPU scaling is hampered by power limitations. We address this problem with *ST^2 GPU*, a GPU architecture that employs history-based speculative adders that produce guaranteed correct results while saving power. We explore the design space of the speculative mechanisms and arrive at an adder design that shows high accuracy (91% on average) and high power savings (70% of the nominal adder power). Overall, ST^2 GPU reduces the energy consumed by a GPU chip by 21% (and chip+DRAM by 19%), with minimal area overhead and practically no performance impact.

4.2 AccelWattch: A Power Modeling Framework for Modern GPUs[2]

4.2.1 Introduction

As the proliferation of GPUs grows to satisfy the demand for higher performance, they are fast becoming a major consumer of power. Thus, it is not surprising that performance per watt, together with peak performance, have emerged as indispensable metrics for evaluating the efficiency of GPU architectures. As such, GPU architects require robust tools that will enable them to quickly and accurately model both the performance and the power consumption of modern GPUs.

However, while GPU performance modeling has progressed in great strides [24], GPU power modeling has lagged. GPUWattch [124] has been an indispensable tool for modeling the power consumption of new innovations in GPU architectures, but it was designed to model (and validated against) older architectures with fewer energy efficiency optimizations.

Attempting to model recent GPUs such as Pascal [23], Volta [22] and Turing [125] using the methodology employed by GPUWattch produces significant inaccuracies, both in terms of absolute numbers and in terms of the relative power consumption of individual hardware components. This can lead to inadvertently optimizing components that may not be as important for energy efficiency in real systems as the model may allude. We find that a key source of errors is the lack of a model for Dynamic Voltage and Frequency Scaling (DVFS). Lacking a DVFS model results in recent GPUs being reported to have a negative constant power term. Our insight, that power under V-F scaling is better modeled by a 3^{rd}-degree polynomial missing a quadratic term (Section 4.2.4.2), allows AccelWattch to accurately estimate constant power.

Another key source of error is the lack of a model for capturing the power-down of hardware components, and their contribution to static power when powered-up but inactive. In the absence

[2]This section is based on our MICRO'21 paper about AccelWattch [121].

of such a model, static power is lumped into a single constant, an oversimplification for modern chips with aggressive power gating. We infer, for the first time to our knowledge, how modern GPUs power-gate chip-wide hardware components (e.g., L2 cache), Streaming Multiprocessor (SM)-wide components (e.g., L1 caches) and lane-specific components (e.g., FPUs). AccelWattch accurately models the effect of reactivating power-gated structures. It does so by capitalizing on our insights (Section 4.2.4.3) that: (1) activating the first SM powers up global chip components which leak when not switching; (2) activating the first lane of an SM powers up SM-wide components, which leak when inactive; and (3) activating subsequent lanes additionally powers up only those lanes' execution units. Our insight, that the simultaneous execution of operations within a warp (now supported by modern GPUs [22]) presents a counter-intuitive sawtooth pattern of power consumption (Section 4.2.4.4), allows AccelWattch to accurately model thread divergence in the presence of Instruction-Level Parallelism (ILP) (Section 4.2.4.5).

Some recently-proposed power models [126]–[128] target modern GPUs, but they, as well as earlier works [129], [130] are provided only as analytic models over average behavior, which hinders research that requires cycle-level accuracy (e.g., research on DVFS). Moreover, analytic models are hard to extend to describe novel architectural components; often it is easier to build a cycle-level model that emulates the component's behavior and use it for evaluation. To support cycle-level research, the computer architecture community needs a robust and configurable power modeling tool, capable of supporting cycle-accurate simulation.

We address the lack of cycle-level power modeling tools for modern GPUs by introducing *AccelWattch*, a new GPU power model that is configurable, capable of cycle-level calculations in emulation and trace-driven environments, and supports DVFS. To the best of our knowledge, AccelWattch is the only power model capable of modeling both PTX (virtual ISA) and SASS (native machine ISA) instructions, and the only open-source tool capable of modeling closed-source

workloads with hand-tuned SASS instructions—it only needs a binary. In addition, AccelWattch is the only GPU power model that can be driven by either pure software performance models (e.g., Accel-Sim [24]), or hardware performance counters commonly found in modern GPUs (thereby capturing execution on real silicon), or a combination of the two. These AccelWattch variants allow researchers to balance the trade-off between power model accuracy and performance modeling effort.

We validate AccelWattch against hardware power measurements on an NVIDIA Volta GV100 [22] GPU running a suite of 26 kernels from NVIDIA CUDA Samples [107], Rodinia 3.1 [106], Parboil [108], and CUTLASS 1.3 [131] suites. AccelWattch yields a mean absolute percentage error (MAPE [132]) between 7.5–9.2 ± 2.1–3.1%, depending on the AccelWattch variant, achieving a Pearson r coefficient of 0.83–0.91. These errors are a factor of 22–$24\times$ lower than GPUWattch's when targeting the same architecture. As a case study, we apply AccelWattch on kernels from DeepBench [133] workloads, and find that it obtains 12.79% MAPE over hardware power measurements despite the significant limitations of existing performance models. We demonstrate the reliability of AccelWattch for design space exploration by applying our validated AccelWattch Volta model (i.e., without retraining or needing new hardware measurements) to model the power of two GPU architectures: a Pascal TITAN X [23], and a Turing RTX 2060S [125]. AccelWattch accurately predicts the power consumption of these new architectures, achieving $11 \pm 3.8\%$ and $13 \pm 4.7\%$ MAPE, respectively.

In summary, we make the following contributions:

- For the first time to our knowledge, we infer and introduce an analytic model that explains and accurately captures constant, static, and dynamic power consumption in the presence of DVFS, thread divergence, intra-warp functional unit overlap, variable SM occupancy, and power gating.

- We introduce AccelWattch, a cycle-level constant, static and dynamic power model for the NVIDIA Volta GPU architecture. AccelWattch resolves long-standing needs for modern GPU architectures: the lack of a cycle-level power model, and the inability to capture the constant and static power with existing methodologies. We validate AccelWattch and show it achieves high correlation to hardware measurements.

- To the best of our knowledge, AccelWattch is the only GPU power model that can be directed by emulation (PTX) or trace-driven (SASS) software performance models, or by hardware performance counters, or by a combination of the above. This allows for the study of discrete hardware components without the need to develop performance models of the entire architecture.

- We demonstrate that AccelWattch can enable reliable design space exploration. Directly applying the Volta power model on a GPU configuration resembling the Pascal and Turing architectures results in accurate power models for these architectures without tuning specifically for them.

4.2.2 AccelWattch Modeling Workflow

We follow the process shown in Figure 4.8 to develop a model that accurately estimates: (a) constant power, for example by board fans and peripheral circuitry, in the presence of DVFS ①; (b) static power in the presence of execution divergence, the simultaneous execution of operations within the same warp [22], variability in SM occupancy, and the power gating of lanes, SMs, and global chip hardware components ②–④; and (c) dynamic power consumption for each individual hardware component ⑤–⑧. To model dynamic power, we develop a suite of 102 microbenchmarks that isolate and stress the various components of a modern GPU ⑤. We use them, together

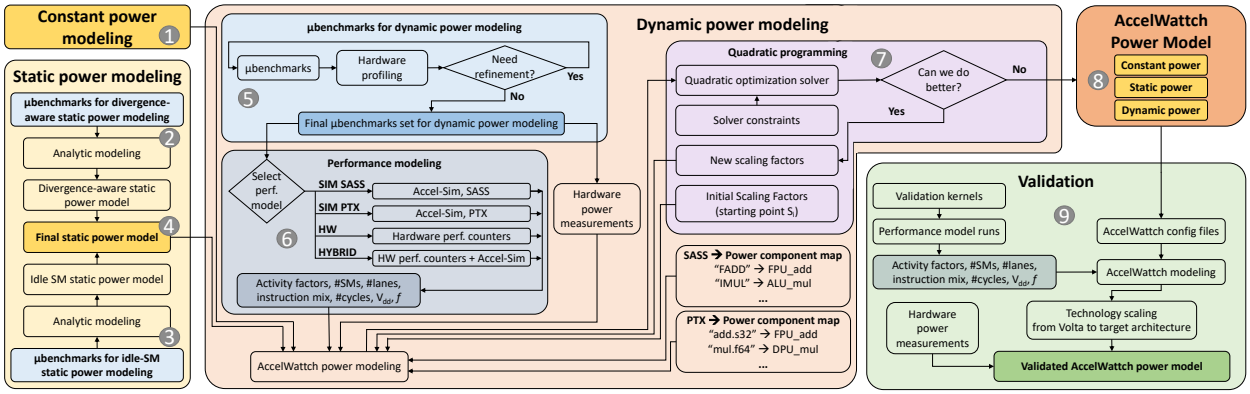


Figure 4.8: AccelWattch power modeling flowchart.

with hardware power measurements and execution statistics ⑥, to bound any modeling inaccuracies using quadratic programming ⑦. AccelWattch is driven by a performance model, which provides AccelWattch with statistics on hardware component activity, active SMs and lanes, voltage-frequency parameters, and cycle count ⑥. We integrate AccelWattch with GPGPU-Sim [118] and Accel-Sim [24] to facilitate its use for both PTX [110] and SASS [134] simulations, producing the *AccelWattch PTX SIM* and *AccelWattch SASS SIM* variants, respectively.

AccelWattch can also be driven by hardware performance counters collected during execution on real silicon, either entirely (*AccelWattch HW*) or in combination with software-modelled ones (*AccelWattch HYBRID*). This allows for the study of discrete hardware components without the need to develop accurate software performance models for the entire architecture. Building comprehensive and accurate performance models for GPUs is a painstaking and time-consuming process. In the absence of sophisticated software performance models like Accel-Sim [24] for future GPUs, one can use hardware performance counters and execution on real silicon to model the power of a GPU architecture, and replace the hardware performance counters of the component targeted by the research with counters obtained from a model of only that component.

The AccelWattch framework can be used to estimate the power consumption of a kernel run-

ning on a new architecture by first initializing it with the AccelWattch model ⑧. Then, a performance model provides AccelWattch with the kernel's execution statistics (e.g., through simulation or hardware counters from execution on real silicon) ⑨. If needed, the resulting power estimates are scaled to a new technology node.

A salient feature of AccelWattch is its longevity. As architectures and technology continue to evolve, power modeling tools must adapt to match their target systems. A key feature of AccelWattch is that it is software-only; it makes use of integrated power monitoring tools in modern GPUs, and requires no external equipment beyond a GPU card. Our power modeling framework is equipped with a suite of microbenchmarks, analytical models, an optimization solver, and a validation methodology that can support future GPUs.

4.2.3 The Architecture of NVIDIA Volta

GPUs execute programs known as "kernels", which comprise several threads, often thousands. Threads do not execute instructions independently; rather, sets of 32 threads (warps) execute the same instruction on different data by using their thread ID to select the data items to work on.

A Volta GV100 GPU chip consists of 80 Streaming Multiprocessors (SMs). Each SM is partitioned into four processing blocks [22], each with 16 INT32 cores for integer arithmetic, 16 FP32 and 8 FP64 cores for 32- and 64-bit floating-point, two tensor cores for matrix arithmetic, one special function unit (SFU) for complex operations (e.g., log), one warp scheduler, one dispatch unit, and a 64KB register file [22]. The INT32, FP32 and FP64 cores have adders, multipliers, and fused-multiply-add (FMA) units. The warp scheduler and dispatch unit can issue one instruction per clock to 32 execution lanes. Each of the 4 processing blocks per SM has 8 LD/ST units and a 12KB L0 instruction cache [135]. Each SM features a 128KB L1 data cache/shared memory, 2KB L1 and 64KB L1.5 constant caches, and a 128KB L1 instruction cache. The GPU has a 6144KB

unified L2 cache at the chip level, and a 32GB GPU DRAM off chip [135].

4.2.4 Constant, Static and Idle Power Modeling

4.2.4 Hardware Experimentation Methodology

We use NVIDIA Management Library (NVML) [136] and the higher-level API, NVIDIA System Management Interface (nvidia-smi) [137] interchangeably for collecting power measurements on silicon for all of our experiments. If needed, we vary the processor frequency with nvidia-smi (e.g., for the constant power modeling experiments described in Section 4.2.4.2). When possible, we lock the processor frequency to the default applications clock frequency while collecting power measurements for microbenchmark and validation suite kernels. We ensure that the microbenchmarks present the desired behavior by profiling them using hardware performance counters provided by NVIDIA Nsight Compute [138].

To protect our experiments from the impact of temperature variations, we bring the GPU chip to 65^{o}C before taking power measurements for a target kernel. Temperature variability affects static power exponentially. Keeping a constant temperature during hardware measurements eliminates this noise. After AccelWattch learns a model assuming constant temperature, one can model temperature variations by multiplying the modeled static power with an experimentally-derived temperature-dependent factor.

4.2.4 DVFS-Aware Constant Power Modeling

At a high level, GPU power can be described by Eq. (4.2):

$$P_{total} = P_{proc,dyn} + P_{mem,dyn} + P_{proc,static} + P_{mem,static} + P_{const} \tag{4.2}$$

The terms $P_{proc,dyn}$ and $P_{mem,dyn}$ comprise the dynamic power consumption of the GPU chip and memory, respectively, and depend on the respective component's frequency, voltage and technology parameters (e.g., capacitance, gate length). The $P_{proc,static}$ and $P_{mem,static}$ terms comprise the static power consumption of the GPU chip and memory, respectively, and depend on voltage and technology parameters only. GPUs consume power not only by activating microarchitectural components (e.g., ALUs, caches) or through leakage currents at inactive components (static power), but also by peripheral components such as GPU board fans and other auxiliary support circuitry. We capture the power consumption of these components in the constant power term P_{const}. We rewrite Eq. (4.2) to reflect these dependencies and obtain Eq. (4.3), in which C refers to the gate capacitance, V the supply voltage, and f the clock frequency. The terms a, a', b, b', m, and n are constants that abstract away environmental, technology and design factors (a, b for GPU chip dynamic and static power; a', b' for memory):

$$\begin{aligned} P_{total} &= aCV^2f + a'CV^2f + bV + b'V + P_{const} \\ &= mCV^2f + nV + P_{const} \end{aligned} \tag{4.3}$$

The methodology employed by prior cycle-level models like GPUWattch [124] to estimate constant power is based on Eq. (4.3) and **does not work on recent GPUs** [126]. This methodology relies on scaling down the frequency f, which linearly reduces the first term (dynamic power). By running kernels at varying frequencies and measuring the GPU's power consumption each time, one could estimate this linear relationship. Extrapolating this line to $f = 0$ eliminates the mCV^2f term and leaves only the $nV + P_{const}$ term, providing an estimate of the static and constant power.

Modern GPUs employ DVFS to scale voltage with frequency and this invalidates the underlying assumptions of the above methodology. Fitting the experimental results from a GPU employing DVFS (like Volta) to a linear model (as in GPUWattch) results in a negative constant and

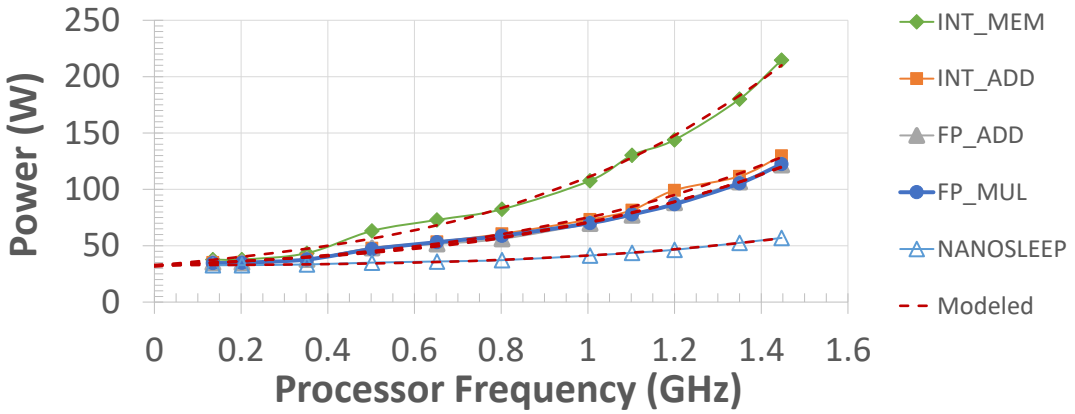


Figure 4.9: Measured and curve-fitted total power with varying processor frequency on GV100.

static power estimate, which is clearly incorrect. Recently-published data for fully-realized processors show a near-linear relationship in the frequency-voltage curve [139], [140]. Hence, we can approximate the voltage scaling as a linear function of frequency, $V \approx kf$, and rewrite Eq. (4.3) as:

$$P_{total} = \beta C f^3 + \tau f + P_{const} \tag{4.4}$$

Thus, total power can be approximated by a cubic polynomial with a missing quadratic term (β, τ are constants). Armed with Eq. (4.4), we perform hardware power measurements at varying clock frequencies of kernels running on a Volta GV100 following the methodology at Section 4.2.4.1. We then curve-fit the experimentally-measured data on curves of the form of Eq. (4.4). Figure 4.9 shows the experimental results, along with the fitted cubic polynomials. We evaluate a mixture of high-power microbenchmarks (e.g., INT_MEM, which executes a mix of integer and memory operations and exceeds 200 W), light workloads (e.g., NANOSLEEP, which executes only nanosleep instructions), and moderate workloads (e.g., INT_ADD, FP_ADD, and FP_MUL which execute integer adds, FP adds, and FP muls, respectively).

The fitted polynomials of the form of Eq. (4.4) show strong correlation to the hardware power measurements (0.998 Pearson r coefficient). By extrapolating the fitted curves all the way to the y-axis intercept point, we can estimate the total power of the GPU when $f = 0$. That is, the y-intercept corresponds to P_{const}. Following this methodology, we estimate that the constant power P_{const} for a Volta GV100 is 32.5 W.

4.2.4 Power-Gating-Aware Static Power Model

After estimating constant power, i.e., P_{const} in Eq. (4.4), we next consider the second term, τf, which models static power. First, we turn our attention to modeling static power in the presence of power gating. We infer how modern GPUs are gating chip-wide, SM-wide and lane-specific hardware components. We measure the impact on power consumption of re-activating these components on real hardware, and introduce an analytic model that explains the power-gating behavior of GPUs and accurately captures their power consumption in the presence of power gating. To the best of our knowledge, this is the first time that the power-gating behavior of GPUs is inferred, measured and modeled analytically.

When lanes or SMs are inactive, modern GPUs gate them to conserve power. There are some components that all SMs share (e.g., L2 cache). These global chip components are powered up even when there is *only one* SM active on the GPU. When they are activated, these components leak power when they are not switching. Similarly, there are components that all lanes share in an SM (e.g., L1 caches, shared memory. These SM-wide components are active even when there is *only one* thread active in an SM, while the remaining lanes are power gated. These components also leak power when they are not switching. As additional lanes become active, they power up their own lane-specific functional units (e.g., INT32 and FP32 cores) which may also leak power when inactive.

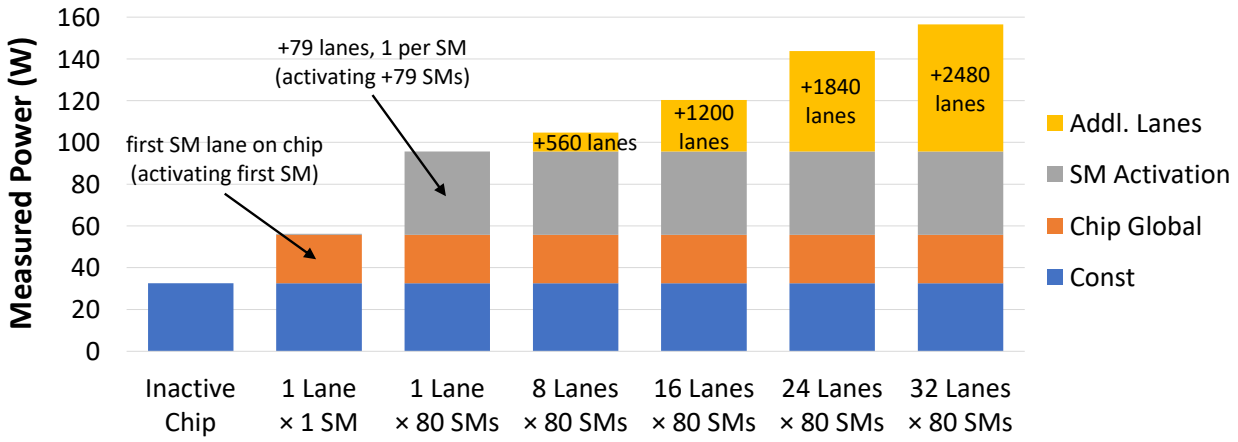


Figure 4.10: Inferring the power consumption of activating power-gated chip-wide and SM-wide components.

Figure 4.10 shows the hardware-measured power of a microbenchmark that issues integer operations to a varying number of SMs and lanes per SM. When no SM is active on the chip, it consumes only constant power (estimated in Section 4.2.4.2). When the microbenchmark runs on only one lane on one SM (*1 Lane* $\times$ *1 SM*), that first SM activation powers up that SM's structures, but also activates global chip structures shared by all SMs. When the microbenchmark utilizes additional SMs (*1 Lane* $\times$ *80 SMs*), it additionally powers up only the SM-wide components of the additional SMs. As a result, the first activated SM on a GPU consumes $47\times$ more power than each one of the subsequently-activated SMs. For example, *1 Lane* $\times$ *80 SMs* consumes 70% more power than *1 Lane* $\times$ *1 SM*, even though it utilizes $79\times$ more SMs.

Similarly, the first lane activation in an SM also activates SM-wide structures that are shared by all lanes. In contrast, activating additional lanes also powers up only those lanes' functional units. As a result, the first activated lane in an SM demands $31\times$ more power than lanes activated after it. For example, *8 Lanes* $\times$ *80 SMs* consumes 10% more power than *1 Lane* $\times$ *80 SMs*, even though it utilizes $7\times$ more lanes.

The increased power consumption after re-activating a component is both due to higher dynamic power as the component is utilized (captured by our dynamic power model), as well as higher static power. We capture the latter in the analytic model we introduce in the next section, as the effects of power gating are inherently linked to the effects of execution divergence.

4.2.4 Divergence-Aware Static Power Modeling

When a warp executes in an SM, it may leave some lanes inactive due to execution divergence, which may be gated to conserve power. The active lanes, however, still leak power as not all of their components are continuously utilized. We capture this behavior of a warp with y active lanes in Eq. (4.5).

$$\begin{aligned} P_{static,addLane} &= (P_{static,32Lanes} - P_{static,firstLane}) \;/\; 31 \\ P_{static,yLanes} &= P_{static,firstLane} + P_{static,addLane} \cdot (y-1) \end{aligned} \tag{4.5}$$

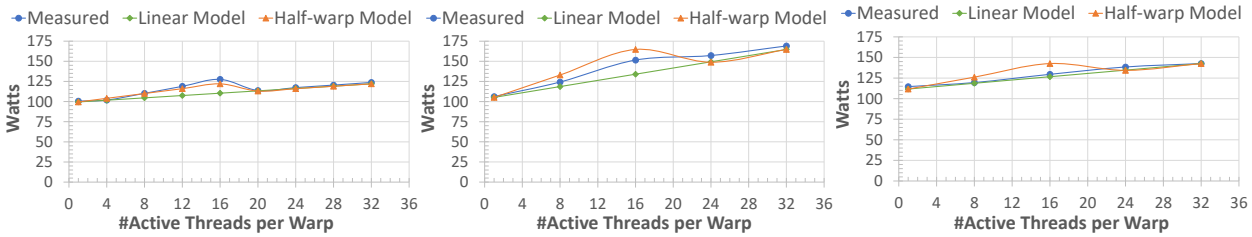
The $P_{static,firstLane}$ term captures the static power of the first active lane, to which we attribute the static power of all the SM-wide components that lanes share. Each additional active lane is only responsible for its own functional units' static power, $P_{static,addLane}$. The $P_{static,32Lanes}$ term refers to the static power when 32 lanes are active. We refer to Eq. (4.5) as the *Linear static power model*, as it distributes equally the static power among all lanes of a warp except the first lane.

However, the linear model of Eq. (4.5) does not always match real-world observations. An SM in Volta comprises 4 processing blocks [22], each with 16 CUDA cores. A warp executes by running two 16-thread half-warps, one after the other. If a warp has $y \leq 16$ threads active, the processing block executes the active half-warp but forgoes the execution of the empty one. Thus, the same fraction of lanes is active on every cycle, and the power consumption rises as y grows. At $y = 16$, all 16 cores on all processing blocks are always active, consuming maximum power.

If a warp has $16 < y < 32$ active threads, then "full half-warps" with 16 active threads (maximum power consumption) alternate with "partial half-warps" with the remaining active threads (lower power consumption). Thus, the power consumption for $16 < y < 32$ will be lower than the power consumption for $y = 16$ (note that the energy consumption will still be higher). When $y = 32$, all processing clusters once again execute "full half-warps" and reach maximum power. We capture this counter-intuitive behavior in Eq. (4.6), to which we refer as the *Half-warp static power model*.

$$P_{static,yLanes} = \begin{cases} P_{static,firstLane} \\ +P_{static,addLane} \cdot (y-1), & \text{if } y \leq 16 \\ \\ P_{static,firstLane} \\ +\frac{1}{2}P_{static,addLane} \cdot 15 \\ +\frac{1}{2}P_{static,addLane} \cdot (y-17), & \text{if } y > 16 \end{cases} \tag{4.6}$$

To study this behavior experimentally, we follow the process shown in Figure 4.8-② and the methodology described in Section 4.2.4.1. We develop microbenchmarks that utilize all SMs but with configurable thread divergence. We run each microbenchmark at varying clock frequencies and thread divergence, collect hardware power measurements, and curve-fit them to Eq. (4.4) (the fitted curve has 1% MAPE). From the fitted Eq. (4.4) for each microbenchmark, we estimate its static power (fitted τf term) when only one lane is active per warp ($P_{static,firstLane}$), and when 32 lanes are active per warp ($P_{static,32Lanes}$). Then, we replace Eq. (4.4)'s τf term with the linear and the half-warp models from Eqs. (4.5) and (4.6) to obtain an analytic model for total power that is divergent-aware.



(a) INT_MUL (b) INT_FP (c) INT_FP_SFU

Figure 4.11: Hardware measurements and modeled power with varying number of active threads in each warp.

We validate the half-warp power model in Figure 4.11a, which compares the power estimated by AccelWattch with hardware power measurements for a microbenchmark that issues INT_MUL instructions. The power for this microbenchmark strongly follows the half-warp model. We emphasize that the blue line in Figure 4.11a represents hardware measurements. It is indeed the case that 16-lane and 32-lane warps consume the maximum power on real hardware, and all other configurations consume less, giving rise to a sawtooth pattern. However, other cases (e.g., Figure 4.11c) follow the linear model instead. The reasons behind this behavior are discussed next.

4.2.4 ILP and Execution Divergence

When a kernel uses only one functional unit, power strongly follows the half-warp model (Figure 4.11a). When exercising two units, (e.g., when the kernel issues both INT32 and FP32 instructions—Figure 4.11b), the half-warp behavior is less pronounced. This happens because Volta can simultaneously execute operations in the same warp by running multiple functional units concurrently [22]. A kernel with ILP can exploit this behavior to execute faster. As different operations usually have different latencies, their executions become interleaved in time. Hence, on every cycle we observe a statistical mix of full and partial half-warps: if we take a snapshot of the GPU, we will observe processing blocks executing “full half-warps” (Section 4.2.4.4) of one

instruction concurrently with "partial half-warps" of the other. This statistical mix smooths out the sawtooth-like pattern of power consumption. When more units are employed (Figure 4.11c) the behavior becomes almost purely linear. Thus, static power for active SMs gradually drifts from the half-warp to the linear model, depending on the instruction mix. To the best of our knowledge, this is the first time this behavior is inferred and modeled.

Capitalizing on this observation, we assess the typical instruction patterns in GPU kernels and develop microbenchmarks that selectively stress them. We identified a total of 9 instruction mix categories, ranging from homogeneous categories with only integer ADD or only integer MUL instructions, to categories comprising a mix of instructions: int, int/FP, int/FP/DP, int/FP/SFU, int/FP/TEX, int/FP/tensor, and a category of only light instructions (e.g., nanosleep). We create the appropriate half-warp or linear models for each instruction mix and integrate them in AccelWattch. During an AccelWattch run, the performance model (simulator or hardware counters) reports the lane occupancy and instruction mix to AccelWattch, which then picks the appropriate power model.

4.2.4 Power Modeling for Idle SMs

Following a similar methodology, shown in Figure 4.8-③, we develop a model that captures the power consumption of SMs that are idle. We follow the methodology in Section 4.2.4.1 to develop microbenchmarks that vary the number of active SMs but use all 32 lanes of each warp (so thread divergence does not perturb our results). For simplicity, we assume that all SMs contribute equally to power consumption when they are occupied with the same microbenchmark. Thus, we estimate the dynamic plus static power per active SM when running microbenchmark i, $P_{dyn+static,perActiveSM,i}$, through Eq. (4.7), where $P_{total,80SMs,i}$ is the hardware power measurement of microbenchmark i with all SMs active (GV100 has 80) and P_{const} is the constant power estimated in Section 4.2.4.2.

$$P_{dyn+static,perActiveSM,i} = (P_{total,80SMs,i} - P_{const}) \ / \ 80 \tag{4.7}$$

When the same microbenchmark i is configured to occupy fewer SMs, $N_{activeSMs}$, we still expect each active SM to expend the power shown by Eq. (4.7). With that in mind, Eq. (4.8) models the power of all idle SMs (combined), where P_{total} is the hardware power measurement of that experiment.

$$P_{idleSMs,i} = P_{total,i} - P_{const} - P_{dyn+static,perActiveSM,i} \cdot N_{activeSMs} \tag{4.8}$$

We model the static power consumption per idle SM for microbenchmark i as a linear model in which each idle SM contributes equally: $P_{perIdleSM,i} = P_{idleSMs,i}/N_{idleSMs}$. We repeat this process for all n microbenchmarks, and use the geomean in Eq. (4.9) as the final estimate of idle SM power.

$$P_{perIdleSM} = \sqrt[n]{\prod_{i=1}^{n} P_{perIdleSM,i}} \tag{4.9}$$

Figure 4.12 shows that AccelWattch exhibits strong correlation with hardware measurements of the total power when running the INT_MUL microbenchmark, validating our model.

4.2.4 Putting It All Together

We combine Eq. (4.5), Eq. (4.6), and Eq. (4.7) and model the static power per active SM with y active lanes per warp in Eq. (4.10). The term $P_{static,yLanes,80SMs}$ is identical to $P_{static,yLanes}$ in Eqs. (4.5) and (4.6); we just make it explicit that the term is for 80 SMs.

$$P_{static,yLanes,perActiveSM} = P_{static,yLanes,80SMs} \ / \ 80 \tag{4.10}$$

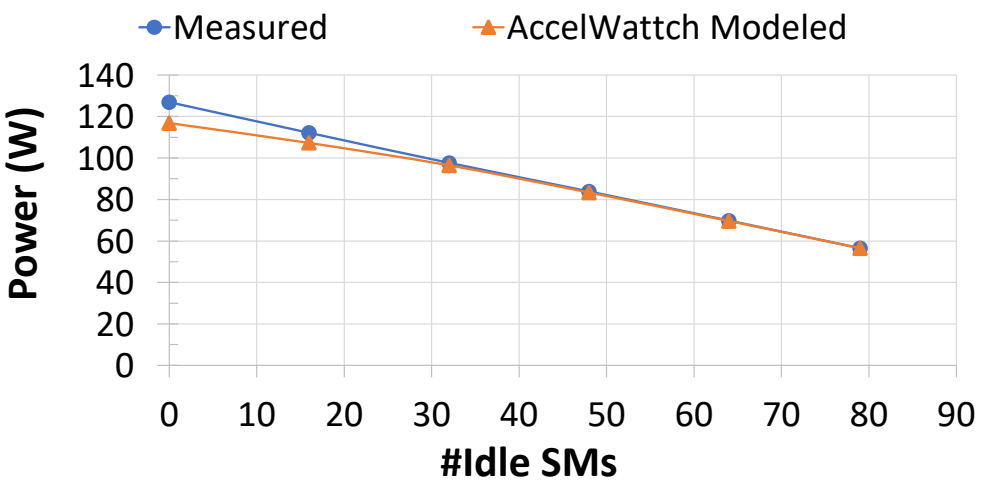


Figure 4.12: Validation of Idle SM static power model.

Taking all of the above into account, AccelWattch's overall power model is shown in Eq. (4.11).

$$P_{total,yLanes,kSMs} = P_{dyn} + P_{static,yLanes,perActiveSM} \cdot k + P_{perIdleSM} \cdot (80 - k) + P_{const} \quad (4.11)$$

The two middle terms of Eq. (4.11) comprise AccelWattch's final static power model (Figure 4.8-④) for y active lanes and k active SMs. The term P_{dyn} corresponds to the dynamic power, and is the subject of the next section.

4.2.5 Dynamic Power Modeling

4.2.5 Dynamic Power Model Formulation

AccelWattch employs an iterative approach to tune its parameters for dynamic power modelling, similar to GPUWattch, but it uses quadratic programming [141] instead of a least-squares solver. Given N microarchitectural components, the dynamic power consumed by a kernel can be described by Eq. (4.12) as a function of each component's i energy per access E_i, its activity factor

a_i (i.e., the number of accesses to it during execution), and the run time $T_{elapsedTime}$.

$$P_{dyn} = \sum_{i=1}^{N} \frac{a_i \cdot E_i}{T_{elapsedTime}} \tag{4.12}$$

The initial estimate $\hat{E}_i$ of component i's energy consumption per access is likely to be inaccurate. We consider this inaccuracy as an unknown variable x_i in equation $E_i = \hat{E}_i \cdot x_i$ and rewrite Eq. (4.11) for y active lanes and k SMs as:

$$\begin{aligned} P_{est.} = & \sum_{i=1}^{N} \frac{a_i \cdot \hat{E}_i}{T_{elapsedTime}} \cdot x_i + P_{static,yLanes,perActiveSM} \cdot k \\ & + P_{perIdleSM} \cdot (80 - k) + P_{const} \end{aligned} \tag{4.13}$$

One can view Eq. (4.13) as a dot product between a vector of power coefficients $\hat{P}_i \cdot x_i$, ..., $P_{static,activeSM} \cdot 1$, $P_{idleSM} \cdot 1$, $P_{const} \cdot 1$ (which remain constant for a given GPU) and activity factors, a_i, y and k, which vary across kernels. These vectors have $N + 3$ dimensions. N components of the vectors relate to estimated dynamic power, where each component contains an unknown factor, x_i. The three remaining components (modeled in Section 4.2.4) relate to constant, static, and idle SM power (these components effectively have $x_i = 1$). A single kernel will produce a vector with $N + 3$ activity factors, which can be used to estimate that workload's power consumption on the GPU. A collection of M kernels will provide $M \times (N + 3)$ set of equations, shown in Eq. (4.14). In this equation, the initially-inaccurate power estimates for the power components in $P_{est.}$ are corrected by the parameter vector X, to obtain a power for each kernel that approximates its hardware power measurement, $P_{meas.}$.

$$P_{est.}^{M \times (N+3)} \times X^{(N+3) \times 1} = P_{meas.}^{M \times 1} \tag{4.14}$$

Table 4.1: Dynamic power components in AccelWattch.

AccelWattch Dynamic Power Component	**Hardware Unit on Volta**	**AccelWattch Dynamic Power Component**	**Hardware Unit on Volta**
Instruction Buffer	L0 Inst. Cache	sqrt	SFU
Instruction Cache	L1i	log	
Constant Cache	Constant Cache	sin/cos	
L1d Cache	L1d Cache/ Shared Memory	exp	
Shared Memory		Tensor Core	Tensor Core
Register File	Register File	Texture Unit	Texture Unit
ALU	INT32 core	Scheduler	Sched. & Dispatch
int mul/mad		SM Pipeline	SM Pipeline
FPU	FP32 core	L2 Cache	L2 Cache
fp mul/mad		NoC	NoC
DPU	FP64 core	Dram	DRAM
dp mul/mad		Memory Controller	Memory Controller

Given enough workloads M, we can solve the system of Eq. (4.14) and obtain the best estimates X^*. We model dynamic power as an equation system with 22 parameters (Table 4.1). We tune these parameters using 102 microbenchmarks, each stressing specific microarchitectural components (Table 4.2), producing a 102×22 set of linear equations. Eq. (4.15) formulates the complete power model as an optimization problem, which AccelWattch solves using quadratic programming [141].

$$
\begin{aligned}
X^* = \arg\min_X \left(X^T \times P_{est.}^T \times P_{est.} \times X - (P_{est.}^T \times P_{meas.})^T \times X \right) \\
\textit{s.t. } \forall i : 0.001 \leq X_i \leq 1000 \ \wedge \ X_{static} = X_{idleSM} = X_{const} = 1 \\
X_{ALU} \leq X_{FPU} \leq X_{DPU} \ \wedge \ X_{ALU} \leq X_{imul} \\
X_{fpmul} \leq \{X_{imul}, X_{dpmul}, X_{sqrt}, X_{log}, X_{sin}, X_{exp}, X_{tensor}, X_{tex}\}
\end{aligned} \tag{4.15}
$$

Table 4.1 lists the dynamic power components (corresponding components of X) modeled

Table 4.2: AccelWattch tuning μBenchmarks.

Hardware Comp. Category	μBench Count	Hardware Comp. Category	μBench Count
Active/Idle SMs	12	Register File	1
INT32 core	9	dCaches + Sh.Mem. + NoC	11
FP32 core	8	DRAM + MC	2
FP64 core	8	Tensor core	6
SFU	9	Mix	29
Texture Unit	7	Other (L0, L1i, Pipeline, Scheduler)	102

by AccelWattch. Each INT32, FP32, and FP64 unit in Volta can perform additions, multiplications, FMAs, and a few other operations. Each of these operations activates different parts of the functional unit's circuitry, which results in a different power consumption per operation. Thus, AccelWattch tracks these operations separately. Similarly, AccelWattch tracks SFU operations separately (e.g., log, sqrt). The L2 Cache and NoC components cannot be distinguished from each other, hence we model them together (similarly for DRAM and Memory Controller).

The AccelWattch HW model collects the information shown in Table 4.1 from hardware counters, except for the shaded components. There are no hardware counters for L1i and register file activity, and while DRAM read and write counters exist, there is no DRAM precharge counter. AccelWattch HYBRID is built similarly, but with the L2 Cache and NoC counters derived from Accel-Sim simulations.

4.2.5 Performance Modeling Framework

We developed AccelWattch by extensively modifying the dormant McPAT-based [142] GPUWattch [124] power model that comes packaged with the underlying GPGPU-Sim v4.0.1 [118] integrated in Accel-Sim v1.1.0 [24]. AccelWattch is driven by a performance model, which pro-

vides AccelWattch with statistics on hardware component activity, active SMs and lanes, voltage-frequency parameters, and cycle count (Figure 4.8-⑥).

For the performance model of AccelWattch variants driven fully or partially by software simulations (SASS SIM, PTX SIM, HYBRID) we use the latest publicly-available version of Accel-Sim v1.1.0 [24]. Accel-Sim has been extensively validated against Volta, showing strong performance correlation with > 0.97 Pearson r coefficient. For SASS simulations we feed Accel-Sim with SASS traces generated by the NVIDIA Binary Instrumentation Tool (NVBit) [143]. For PTX simulations, Accel-Sim invokes the underlying GPGPU-Sim.

At each sampling period (500 cycles) Accel-Sim provides execution statistics to AccelWattch, which uses them to estimate the workload's power for each sampling period. As the performance model provides AccelWattch with frequency and voltage settings at each sampling interval, AccelWattch can scale the estimated power for that interval following Eq. (4.3). Thus, if the performance model is DVFS-capable, AccelWattch will calculate all power transitions.

Similarly, for the AccelWattch variants driven by hardware (HW and HYBRID), AccelWattch collects hardware activity and execution statistics from kernel runs on real silicon using hardware counters provided by NVIDIA Nsight Compute [138]. For these models, AccelWattch collects dynamic instruction information and lane activity from the SASS traces (which are also obtained from execution on real silicon). For AccelWattch HYBRID, we follow the same methodology as for AccelWattch HW for all available hardware counters, but we utilize the methodology for AccelWattch SASS SIM to obtain the activity counters for the L2 Cache and NoC.

4.2.5 Microbenchmarking for Dynamic Power

Following the process shown in Figure 4.8-⑤ and the methodology in Section 4.2.4.1, we build a suite of 102 microbenchmarks that stress target hardware components to estimate their power

consumption. We use a mixture of compiler options, inline-assembly (PTX), and pointer-chasing (for microbenchmarks that stress the memory hierarchy) to work around default compiler optimizations. We place the Region of Interest (ROI) of these kernels inside an unrolled loop, and run them on silicon with a high loop iteration count.

Table 4.1 lists the 22 microarchitectural components that AccelWattch tracks. Table 4.2 combines them into coarser-grain categories and lists the number or microbenchmarks that target components within each category. The *mix* microbenchmarks target combinations of these categories. The *Other* category includes components such as the SM pipeline, scheduler, and L0 and L1i instruction caches. All microbenchmarks stress this category, so all are included.

Figure 4.13 shows the dynamic power heat-map of microbenchmarks based on the hardware component categories they target. Each cell's color encodes the fraction of dynamic power a microbenchmark spends on the corresponding GPU component, as estimated by AccelWattch SASS SIM. It is important to note that even if a hardware unit is stressed by continuously issuing instructions to it, other units may also be accessed with a high frequency (e.g., register file, L1i), and the power consumed by these other units may be higher than the targeted one's. In such cases, the heat-map cell corresponding to the targeted unit may not appear "hot". For example, this behavior appears with units that do not consume much power, e.g., the Texture Unit, or with microbenchmarks that stress the INT32 Core with highly-divergent code. Overall, the heat-map demonstrates that each microbenchmark exercises the corresponding GPU component it targets, and our suite adequately exercises all components.

To build the power model of an AccelWattch variant (Section 4.2.2), we collect each hardware component's activity when running a microbenchmark from the respective performance model, and generate a set of per-component power estimates. These estimates, together with hardware measurements of total power consumption, are used in the quadratic programming optimization of

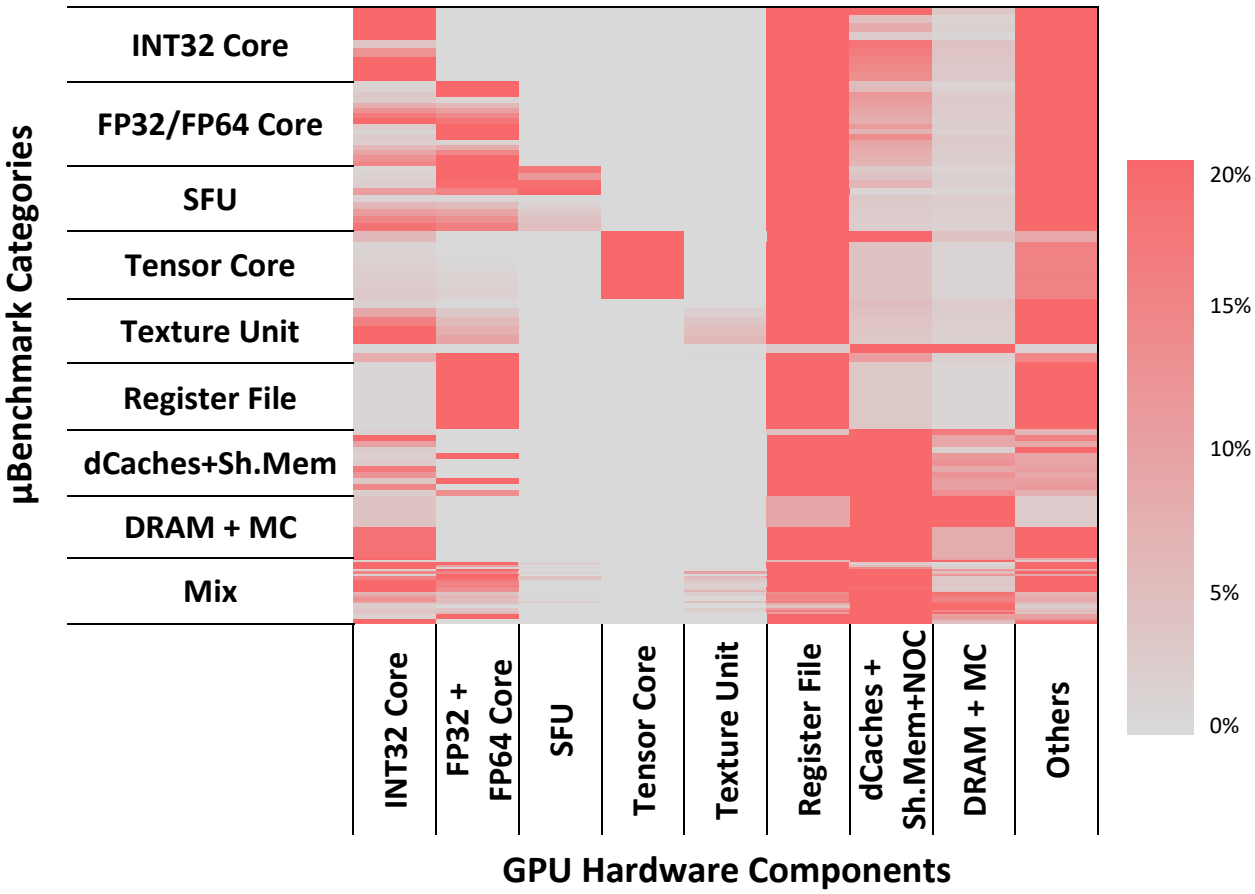


Figure 4.13: Dynamic power heat-map of GPU hardware component categories exercised by microbenchmarks.

Eq. (4.15).

4.2.5 Quadratic Programming Optimization

We perform quadratic programming optimization to minimize the relative error between the modeled system power and the measured hardware power (Eq. (4.15)). At each step of the regression, we obtain new per-component scaling factors that we supply to AccelWattch and re-iterate, until the solver can no longer reduce the relative errors (Figure 4.8-⑦). We enforce per-scaling-factor constraints on the quadratic solver to guard against unrealistic component power estimates. In

Table 4.3: Target GPUs for validation and case studies.

GPU	Tech. Node	Clock Frequency for HW Power Measurement	Power Limit	Case Study?
Quadro GV100 (Volta)	12 nm	1417 MHz	250 W	N
TITAN X (Pascal)	16 nm	1470 MHz	250 W	Y
RTX 2060S (Turing)	12 nm	1905 MHz	175 W	Y

particular, we ensure all scaling factors are positive, and we constrain the energy cost of execution units to guard against unrealistic estimates (see constraints in Eq. (4.15)).

We use two different starting points for the scaling factors used in the initial iteration of this process. For one of the starting points, all initial scaling factors are set to one, thus there is no scaling taking place initially. The other starting point is obtained from the GPUWattch model for NVIDIA Fermi GTX 480 [144] which has been independently validated [124]. Due to having two starting points, we end up with two AccelWattch models. The model obtained from the Fermi starting point achieves higher accuracy (9.2% vs. 14.8% MAPE on the validation set for SASS SIM). Thus, for each AccelWattch variant, we adopt the model obtained from the Fermi starting point as the final AccelWattch model (Figure 4.8-⑧).

4.2.6 Validation

4.2.6 Target Architecture and Workloads

We validate AccelWattch against a Volta GV100 GPU (Table 4.3) by applying the AccelWattch power models on a suite of validation kernels that are not part of the training set.

Our validation suite consists of a wide range of kernels selected from NVIDIA CUDA Samples (SDK) [107], Rodinia 3.1 [106], Parboil [108], and CUTLASS 1.3 [131]. Including NVIDIA SDK is important for the unbiased evaluation of AccelWattch. It does not skew results to AccelWattch's

favor: AccelWattch SASS SIM achieves 9.91% MAPE for NVIDIA SDK, compared to 4.9% (Parboil), 9.55% (Rodinia), and 9.98% (CUTLASS). On the contrary, AccelWattch experiences its largest error in one of the SDK workloads (dct_k2). All workloads are compiled with NVCC V11.0.167 [145] with compute-capability support for Volta (`code=sm_70`).

To collect hardware power measurements, we launch the target validation kernel to the default CUDA stream repeatedly in a loop so that it runs sufficiently long for accurate hardware measurements using NVML. We ensure that the kernel runs for NVML's entire sampling period through nvidia-smi. We wait until the chip reaches 65°C, then collect several power measurements. We let the chip cool down back to its idle-state temperature, repeat at least 5 times, and report the average across all measurements. We observe that measurements at the NVML resolution frequency are stable throughout kernel execution (0.0018–1.9% variance across all measurements and repetitions). If a target kernel cannot reach 65°C, we use a power-hungry kernel first to heat up the chip to much higher than 65°C, and then switch to the target kernel and collect power measurements at 65°C as the chip cools down.

NVML has a low sampling frequency of 50-100 Hz. Thus, we are unable to collect accurate hardware power measurements of short-running kernels ($< 2\mu s$ run time) because their measurements are perturbed by other events (e.g., invocation/setup overheads to prepare the next kernel iteration, host synchronization, PCI transfers). We exclude such kernels from our suite. We also exclude kernels that are impractical to simulate (> 2 days per run) due to exceedingly-long simulation times and multi-TB instruction trace storage requirements. All kernels are run to completion.

Table 4.4 lists our full evaluation suite of 26 kernels from 18 workloads along with their runtime coverage of the respective workloads they are from. We use the largest available input configuration for all workloads except CUTLASS, for which, we use three different input matrix sizes. We exclude CUTLASS, *hotspot*, and *pathfinder* from the PTX SIM validation suite because they

Table 4.4: List of kernels in validation suite.

Kernel	Run-time Coverage	Benchmark	Kernel	Run-time Coverage	Benchmark
CUDA Samples 11.0					
tensor_K1	100%	cudaTensorCoreGemm	dct_K1	19.6%	dct8x8
binOpt_K1	100%	BinomialOptions	dct_K2	72.3%	dct8x8
walsh_K1	47.8%	fastWalshTransform	histo_K1	52.9%	histogram
walsh_K2	49.4%	fastWalshTransform	msort_K1	71.8%	mergesort
qrng_K1	66.4%	quasirandomGenerator	msort_K2	26.3%	mergesort
qrng_K2	33.6%	quasirandomGenerator	sobol_K1	100%	SobolQRNG
Rodinia 3.1					
kmeans_K1	91.6%	kmeans	sradv1_K1	53.9%	sradv1
bprop_K1	75.7%	backprop	hspot_K1	100%	hotspot
bprop_K2	24.3%	backprop	b+tree_K1	48.5%	b+tree
pfind_K1	100%	pathfinder	b+tree_K2	51.5%	b+tree
CUTLASS 1.3 (cutlass-wmma)			Parboil		
cutlass_K1	100%	input: 2560x16x2560	sgemm_K1	100%	sgemm
cutlass_K2	100%	input: 4096x128x4096	mri-q_K1	100%	mri-q
cutlass_K3	100%	input: 2560x512x2560	sad_K1	95.9%	sad

do not compile for Accel-Sim's PTX mode We exclude *pathfinder* from the HW and HYBRID validation suites because NVIDIA Nsight Compute fails to provide hardware performance counters for this workload.

To validate AccelWattch, we follow the flow in Figure 4.8-⑨ and discussed in Section 4.2.2, with a NVIDIA Volta GV100 GPU as the target architecture. We compare AccelWattch's power estimates with hardware measurements obtained for the validation suite kernels running on a GV100 GPU.

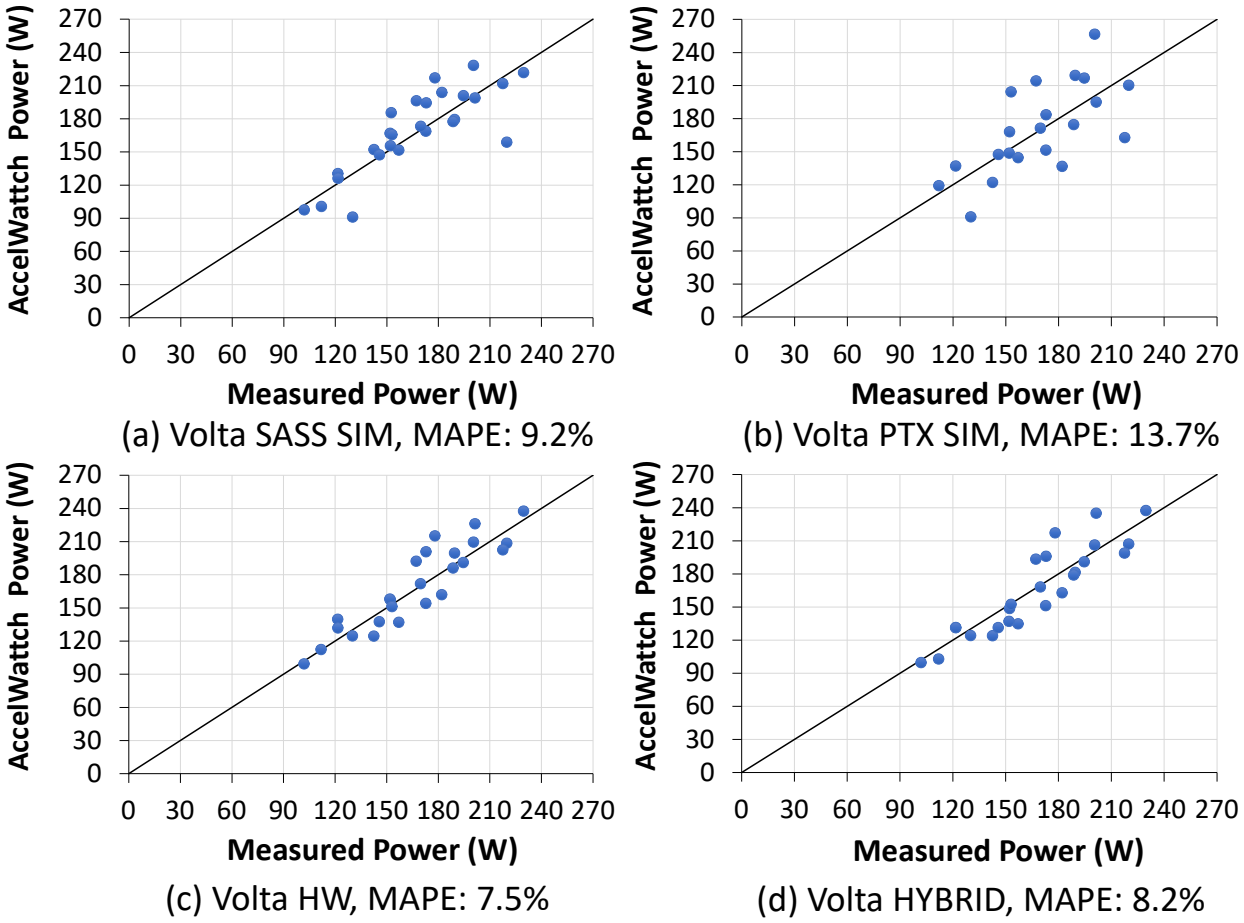


Figure 4.14: Correlation plots for AccelWattch validation.

4.2.6 Validation Results

Figure 4.14 shows the correlation plots of the estimated vs. measured power consumption and the corresponding MAPE for AccelWatch SASS SIM and PTX SIM for Volta. Overall, AccelWattch exhibits stronger correlation for SASS than PTX. We attribute this to the fact that PTX instructions do not map 1:1 to SASS instructions; prior work [146] demonstrates that simulating a virtual ISA (PTX) introduces inaccuracies compared to simulating the native ISA (SASS), which directly corresponds to execution on hardware units.

AccelWattch SASS SIM modeling a Volta GV100 compared to hardware measurements attains a MAPE of $9.2\% \pm 3.12\%$ (95% confidence interval) with a maximum relative error of 30%. Two thirds of our validation suite kernels (17 out of 26) have $< 10\%$ absolute relative error, while only 4 kernels have an absolute relative error of $> 20\%$. In comparison, when modeling a Volta architecture, GPUWattch reaches a MAPE of 219% with maximum error 447% (Section 4.2.7.3). For the two GPUs it is trained for (GTX 480 and Quadro FX5600), GPUWattch achieves average error of 9.9% and 13.4%, respectively, but with a maximum relative error of 57.8%. Overall, AccelWattch successfully tracks the high variability in measured power across our validation suite (from 90 W up to 230 W).

Among all variants, AccelWattch HW achieves the lowest MAPE (7.5%). This is expected, as it is driven by performance counters collected from execution on real silicon, and does not suffer from the inevitable inaccuracies of software performance models (e.g., Accel-Sim). However, AccelWattch HW is the most restrictive model, as it does not lend the same modeling flexibility as software simulators.

AccelWattch HYBRID is introduced as a way to alleviate this problem. The HYBRID variant utilizes hardware performance counters for all components except the ones that the user decides to replace with their own models. Thus, users may partially avoid the inordinate effort required to build, tune and validate highly sophisticated software models of entire GPU architectures, and instead focus their attention on only the few components that are relevant to their research target. As an example of a HYBRID model in this work, we target the L2 and NoC hardware components and replace their statistics with ones obtained from Accel-Sim. AccelWattch HYBRID achieves a MAPE of 8.2%, showing that it can successfully trade-off accuracy for modeling flexibility.

Figure 4.15 (left) shows the normalized power breakdown estimated by AccelWattch for Volta averaged across all kernels in the validation set. The register file, static power, and constant power

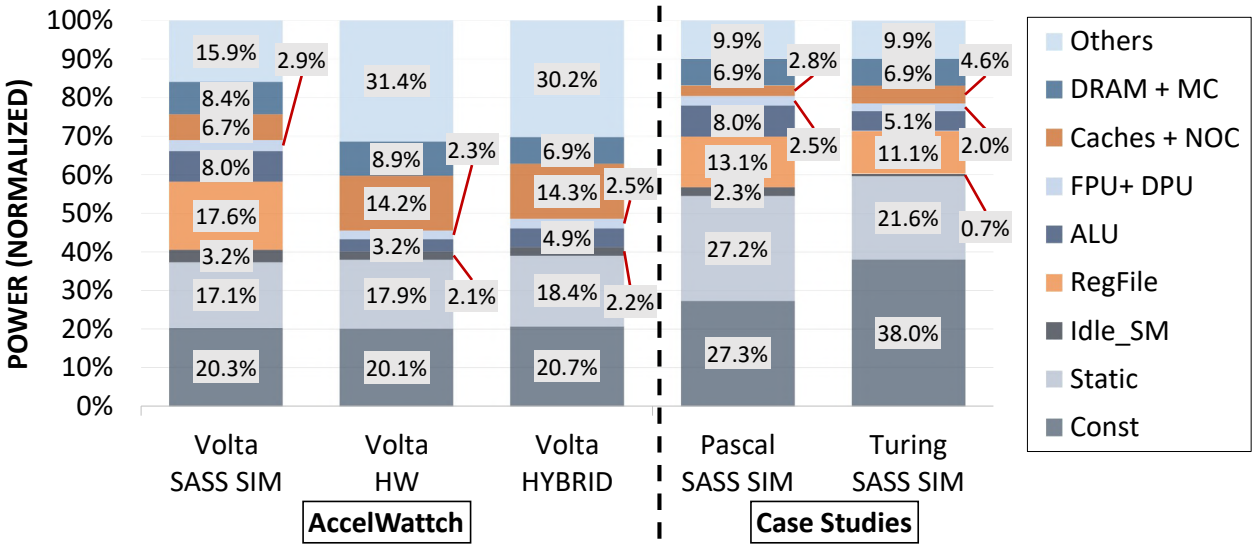


Figure 4.15: Normalized per-component power breakdown.

are the most significant power contributors, together consuming 55% of the total system power on average. Volta also has a measurable *Idle_SM* power component, owing to having a high number of SMs which our validation suite kernels do not always fully utilize. The *Others* category comprises of the instruction buffer, scheduler, SM pipeline, texture units, and tensor cores (exercised by only 4 out of 26 kernels). As Volta does not have hardware performance counters for register file and L1i, the AccelWattch HW solver minimizes error by lumping their power to other commonly-accessed ones: instruction buffer, scheduler, and SM pipeline. Hence, the Others category grows proportionally to accommodate this reassignment. AccelWattch HYBRID shows a similar trend. Figure 4.15 also shows that replacing the hardware counters for L2 and NoC hardly changes the power breakdown compared to HW, suggesting that HYBRID is likely to work well when the software model of the targeted component closely approximates its behavior on real silicon.

Figure 4.16 shows the power breakdown for each kernel. Tensor cores consume a significant portion of total system power (geomean 28.7%) for the kernels that use them. Note that tensor cores are not part of the *Others* category in Figure 4.16. Also, several kernels (backprop_k1, hotspot_k1,

sgemm_k1) consume over 90% of the peak power. We postulate this happens because these kernels keep resources busier than the rest due to high thread IPCs (5690, 7157, and 4668, respectively) and have a nearly even split of ALU and FPU (DPU) instructions, which can execute concurrently on Volta.

4.2.7 Case Studies

4.2.7 Modeling Pascal and Turing Architectures

An important goal of an architecture power model is to enable accurate design space exploration. For the use-case scenarios, we envision an architect who starts with the Volta architecture and uses AccelWattch to estimate the power consumption for a new architecture with different parameters. While GPUWattch is configurable, its accuracy when varying parameters has been identified as a weakness [147] (see also Section 4.2.7.3). Thus, as our first case study, we apply AccelWattch to estimate the power consumption of two new architectures for which it has **not** been tuned. We emphasize that if we directly tuned models for these GPUs they would likely result in more accurate models.

Ideally, we would validate AccelWattch predictions for these new architectures against Volta chips that employ these different configurations. However, such chips do not exist. As a proxy, we select configuration parameters similar to the NVIDIA Pascal and Turing architectures, and

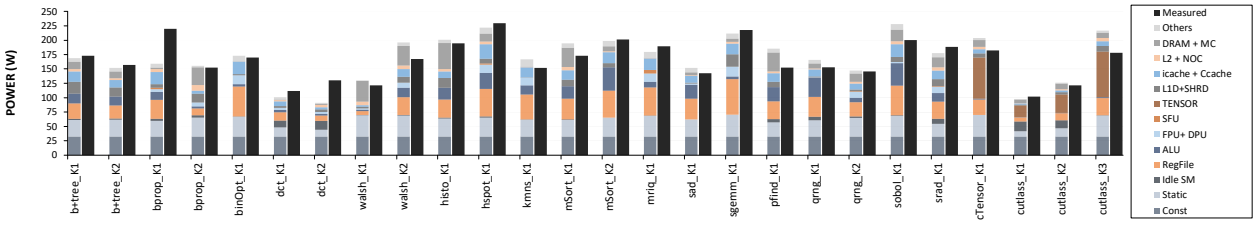


Figure 4.16: AccelWattch validation: AccelWattch SASS SIM modeling a Volta GV100.

compare against real Pascal and Turing chips. The Pascal and Turing architectures are the nearest to Volta. Hence, they are the most likely to have similar hardware implementations. Differences in the implementation of hardware units and the ISA between Volta and these architectures will manifest as modeling error. Table 4.3 lists the parameters of the target GPUs.

The evaluation closely follows the validation flow shown in Figure 4.8-⑨, with small modifications. First, the workloads are compiled with compute-capability support for Pascal and Turing (`code=sm_61` and `code=sm_75` compiler options, respectively). Second, Volta, Pascal and Turing implement different ISAs. Thus, comparing hardware runs on Pascal (or Turing) with Volta-derived traces in Accel-Sim would introduce spurious inaccuracies; we only attempt to model different architectural parameters with AccelWattch in these use cases after all, not different ISAs. To avoid such spurious inaccuracies, we re-extract traces for Pascal and Turing GPUs to use in the validation of the corresponding use cases. AccelWattch is still using the Volta-trained model—the traces are used only for validation purposes. Differences in hardware implementations still manifest as errors, as we do not attempt to model different functional unit implementations. We exclude all workloads that use tensor cores (CUTLASS and *cudaTensorCoreGemm*) from our validation suite for Pascal, since Pascal does not have tensor cores.

AccelWattch is based on the Volta architecture which is implemented at 12 nm, while Pascal is at 16 nm. Thus, following the flow in Figure 4.8-⑨, after we collect power estimates for Pascal from AccelWattch, we apply technology scaling based on published IRDS [148] parameters. Technology scaling reduces MAPE by 1.22% for PTX and 1.85% for SASS compared to the non-scaled models. Turing is also at 12 nm so it does not need technology scaling. We also set the constant power for Turing at $1.7\times$ higher than Volta's to approximate the new board's fans and peripheral circuitry. We only make this change so we can compare against a real-world chip without perturbations from unrelated components; architecture research does not typically modify fans and

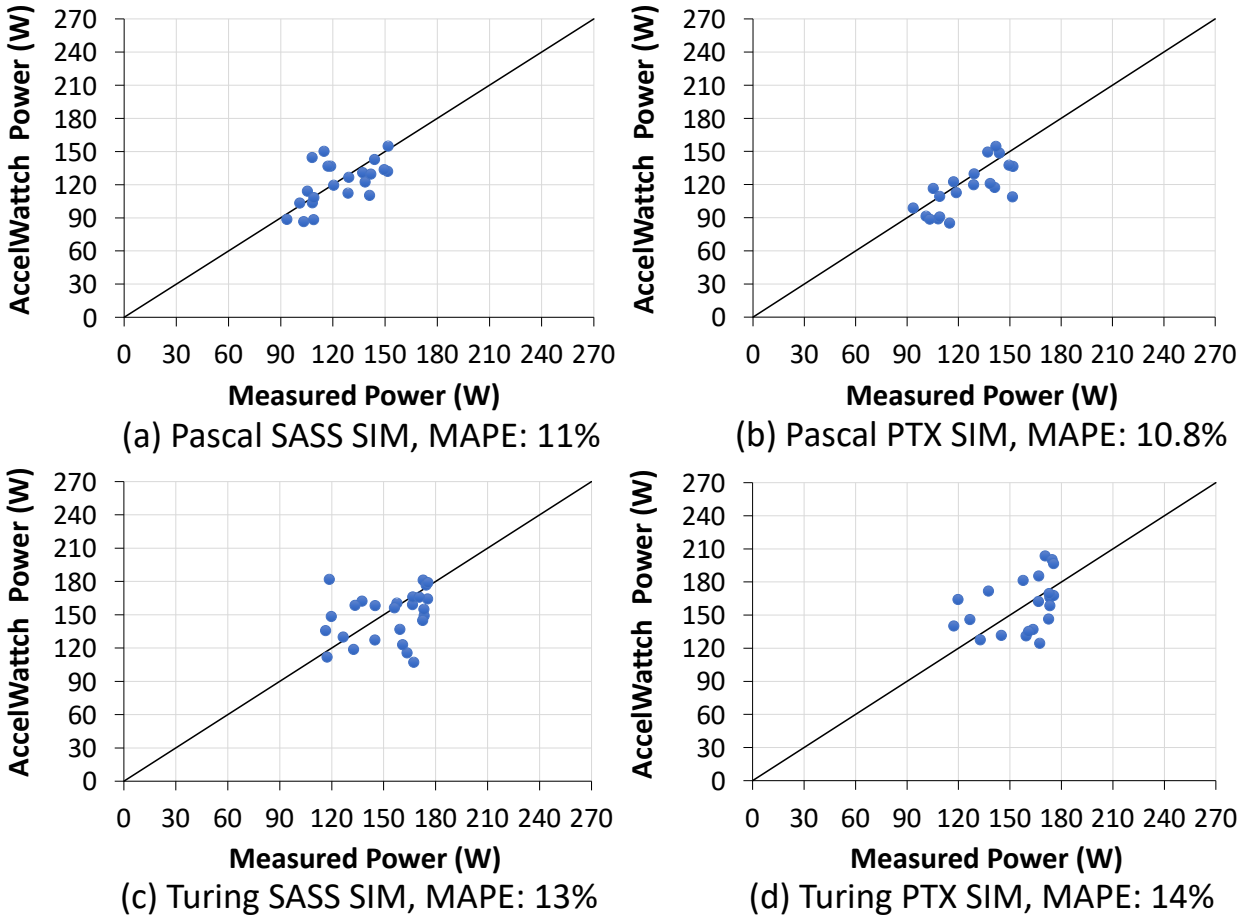


Figure 4.17: Correlation plots for case studies.

peripherals.

Figure 4.17 shows the correlation plots of the estimated vs. measured power consumption for the two case studies, and Figure 4.18 shows the per-component breakdowns for all kernels. Overall, AccelWattch shows strong correlation to hardware power measurements for both Pascal and Turing. Figure 4.15 (right) shows the normalized average power breakdown. Similarly to Volta, the register file, static power and constant power are the three most significant components, together consuming 67.7% and 70.7% of the average total system power on Pascal and Turing, respectively.

Computer architects typically evaluate their designs by comparing a figure of merit (e.g., power) relative to a baseline design across workloads. As all simulators occasionally exhibit high errors, architects typically also consider averages across many workloads, rather than only a few worst-case applications. Following this common pattern, Figure 4.19a shows the estimated power of Pascal relative to the estimated power of Volta across workloads, and on average (red bar). The figure also shows the hardware-measured power of Pascal relative to Volta chips running the same workloads for comparison, along with an average of the relative hardware measurements. Similarly, Figures 4.19b and 4.19c show the estimated and hardware-measured power of Turing relative to Volta and Turing relative to Pascal. While AccelWattch's error varies by workload, the aggregate estimate closely tracks real hardware measurements for all architectures. Across all workloads (26 for Turing, 22 for Pascal), the average relative power as estimated by AccelWattch differs from the average relative power measured in hardware by 1% for Pascal vs. Volta, 3% for Turing vs. Volta, and 1% for Turing vs. Pascal. High errors are rare in AccelWattch: only 9 out of 26 workloads exhibit error over 10%.

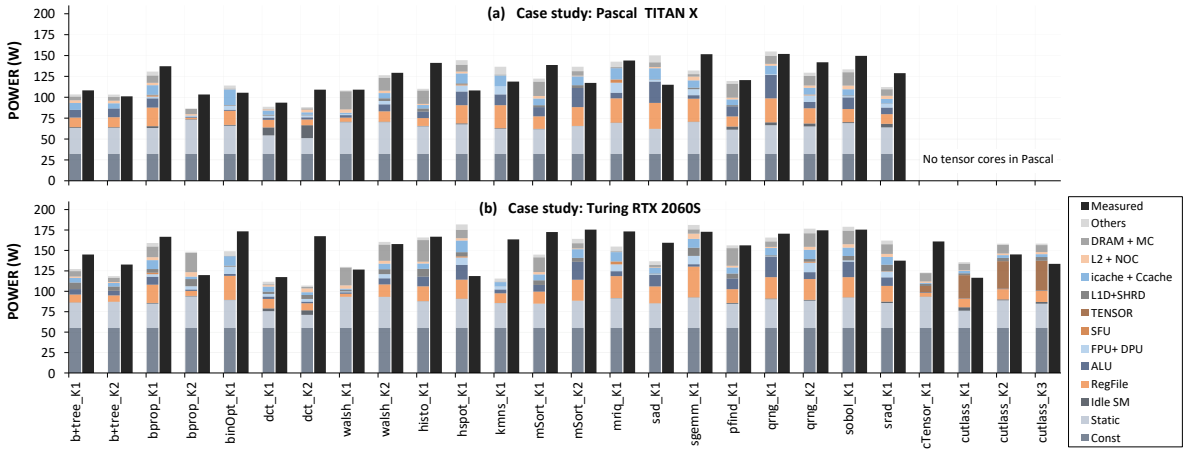
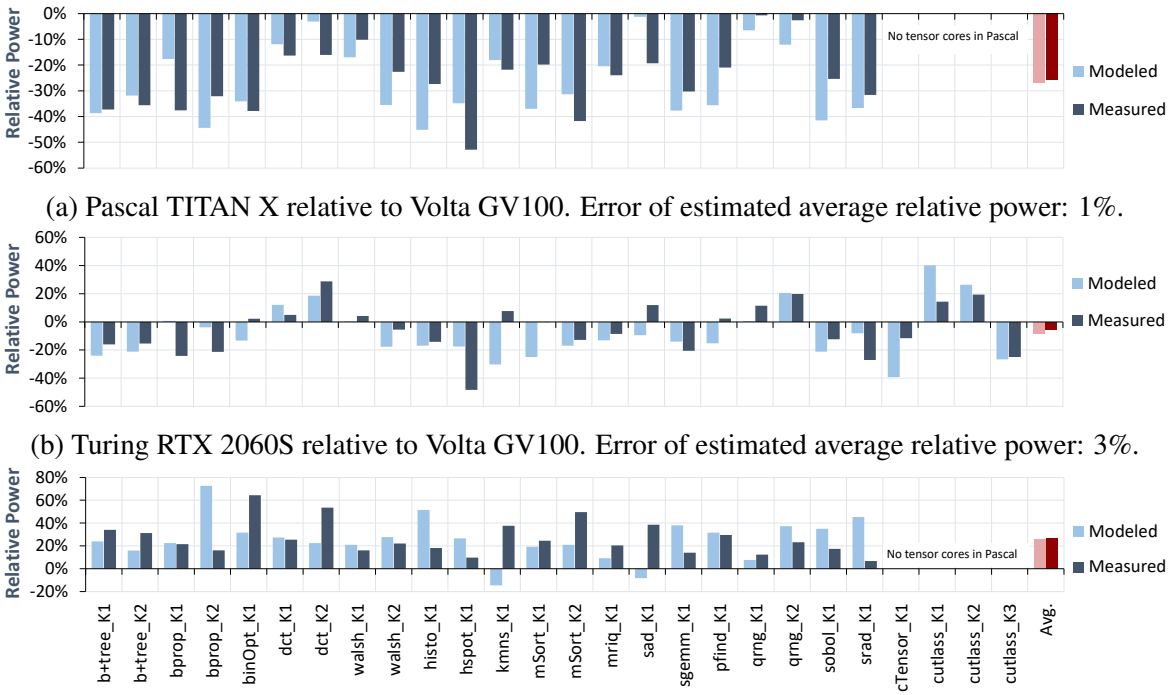


Figure 4.18: Case studies: AccelWattch SASS SIM (tuned for Volta), applied to model Pascal and Turing architectures.



(a) Pascal TITAN X relative to Volta GV100. Error of estimated average relative power: 1%.

(b) Turing RTX 2060S relative to Volta GV100. Error of estimated average relative power: 3%.

(c) Turing RTX 2060S relative to Pascal TITAN X. Error of estimated average relative power: 1%.

Figure 4.19: Relative Modeled and Measured Power across three architectures for AccelWattch SASS SIM.

The error is higher for Turing over Volta (Figure 4.19b) largely due to inaccuracies in Accel-Sim (Accel-Sim is not part of this work). For example, the L1d miss rate for *kmeans_K1* on a Turing RTX 2060S is 10× higher than the one estimated by Accel-Sim, leading to a 1.7× error in the run time estimate. As AccelWattch depends on the run time estimate to convert energy to power (Eq. (4.13)), inaccuracies in the performance model can adversely affect the power estimates. In addition, some errors are artifacts of ISA and hardware changes, which we did not intend to capture in these use cases, and are therefore spurious (Section 4.2.7.1). The Turing power relative to Volta is also concentrated in relatively small changes around zero (Figure 4.19b), which makes it easy even for small inaccuracies to result in estimates pointing in the opposite direction (*kmns_k1, sad_k1, pfind_k1*). Even with the adverse impact of performance model inaccuracies and treading

around zero, only in 4 out of 26 workloads (i.e., 15% of the time) AccelWattch's prediction points in the opposite direction than the hardware measurement; in 85% of the time the predictions are tracking the measurements on real silicon. This fraction grows for Turing relative to Pascal to 91%, and becomes 100% for Pascal relative to Volta.

4.2.7 AccelWattch for Deep Learning Workloads

GPUs are one of the dominant forces in Machine Learning (ML) acceleration [149]. To assess AccelWattch's accuracy in the ML space, our validation suite already includes workloads from CUTLASS, which primarily consists of general matrix multiply (GEMM) operations. Many operations in modern deep neural networks are either defined as GEMMs or can be cast as such, thus CUTLASS is representative of many ML workloads. In this section we delve deeper into the ML space and evaluate the accuracy of AccelWattch not only in GEMMs, but also in benchmarks implementing Convolutional (CNN) and Recurrent Long-Short Term Memory Neural Networks (RNN-LSTM). For this purpose we use DeepBench [133], a widely-used deep learning benchmark suite.

DeepBench utilizes closed-source, hand-tuned SASS kernels from the cuBLAS [150] and cuDNN [151] libraries, for which there are no PTX representations. While AccelWattch can execute closed-source kernels and estimate their power consumption, validating AccelWattch predictions on DeepBench is challenging and error-prone. Each DeepBench workload issues 10–130 kernels (geomean 33), and each kernel only uses about 12 SMs. The GPU hardware executes several kernels concurrently. However, Accel-Sim executes kernels only sequentially, leaving most of the simulated GPU idle and misleading AccelWattch to report significantly lower power. This is a limitation of Accel-Sim and not of AccelWattch. To mitigate this problem, we hand-construct a possible concurrent kernel execution schedule for each DeepBench benchmark, and then use

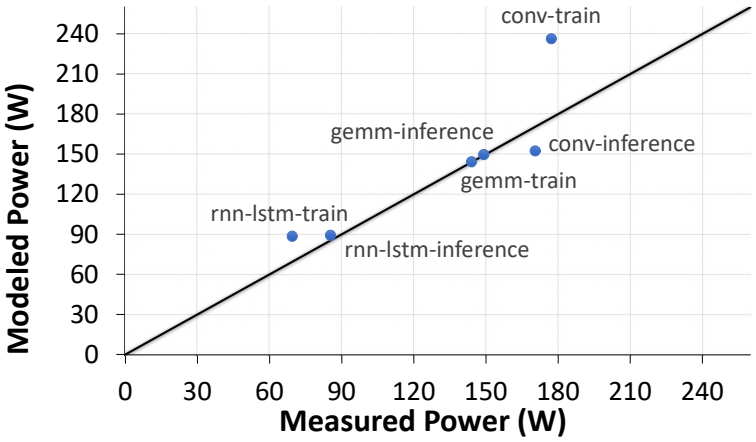


Figure 4.20: Correlation plot for DeepBench benchmarks.

AccelWattch to estimate its power consumption.

Even then, there is no guarantee the schedule is viable or that it matches the hardware-executed schedule. Kernel dependencies are unknown (cuDNN and cuBLAS are closed-source) and are thus not considered in the schedule. Also, cuDNN probes the hardware to decide which propagation algorithm to use, and may pick different algorithms for Accel-Sim and hardware execution, leading to hard-to-compare divergent behaviors. Similarly, AccelWattch HW reads hardware counters from Nsight, which is also constrained to serial kernel execution. Since we cannot design a validation process as precise as for the other benchmark suites, we exclude DeepBench from the validation suite and only present it as a case study.

We emphasize that these are not constraints imposed by AccelWattch, but by the existing performance simulators and profiling tools, and only apply to validating AccelWattch's estimates. AccelWattch can predict the power of individual cuDNN and cuBLAS kernels just fine. With these restrictions in mind, we perform a best-effort experiment in which we evaluate the application of AccelWattch on 6 DeepBench benchmarks: train and inference for CONV, RNN-LSTM, and GEMM. Figure 4.20 presents the results. Overall, AccelWattch SASS SIM obtains 12.79%

MAPE over Quadro GV100 hardware measurements for the DeepBench benchmarks.

4.2.7 Comparison to GPUWattch

To compare AccelWattch with GPUWattch, we apply GPUWattch's Fermi (NVIDIA GTX480) configuration to model Volta. As GPUWattch does not model tensor cores, we enhance it with AccelWattch's estimates for them. In fact, this is our Fermi starting point described in Section 4.2.5.4, with updates for the components that GPUWattch does not model (i.e., tensor cores). Running SASS and PTX simulations with this configuration for Volta architectures results in a MAPE of 219% and 225%, respectively, on the same validation suite of kernels. GPUWattch calculates unrealistically high power consumption for all kernels. The average power consumption it estimates is 530 W, with all but three of the kernels scoring above 300 W and a maximum of 926 W.

Moreover, in some cases GPUWattch reports unrealistic power consumption for particular components. For example, GPUWattch reports that constant and static power together account for 10.45 W on all validation kernels, which corresponds to 2.4% of the total system power on average. This contradicts our hardware power measurements on Volta, where even the lightest workload possible at the lowest frequency setting consumes >30 W. In addition, GPUWattch estimates that an average 14% of the total system power (including DRAM) is spent on INT_MUL units, compared to 1.4–1.8% in all AccelWattch variants. We believe the high power consumption that GPUWattch attributes to multipliers is unrealistic, as they would consume more power than GPUWattch's estimate for the register file (9.1%), pointing to a GPUWattch inaccuracy. Another notable difference includes GPUWattch's estimate of 27% of the system power spent on DRAM, compared to 8.4–9% in AccelWattch.

4.2.8 Related Work

GPU architecture research has been largely enabled by event-driven cycle-accurate simulators such as GPGPU-Sim [118], Multi2Sim [152], and MGPUSim [153]. The lack of a fast, SASS-capable simulator was only recently resolved by Accel-Sim [24], into which AccelWattch integrates. Validating such tools against real hardware has always been a crucial component of performance modeling research. Similar to our work, prior studies [154]–[159] present validation methodologies integrated into contemporary CPU simulators.

Wattch [160], McPAT [142], and SimplePower [161] are robust cycle-accurate CPU power modeling frameworks that have enabled a wide range of architecture-level research. Xi *et al.* [162] provided insightful guidelines for creating accurate power models with McPAT, including the use of analytical modeling for power gating and targeted microbenchmarking, which are followed by AccelWattch.

Cycle-accurate GPU power models based on McPAT (Lim *et al.* [163], GPUWattch [124], GPUSimPow [164]) model a decade-old Fermi architecture [144] at the PTX ISA level only. GPUWattch and GPUSimPow estimate constant power based on Eq. (4.3), a methodology no longer applicable to modern GPUs. AccelWattch rectifies this problem and is substantially more accurate than GPUWattch for modern architectures.

The IPP [129] analytic power model achieves high accuracy, but requires source-level PTX analysis that is infeasible for large or closed-source workloads, and PTX may not accurately correspond to real hardware activity [146]. Guerreiro *et al.* [126] present an analytical model that accurately predicts the power consumption of a GPU given a voltage-frequency setting. However, it provides a fixed power component encompassing static, constant and idle SM power, does not account for component power gating, and can only model architectures with a silicon implementation. AccelWattch is not similarly constrained, and also models 25 microarchitectural power

components compared to 8 in Guerreiro *et al*.

In general, analytical models (GPUJoule [127], IPP [129], Guerreiro *et al.* [126]) can only capture program-level average power consumption, which hinders research that requires cycle-accurate simulation. Meanwhile, AccelWattch can provide a power trace at cycle-level granularity. AccelWattch is the first GPU power model, to the best of our knowledge, to have power components mapped to the SASS machine ISA instructions. Owing to this salient feature, AccelWattch can estimate the power consumption of closed-source GPU workloads that contain hand-tuned assembly.

4.2.9 Conclusions

There is a need for robust tools that will enable GPU architects to quickly model both the performance and the power consumption of modern GPUs. In this work, we introduce AccelWattch, a configurable cycle-level power model for modern GPUs that can be directed by emulation and trace-driven simulation environments, hardware performance counters, or a combination of the two, striking a balance between model accuracy and performance modeling effort. AccelWattch is the only open-source tool capable of modeling closed-source workloads with hand-tuned SASS instructions.

We infer, for the first time to our knowledge, how modern GPUs power-gate chip-wide, SM-wide and lane-specific hardware components, and introduce an analytic power model that accurately captures the combined effects of power gating, thread divergence, intra-warp functional unit overlap, and variable SM occupancy. We also introduce a DVFS-aware methodology for modeling constant power. We integrate AccelWattch with Accel-Sim and GPGPU-Sim to facilitate its widespread use and release it in the public domain, along with its microbenchmarks and support infrastructure, as an open-source research tool for the computer architecture community. We ex-

tensively validate AccelWattch and find that it is within 7.5–9.2 ± 2.1–3.1% of hardware power measurements on a NVIDIA Volta Quandro GV100 GPU. Finally, we demonstrate that AccelWattch can enable reliable design space exploration.

CHAPTER 5
CONCLUSIONS AND FUTURE WORK

This chapter concludes my dissertation by summarizing my contributions and discusses directions for future work.

5.1 Conclusions

Modern systems have become increasingly more parallel since the breakdown of Dennard Scaling. However, today's hardware and software do not fully take advantage of these parallel capabilities, resulting in a lot of untapped system performance and energy efficiency. My dissertation focuses on improving the performance and energy efficiency of modern systems by leveraging information across system abstraction layers to extract latent parallelism in hardware and software.

One avenue of parallelism in modern systems is at the SIMD level. Maximizing the utilization of these SIMD units is necessary to maximize total system throughput in modern CPUs. To make today's software explicitly target these SIMD units and improve SIMD utilization, I propose Parsimony [25]. Parsimony is a well-specified programming model and compiler framework designed to remain fully compatible with standard language semantics and compiler flows. We demonstrate a prototype implementation of Parsimony in LLVM, with performance results showing that our SPMD variant performs as well as state-of-the-art SPMD frameworks (i.e., `ispc`) and custom AVX-512 code, without requiring the use of a specialized programming language or compiler.

Improving SIMD utilization alone is not enough to maximize system throughput. Unfortunately, we are still bottlenecked by the performance of the memory subsystem for several important applications. To improve memory performance, I turn towards increasing memory level parallelism

without increasing programmer burden by closing the performance gap between strong and weak memory consistency models. I propose Hybrid Consistency (HC) [68] based on the observation that we can allow memory operations to be executed out-of-order when their reordering does not affect observable program behavior. HC is an efficient hardware design that blends strong and weak MCMs by enabling a fine grained non-speculative reordering of memory operations at the load and store buffers. We demonstrate that HC's dynamic temporality-aware classification can achieve the performance benefits of memory reordering with a cache line granularity classification scheme without needing to keep track of state information at the cache line level. HC outperforms the current state-of-the-art dynamic memory reordering scheme by 24% whilst maintaining a complexity-effective design with minimal area overhead and negligible energy overhead.

The techniques I have discussed so far aim to improve system performance. However, improving performance alone is not enough; we also need to improve system energy efficiency to be able to meet today's performance demands while staying within a practical power budget. It is not surprising that performance per watt has become a fundamental metric for evaluating the efficiency of modern systems, especially highly parallel accelerators such as GPUs. I propose to improve GPU energy efficiency by leveraging the parallelism within GPU computation structures such as arithmetic units and introduce ST^2 adders. ST^2 is a speculative adder design that exploits spatio-temporal value correlation to perform carry speculation and reduce power consumption. ST^2 adders guarantee correctness and outperform state-of-the-art designs. Furthermore, we propose ST^2 GPU [105], an architecture that integrates ST^2 adders and carry speculation units into the warp pipeline, and show that it achieves significant energy savings with negligible overheads.

To evaluate the performance and energy efficiency of hardware advancements such as ST^2 GPU, architects need robust tools that will allow them to quickly model both the performance and the power consumption of their designs. However, while GPU performance modeling has advanced

in great strides, the community lacks an accurate power model for modern GPUs. I address this issue by introducing AccelWattch [121]. AccelWattch is a cycle-level power model for modern GPU architectures. We validate AccelWattch and show that it achieves a high correlation with hardware measurements. Additionally, we demonstrate that AccelWattch enables reliable design space exploration.

5.2 Other Contributions from Collaborative Work

To further improve memory performance by leveraging parallelism in memory, in collaboration with other researchers, I propose WARDen [96]. WARDen is a cache coherence protocol that leverages properties of High-Level Parallel Languages (HLPs) to selectively deactivate cache coherence when not required. We demonstrate that WARDen improves the performance of HLPL benchmarks by 46% and reduces energy by 23% due to the elimination of unnecessary data movement and coherence messages.

5.3 Future Directions

Hybrid Consistency (HC) [68] is an efficient hardware design that blends strong and weak MCMs to extract memory level parallelism. As discussed in Section 3.1.3, HC allows eager re-classifications to enable further memory reordering. As discussed in Section 3.1.3, we apply the same limit for the number of eager re-classifications (*PRV* to *PRV* transitions) to all HC regions. However, each HC region might benefit from a different re-classification threshold. A direction to extend HC would be to design an adaptive re-classification scheme that uses a cost model to dynamically adjusts the re-classification threshold for each HC region. The adaptive re-classification scheme could rely on past state transition behavior of HC regions to predict their future behavior. For instance, the scheme could decide to stop further re-classifications of a HC region if its prior re-classification

costs already outweigh the performance benefit of allowing the previous re-classifications; i.e., the HC region did not see enough reuse by a single thread to justify further re-classifications.

HC is a dynamic OS/hardware-only approach to enable memory reordering. The compiler, or programming language, or even the user could provide HC with a fixed decision point to statically classify accesses as reorder-safe or reorder-unsafe. This could allow us to save the limited hardware resources for only accesses that we cannot classify statically. We could leverage programming model properties or have user annotations to mark entire code regions as reorder-safe; i.e., all accesses to reorder-safe regions can execute out-of-order. An interesting future direction would be to leverage Parsimony's [25] SPMD programming model guarantees to get a fixed decision point for access classification. Parsimony does not provide any ordering guarantees among its gangs. Thus, Parsimony gangs could be running on entirely different OS threads on different processing cores with no gurantees provided to the programmer about ordering among these gangs. Thus, all memory accesses within Parsimony's SPMD regions could be considered as reorder-safe by HC and be allowed to execute out of order. Care must be taken to ensure sequential semantics by not reordering a memory accesses within a Parsimony SPMD region with another access to the same memory location outside the SPMD region (scalar code). One possible HC design that leverages Parsimony's programming model guarantees could be a Parsimony compiler extension that informs the hardware about the start and end of each SPMD region with a dedicated instruction. All memory operations issued out of the processor's ROB while within a SPMD region can be inserted into HC's logically weak-ordered load or store buffer. Memory barriers can be inserted at the boundaries of SPMD regions to preserve sequential semantics. Thus, a hardware/software co-design that allows for Parsimony and HC to work in cohesion can leverage SIMD-level parallelism within Parsimony gangs, and OS thread-level and memory-level parallelism across Parsimony gangs that execute on different processing cores.

5.4 Acknowledgements of Funding Sources

I thank the National Science Foundation (NSF), Northwestern University, NVIDIA, Intel, and Synopsys for supporting my thesis work. ST^2 GPU [105] was partially funded by NSF awards CCF-1453853, CNS-1763743. AccelWattch [121] was partially funded by NSF awards CCF-1453853, CNS- 1763743, CCF-1910924, and by a Discovery Grant from the Natural Sciences and Engineering Research Council of Canada. Parsimony [25] was partially funded by NSF awards CCF-2119069, CCF-2028851, and CNS-1763743. HC [68] was partially funded by NSF award CCF-2119069, and by the Computer Science Department at Northwestern University. WARDen [96] was partially supported by the Exascale Computing Project (17-SC-20-SC), a collaborative effort of the U.S. Department of Energy Office of Science and the National Nuclear Security Administration, by the U.S. Department of Energy, Office of Science, under Contract DE-AC02-06CH11357, and by NSF awards CNS-1763743, CCF-2028851, CCF-2119069, CCF- 2115104, CCF-2119352, CCF-1901381, CCF-2107241, CCF- 2107042, CCF-1908488, and CCF-2118708.

REFERENCES

[1] R. Dennard, F. Gaensslen, H.-N. Yu, V. Rideout, E. Bassous, and A. LeBlanc, "Design of ion-implanted mosfet's with very small physical dimensions," *IEEE Journal of Solid-State Circuits*, vol. 9, no. 5, pp. 256–268, 1974.

[2] S. Borkar and A. A. Chien, "The future of microprocessors," *Commun. ACM*, vol. 54, no. 5, pp. 67–77, May 2011.

[3] Intel, *Intel® 64 and ia-32 architectures software developer's manual - Volume 1-4*, https://cdrdv2.intel.com/v1/dl/getContent/671200, Accessed: 2022-11-9, 2022.

[4] Ribbens, Cal, *High Performance Computing*, https://people.cs.vt.edu/~ribbens/papers/vtreview.html, Accessed: 2022-11-9, 2022.

[5] OpenMP Architecture Review Board, *OpenMP Application Program Interface Version 4.0*, https://www.openmp.org/wp-content/uploads/OpenMP4.0.0.pdf, Accessed: 2022-1-9, May 2013.

[6] Rupp, Karl, *42 Years of Microprocessor Trend Data*, https://www.karlrupp.net/2018/02/42-years-of-microprocessor-trend-data/, Accessed: 2022-11-9, 2022.

[7] WikiChip, *Cascade Lake - Microarchitectures - Intel*, https://en.wikichip.org/wiki/intel/microarchitectures/cascade_lake, Accessed: 2022-11-9, 2019.

[8] N. Stephens, S. Biles, M. Boettcher, *et al.*, "The arm scalable vector extension," *IEEE micro*, vol. 37, no. 2, pp. 26–39, 2017.

[9] RISC, *RISC-V "V" Vector Extension*, https://github.com/riscv/riscv-v-spec/releases/tag/v1.0, Accessed: 2022-1-9, 2021.

[10] M. Pharr and W. R. Mark, "Ispc: A spmd compiler for high-performance cpu programming," in *2012 Innovative Parallel Computing (InPar)*, 2012, pp. 1–13.

[11] C. Lattner, "LLVM: An Infrastructure for Multi-Stage Optimization," M.S. thesis, Computer Science Dept., University of Illinois at Urbana-Champaign, Urbana, IL, Dec. 2002.

[12] S. Salehian and Y. Yan, "Evaluation of knight landing high bandwidth memory for hpc workloads," in *Proceedings of the Seventh Workshop on Irregular Applications: Architectures and Algorithms*, ser. IA3'17, Denver, CO, USA, 2017.

[13] S. A. McKee, "Reflections on the memory wall," in *Proceedings of the 1st Conference on Computing Frontiers*, ser. CF '04, Ischia, Italy, 2004, p. 162.

[14] A. Rodrigues, M. Gokhale, and G. Voskuilen, "Towards a scatter-gather architecture: Hardware and software issues," in *Proceedings of the International Symposium on Memory Systems*, ser. MEMSYS '19, Washington, District of Columbia, USA, 2019, pp. 261–271.

[15] S. Owens, S. Sarkar, and P. Sewell, "A better x86 memory model: X86-tso," in *Theorem Proving in Higher Order Logics*, S. Berghofer, T. Nipkow, C. Urban, and M. Wenzel, Eds., 2009, pp. 391–407.

[16] IBM, *Power ISA™Version 3.0 B. Technical Report.* `https://ibm.ent.box.com/s/1hzcwkwf8rbju5h9iyf44wm94amnlcrv`, Accessed: 2022-1-9, 2017.

[17] S. Sarkar, P. Sewell, J. Alglave, L. Maranget, and D. Williams, "Understanding power multiprocessors," ser. PLDI '11, San Jose, California, USA, 2011, pp. 175–186.

[18] J. Alglave, L. Maranget, and M. Tautschnig, "Herding cats: Modelling, simulation, testing, and data mining for weak memory," *ACM Trans. Program. Lang. Syst.*, vol. 36, no. 2, Jul. 2014.

[19] TOP500.org, *TOP500 List*, `https://www.top500.org/lists/top500/2022/11/`, Accessed: 2022-11-9, Nov. 2022.

[20] A. Snell and L. Segervall, *HPC application support for GPU computing*, `https://www.nvidia.com/content/intersect-360-HPC-application-support.pdf`, Accessed: 2022-11-9, Nov. 2017.

[21] Forbes, *NVIDIA Dominates The Market For Cloud AI Accelerators More Than You Think*, `https://www.forbes.com/sites/paulteich/2019/06/17/nvidia-dominates-the-market-for-cloud-ai-accelerators-more-than-you-think/#676dea375edb`, Accessed: 2022-11-9, Jun. 2019.

[22] NVIDIA, *Whitepaper: NVIDIA Telsa V100 GPU Architecture*, http://images.nvidia.com/content/volta-architecture/pdf/volta-architecture-whitepaper.pdf, Accessed: 2022-11-9, Aug. 2017.

[23] NVIDIA, *Whitepaper: NVIDIA Tesla P100*, https://images.nvidia.com/content/pdf/tesla/whitepaper/pascal-architecture-whitepaper.pdf, Accessed: 2022-11-9, 2016.

[24] M. Khairy, Z. Shen, T. M. Aamodt, and T. G. Rogers, "Accel-sim: An extensible simulation framework for validated gpu modeling," in *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*, 2020, pp. 473–486.

[25] V. Kandiah, D. Lustig, O. Villa, D. Nellans, and N. Hardavellas, "Parsimony: Enabling SIMD/Vector Programming in Standard Compiler Flows," in *IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, 2023, To Appear.

[26] P. Papaphilippou, P. H. Kelly, and W. Luk, "Extending the risc-v isa for exploring advanced reconfigurable simd instructions," *arXiv preprint arXiv:2106.07456*, 2021.

[27] G. Tagliavini, S. Mach, D. Rossi, A. Marongiu, and L. Benini, "Design and evaluation of smallfloat simd extensions to the risc-v isa," in *2019 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, 2019, pp. 654–657.

[28] K. Patsidis, C. Nicopoulos, G. C. Sirakoulis, and G. Dimitrakopoulos, "Risc-v2: A scalable risc-v vector processor," in *2020 IEEE International Symposium on Circuits and Systems (ISCAS)*, 2020, pp. 1–5.

[29] Intel, *Intel® 64 and ia-32 architectures software developer's manual - Volume 1-4*, https://cdrdv2.intel.com/v1/dl/getContent/671200, Accessed: 2022-1-9, 2022.

[30] N. Stephens, S. Biles, M. Boettcher, *et al.*, "The arm scalable vector extension," *IEEE micro*, vol. 37, no. 2, pp. 26–39, 2017.

[31] RISC, *RISC-V "V" Vector Extension*, https://github.com/riscv/riscv-v-spec/releases/tag/v1.0, Accessed: 2022-1-9, 2021.

[32] O. Reiche, C. Kobylko, F. Hannig, and J. Teich, "Auto-vectorization for image processing dsls," in *Proceedings of the 18th ACM SIGPLAN/SIGBED Conference on Languages, Compilers, and Tools for Embedded Systems*, 2017, pp. 21–30.

[33] A. Pohl, B. Cosenza, M. A. Mesa, C. C. Chi, and B. Juurlink, "An evaluation of current simd programming models for c++," in *Proceedings of the 3rd Workshop on Programming Models for SIMD/Vector Processing*, 2016, pp. 1–8.

[34] Wenzel Jakob, *Enoki: structured vectorization and differentiation on modern processor architectures*, `https://github.com/mitsuba-renderer/enoki`, Accessed: 2022-1-9, 2019.

[35] N. Shibata and F. Petrogalli, "Sleef: A portable vectorized library of c standard mathematical functions," *IEEE Transactions on Parallel and Distributed Systems*, vol. 31, no. 6, pp. 1316–1327, 2020.

[36] NVIDIA, *CUDA C programming guide, v10.2*, `https://docs.nvidia.com/pdf/CUDA_C_Programming_Guide.pdf`, Accessed: 2020-4-21, May 2019.

[37] R. Karrenberg, "Whole-function vectorization," in *Automatic SIMD vectorization of SSA-based control flow graphs*, 2015, pp. 85–125.

[38] S. Moll and S. Hack, "Partial control-flow linearization," *ACM SIGPLAN Notices*, vol. 53, no. 4, pp. 543–556, 2018.

[39] R. Allen and K. Kennedy, "Automatic translation of fortran programs to vector form," *ACM Transactions on Programming Languages and Systems (TOPLAS)*, vol. 9, no. 4, pp. 491–542, 1987.

[40] K. Trifunovic, D. Nuzman, A. Cohen, A. Zaks, and I. Rosen, "Polyhedral-model guided loop-nest auto-vectorization," in *2009 18th International Conference on Parallel Architectures and Compilation Techniques*, 2009, pp. 327–337.

[41] D. Nuzman, I. Rosen, and A. Zaks, "Auto-vectorization of interleaved data for simd," *ACM SIGPLAN Notices*, vol. 41, no. 6, pp. 132–143, 2006.

[42] Standard C++ Foundation, *ISO International Standard ISO/IEC 14882:2020(E) – Programming Language C++*, `https://isocpp.org/std/the-standard`, Accessed: 2022-1-9, 2020.

[43] D. Nuzman and A. Zaks, "Outer-loop vectorization: Revisited for short simd architectures," in *Proceedings of the 17th international conference on Parallel architectures and compilation techniques*, 2008, pp. 2–11.

[44] S. Larsen and S. Amarasinghe, "Exploiting superword level parallelism with multimedia instruction sets," *ACM SIGPLAN Notices*, vol. 35, no. 5, pp. 145–156, 2000.

[45] V. Porpodas, A. Magni, and T. M. Jones, "Pslp: Padded slp automatic vectorization," in *2015 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, 2015, pp. 190–201.

[46] J. Ren, S. Rajbhandari, R. Y. Aminabadi, *et al.*, "ZeRO-Offload: Democratizing Billion-Scale model training," in *2021 USENIX Annual Technical Conference (USENIX ATC 21)*, 2021, pp. 551–564.

[47] Y. Chen, C. Mendis, M. Carbin, and S. Amarasinghe, "Vegen: A vectorizer generator for simd and beyond," in *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, 2021, pp. 902–914.

[48] W. W. Fung and T. M. Aamodt, "Thread block compaction for efficient simt control flow," in *2011 IEEE 17th International Symposium on High Performance Computer Architecture*, 2011, pp. 25–36.

[49] Y. Lee, V. Grover, R. Krashinsky, M. Stephenson, S. W. Keckler, and K. Asanović, "Exploring the Design Space of SPMD Divergence Management on Data-Parallel Architectures," in *Proceedings of the 47th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2014, pp. 101–113.

[50] Intel, *ispc: Intel SPMD Program Compiler*, `https://github.com/ispc/ispc/blob/v1.18.0/src/opt.cpp#L466`, Accessed: 2022-1-9, 2022.

[51] LLVM Community, *LLVM 15.0.1*, `https://github.com/llvm/llvm-project/releases/tag/llvmorg-15.0.1`, Accessed: 2022-1-10, 2022.

[52] B. Coutinho, D. Sampaio, F. M. Q. Pereira, and W. Meira Jr, "Divergence analysis and optimizations," in *2011 International Conference on Parallel Architectures and Compilation Techniques*, 2011, pp. 320–329.

[53] D. Sampaio, R. Martins, S. Collange, and F. M. Q. Pereira, "Divergence analysis with affine constraints," in *2012 IEEE 24th International Symposium on Computer Architecture and High Performance Computing*, 2012, pp. 67–74.

[54] D. Sampaio, R. M. d. Souza, C. Collange, and F. M. Q. Pereira, "Divergence analysis," *ACM Transactions on Programming Languages and Systems (TOPLAS)*, vol. 35, no. 4, pp. 1–36, 2014.

[55] Y. Lee, R. Krashinsky, V. Grover, S. W. Keckler, and K. Asanović, "Convergence and Scalarization for Data-Parallel Architectures," in *Proceedings of the 2013 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, 2013, pp. 1–11.

[56] L. d. Moura and N. Bjørner, "Z3: An efficient smt solver," in *International conference on Tools and Algorithms for the Construction and Analysis of Systems*, 2008, pp. 337–340.

[57] Ihar Yermalayeu et al., *The Simd Library*, `https://github.com/ermig1979/Simd`, Accessed: 2022-1-9, 2019.

[58] Intel, *ispc: Intel SPMD Program Compiler*, `https://github.com/ispc/ispc/releases/tag/v1.18.0`, Accessed: 2022-1-9, 2022.

[59] P. B. Schneck, "Automatic recognition of vector and parallel operations in a higher level language," *ACM SIGPLAN Notices*, vol. 7, no. 11, pp. 45–52, 1972.

[60] D. Wedel, "Fortran for the texas instruments asc system," *ACM SIGPLAN Notices*, vol. 10, no. 3, pp. 119–132, 1975.

[61] D. J. Kuck, R. H. Kuhn, B. Leasure, and M. Wolfe, "The Structure of an Advanced Vectorizer for Pipelined Processors," in *Proceedings of the 4th International Conference on Computer Software and Applications (COMPSAC)*, 1980, pp. 709–715.

[62] R. G. Scarborough and H. G. Kolsky, "A vectorizing fortran compiler," *IBM Journal of Research and Development*, vol. 30, no. 2, pp. 163–171, 1986.

[63] A. E. Eichenberger, P. Wu, and K. O'brien, "Vectorization for simd architectures with alignment constraints," *ACM SIGPLAN Notices*, vol. 39, no. 6, pp. 82–93, 2004.

[64] Y. Chen, C. Mendis, and S. Amarasinghe, "All you need is superword-level parallelism: Systematic control-flow vectorization with slp," in *Proceedings of the 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation*, 2022, pp. 301–315.

[65] R. Leißa, I. Haffner, and S. Hack, "Sierra: A simd extension for c++," in *Proceedings of the 2014 Workshop on Programming models for SIMD/Vector processing*, 2014, pp. 17–24.

[66] M. Haidl, S. Moll, L. Klein, H. Sun, S. Hack, and S. Gorlatch, "Pacxxv2+ rv: An llvm-based portable high-performance programming model," in *Proceedings of the Fourth Workshop on the LLVM Compiler Infrastructure in HPC*, 2017, pp. 1–12.

[67] J. A. Stratton, V. Grover, J. Marathe, *et al.*, "Efficient Compilation of Fine-Grained SPMD-Threaded Programs for Multicore CPUs," in *Proceedings of the 8th Annual IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, 2010, pp. 111–119.

[68] V. Kandiah, A. Patel, T. Dempski, and N. Hardavellas, "Hybrid Consistency: Fine grained Dynamic Blending of Memory Consistency Models," in *To Be Submitted IEEE/ACM International Symposium on Computer Architecture (ISCA)*, 2024, To Be Submitted.

[69] C. Bienia, S. Kumar, J. P. Singh, and K. Li, "The parsec benchmark suite: Characterization and architectural implications," in *Proceedings of the 17th International Conference on Parallel Architectures and Compilation Techniques*, ser. PACT '08, Toronto, Ontario, Canada: Association for Computing Machinery, 2008, pp. 72–81, ISBN: 9781605582825.

[70] D. Bailey, T. Harris, W. Saphir, R. Van Der Wijngaart, A. Woo, and M. Yarrow, "The nas parallel benchmarks 2.0," Technical Report NAS-95-020, NASA Ames Research Center, Tech. Rep., 1995.

[71] C. Pulte, S. Flur, W. Deacon, J. French, S. Sarkar, and P. Sewell, "Simplifying arm concurrency: Multicopy-atomic axiomatic and operational models for armv8," *Proceedings of the ACM on Programming Languages*, vol. 2, no. POPL, pp. 1–29, 2017.

[72] K. Gharachorloo, A. Gupta, and J. L. Hennessy, "Two techniques to enhance the performance of memory consistency models," in *International Conference on Parallel Processing*, 1991.

[73] A. Singh, S. Narayanasamy, D. Marino, T. Millstein, and M. Musuvathi, "End-to-end sequential consistency," in *2012 39th Annual International Symposium on Computer Architecture (ISCA)*, 2012, pp. 524–535.

[74] S. Singh, A. Jimborean, and A. Ros, "Regional out-of-order writes in total store order," in *Proceedings of the ACM International Conference on Parallel Architectures and Compilation Techniques*, ser. PACT '20, Virtual Event, GA, USA: Association for Computing Machinery, 2020, pp. 205–216, ISBN: 9781450380751.

[75] P. Ranganathan, V. S. Pai, and S. V. Adve, "Using speculative retirement and larger instruction windows to narrow the performance gap between memory consistency models," in *Proceedings of the Ninth Annual ACM Symposium on Parallel Algorithms and Architectures*, ser. SPAA '97, Newport, Rhode Island, USA: Association for Computing Machinery, 1997, pp. 199–210, ISBN: 0897918908.

[76] C. Gniady, B. Falsafi, and T. N. Vijaykumar, "Is sc + ilp = rc?" *SIGARCH Comput. Archit. News*, vol. 27, no. 2, pp. 162–171, May 1999.

[77] C. Gniady and B. Falsafi, "Speculative sequential consistency with little custom storage," in *Proceedings.International Conference on Parallel Architectures and Compilation Techniques*, 2002, pp. 179–188.

[78] T. F. Wenisch, A. Ailamaki, B. Falsafi, and A. Moshovos, "Mechanisms for store-wait-free multiprocessors," in *Proceedings of the 34th Annual International Symposium on Computer Architecture*, ser. ISCA '07, San Diego, California, USA: Association for Computing Machinery, 2007, pp. 266–277, ISBN: 9781595937063.

[79] L. Ceze, J. Tuck, P. Montesinos, and J. Torrellas, "Bulksc: Bulk enforcement of sequential consistency," in *Proceedings of the 34th Annual International Symposium on Computer Architecture*, ser. ISCA '07, San Diego, California, USA: Association for Computing Machinery, 2007, pp. 278–289, ISBN: 9781595937063.

[80] W. Ahn, S. Qi, M. Nicolaides, *et al.*, "Bulkcompiler: High-performance sequential consistency through cooperative compiler and hardware support," in *2009 42nd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2009, pp. 133–144.

[81] C. Blundell, M. M. Martin, and T. F. Wenisch, "Invisifence: Performance-transparent memory ordering in conventional multiprocessors," in *Proceedings of the 36th Annual International Symposium on Computer Architecture*, ser. ISCA '09, Austin, TX, USA: Association for Computing Machinery, 2009, pp. 233–244, ISBN: 9781605585260.

[82] Y. Duan, A. Muzahid, and J. Torrellas, "Weefence: Toward making fences free in tso," in *Proceedings of the 40th Annual International Symposium on Computer Architecture*, ser. ISCA '13, Tel-Aviv, Israel: Association for Computing Machinery, 2013, pp. 213–224, ISBN: 9781450320795.

[83] Lamport, "How to make a multiprocessor computer that correctly executes multiprocess programs," *IEEE Transactions on Computers*, vol. C-28, no. 9, pp. 690–691, 1979.

[84] D. L. Black, R. F. Rashid, D. B. Golub, and C. R. Hill, "Translation lookaside buffer consistency: A software approach," *SIGARCH Comput. Archit. News*, vol. 17, no. 2, pp. 113–122, Apr. 1989.

[85] C. Villavieja, V. Karakostas, L. Vilanova, *et al.*, "Didi: Mitigating the performance impact of tlb shootdowns using a shared tlb directory," in *2011 International Conference on Parallel Architectures and Compilation Techniques*, 2011, pp. 340–349.

[86] OSDev, *Paging*, `https://wiki.osdev.org/Paging`, Accessed: 2023-11-01.

[87] Intel, *Whitepaper: 5-level paging and 5-level ept*, `https://cdrdv2.intel.com/v1/dl/getContent/671442?fileName=5-level-paging-white-paper.pdf`, Accessed: 2023-11-01, 2017.

[88] T. E. Carlson, W. Heirman, and L. Eeckhout, "Sniper: Exploring the level of abstraction for scalable and accurate parallel multi-core simulation," in *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, ser. SC '11, Seattle, Washington, 2011, ISBN: 9781450307710.

[89] T. E. Carlson, W. Heirman, S. Eyerman, I. Hur, and L. Eeckhout, "An evaluation of high-level mechanistic core models," *ACM Transactions on Architecture and Code Optimization*, vol. 11, no. 3, Aug. 2014.

[90] N. Muralimanohar, R. Balasubramonian, and N. P. Jouppi, "Architecting efficient interconnects for large caches with cacti 6.0," *IEEE Micro*, vol. 28, no. 1, pp. 69–79, 2008.

[91] A. Ros and S. Kaxiras, "Non-speculative store coalescing in total store order," in *2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA)*, 2018, pp. 221–234.

[92] A. Ros, T. E. Carlson, M. Alipour, and S. Kaxiras, "Non-speculative load-load reordering in tso," in *2017 ACM/IEEE 44th Annual International Symposium on Computer Architecture (ISCA)*, 2017, pp. 187–200.

[93] A. Ros and S. Kaxiras, "The superfluous load queue," in *2018 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2018, pp. 95–107.

[94] N. Hardavellas, M. Ferdman, B. Falsafi, and A. Ailamaki, "Reactive nuca: Near-optimal block placement and replication in distributed caches," in *Proceedings of the 36th Annual*

International Symposium on Computer Architecture, ser. ISCA '09, Austin, TX, USA: Association for Computing Machinery, 2009, pp. 184–195, ISBN: 9781605585260.

[95] A. Esteve, A. Ros, A. Robles, and M. E. Gómez, "Tokentlb+cup: A token-based page classification with cooperative usage prediction," *IEEE Transactions on Parallel and Distributed Systems*, vol. 29, no. 5, pp. 1188–1201, 2018.

[96] M. Wilkins, S. Westrick, V. Kandiah, *et al.*, "Warden: Specializing cache coherence for high-level parallel languages," in *Proceedings of the 21st ACM/IEEE International Symposium on Code Generation and Optimization*, ser. CGO 2023, Montréal, QC, Canada: Association for Computing Machinery, 2023, pp. 122–135, ISBN: 9798400701016.

[97] J. Meng and K. Skadron, "Avoiding cache thrashing due to private data placement in last-level cache for manycore scaling," in *2009 IEEE International Conference on Computer Design*, 2009, pp. 282–288.

[98] S. H. Pugsley, J. B. Spjut, D. W. Nellans, and R. Balasubramonian, "Swel: Hardware cache coherence protocols to map shared data onto shared caches," in *2010 19th International Conference on Parallel Architectures and Compilation Techniques (PACT)*, 2010, pp. 465–475.

[99] B. Cuesta, A. Ros, M. E. Gómez, A. Robles, and J. Duato, "Increasing the effectiveness of directory caches by deactivating coherence for private memory blocks," in *2011 38th Annual International Symposium on Computer Architecture (ISCA)*, 2011, pp. 93–103.

[100] H. Hossain, S. Dwarkadas, and M. C. Huang, "Pops: Coherence protocol optimization for both private and shared data," in *2011 International Conference on Parallel Architectures and Compilation Techniques*, 2011, pp. 45–55.

[101] Y. Li, R. Melhem, and A. K. Jones, "Practically private: Enabling high performance cmps through compiler-assisted data classification," in *2012 21st International Conference on Parallel Architectures and Compilation Techniques (PACT)*, 2012, pp. 231–240.

[102] A. Ros, B. Cuesta, M. E. Gómez, A. Robles, and J. Duato, "Temporal-aware mechanism to detect private data in chip multiprocessors," in *2013 42nd International Conference on Parallel Processing*, 2013, pp. 562–571.

[103] A. Ros and A. Jimborean, "A hybrid static-dynamic classification for dual-consistency cache coherence," *IEEE Transactions on Parallel and Distributed Systems*, vol. 27, no. 11, pp. 3101–3115, 2016.

[104] A. Esteve, A. Ros, M. E. Gómez, A. Robles, and J. Duato, "Tlb-based temporality-aware classification in cmps with multilevel tlbs," *IEEE Transactions on Parallel and Distributed Systems*, vol. 28, no. 8, pp. 2401–2413, 2017.

[105] V. Kandiah, A. M. Gök, G. Tziantzioulis, and N. Hardavellas, "ST2 GPU: An Energy-Efficient GPU Design with Spatio-Temporal Shared-Thread Speculative Adders," in *2021 58th ACM/IEEE Design Automation Conference (DAC)*, 2021, pp. 271–276.

[106] S. Che, M. Boyer, J. Meng, *et al.*, "Rodinia: A benchmark suite for heterogeneous computing," in *IISWC*, 2009.

[107] NVIDIA, *CUDA-9.1_Samples*.

[108] J. A. Stratton, C. Rodrigues, I.-J. Sung, *et al.*, "Parboil: A revised benchmark suite for scientific and commercial throughput computing," in *IMPACT Technical Report, IMPACT-12-01, UIUC*, 2012.

[109] A. M. Gok and N. Hardavellas, "VaLHALLA: Variable latency history aware local-carry lazy adder," in *GLSVLSI*, Banff, Alberta, Canada, 2017, ISBN: 978-1-4503-4972-7.

[110] NVIDIA, *Parallel thread execution ISA version 7.0*, 2020.

[111] J. M. Rabaey, *Digital Integrated Circuits: A Design Perspective*. Prentice-Hall, 1996, ISBN: 0-13-178609-1.

[112] A. B. Kahng and S. Kang, "Accuracy-configurable adder for approximate arithmetic designs," in *DAC*, 2012.

[113] J. Hu and W. Qian, "A new approximate adder with low relative error and correct sign calculation," in *DATE*, 2015.

[114] X. Chen, A. M. Eltawil, and F. J. Kurdahi, "Low latency approximate adder for highly correlated input streams," in *ICCD*, 2017.

[115] G. Liu, Y. Tao, M. Tan, and Z. Zhang, "CASA: Correlation-aware speculative adders," in *ISLPED*, 2014.

[116] A. K. Verma, P. Brisk, and P. Ienne, "Variable latency speculative addition: A new paradigm for arithmetic circuit design," in *DATE*, 2008.

[117] O. J. Bedrij, "Carry-select adder," *IRE Trans. Electr. Computers*, 1962.

[118] A. Bakhoda, G. Yuan, W. Fung, H. Wong, and T. M. Aamodt, "Analyzing CUDA workloads using a detailed GPU simulator," in *ISPASS*, 2009.

[119] M. Khairy, A. Jain, T. M. Aamodt, and T. G. Rogers, "A detailed model for contemporary GPU memory systems," in *ISPASS*, 2019.

[120] Synopsys, *Synopsys DesignWare library*.

[121] V. Kandiah, S. Peverelle, M. Khairy, *et al.*, "AccelWattch: A Power Modeling Framework for Modern GPUs," in *MICRO-54: 54th Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO '21, Virtual Event, Greece, 2021, pp. 738–753.

[122] W. Liu, E. Salman, C. Sitik, and B. Taskin, "Enhanced level shifter for multi-voltage operation," in *ISCAS*, 2015.

[123] A. Shapiro and E. G. Friedman, "Power efficient level shifter for 16 nm FinFET near threshold circuits," *IEEE Trans. VLSI Systems*, 2016.

[124] J. Leng, T. Hetherington, A. ElTantawy, *et al.*, "GPUWattch: Enabling energy optimizations in GPGPUs," in *ISCA*, Tel-Aviv, Israel, 2013, ISBN: 978-1-4503-2079-5.

[125] NVIDIA, *Whitepaper: Nvidia turing gpu architecture*, `https://www.nvidia.com/content/dam/en-zz/Solutions/design-visualization/technologies/turing-architecture/NVIDIA-Turing-Architecture-Whitepaper.pdf`, Accessed: 2020-11-21, 2018.

[126] J. Guerreiro, A. Ilic, N. Roma, and P. Tomas, "Gpgpu power modeling for multi-domain voltage-frequency scaling," in *2018 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, 2018, pp. 789–800.

[127] A. Arunkumar, E. Bolotin, D. Nellans, and C.-J. Wu, "Understanding the future of energy efficiency in multi-module gpus," in *2019 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, 2019, pp. 519–532.

[128] G. Wu, J. L. Greathouse, A. Lyashevsky, N. Jayasena, and D. Chiou, "Gpgpu performance and power estimation using machine learning," in *2015 IEEE 21st International Symposium on High Performance Computer Architecture (HPCA)*, 2015, pp. 564–576.

[129] S. Hong and H. Kim, "An integrated gpu power and performance model," in *Proceedings of the 37th Annual International Symposium on Computer Architecture*, ser. ISCA '10, Saint-Malo, France, 2010, pp. 280–289, ISBN: 9781450300537.

[130] Y. Zhang, Y. Hu, B. Li, and L. Peng, "Performance and power analysis of ati gpu: A statistical approach," in *2011 IEEE Sixth International Conference on Networking, Architecture, and Storage*, 2011, pp. 149–158.

[131] NVIDIA, *CUTLASS: CUDA template library for dense linear algebra at all levels and scales*, https://github.com/NVIDIA/cutlass, Accessed: 2020-11-24, 2019.

[132] A. de Myttenaere, B. Golden, B. Le Grand, and F. Rossi, "Mean absolute percentage error for regression models," *Neurocomputing*, vol. 192, pp. 38–48, 2016.

[133] S. Narang, *Deepbench*, https://svail.github.io/DeepBench/, Accessed: 2021-09-08, 2016.

[134] NVIDIA, *Instruction Set Reference*, https://docs.nvidia.com/cuda/cuda-binary-utilities/index.html#instruction-set-ref, Accessed: 2020-6-5, Jun. 2020.

[135] Z. Jia, M. Maggioni, B. Staiger, and D. P. Scarpazza, "Dissecting the NVIDIA volta GPU architecture via microbenchmarking," *CoRR*, vol. abs/1804.06826, Apr. 2018. arXiv: 1804.06826.

[136] NVIDIA, *NVML API Reference*, https://docs.nvidia.com/deploy/nvml-api/nvml-api-reference.html, Accessed: 2020-11-24, May 2019.

[137] NVIDIA, *nvidia-smi - NVIDIA System Management Interface*, http://developer.download.nvidia.com/compute/DCGM/docs/nvidia-smi-367.38.pdf, Accessed: 2020-11-24, Jul. 2016.

[138] NVIDIA, *Nsight Compute*, https://docs.nvidia.com/nsight-compute/NsightCompute/index.html, Accessed: 2021-9-5, Jul. 2021.

[139] S. Jain, S. Khare, S. Yada, *et al.*, "A 280mv-to-1.2v wide-operating-range ia-32 processor in 32nm cmos," in *2012 IEEE International Solid-State Circuits Conference*, 2012, pp. 66–68.

[140] B. Zimmer, Y. Lee, A. Puggelli, *et al.*, "A risc-v vector processor with tightly-integrated switched-capacitor dc-dc converters in 28nm fdsoi," in *2015 Symposium on VLSI Circuits (VLSI Circuits)*, 2015, pp. C316–C317.

[141] S. Boyd and L. Vandenberghe, *Convex Optimization*. Cambridge, UK: Cambridge University Press, 2004.

[142] S. Li, J. H. Ahn, R. D. Strong, J. B. Brockman, D. M. Tullsen, and N. P. Jouppi, "Mcpat: An integrated power, area, and timing modeling framework for multicore and manycore architectures," in *2009 42nd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2009, pp. 469–480.

[143] O. Villa, M. Stephenson, D. W. Nellans, and S. W. Keckler, "Nvbit: A dynamic binary instrumentation framework for NVIDIA gpus," in *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2019, pp. 372–383.

[144] NVIDIA, *Whitepaper: NVIDIA's Next Generation CUDA Compute Architecture: Fermi*, https://www.nvidia.com/content/PDF/fermi_white_papers/NVIDIA_Fermi_Compute_Architecture_Whitepaper.pdf, Accessed: 2020-6-5, 2009.

[145] NVIDIA, *CUDA Compiler Driver NVCC, v9.1*, https://docs.nvidia.com/cuda/archive/9.1/cuda-compiler-driver-nvcc/index.html, Accessed: 2020-4-21, May 2019.

[146] A. Gutierrez, B. M. Beckmann, A. Dutu, *et al.*, "Lost in abstraction: Pitfalls of analyzing gpus at the intermediate language level," in *2018 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, 2018, pp. 608–619.

[147] T. Nowatzki, J. Menon, C.-H. Ho, and K. Sankaralingam, "Architectural simulators considered harmful," *IEEE Micro*, vol. 35, no. 6, pp. 4–12, 2015.

[148] IEEE, *International Roadmap for Devices and Systems*, https://irds.ieee.org/editions/2016/, Accessed: 2020-11-24, 2016.

[149] Forbes, *NVIDIA Dominates The Market For Cloud AI Accelerators More Than You Think*, https://www.forbes.com/sites/paulteich/2019/06/17/nvidia-dominates-the-market-for-cloud-ai-accelerators-more-than-you-think/#676dea375edb, Accessed: 2020-4-21, Jun. 2019.

[150] NVIDIA, *cuBLAS*, `https://developer.nvidia.com/cublas/`, Accessed: 2021-09-08, 2021.

[151] S. Chetlur, C. Woolley, P. Vandermersch, *et al.*, "cuDNN: Efficient primitives for deep learning," *CoRR*, vol. abs/1410.0759, 2014. arXiv: `1410.0759`.

[152] R. Ubal, B. Jang, P. Mistry, D. Schaa, and D. Kaeli, "Multi2sim: A simulation framework for cpu-gpu computing," in *2012 21st International Conference on Parallel Architectures and Compilation Techniques (PACT)*, 2012, pp. 335–344.

[153] Y. Sun, T. Baruah, S. A. Mojumder, *et al.*, "Mgpusim: Enabling multi-gpu performance modeling and optimization," in *2019 ACM/IEEE 46th Annual International Symposium on Computer Architecture (ISCA)*, 2019, pp. 197–209.

[154] J. Gibson, R. Kunz, D. Ofelt, M. Horowitz, J. Hennessy, and M. Heinrich, "Flash vs. (simulated) flash: Closing the simulation loop," in *Proceedings of the 9th International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS IX, Cambridge, Massachusetts, USA, 2000, pp. 49–58, ISBN: 1581133170.

[155] R. Desikan, D. Burger, and S. W. Keckler, "Measuring experimental error in microprocessor simulation," in *Proceedings of the 28th Annual International Symposium on Computer Architecture*, ser. ISCA '01, Göteborg, Sweden, 2001, pp. 266–277, ISBN: 0769511627.

[156] A. Gutierrez, J. Pusdesris, R. G. Dreslinski, *et al.*, "Sources of error in full-system simulation," in *2014 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, 2014, pp. 13–22.

[157] M. Walker, S. Bischoff, S. Diestelhorst, G. Merrett, and B. Al-Hashimi, "Hardware-validated cpu performance and energy modelling," in *2018 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, 2018, pp. 44–53.

[158] A. Adileh, C. González-Álvarez, J. Miguel De Haro Ruiz, and L. Eeckhout, "Racing to hardware-validated simulation," in *2019 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, 2019, pp. 58–67.

[159] T. E. Carlson, W. Heirman, S. Eyerman, I. Hur, and L. Eeckhout, "An evaluation of high-level mechanistic core models," *ACM Transactions on Architecture and Code Optimization*, vol. 11, no. 3, Aug. 2014.

[160] D. Brooks, V. Tiwari, and M. Martonosi, "Wattch: A framework for architectural-level power analysis and optimizations," in *Proceedings of the 27th Annual International Symposium on Computer Architecture*, ser. ISCA '00, Vancouver, British Columbia, Canada, 2000, pp. 83–94, ISBN: 1581132328.

[161] W. Ye, N. Vijaykrishnan, M. T. Kandemir, and M. J. Irwin, "The design and use of simplepower: A cycle-accurate energy estimation tool," in *Proceedings 37th Design Automation Conference*, 2000, pp. 340–345.

[162] S. L. Xi, H. Jacobson, P. Bose, G.-Y. Wei, and D. Brooks, "Quantifying sources of error in mcpat and potential impacts on architectural studies," in *2015 IEEE 21st International Symposium on High Performance Computer Architecture (HPCA)*, 2015, pp. 577–589.

[163] J. Lim, N. B. Lakshminarayana, H. Kim, W. Song, S. Yalamanchili, and W. Sung, "Power modeling for gpu architectures using mcpat," *ACM Trans. Des. Autom. Electron. Syst.*, vol. 19, no. 3, Jun. 2014.

[164] J. Lucas, S. Lal, M. Andersch, M. Alvarez-Mesa, and B. Juurlink, "How a single chip causes massive power bills gpusimpow: A gpgpu power simulator," in *2013 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, 2013, pp. 97–106.