ARCHITECTURAL SUPPORT FOR VIRTUAL MEMORY IN GPUs

BY BHARATH SUBRAMANIAN PICHAI

A thesis submitted to the

Graduate School—New Brunswick

Rutgers, The State University of New Jersey

in partial fulfillment of the requirements

for the degree of

Master of Science

Graduate Program in Computer Science

Written under the direction of

Dr.Abhishek Bhattacharjee

and approved by

_____

_____

_____

New Brunswick, New Jersey

October, 2013

ABSTRACT OF THE THESIS

Architectural Support for Virtual Memory in GPUs

by Bharath Subramanian Pichai

Thesis Director: Dr.Abhishek Bhattacharjee

The proliferation of heterogeneous compute platforms, of which CPU/GPU is a prevalent example, necessitates a manageable programming model to ensure widespread adoption. A key component of this is a shared unified address space between the heterogeneous units to obtain the programmability benefits of virtual memory. Indeed, processor vendors have already begun embracing heterogeneous systems with unified address spaces (e.g., Intel's Haswell, AMD's Berlin processor, and ARM's Mali and Cortex cores).

We are the first to explore GPU Translation Lookaside Buffers (TLBs) and page table walkers for address translation in the context of shared virtual memory for heterogeneous systems. To exploit the programmability benefits of shared virtual memory, it is natural to consider mirroring CPUs and placing TLBs prior (or parallel) to cache accesses, making caches physically addressed. We show the performance challenges of such an approach and propose modest hardware augmentations to recover much of this lost performance.

We then consider the impact of this approach on the design of general purpose GPU performance improvement schemes. We look at: (1) warp scheduling to increase cache hit rates; and (2) dynamic warp formation to mitigate control flow divergence overheads. We show that introducing cache-parallel address translation does pose challenges, but

that modest optimizations can buy back much of this lost performance.

In the CPU world, the programmability benefits of address translation and physically addressed caches have outweighed their performance overheads. This paper is the first to explore similar address translation mechanisms on GPUs. We find that while cache-parallel address translation does introduce non-trivial performance overheads, modestly TLB-aware designs can move performance losses into a range deemed acceptable in the CPU world. We presume this stake-in-the-ground design leaves room for improvement but hope the larger result, that a little TLB-awareness goes a long way in GPUs, sets the stage for future work in this fruitful area.

List of Figures

Acknowledgements

I thank my advisor Dr. Abhishek Bhattacharjee, whose guidance enabled me to pursue this study. I express my gratitude to Dr. Lisa Hsu of Qualcomm for the insightful feedback during the course of my thesis work. Further, I would like to express my gratitude to my lab mates Binh Pham and Zi Yan for the meaningful discussions we had on the project, life, the universe and everything. I would like to thank Dr. Ricardo Bianchini and Dr. Thu Nguyen for supervising my thesis defense. Finally, I would like to thank the Department of Computer Science, Rutgers University for providing the necessary research infrastructure.

Table of Contents

Chapter 1

Introduction

The advent of the dark silicon era [57] has generated research on hardware accelerators. Recent heterogeneous systems include accelerators for massive multidimensional data-sets [59], common signal processing operations [52], object caching [40], and spatially-programmed architectures [50]. To ensure widespread adoption of accelerators, their programming models must be efficient and easy to use.

One option is to have unified virtual and physical address spaces between CPUs and accelerators. Unified address spaces enjoy many benefits; they make data structures and pointers globally visible among compute units, obviating the need for expensive memory copies between CPUs and accelerators. They also unburden CPUs from pinning data pages for accelerators in main memory, improving memory efficiency. Unified address spaces also require architectural support for virtual-to-physical address translation.

CPUs currently use per-core Translation Lookaside Buffers (TLBs) and hardware page table walkers (PTWs) to access frequently-used address translations from operating system (OS) page tables. CPU TLBs and PTWs are accessed before (or in parallel with) hardware caches, making caches physically-addressed. This placement restricts TLB size (so that its access time does not overly increase cache access times), but efficiently supports multiple contexts, dynamically-linked libraries, cache coherence, and unified virtual/physical address spaces. Overall, CPU TLB misses expend 5-15% of runtime [7, 9, 10, 11, 12, 51], which is considered acceptable for their programmability benefits.

As accelerators proliferate, we must study address translation and its role in the impending unified virtual address space programming paradigm. We are the first to explore key accelerator TLB and PTW designs in this context. We focus on general

purpose programming for graphics processing units (GPUs) [15, 19, 49, 48] for two reasons. First, GPUs are a relatively mature acceleration technology that have seen significant recent research [20, 21, 31, 41, 45, 54, 55]. Second, processor vendors like Intel, AMD, ARM, Qualcomm, and Samsung are embracing integrated CPU/GPUs and moving towards fully unified address space support, as detailed in Heterogeneous Systems Architecture (HSA) [39] specifications. For example, AMD's upcoming Berlin processor commits to fully unified address spaces using heterogeneous uniform memory access (hUMA) technology [53].

Today's GPUs and CPUs typically use separate virtual and physical address spaces. Main memory may be physically shared, but is usually partitioned, or allows unidirectional coherence (e.g., ARM allows accelerators to snoop CPU memory partitions but not the other way around). GPU address translation has traditionally been performed using Input/Output Memory Management Unit (IOMMU) TLBs at the memory controller, leaving caches virtually-addressed. This approach sufficed in the past because there was no concept of coherence or shared memory support with the host CPU. However, as vendors pursue address space unification with specifications like HSA, this approach becomes insufficient.

We are the first to consider alternatives to this approach, by placing GPU TLBs and PTWs before (or in parallel with) cache access (i.e., realizing physically-addressed caches). This mirrors CPU TLBs and achieves the same programmability benefits; for example, efficient support for multiple contexts and application libraries, and cache coherence between CPU and GPU caches (all of which are industry goals [39]).

We show that vanilla GPU TLBs degrade performance, demonstrating that the success of accelerator platforms may hinge upon more thoughtful TLB designs. In response, we propose augmentations that recover much of this lost performance. We then show how address translation affects: (1) warp scheduling to increase cache hit rates; and (2) dynamic warp formation to mitigate control flow divergence overheads. Specifically, our contributions are as follows:

First, we show that placing TLBs and PTWs before (or in parallel with) cache access degrades performance by 20-50%. This is because cache-parallel accesses mean

latency (and thus sizing) is paramount. Meanwhile, GPU SIMD execution, in which multiple warp threads execute in lock-step, means a TLB miss from a single thread can stall all warp threads, magnifying miss penalties. Fortunately, we show that modest optimizations recover most lost performance.

Second, we show the impact of TLBs on cache-conscious warp/wavefront scheduling (CCWS) [54], recently-proposed to boost GPU cache hit rates. While naively adding TLBs offsets CCWS, simple modifications yield close to ideal CCWS (without TLBs) performance. We also study TLB-based CCWS schemes that are simpler *and* higher-performance.

Third, we show how TLBs affect dynamic warp formation for branch divergence. Using thread block compaction (TBC) [20], we find that dynamically assimilating threads from disparate warps increases memory divergence and TLB misses. Though naive designs degrade performance by over 20%, adding TLB-awareness mitigates these overheads, boosting performance close to ideal TBC (without TLBs).

Overall, this work is the first to study GPU address translation in a unified virtual/physical memory space paradigm. While our work is relevant to server and client systems, our evaluations focus on server workloads. Our insights, however, are relevant across the compute spectrum, highlighting TLB, PTW, and warp scheduling interactions. All address translation support degrades performance to an extent; the question is whether the overheads are worth the associated programmability benefits. Current CPUs generally deem 5-15% performance degradations acceptable [7, 8, 11, 12, 18]. While our GPU TLB proposals meet these ranges, we shed light on various GPU address translation design issues rather than advocate a specific implementation. In fact, we believe that there is plenty of room for improvement and even alternate GPU TLB designs and placement (e.g., opportunistic virtual caching [9]). This paper provides valuable insights for future studies and methodology for reasoning about the impact of address translation on GPU performance.

Chapter 2

Background, Our Approach, Methodology

In this chapter, we discuss the contemporary translation schemes in CPUs and GPUs. Further, we elucidate the need for unified virtual address space and explain the goals of our work followed up with the methodology adopted to perform this study.

2.1 Address Translation on CPUs

CPU TLBs have long been studied by the academic community [7, 8, 11, 12, 33, 51]. Most CPUs currently access TLBs prior to (or in parallel with) L1 cache access, realizing physically-addressed caches. This approach dominates commercial systems (over virtually-addressed caches [16, 37]) because of programmability benefits. Physically-addressed caches prevent coherence problems from address synonyms (multiple virtual addresses mapping to the same physical address) and homonyms (a single virtual address mapping to multiple physical addresses) [16]. This efficiently supports multiple contexts, dynamically-linked libraries, cache coherence among multiple cores and with DMA engines.

2.2 Address Translation on CPU/GPUs

Recent interest in general purpose programming for GPUs has spurred commercial CPU/GPU implementations and research to program them [15, 19, 49, 48]. Intel, AMD, ARM, Qualcomm, and Samsung already support integrated CPU/GPU systems, with cross-platform standards for parallel programming (e.g., OpenCL [43]). NVidia supports general purpose GPU programming through CUDA [58] and is considering GPU architectures for the cloud (e.g., Kepler [47]), with an eye on general purpose computation.

Current heterogeneous systems use rigid programming models that require separate page tables, data replication, and manual data movement between CPU and GPU. This is especially problematic for pointer-based data structures (e.g., linked lists, trees) [1]. Recent research [22, 23, 28, 27] tries to overcome these issues, but none solve the problem generally by unifying the address space. As such, although GPUs currently use separate address spaces (though latest CUDA releases permit limited CPU/CPU virtual address sharing [58]), vendors have begun supporting address translation [13] as a step towards unified virtual and physical address spaces.

There are many possibilities for address translation in GPUs. AMD and Intel use Input Output Memory Management Units [1, 2, 25] (IOMMUs) with their own page tables, TLBs, and PTWs. IOMMUs with large TLBs are placed in the memory controller, making GPU caches virtually-addressed. In this paper, we study the natural alternative of translating addresses before (or in parallel with) cache access. This has the following programmability benefits.

CPU/GPU systems with full-blown address translation more naturally support a single virtual and physical address space. This eliminates the current need for CPUs to initialize, copy, pin data pages (in main memory), duplicate page tables for GPU IOMMUs, and set up IOMMU TLB entries [13]. It also allows GPUs to support page faults and access memory mapped files, features desired in hUMA specifications [53] (though their feasibility requires a range of hardware/software studies beyond the scope of this work).

Second, physically-addressed caches support multiple contexts (deemed desirable by HSA [39] and NVidia's Kepler whitepaper [47]) more efficiently. Virtually-addressed caches struggle due to incoherence from address synonyms [37]. Workarounds like cache flushing on context switches can ensure correct functionality, but do so at a performance and complexity cost versus physically-addressed caches.

Third, address translation placement that allows physically-addressed caches also

[1]Some platforms provide pinning mechanisms that do not need data transfers. Both CPU and GPU maintain pointers to the pinned data and the offset arithmetic is the same; however, the pointers are distinct. This suffers overheads from pinning and pointer replication (which can lead to buggy code).

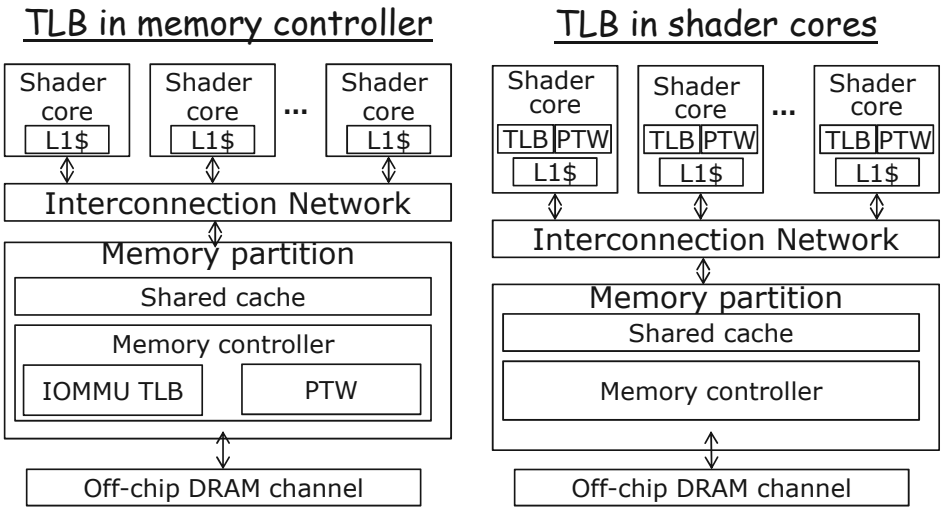

Figure 2.1: The diagram on the left shows conventional GPU address translation with an IOMMU TLB and PTW in the memory controller. Instead, our approach embeds a TLB and PTW per shader core so that all caches become physically-addressed.

efficiently supports application libraries (mentioned by HSA as a design goal). Traditional virtually-addressed GPU caches suffer from address homonyms [16], which can arise when executing libraries.

Finally, cache coherence between CPU and GPU caches has long been deemed desirable [35, 39, 55]. In general, cache coherence is greatly simplified if GPU caches are physically-addressed, in tandem with CPU caches.

2.3 Goals of Our Work

Processor vendors are embracing support for unified address spaces between CPUs and accelerators, making accelerator address translation a first-class design objective. Unfortunately, the programmability benefits of address translation do not come for free. In the past, CPU address translation overheads of 5-15% of system runtime have been deemed a fair tradeoff for its benefits [7, 9, 10, 11, 12, 51]. Given the rise of GPU address translation, our goal is to conduct the first study of GPU TLBs and PTWs.

We have already established that like CPUs, it is desirable for GPUs to have physically-addressed caches for various programmability benefits. A natural design option is to mirror CPUs, placing GPU TLBs prior to (or in parallel with) L1 caches.

The right side of Figure 2.1 shows our approach, with per shader core TLBs and PTWs. Like CPUs, we assume that L1 caches are virtually-indexed and physically-tagged, allowing TLB access to overlap with L1 cache access. This contrasts with past academic work ignoring address translation [20, 21, 54] or placing an IOMMU TLB and PTW in the memory controller (the diagram on the left side of Figure 2.1).

For all its programmability benefits, address translation at the L1-level is challenging because it degrades performance by constraining TLB size. Figure 2.2 shows that naive designs that do not consider the distinguishing characteristics of GPUs can severely degrade performance. The plots show speedups (values higher and lower than 1 are improvements and degradations) of general purpose GPU benchmarks using naive 128-entry, 3-port TLBs with 1 PTW per shader core (`With TLB`). Not only do naive TLBs degrade performance, they also lose 30-50% performance versus conventional cache-conscious wavefront scheduling and thread block compaction [20, 54]. These degradations arise for three reasons, unique to GPUs.

First, GPU memory accesses have low temporal locality, increasing TLB miss rates. Second, shader cores run multiple threads in lock-step in *warps* (in NVIDIA terminology) [21, 20, 41]. Therefore, a TLB miss on one warp thread effectively stalls all warp threads, magnifying the miss penalty. Third, multiple warp threads often TLB miss in tandem, stressing conventional PTWs that serialize TLB miss handling.

Therefore, a key goal of this work is to refashion conventional CPU TLB and PTW hardware to fit GPU characteristics. We show that modest but informed modifications significantly reduces the degradations of Figure 2.2. In fact, simple, thoughtful GPU TLBs reduce GPU address translation overheads to 5-15% of system runtime.

We emphasize that our study is one natural design point to achieve programmability benefits [39]. Whether this is the *best* design point is a function of what performance overhead is considered reasonable, alternate implementations, and emerging classes of GPU workloads. Though this is beyond our scope, we do present broad insights to reason about address translation in GPUs (and more broadly, accelerators).

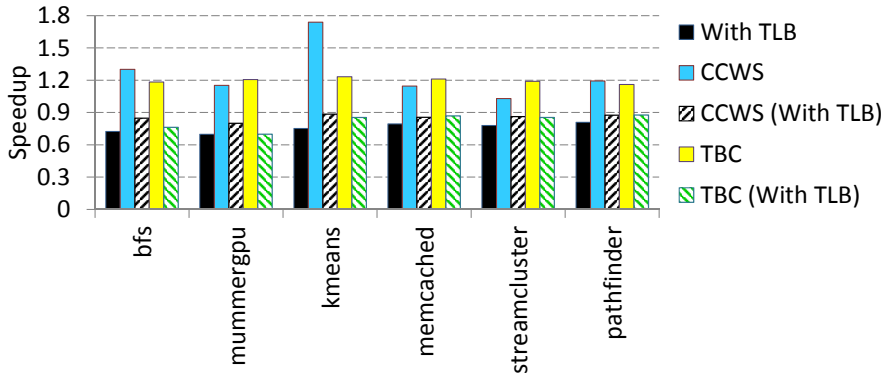

Figure 2.2: Compared to a baseline architecture without TLBs, speedup of naive, 3-ported TLBs per shader core, with and without cache-conscious wavefront scheduling, with and without thread block compaction. Naive TLBs degrade performance in every case.

2.4 Methodology

2.4.1 Evaluation Workloads

We use server workloads from past studies on control-flow divergence and cache scheduling [20, 54]. From the Rodinia benchmarks [17], we use `bfs` (graph traversal), `kmeans` (data clustering), `streamcluster` (data mining), `mummergpu` (DNA sequence alignment), and `pathfinder` (grid dynamic programming). In addition, we use `memcached`, a key-value store and retrieval system, stimulated with a representative portion of the Wikipedia traces [24]. Our baseline version of these benchmarks have memory footprints in excess of 1GB.

Ideally, we would like to run benchmarks that will be prevalent on future integrated CPU/GPU platforms. Unfortunately, there is a dearth of such workloads because of the lack of abstractions that ease CPU/GPU programming (like unified address spaces themselves). We do include applications like `memcached` which will likely run on these systems but expect that future workloads supporting braided parallelism (a mix of task and data parallelism from a single source) [42] will particularly benefit from unified address spaces. Our studies enable these applications and provide insights on their performance with GPU address translation.

2.4.2 Evaluation Infrastructure

We use GPGPU-Sim [5] with parameters similar to past work [20, 54]. In particular, we assume 30 SIMT cores, 32-thread warps, and a pipeline width of 8. We have, per core, 1024 threads, 16KB shared memory, and 32KB L1 data caches (with 128 byte lines and LRU). We also use 8 memory channels with 128KB of unified L2 cache space per channel. We run binaries with CPU and GPU portions, but report timing results for the GPU part. GPGPU-Sim uses single instruction multiple data (SIMD) pipelines, grouping SIMD cores into core clusters, each of which has a port to the interconnection network with a unified L2 cache and the memory controller.

Most of our results focus on 4KB pages due to the additional challenge imposed by small page size; however, we also present initial results for large 2MB pages later in the paper. Note that large pages, while effective, don't come for free and can have their own overheads in certain situations [3, 8, 46]. As a result, many applications are restricted to 4KB page sizes; it is important for our GPUs with address translation to be compatible with them.

Chapter 3

Address Translation for GPUs

We now introduce design options for GPU TLBs and PTWs, showing the shortcomings of blindly porting CPU-style address translation into GPUs. We then present GPU-appropriate address translation.

3.1 Address Translation Design Space

We consider the following address translation design points.

Number and placement of TLBs: CPUs traditionally place a TLB in each processor core so that pipelines can have quick, unfettered address translation. One might consider the same option for each GPU shader core. It is possible to implement one TLB per SIMD lane; this provides the highest performance, at the cost of power and area (e.g., we assume 240 SIMD lanes, so this approach requires 240 TLBs per shader core). Instead, we assume a more power- and area-frugal approach, with one TLB per shader core (shared among lanes).

PTW mechanisms, counts, and placement: It is possible to implement page table walking with either hardware or software (where the operating system is interrupted on a TLB miss [29]). Hardware approaches require more area but perform far better [29]. When using hardware PTWs, it is possible to implement one or many per TLB. While a single PTW per TLB saves area, multiple PTWs can potentially provide higher performance if multiple TLB misses occur. Finally, placing PTWs with TLBs encourages faster miss handling; alternately, using a single shared PTW saves area at the cost of performance.

Size of TLBs: While larger TLBs have a higher hit rate, they also occupy more area and have higher hit times. Since TLB access must complete by the time the L1

cache set is selected (for virtually-indexed, physically-tagged caches), overly-high hit times degrade performance. In addition, TLBs use power-hungry content-addressable memories [9]; larger TLBs hence consume much more power.

TLB port count: More TLB ports permit parallel address translation, which can be crucial in high-throughput shader cores. Unfortunately, they also consume area and power.

Blocking versus non-blocking TLBs: One way of reducing the impact of TLB misses is to overlap them with useful work. Traditional approaches involve augmenting TLBs to support hits under the original miss or additional misses under the miss. Both approaches improve performance, but require additional hardware like Miss Status Holding Registers (MSHRs) and access ports.

Page table walk scheduling: Each page table walk requires multiple memory references (four in x86 [6]) to find the desired PTE, some of which may hit in caches. In multicore systems, it is possible that more than one core concurrently experience TLB misses. In response, one option (which has not been studied to date) is to consider the page table walks of the different cores and see if some of their memory references are to the same locations or cache lines. In response, it is possible to interleave PTW memory references from different cores to reduce the number of memory references and boost cache hit rates (while retaining functional correctness).

System-level issues (TLB shootdowns, page faults): The adoption of unified address spaces means that there are many options for handling TLB shootdowns and page faults. In one possibility, we interrupt a CPU to execute the shootdown code or page fault handler on the GPU's behalf. This CPU could be the one that launched the GPU kernel or any other idle core. Alternately, GPUs could themselves run the shootdown code or the page fault handler, as suggested by hUMA [53].

3.2 CPU-Style Address Translation in GPUs

Since GPU address translation at the L1 cache level has not previously been studied, the design space is a blank slate. As such, it is natural to begin with CPU-style TLBs.

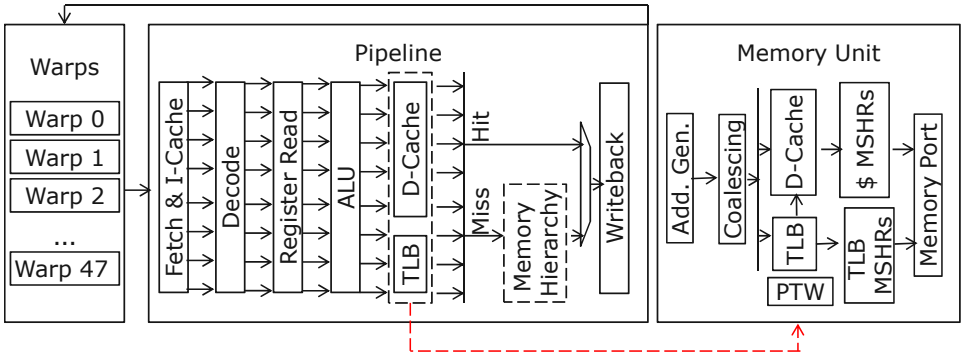

Figure 3.1: Shader core pipeline with address translation. We assume that L1 data caches are virtually-indexed and physically-tagged (allowing TLB lookup in parallel with cache access). All caches (L1 and shared caches, which are not shown) are physically-addressed. Note that the diagram zooms in on the memory unit.

We now discuss this baseline naive architecture.

Figure 3.1 shows our design. Each shader core maintains a TLB/PTW and has 48 warps (the minimum scheduling unit) per shader core. Threads of a warp execute the same instruction in lock-step. Instructions are fetched from an I-cache and operands read from a banked register file. Loads and stores access the memory unit (which Figure 3.1 zooms on).

The memory unit's address generation unit calculates virtual addresses, which are coalesced to unique cache line references. We enhance conventional coalescing logic to also coalesce multiple intra-warp requests to the same virtual page (and hence PTE). This is crucial towards reducing TLB access traffic and port counts. At this point, two sets of accesses are available: (1) unique cache accesses; and (2) unique PTE accesses. These are presented in parallel to the TLB and data cache. Note that we implement virtually-indexed, physically-tagged L1 caches. We now elaborate on the design space options from Section 3.1, referring to Figure 3.1 when necessary.

Number and placement of TLBs: As previously detailed, we assume 1 TLB per shader core shared among SIMD lanes, to save power and area.

PTW mechanisms, counts, and placement: Our GPU, like most CPUs today,

assumes hardware PTWs because: (1) they achieve higher performance than software-managed TLBs [29]; and (2) they do not need to run OS code (which GPUs cannot currently execute), unlike software-managed TLBs.

CPUs usually place PTWs close to each TLB so that page table walks can be quickly initiated on misses. We use the same logic to place PTWs next to TLBs. Finally, CPUs maintain one PTW per TLB. While it is possible to consider multiple PTWs per GPU TLB, our baseline design mirrors CPUs and similarly has one PTW per shader core. We investigate the suitability of this decision in subsequent sections.

Size of TLBs: Commercial CPUs currently implement 64-512 entry TLBs per core [11, 26]. These sizes are typically picked to be substantially smaller and lower-latency than CPU L1 caches. Using a similar methodology with CACTI [44], we have found that 128-entry TLBs are the are the largest possible structures that do not increase the access time of 32KB GPU L1 data caches. We use 128-entry TLBs in our naive baseline GPU design, studying other sizes later.

TLB port count: Intel's CPU TLBs support three ports in today's systems [26]. Past work has found that this presents a performance/power/area tradeoff appropriate for CPU workloads [4]. Our baseline design assumes 3-ported TLBs for GPUs too but we revisit this in subsequent sections.

Blocking versus non-blocking: There is evidence that that some CPU TLBs do support hits under misses because of their relatively simple hardware support [30]. However, most commercial CPUs typically use blocking TLBs [7, 51] because of their high hit rates. We therefore assume blocking TLBs in our baseline GPU design. This means that similar to cache misses, TLB misses prompt the scheduler to swap another warp into the SIMD pipeline. Swapped-in warps executing non-memory instructions proceed unhindered (until the original warp's page table walks finish and it completes the `Writeback` stage). Swapped-in threads with memory references, however, do not proceed in this naive design as they require non-blocking support.

Figure 3.1 shows that TLBs have their own (MSHRs). We assume, like both GPU caches and past work on TLBs [41], that there is one TLB MSHR per warp thread (32 in total). MSHR allocation triggers page table walks, which inject memory requests to

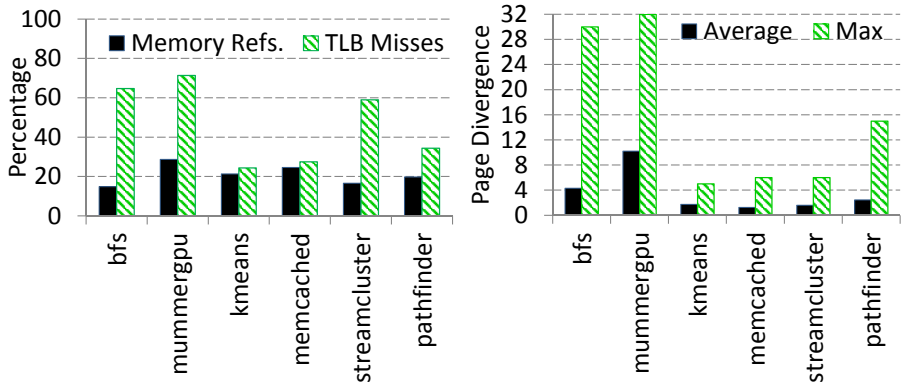

Figure 3.2: The left diagram shows the percentage of total instructions that are memory references and TLB miss rates; the right diagram shows the average number of distinct translations requested per warp (page divergence) and the maximum number of translations requested by any warp through the execution.

the shared caches and main memory.

Page table walk scheduling: CPU PTWs do not currently interleave memory references from multiple concurrent TLB misses for increased cache hit rate and reduced memory traffic. This is because the low incidence of concurrent TLB misses on CPUs [12] isn't worth the complexity and area overheads of such logic (though this is likely to require relatively-simple combinational logic). Similarly, our naive baseline GPU design also doesn't include PTW scheduling logic.

System-level issues (shootdowns, page faults: CPUs usually shootdown TLBs on remote cores when its own TLB updates an entry, using software inter-processor interrupts (IPIs). We assume the same approach for GPUs (i.e., if the CPU that initiated the GPU modifies its TLB entries, GPU TLBs are flushed). Similarly, we assume that a page fault interrupts a CPU to run the handler. In practice, our performance was not affected by these decisions (because shootdowns and page faults almost never occur on our workloads). If these become a problem, as detailed in hUMA, future GPUs may be able to run dedicated OS code for shootdowns and page faults without interrupting CPUs. We leave this for future work.

Performance: Having detailed this naive baseline approach, we profile its performance for our workloads. Our results, already presented in Figure 2.2 in Section 2.3, shows

the inadequacies of naive baseline GPU address translation. Overall, performance is degraded by 20-35%. The graph on the left of Figure 3.2 complements this data by showing: (1) the number of memory references in each workload as a percentage of the total instructions; and (2) miss rates of 128-entry GPU TLBs. While the number of memory references are generally low compared to CPUs (under 25% for all benchmarks), TLB miss rates are very high (ranging from 22% to 70%). This is because GPU benchmarks tend to stream through large quantities of data without much reuse. In addition, we find that average miss penalties are well above 200 cycles for every benchmark (Figure A.1 in the Appendix). To put this in perspective, this is over double the penalty of L1 cache misses.

Fortunately, we now show that simple GPU-aware modifications of naive TLBs and PTWs counter these problems, recovering most of this lost performance.

3.3 Augmenting Address Translation for GPUs

GPU execution pipelines differ widely from CPUs in that they perform data-parallel SIMD operations on warps. Since each warp simultaneously executes multiple threads, it can conceivably prompt multiple TLB misses (up to the warp width, which is 32 in our configuration). In the worst cases, all TLB misses could be to different virtual pages or PTEs. Thus, there is more pressure on TLBs in some ways, but also more opportunity for parallelism, if designed with GPUs in mind. We now detail optimizations to exploit these opportunities, quantifying their improvements.

TLB size and port counts: An ideal TLB is large and low-latency. Furthermore, it is heavily multi-ported, with one port per warp thread (32 in total). While more ports facilitate quick lookups and miss detection, they also significantly increase area and power. Figure 3.3 sheds light on size, access time, and port count tradeoffs for naive baseline GPU address translation. We vary TLB sizes from 64 to 512 entries (the range of CPU TLB sizes) and port count from 3 (like L1 CPU TLBs) to an ideal number of 32. We present speedups versus against the no-TLB case (speedups are under 1 since adding TLBs degrades performance). We use CACTI to assess access time increases

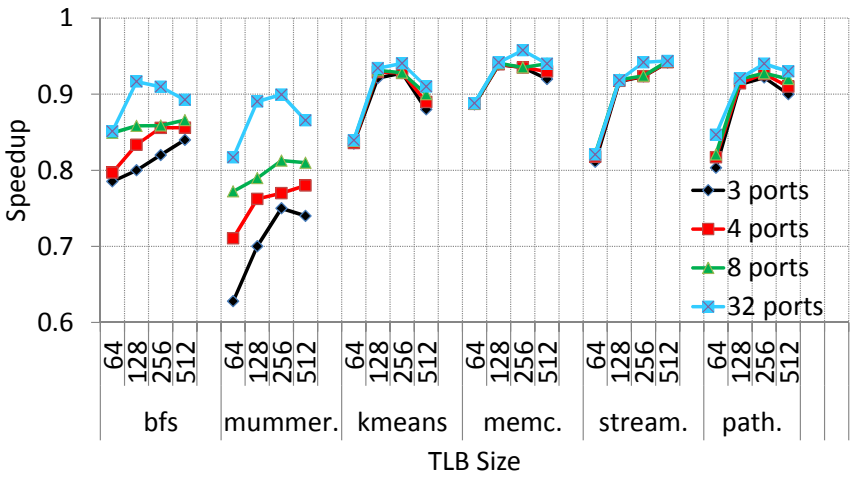

Figure 3.3: Performance for TLB size and port counts, assuming fixed access times. Note that TLBs larger than 128 entries and 4 ports are impractical to implement and actually have much higher access times that degrade performance.

with size.

Figure 3.3 shows that larger sizes and more ports greatly improve GPU TLB performance. In general, 128 entry TLBs perform best; beyond this, increased access times reduce performance. Figure 3.3 also shows that while port counts do impact performance (particularly for `mummergpu` and `bfs`), modestly increasing from 3 ports (in our naive baseline) to 4 ports recovers much of this lost performance. The graph on the right of 3.2 shows why, by plotting page divergence (the number of distinct translations requested by a warp). We show both average page divergence the maximum page divergence of any warp through execution. Figure 3.2 shows warps usually request far less than their maximum of 32 translations. This is because coalescing logic accessed before TLBs reduce requests to the same PTE into a single lookup. Only `bfs` and `mummergpu` have average page divergence higher than 4. While this does mean that some warps have higher lookup time (e.g., when `mummergpu` encounters its maximum page divergence of 32), simply augmenting the port count of the naive implementation to 4 recovers much of the lost performance. Figure A.2 in the Appendix shows more details on the page divergence requirements of different benchmarks.

Blocking versus non-blocking TLBs: Using blocking TLBs, the only way to overlap miss penalties with useful work is to execute alternate warps without memory references.

We have seen however, that GPU TLB miss penalties are extremely long. Therefore, GPU address translation requires more aggressive non-blocking facilities. Specifically, we investigate:

Hits from one warp under misses from another warp: In this approach, the swapped-in warp executes even if it has memory references, as long as they are all TLB hits. As soon as the swapped-in thread TLB misses, it too must be swapped out. This approach leverages already-existing TLB MSHRs and requires only simple combinatorial logic updates to the warp scheduler. In fact, hardware costs are similar to CPU TLBs which already support hits under misses [30]. We leave more aggressive miss under miss support for future work.

Overlapping TLB misses with cache accesses within a warp: Beyond overlapping a warp's TLB miss penalty with the execution of other warps, it is also possible to overlap TLB misses with work from the *warp that originally missed*. Since warps have multiple threads, even when some miss, others may TLB hit. Since hits immediately yield physical addresses, it is possible to look up the L1 cache with these addresses without waiting for the warp's TLB misses to be resolved. This boosts cache hit rates since this warp's data is likelier to be in the cache before a swapped-in warp evicts its data. Furthermore if these early cache accesses do miss, subsequent cache miss penalties can be overlapped with the TLB miss penalties of the warp.

In this approach, TLB hit addresses immediately look up the cache even if the same warp has a TLB miss. Data found in the cache is buffered in the standard warp context state in register files (from past work [21, 54]) before the warp is swapped out. This approach requires no hardware beyond simple combinational logic in the PTW and MSHRs to allow TLB hits to proceed for cache lookup.

Results: Figure 3.4 quantifies the benefits of non-blocking TLBs, normalized to a baseline without TLBs. We first permit hits under misses, then also allow TLB hits to proceed to the cache without waiting for misses to be resolved. We compare these to an *ideal and impractical* 512-entry TLB with 32 ports and *no increased access latencies*.

While hits under misses improve performance, immediately looking up the cache for threads that TLB hit and PTW scheduling is even more effective. For example,

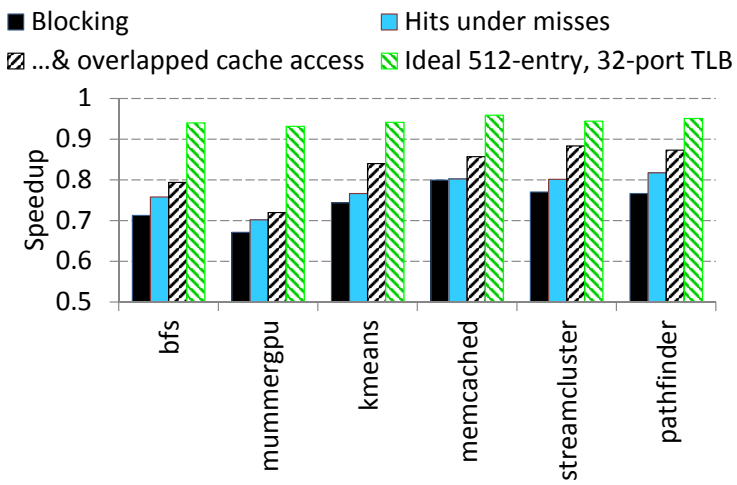

Figure 3.4: On a 128-entry, 4-port TLB, adding non-blocking support improves performance closer to an ideal (no increasing access latency) 32-port, 512-entry TLB.

`streamcluster` gains an additional 8% performance from overlapped cache access. Overall, Figure 3.7 shows that modest TLB design enhancements boost performance substantially. We will show how additional enhancement further bring performance close to the impractical, ideal case.

Page table walk scheduling: Our page divergence results show that GPUs execute warps that can suffer TLB misses on multiple threads. Often, the TLB misses are to distinct PTEs. Consider the example in Figure 3.5, which shows three concurrent x86 page walks. x86 page table walks require a memory reference to the the Page Map Level 4 (PML4), Page Directory Pointer (PDP), Page Directory (PD), and Page Table (PT). A `CR3` register provides the base physical address of the PML4. A nine bit index (bits 47 to 39 of the virtual address) is concatenated with the base physical address to generate a memory reference to the PML4. This finds the base physical address of the PDP. Bits 38-30 of the virtual address are then used to look up the PDP. Bits 29-21, and 20-12 are similarly used for the PD and PT (which has the desired translation).

Figure 3.5 shows multilevel page lookups for a warp that has three threads missing on virtual pages `(0xb9, 0x0c, 0xac, 0x03)`, `(0xb9, 0x0c, 0xac, 0x04)`, and `(0xb9, 0x0c, 0xad, 0x05)`. We present addresses in groups of 9-bit indices as these correspond directly to the page table lookups. Naive baseline GPU PTWs perform the three page table walks serially (shown with dark bubbles). This means that each page

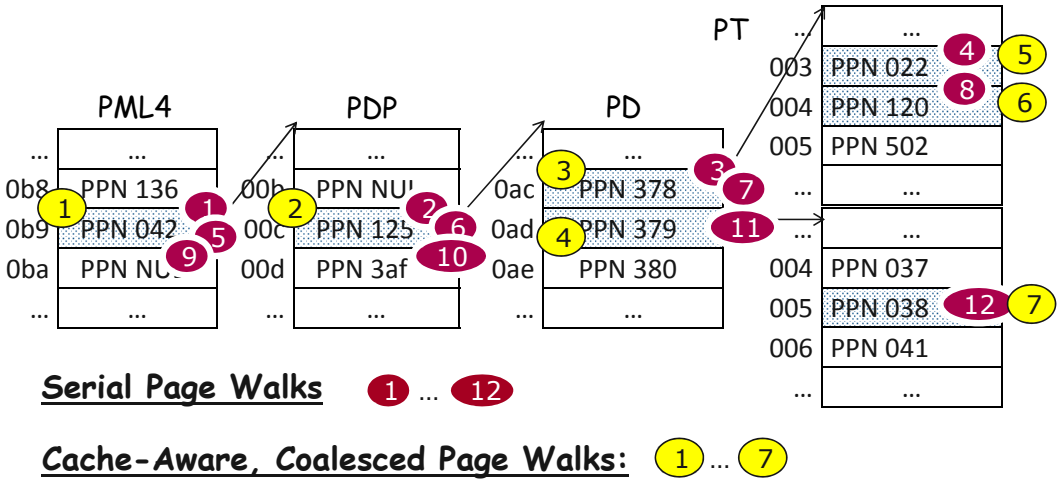

Figure 3.5: Three threads from a warp TLB miss on addresses `(0xb9, 0x0c, 0xac, 0x03)`, `(0xb9, 0x0c, 0xac, 0x04)`, and `(0xb9, 0x0c, 0xad, 0x05)`. A conventional page walker carries out three serial page walks (shown with dark bubbles) , making references to (1-4), (5-8), and (9-12), a total of 12 loads. Our cache-aware coalesced page walker (shown with light bubbles) reduces this to 7 and achieves better cache hit rate.

table walk requires four memory references; (1-4) for `(0xb9, 0x0c, 0xac, 0x03)`, (5-8) for `(0xb9, 0x0c, 0xac, 0x04)`, and (9-12) for `(0xb9, 0x0c, 0xad, 0x05)`. Each of the references hits in the shared cache (several tens of cycles) or main memory. Fortunately, simple PTW scheduling exploiting commonality across concurrent page table walks improves performance in two ways:

Reducing the number of page table walk memory references: Higher order virtual address bits tend to be constant across memory references. For example, bits 47-39, and 38-30 of the virtual address change infrequently as lower-order bits (29 to 0) cover a 1GB address space. Since these bits index the PML4 and PDP during page table walks, multiple page table walks usually traverse similar paths. In Figure 3.5, all three page walks read the same PML4 and PDP locations. We leverage this observation by remembering PML4 and PDP reads so that they can be reused among page walks. This means that three PML4 and PDP reads are replaced by a single reads.

Increasing page table walk cache hit rates: 128-byte cache lines hold 16 consecutive 8-byte PTEs. Therefore, there is potential for cache line reuse across different page table walks. For example, in Figure 3.5, two PD entries from the same cache line are used for

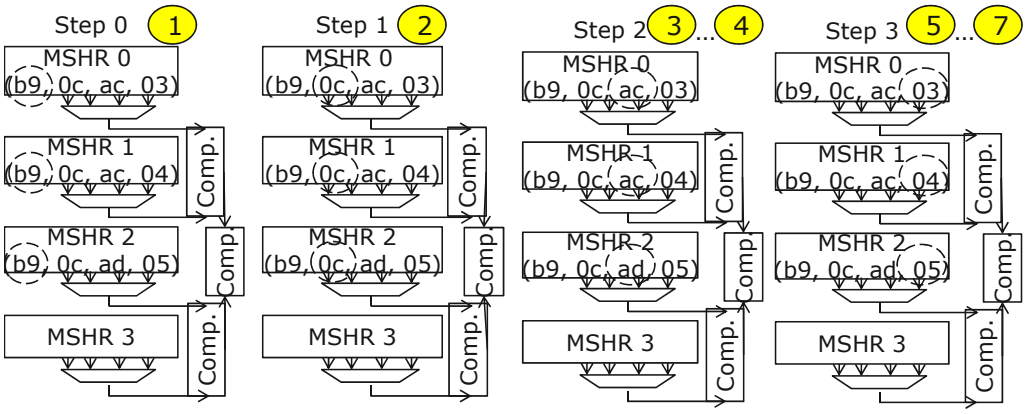

Figure 3.6: Our page table walk scheduler attempts to reduce the number of memory references and increase cache hit rate. The highlighted entries show what memory references in the page walks are performed and in what order. This hardware assumes a mux per MSHR, with a tree-based comparator circuit. We show 4 MSHRs though this approach generalizes to 32 MSHRs.

all walks. Similarly, the PT entries for virtual pages `(0xb9, 0x0c, 0xac, 0x03)` and `(0xb9, 0x0c, 0xac, 0x04)` are on the same cache line. We exploit this observation by interleaving memory references from different page table walks (shown in lighter bubbles). In Figure 3.5, references 3 and 4 (from three different page table walks) are handled successively, as are references 5 and 6 (from page walks for virtual pages `(0xb9, 0x0c, 0xac, 0x03)` and `(0xb9, 0x0c, 0xac, 0x04)`), boosting hit rates.

Implementation: Figure 3.6 shows PTW scheduling (TLBs and PTWs are present, though not shown). MSHRs store the virtual page numbers causing TLB misses. We propose combinational hardware that scans the MSHRs, extracting, in four consecutive steps, PML4, PDP, PD, and the PT indices. Each stage checks whether the memory access for its level are amenable to coalescing (they are repeated) or lie on the same cache line. The PTW then injects references (for each step, we show the matching memory reference from Figure 3.5).

Figure 3.6 shows that a comparator tree matches indices in each stage of the algorithm. The It scans the PML4 indices looking for a match in bits 47-44 because both a repeated memory reference and PTEs within the same cache line share all but the bottom 4 index bits. Once the comparator discovers that all the page walks can be satisfied from the same cache line in step 0, a memory reference for PML4 commences.

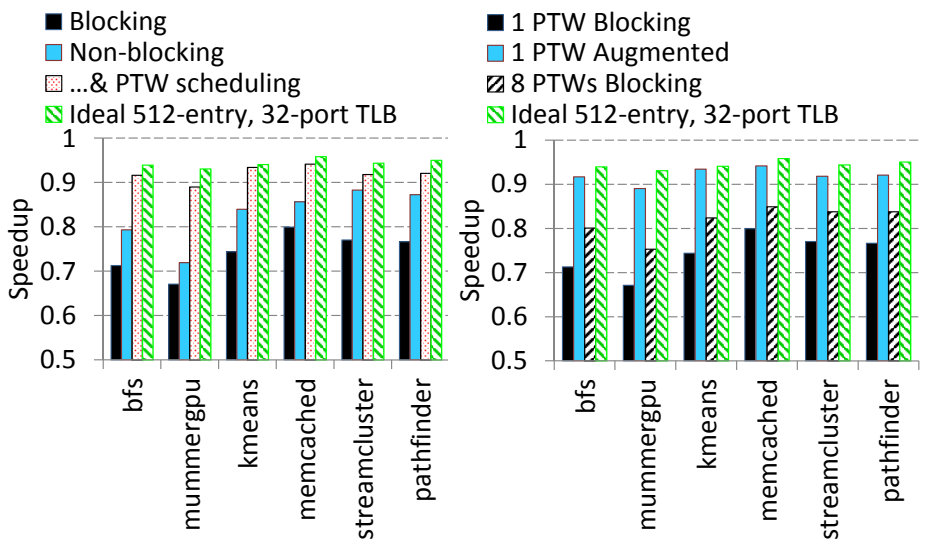

Figure 3.7: (Left) On a 128-entry, 4-port TLB, adding non-blocking and PTW scheduling logic achieves close to the performance of an ideal (no increasing access latency) 32-port, 512-entry TLB. (Right) Our augmented TLB with 1 PTW consistently outperforms 8 PTWs.

In parallel with this memory reference, the same comparator tree now compares the upper 5 bits of the PDP indices (bits 38-34). Again, all indices match, meaning that in step 1, only one PDP reference is necessary. In step 2, PD indices are studied (in parallel with the PDP memory reference). We find that the top 5 bits match but that the bottom 4 bits don't (indicating that they are multiple accesses to the same cache line). Therefore step 2 injects two memory references (3 and 4) successively. Step 3 uses similar logic to complete the page walks for `(0xb9, 0x0c, 0xac, 0x03)`, `(0xb9, 0x0c, 0xac, 0x04)`, and `(0xb9, 0x0c, 0xad, 0x05)`.

To reduce hardware, we use a comparator tree rather than implementing a comparator between every pair of MSHRs (which provides maximum performance). We have found that this achieves close to pairwise comparator performance. Also, we share one comparator tree with muxes for all page table levels. This is possible because MSHRs scans can proceed in parallel with loads from the previous step.

Results: Figure 3.7 (left) shows that PTW scheduling significantly boosts GPU performance. For example, `bfs` and `mummergpu` gain from PTW scheduling because they have a higher page divergence (so there are more memory references from different TLB

misses to schedule). We have found that PTW scheduling achieves its performance benefits by completely eliminating 10-20% of the PTW memory references and boosting PTW cache hit rates by 5-8% across the workloads. Consequently the number of idle cycles (due in large part to TLB misses) reduces from 5-15% to 4-6% (see Figure A.3 in the Appendix for more details), boosting performance.

Overall, Figure 3.7 (left) shows that thoughtful non-blocking and PTW scheduling extensions to naive baseline GPUs boosts performance to the extent that it is within 1% of an ideal, impractical, large and heavily-ported 512-entry, 32-port TLB with no access latency penalties. In fact, all the techniques reduce GPU address translation overheads under 10% for all benchmarks, well within the 5-15% range considered acceptable on CPUs.

Multiple PTWs: Multiple PTWs are an alternative to PTW scheduling. Unfortunately, their higher performance comes with far higher area and power overheads. In our studies, we have found that single PTW with augmented TLBs (4-ports with non-blocking and PTW scheduling extensions) consistently outperforms naive TLBs with more PTWs. Specifically, Figure 3.7 (right) shows a 10% performance gap between the augmented 1 PTW approach and 8 naive PTWs. We therefore assume (for the remainder of the paper), a single lower-overhead PTW with non-blocking support (hit under miss and cache overlap) and PTW scheduling augmentations.

Chapter 4

TLBs and Cache-Conscious Warp Scheduling

Having showcased the GPU address translation design space, we now move to our second contribution. There are many recent studies on improving GPU cache performance [32, 31, 36, 45, 54] via better warp scheduling. We focus on the impact of address translation on cache-conscious wavefront scheduling (CCWS) [54]. Our insights, however, are general and hold for other approaches too.

4.1 Baseline Cache-Conscious Wavefront Scheduling

Basic operation: CCWS observes that conventional warp scheduling (e.g., round robin) is oblivious to intra-warp locality, touching data from enough threads to thrash the L1 cache [54]. Carefully limiting the warps that overlap with one another promotes better cache reuse and boosts performance. CCWS accomplishes this using the baseline hardware shown in Figure 4.1. The cache holds tags and data, but also a warp identifier for the warp that allocated the cache line. A lost locality detector (LLD) maintains per-warp, set-associative cache victim tag arrays (VTAs), which store the tags of evicted cache lines. The LLD, with lost locality scoring (LLS), identifies warps that share cache working sets and those that increase cache thrashing. CCWS scheduling logic encourages warps that share working sets to run together.

CCWS operates as follows. Suppose a warp issues a memory reference. After coalescing, the address is presented to the data cache. On a cache miss, CCWS logic is invoked to determine whether multiple warps are thrashing the cache. The VTA of the current warp is probed to see if the desired cache line was recently evicted. A hit indicates the possibility of inter-warp interference. If the warp making the current request were prioritized by the scheduler, intra-warp reuse would be promoted and

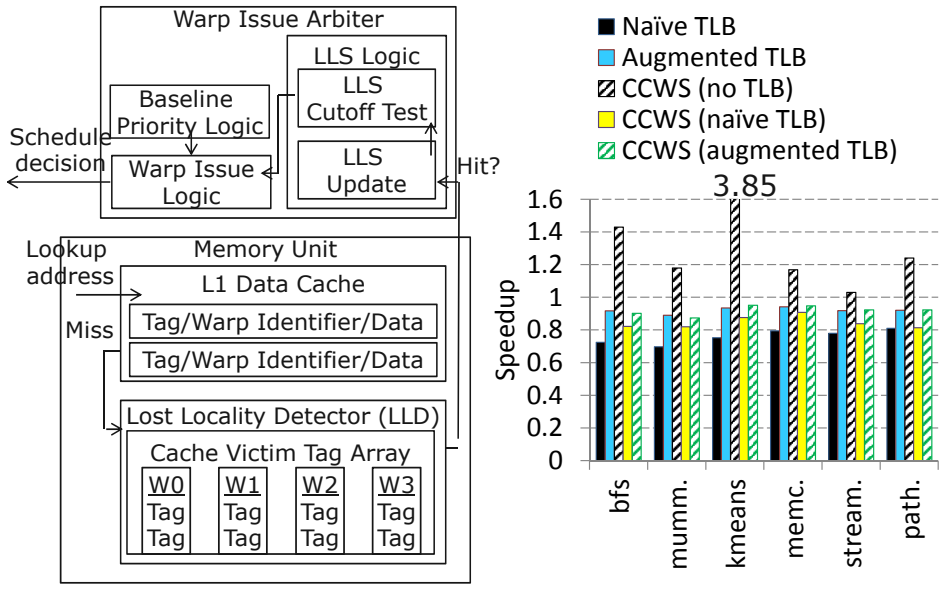

Figure 4.1: The left diagram shows conventional CCWS, with a cache victim tag array. The right diagram shows, compared to a baseline architecture without TLBs, speedup of naive, 4-ported TLBs per shader core, augmented TLBs and PTWs, CCWS without TLBs, CCWS with naive TLBs and PTWs, and CCWS with augmented TLBs and PTWs.

cache misses reduced. This information is communicated to the LLS, which maintains a counter per warp. A VTA hit increments the warp counter; whenever this happens, LLS logic sums all counter values. The LSS cutoff logic checks if the total is larger than a predefined cutoff; if so, warps with the highest counter values are prioritized since they hit most in VTAs, indicating that their lines are most-recently evicted and hence most likely to gain if not swapped out. We refer readers to the CCWS paper [54] for more details and sensitivity studies on the update and cutoff values, LSS, and LLD hardware overheads. The remainder of our work assumes 16-entry, 8-way victim VTAs per warp.

Performance of basic approach: While baseline CCWS ignored address translation [54], one might expect that boosting cache hit rate should also increase TLB hit rates. Figure 4.1 (right) quantifies the speedup (against a baseline without TLBs) of (1) naive blocking 128-entry, 4-port TLBs with one PTW (no non-blocking or PTW scheduling); (2) augmented TLBs that overlap misses with cache access and allow hits under misses (non-blocking), with PTW scheduling; (3) CCWS without TLBs; (4) CCWS with naive TLBs; and (5) CCWS with augmented TLBs.

Baseline CCWS (without TLBs) improves performance for all benchmarks by at least 20%. However, adding CCWS to naive TLBs and augmented TLBs outperform vanilla naive and augmented versions by only 5-10%. Also, the gap between CCWS with and without TLBs remains large.

4.2 Adding Address Translation Awareness

In response, we propose two CCWS schemes with TLB information.

TLB-aware CCWS (TA-CCWS): CCWS loses performance when integrating TLBs (even augmented ones) because it treats all cache misses equivalently. In reality, some cache misses are accompanied by TLB misses, others with TLB hits. In this light, baseline CCWS should be modified so that LSS logic weighs cache misses with TLB misses more heavily than those with TLB hits. TA-CCWS does exactly this, prompting more frequent TLB misses to cause the LSS counter sum to go over the threshold faster. This in turn ensures that the final pool of warps identified as scheduling candidates enjoys intra-warp cache *and intra-warp TLB reuse*.

The diagram on the left of Figure 4.1 shows TA-CCWS hardware, with practically no changes to baseline CCWS. We consider only TLB weights that are multiple of 2 so that shifters can perform the counter updates.

TLB conscious warp scheduling (TCWS): TCWS goes beyond TA-CCWS by observing that TLB and cache behavior are highly correlated. For example, TLB misses are frequently accompanied by cache misses because a TLB miss indicates that cache lines from its physical page were references far back in the past. Therefore, it is possible to subsume cache access behavior by analyzing the intra-warp locality lost on TLB behavior. TCWS exploits this correlation by replacing cache line based VTAs with smaller, lower-overhead, yet more performance-efficient TLB-based VTAs.

Figure 4.2 (right) illustrates TWCS with TLB-based VTAs containing virtual address tags. VTAs are now looked up on TLB misses rather than cache misses. VTA hits are communicated to LSS update logic, where per-warp counters are updated (similar to baseline CCWS). When the sum of the counters exceeds the prefedined cutoff, warps

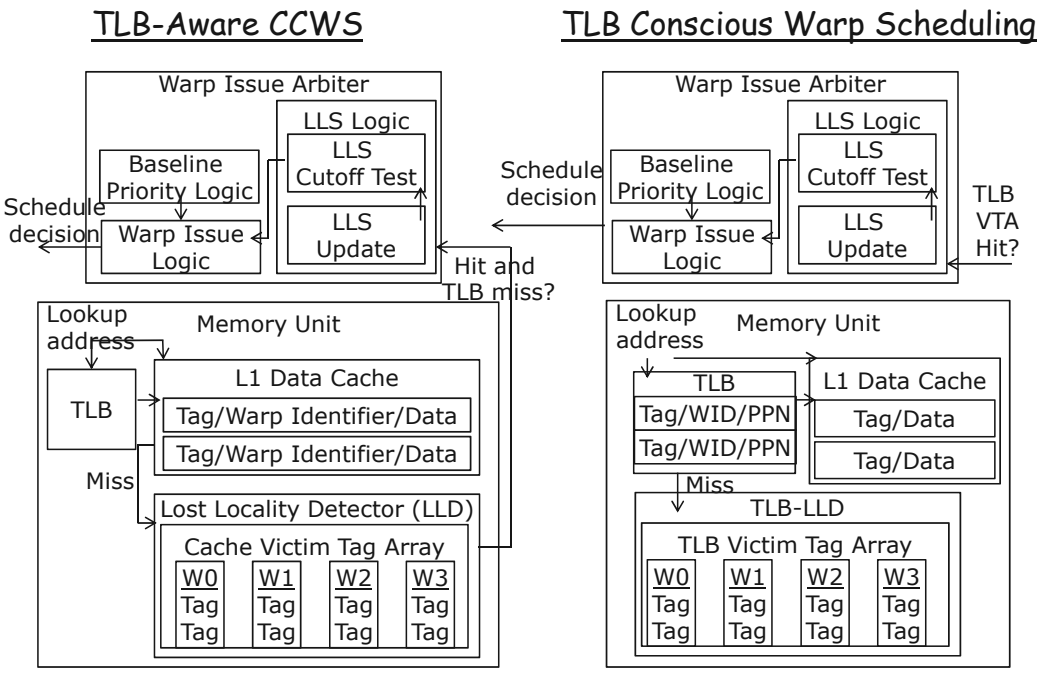

Figure 4.2: On the left, we show TA-CCWS which updates locality scores with TLB misses. On the right, we show TLB conscious warp scheduling, which replaces cache VTAs with TLB VTAs to outperform TA-CCWS, despite using less hardware.

with the highest LLS counters are prioritized.

This initial approach uses baseline CCWS but with TLB VTAs. There is, however, one problem with this approach. As described, we now update LSS scores only on TLB misses so warp scheduling decisions are relatively infrequent compared to conventional CCWS, which updates scores on cache misses. This makes CCWS less rapidly-adaptive, possibly degrading performance. Therefore, we force more frequent scheduling decisions by also updating LSS counters on TLB hits. We update by observing that each TLB set has an LRU stack (logically) of PTEs. This study assumes 128-entry, 4-way associative GPU TLBs. To update LSS logic sufficiently often, we track the LRU depth of TLB hits. Then, we update the LLS scoring logic, weighting a deeper hit more heavily since it indicates that the PTEs are closer to eviction (and TLBs/caches are likelier to suffer thrashing). A key parameter is how to vary weights with LRU depth.

Interestingly, TCWS requires less hardware than CCWS. Since TCWS VTAs maintain tags for 4KB pages, fewer of them are necessary compared to cache line VTAs. We

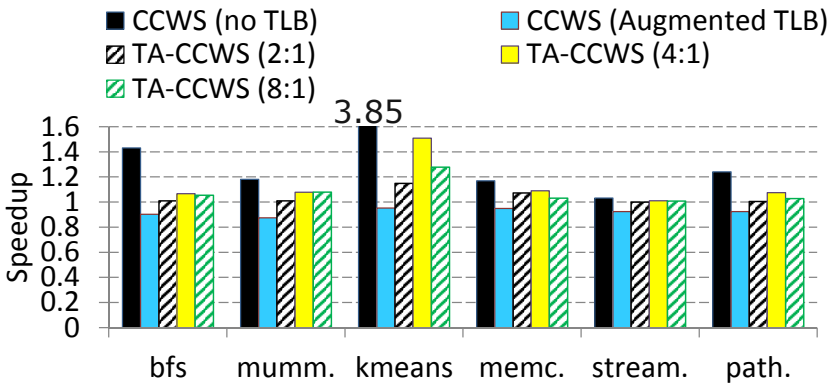

Figure 4.3: TLB-aware CCWS performance for varying weights of TLB misses versus cache misses. TA-CCWS (x:y) indicates that the TLB miss is weighted x times as much as y by the LLS scoring logic.

find that TLB-based VTAs in TCWS require half the area overhead of cache line-based CCWS.

4.3 Performance of CCWS with TLB Information

We now quantify how well TA-CCWS and TCWS perform.

TLB-aware cache conscious scheduling results: Figure 4.3 shows that updating LLS scoring logic with not just cache misses but also TLB miss information improves performance substantially. The graph separates the speedups of baseline CCWS (no TLB) and CCWS with augmented TLBs with non-blocking and PTW scheduling logic. It then shows TA-CCWS, with ratios of how much a TLB miss is weighted versus a cache miss. Clearly, weighting TLB misses more heavily improves performance. When they are weighted 4 times as much as cache misses, TA-CCWS achieves within 5-10% of CCWS without TLBs for four benchmarks (`mummergpu`, `memcached`, `streamcluster` and `pathfinder`). While `bfs` and `kmeans` still suffer degradations, we will show that TLB conscious cache scheduling boosts performance even for these benchmarks significantly.

TLB conscious cache scheduling results: Figure 4.4 shows that TCWS comprehensively achieves high performance. The graph on the left shows how varying the number of entries per warp (EPW) in the TLB VTA affects performance (this graph

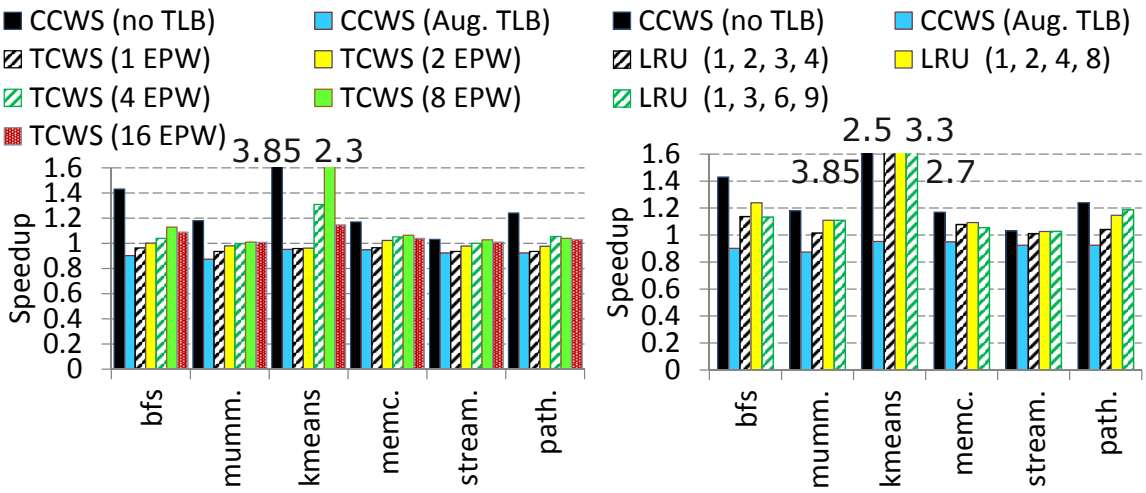

Figure 4.4: TLB conscious warp scheduling achieves within 5-15% of baseline CCWS without TLBs. The left diagram shows TCWS performance as the number of entries per warp (EPW) in the VTA is varied. The right diagram adds LRU depth weights to LLS scoring.

isolates the impact of EPWs alone without LRU depth weighting) Typically, 8 EPWs per warp VTA does best, consistently outperforming TA-CCWS.

Figure 4.4 (right) then shows how updating LLS scores based on the depth of the hit in the TLB set's LRU stack improves performance. We consider many LRU stack weights but only show three of them due to space constraints. The first scheme weights hits on the first entry of the set (MRU) with a score of 1, the second a score of 2, the third a score of 3, and the fourth a score of 4 (`LRU(1, 2, 3, 4)`). Similarly, we also show `LRU(1, 2, 4, 8)` and `LRU(1, 3, 6, 9)`. TLB misses that result in TLB VTA hits are scored as before.

`LRU(1, 2, 4, 8)` typically performs best, consistently getting within 1-15% of the baseline CCWS performance without TLBs. In fact, *not only does TCWS require only half the hardware of TA-CCWS or even CCWS, it outperforms TA-CCWS consistently.*

At a high level, these results make two points. First, warp scheduling schemes that improve cache hit rates are intimately affected by TLB hit rates too. Second, overheads from this can be effectively countered with simple, thoughtful TLB and PTW awareness. Our approach, like CPU address translation, consistently reduces overheads to 5-15% of runtime.

Chapter 5

TLBs and Thread Block Compaction

Our final contribution is to show how address translation affects mechanims to reduce branch divergence overheads. Traditionally, SIMD architectures have supported divergent branch execution by masking vector lanes and stack reconvergence [14, 21], significantly reducing SIMD throughput. Proposed solutions have included stream programming language extensions [34], allowing vector lanes to execute scalar codes for short durations [38], or more recently, dynamic warp formation techniques [21] which assimilate threads with similar control flow paths to form new warps. While address translation affects all of these approaches, we focus on the best known dynamic warp formation scheme, Thread Block Compaction (TBC) [20].

5.1 Baseline Thread Block Compaction

Basic operation: We now briefly outline the operation of baseline thread block compaction. We refer readers to the original paper for complete details [20]. In CUDA and OpenCL, threads are issued to SIMD cores in units of thread blocks. Warps within a thread block can communicate through shared memory. TBC essentially also proposes control flow locality within a thread block and is implemented using block-wide reconvergence stacks for divergence handling [20]. At a divergent branch, all warps of a thread block synchronize. TBC hardware scans the thread block's threads (which can be across multiple warps) to identify which ones follow the same control flow path. Threads are compacted into new dynamic warps according to branch outcomes and executed until the next branch or reconvergence point (where they are synchronized again for compaction). Overall, this approach increases SIMD utilization.

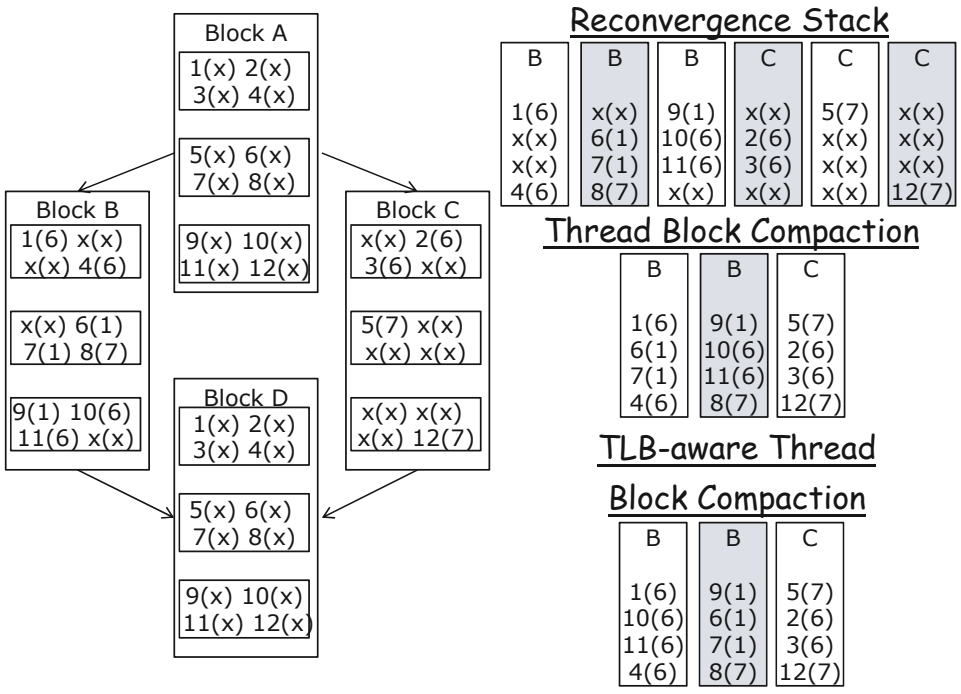

Figure 5.1: Comparison of warp execution when using reconvergence stacks, thread block compaction, and TLB-aware thread block compaction. While TLB-TBC may execute more warps, its higher TLB hit rate provides higher overall performance.

Unfortunately, blindly adding address translation has problems. Dynamically assimilating threads from different warps into new warps increases both TLB miss rates and warp page divergence (which amplifies the latency of one thread's TLB miss on all warp threads). Consider, for example, the control flow graph of Figure 5.1. In this example, each thread block contains three warps of 4 threads. Each thread is given a number, along with the virtual page it is accessing if it is a memory operation. For example, 1(6) refers to thread 1 accessing virtual page 6, 1(x) means that thread 1 is execution a non-memory instruction, and x(x) means that the thread is masked off through branching.

All threads execute blocks A and D but only threads 2, 3, 5, and 12 execute block C due to a branch divergence at the end of A (the rest execute block B). Blocks B and C consist of a memory operation. Figure 5.1 shows the order in which warps execute blocks B and C, using conventional stack reconvergence. Since there is no dynamic warp formation, it takes six distinct warp fetches to execute both branch paths. Instead, Figure 5.1 shows that forming TBC reduces warp fetches to just three, fully utilizing

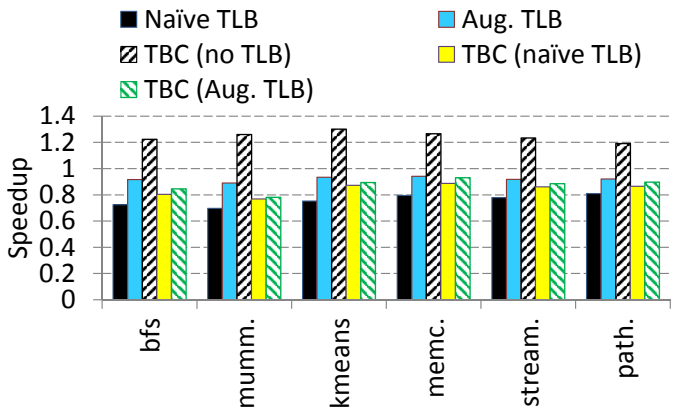

Figure 5.2: Performance of TBC without TLBs with TBC when using naive 128-entry, 4-port blocking TLBs, and when augmenting TLBs with nonblocking and PTW scheduling facilities.

SIMD pipelines.

While TBC may at first seem ideal, address translation poses problems. For example, the first dynamic warp now requires virtual pages 1 and 6. If we consider a 1-entry TLB which is initially empty, the first warp takes 2 TLB misses, the second 3, and the third 2. Instead, Figure 5.1 shows a TLB-aware scheme that potentially performs better by forming a first dynamic warp with threads requiring only virtual page 6 and a second warp requesting virtual page 1. Now, the first two warps suffer 2 TLB misses as opposed to 5 (for baseline, TLB-agnostic TBC), without sacrificing SIMD utilization.

Performance of basic approach: Figure 5.2 quantifies the performance of TLB-agnostic TBC when using address translation. Against a baseline without TLBs, we plot the speedup of TBC without TLBs, TBC with naive 128-entry, 4-port TLBs (blocking, no PTW scheduling), TBC with augmented TLBs (nonblocking, overlap misses with cache access, PTW scheduling). We also show naive TLBs and augmented TLBs without TBC. There is a significant performance gap between TBC with and without TLBs. Even with augmented TLBs, an average of 20% performance is lost compared to ideal TBC (without TLBs). *In fact, augmented TLBs without TBC actually outperform augmented TLBs with TBC.* We have found that this occurs primarily because TBC increases per-warp page divergence (by an average of 2-4 for our workloads). This further increases TLB miss rates by 5-10%. In response, we study TLB-aware TBC, assuming block-wide reconvergence stacks and age-based scheduling [20].

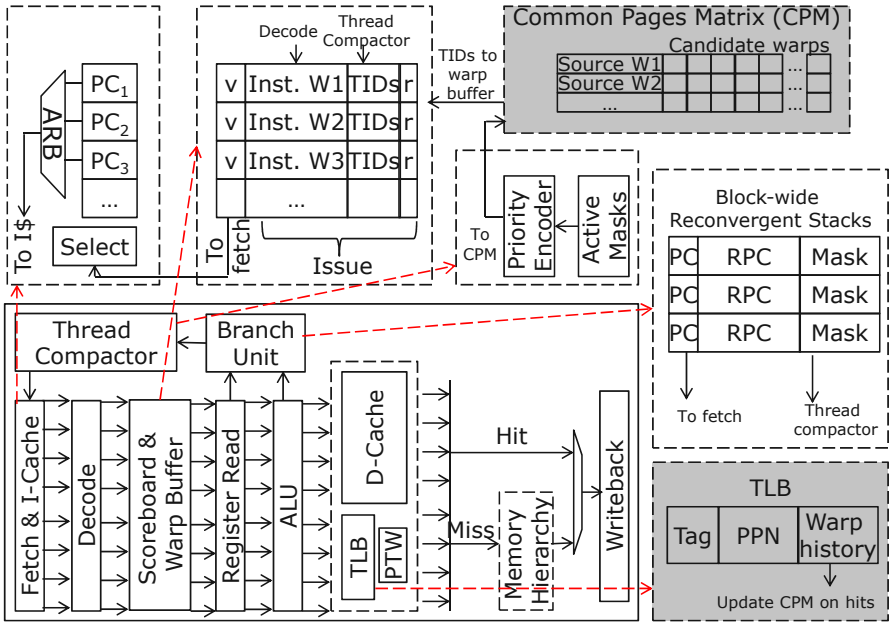

Figure 5.3: Hardware implementation of TLB-aware TBC. We add only the combinational logic in the common page matrix (CPM) and a warp history field per TLB entry. The red dotted arrows zoom into different hardware modules.

5.2 Address Translation Awareness

Hardware details: Figure 5.3 shows how we make TBC aware of address translation awareness with minimal additional hardware. The diagram shows the basic SIMD pipeline, with red dotted arrows zooming on specific modules. We add only the shaded hardware to basic TBC from past work [20].

In baseline TBC, on a divergent branch, a branch unit generates active masks based on branch outcomes. A block-wide active mask is sent to the thread compactor and is stored in multiple buffers. Each cycle, a priority encoder selects at most one thread from its corresponding inputs and sends its ID to the warp buffer. Bits corresponding to the selected threads are reset, allowing encoders to select from remaining threads in subsequent cycles. At the end of this step, threads have been formed into dynamic warps based on branch outcomes. When these dynamic warps becomes ready (indicated by the r bit per warp buffer entry) they are sent to the fetch unit, which stores program counters and initiates warp fetch. We refer readers to the TBC paper [20] for details on how the priority encoder and compactor assimilate dynamic warps.

Our contribution is to modify this basic design and encourage dynamic warp formation among warps that have historically accessed similar TLB PTEs and are hence likely to do so in the future, minimizing page divergence and miss rates. We use a table called the Common Page Matrix (CPM). Each CPM row holds a tag and multiple saturating counters. The CPM maintains a row for every warp (48 rows for our SIMD cores), each of which has a counter for every other warp (47). Each counter essentially indicates how often the warp ID associated with the row and the column have accessed the same PTEs in the past. We use this structure in tandem with the priority encoder. Threads are compacted into the warp buffer *only if the thread's original warp had accessed PTEs that threads already compacted in the new dynamic warp also accessed*. This information is easily extracted in the CPM when compacting threads; we choose a CPM row with the thread's original warp number. Then, we look up the counters using the original warp numbers of the threads that have already been compacted into the target dynamic warp. We compact the candidate thread into the dynamic warp only if the selected counters are at maximum value.

Figure 5.3 shows that CPM counters are updated on TLB hits. Each TLB entry maintains a history of warp numbers that previously accessed it. Every time a warp hits on the entry, it selects a CPM row and updates the counters corresponding to the warps in the history list. To ensure that CPM continues to adapt to program behavior, the table is periodically flushed (a flush every 500 cycles suffices).

Hardware overheads: TLB-aware TBC adds little hardware to baseline TBC. We track PTE access similarity between warps rather than between threads to reduce hardware costs and because original warps (not dynamic ones) usually have modest page divergence. This means that focusing on warp access patterns provides most of the benefits of per-thread information, but with far less overhead. The CPM has 48x47 entries; we find that 3-bit counters perform well, for a total CPM of 0.8KB. In addition, we use a history length of 2 per TLB; this requires 12 bits (since warp identifier is 6 bits). Fortunately, we observe that PTEs do not actually use full 64-bit address spaces yet, leaving 18 bits unused. We use 12 of these 18 bits to maintain history. All CPM updates and flushes occur off the critical path of dynamic warp formation.

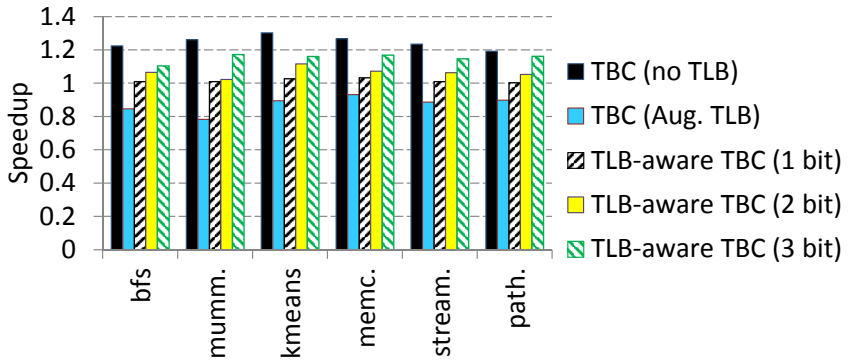

Figure 5.4: Performance of TLB-aware TBC, as the number of bits per CPM counter is varied. With 3-bits per counter, TLB-aware TBC achieves performance within 3-12% of TBC without TLBs.

5.3 Performance of TLB-Aware TBC

Figure 5.4 shows the performance of TLB-aware TBC against baseline TLB-agnostic TBC without TLBs, and with augmented TLBs. We assume 1, 2, and 3 bits per CPM counter; more bits provide greater confidence in detecting whether warps access the same PTEs. Even 1 bit counters drastically improves performance over TLB-agnostic TBC with augmented TLBs (15-20% on average). 3 bits boost performance within 5-12% of baseline TBC *without TLBs*, well within the typical 5-15% range deemed acceptable on CPUs. Like CCWS, these results mean that even though address translation tests conventional dynamic warp formation, simple tweaks recovers most lost performance.

Chapter 6

Discussion and Future Work

Shared last-level TLBs. Recent work has shown the benefits of last-level TLBs shared among multiple CPU cores [11]. We will consider the benefits of this approach in future work.

Memory management unit (MMU) caches. x86 cores use MMU caches to store frequently-used PTEs from upper levels of the page table tree [6]. These structures accelerate page table walks and may be effective for GPUs too. Future studies will consider MMU caches; note though that many benefits of their benefits are similar to adding PTW intelligence.

Large pages. Past work [3, 8, 56, 46] has shown that large pages (2MB/1GB) can potentially improve TLB performance. However, in some cases, their overheads can become an issue; for example, they require specialized OS code, can increase paging traffic, and may require pinning in physical memory (e.g., in Windows). We leave a detailed analysis of the pros and cons of large pages to future work. We do, however, present initial insights on 2MB pages. Specifically, one may, at first blush, expect large pages to dramatically reduce page divergence since it is much likelier that 32 warp threads request the same 2MB chunk rather than the same 4KB chunk. Though this is usually true, we have found that some benchmarks, `mummer` and `bfs`), still suffer high page divergences of 6 and 3 (Figure A.4 in the Appendix). Warp threads in these benchmarks have such far-flung accesses that they essentially span 12MB and 6MB of the address space. We therefore believe that a careful design space study of superpages is a natural next step in the envolution of this work.

Chapter 7

Conclusion

This work examines address translation in CPU/GPUs. We are prompted to study this because of industry trends toward fully-coherent unified virtual address spaces in heterogeneous platforms. The reasons for this trend are well-understood – fully coherent unified virtual address spaces between cores and accelerators simplify programming models and reduce the burden on programmers to manage memory. Unsuprisingly, we find that adding address translation at the L1-level of the GPUs does degrade performance. Moreover, we find that the design of GPU address translation should not be naively borrowed from CPUs (even though CPU address translation is a relatively mature technology) because the resulting overheads are untenable. We conclude that the wide adoption of heterogeneous systems, which rely on a manageable programming model, hinges upon thoughtful GPU-aware address translation.

We show that, fortunately, simple designs mindful of GPU data-parallel execution significantly reduce performance overheads from cache-parallel address translation. We also show that two recent proposals for improving GPU compute execution times (Cache-Conscious Wavefront Scheduling and Thread Block Compaction), have their gains nearly eliminated with the introduction of cache-parallel address translation. However, these schemes can be made TLB-aware with a few simple adjustments, bringing their performance back from the brink and approaching their original gains. Overall, mindful implementation of TLB-awareness in the GPU execution pipeline is not complicated, thus enabling manageable performance degradation in exchange for the industry-driven desire for enhanced programmability. Therefore, we expect there is a body of low-hanging fruit yet to be plucked for enhancing address translation in heterogeneous systems.

Appendix A

Miss Penalty Cycles and Page Divergence CDFs

A.1 TLB Miss Penalties for Blocking TLBs

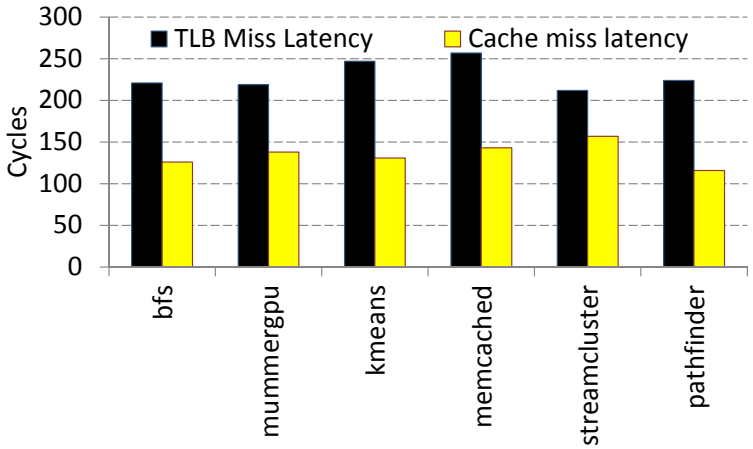

Figure A.1: *Average cycles per TLB miss, compared to L1 cache misses. TLB miss penalties are typically twice as long as L1 cache miss penalties.*

Figure A.1 shows that TLB miss penalties are expensive, well above 200 cycles in all cases. In fact, they are about twice as long as L1 cache misses because they involve multiple memory references to walk the radix tree page table. This, in combination with high miss rates, explains why naive blocking GPU TLBs degrade performance.

A.2 Page Divergence CDFs

Figure A.2 shows how many warps that execute experience page divergence values of 1, 2 to 3 pages, 4 to 7 pages, 8 to 15 pages, and 16 to 32 pages. Higher page divergence values require more ports. In general, 80% of warps for all benchmarks except for `mummergpu` have a page divergence of 3 or under, meaning that 3 ports per

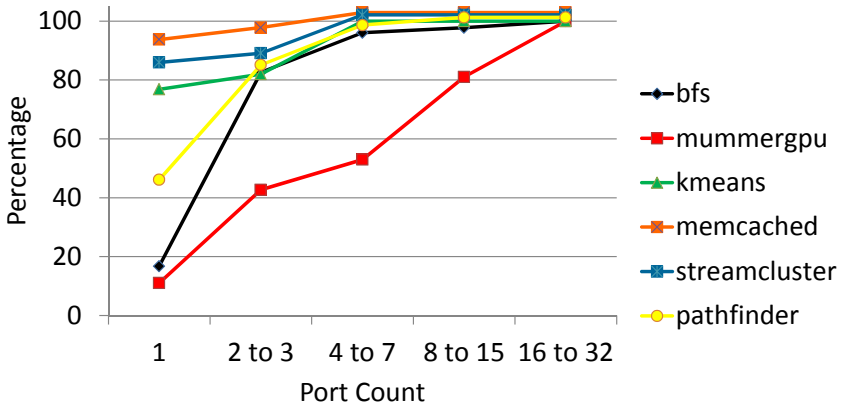

Figure A.2: *Page divergence cumulative distribution functions. We show what percentage of warps have have a page divergence of 1, 2-3, 4-7, 8-15, and 16-32.*

TLB suffices. `mummergpu` is the exception, with a page divergence of about 16 required to cover 80% of the warps.

A.3 Idle Cycle Analysis

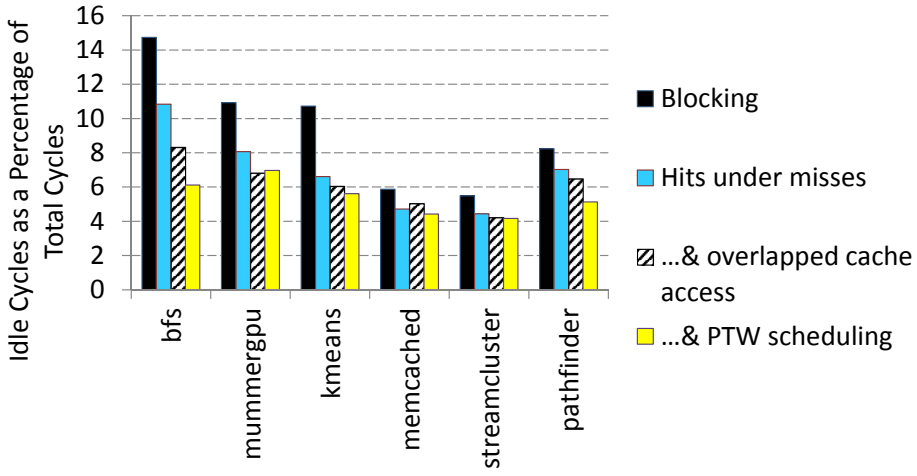

Figure A.3: *Percentage of total cycles that are idle for blocking GPU TLBs and various non-blocking and page table walking optimizations.*

Figure A.3 shows the number of idle cycles as a percentage of total benchmark runtime when using blocking GPU TLBs and as various optimizations are added. As

expected, the number of idle cycle decreases substantially (by 50% in many cases) when using nonblocking and page table walk scheduling features.

A.4 Page Divergence CDFs with Large Pages

Figure A.4 shows CDFs for the page divergence counts assuming 2MB large pages. While one might expect large pages to have sufficient reach to cover all threads of a warp, it still takes 3 pages to cover 80% of the warps from `bfs` and `pathfinder`, and over 8 pages for `mummergpu`.

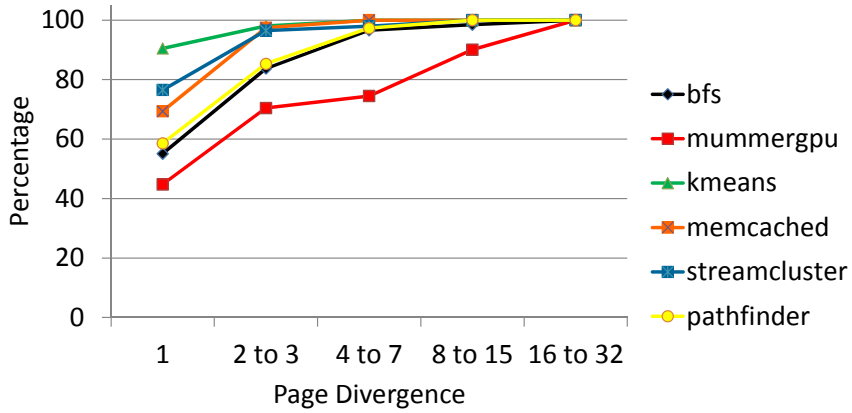

Figure A.4: *Page divergence cumulative distribution functions when using 2MB large pages. We show what percentage of warps have have a page divergence of 1, 2-3, 4-7, 8-15, and 16-32.*

References

[1] AMD. AMD I/O Virtualization Technology (IOMMU) Specification. 2006.

[2] Nadav Amit, Muli Ben Yehuda, and Ben-Ami Yassour. IOMMU: Strategies for Mitigating the IOTLB Bottleneck. *WIOSCA*, 2010.

[3] Andrea Arcangeli. Transparent Hugepage Support. *KVM Forum*, 2010.

[4] Todd Austin and Gurindar Sohi. High-Bandwidth Address Translation for Multiple-Issue Processors. *ISCA*, 1996.

[5] Ali Bakhoda, George Yuan, Wilson Fung, Henry Wong, and Tor Aamodt. Analyzing CUDA Workloads Using a Detailed GPU Simulator. *ISPASS*, 2009.

[6] T. Barr, A. Cox, and S. Rixner. Translation Caching: Skip, Don't Walk (the Page Table). *ISCA*, 2010.

[7] T. Barr, A. Cox, and S. Rixner. SpecTLB: A Mechanism for Speculative Address Translation. *ISCA*, 2011.

[8] Arkaprava Basu, Jayneel Gandhi, Jichuan Chang, Mike Swift, and Mark Hill. Efficient Virtual Memory for Big Memory Servers. *ISCA*, 2013.

[9] Arkaprava Basu, Mark Hill, and Michael Swift. Reducing Memory Reference Energy with Opportunistic Virtual Caching. *ISCA*, 2012.

[10] Ravi Bhargava, Benjamin Serebrin, Francesco Spadini, and Srilatha Manne. Accelerating Two-Dimensional Page Walks for Virtualized Systems. *ASPLOS*, 2008.

[11] A. Bhattacharjee, D. Lustig, and M. Martonosi. Shared Last-Level TLBs for Chip Multiprocessors. *HPCA*, 2010.

[12] A. Bhattacharjee and M. Martonosi. Inter-Core Cooperative TLB Prefetchers for Chip Multiprocessors. *ASPLOS*, 2010.

[13] Pierre Boudier and Graham Sellers. Memory System on Fusion APUs. *Fusion Developer Summit*, 2012.

[14] W Bouknight, Stewart Denenberg, David McIntyre, J Randall, Amed Sameh, and Daniel Slotnick. The Illiac IV System. *Proceedings of the IEEE*, 60(4):369–388, April 1972.

[15] Ian Buck, Tim Foley, Daniel Horn, Jeremy Sugerman, Kayvon Fatahalian, Mike Houston, and Pat Hanrahan. Brook for GPUs: Stream Computing on Graphics Hardware. *SIGGRAPH*, 2004.

[16] M Cekleov and M Dubois. Virtual-Addressed Caches. *IEEE Micro*, 1997.

[17] Shuai Che, Michael Boyer, Jiayuan Meng, David Tarjan, Jeremy Sheaffer, Sang ha Lee, and Kevin Skadron. Rodinia: A Benchmark Suite for Heterogeneous Computing. *IISWC*, 2009.

[18] D. Clark and J. Emer. Performance of the VAX-11/780 Translation Buffers: Simulation and Measurement. *ACM Transactions on Computer Systems*, 3(1), 1985.

[19] William Dally, Pat Hanrahan, Mattan Erez, Timothy Knight, Francois Labonte, Jung-Ho Ahn, Nuwan Jayasena, Ujval Kapasi, Abhishek Das, Jayanth Gummaraju, and Ian Buck. Merrimac: Supercomputing with Streams. *SC*, 2003.

[20] Wilson Fung and Tor Aamodt. Thread Block Compaction for Efficient SIMT Control Flow. *HPCA*, 2011.

[21] Wilson Fung, Ivan Sham, George Yuan, and Tor Aamodt. Dynamic Warp Formation and Scheduling for Efficient GPU Control Flow. *MICRO*, 2007.

[22] Isaac Gelado, Javier Cabezas, Nacho Navarro, John Stone, Sanjay Patel, and Wen mei Hwu. An Asymmetric Distributed Shared Memory Model for Heterogeneous Parallel Systems. *ASPLOS*, 2010.

[23] Blake Hechtman and Daniel Sorin. Evaluating Cache Coherent Shared Virtual Memory for Heterogeneous Multicore Chips. *ISPASS*, 2013.

[24] Taylor Hetherington, Timothy Rogers, Lisa Hsu, Mike O'Connor, and Tor Aamodt. Characterizing and Evaluating a Key-Value Store Application on Heterogeneous CPU-GPU Systems. *ISPASS*, 2012.

[25] Intel. Intel Virtualization Technology for Directed I/O Architecture Specification. 2006.

[26] Intel Corporation. TLBs, Paging-Structure Caches and their Invalidation. *Intel Technical Report*, 2008.

[27] Thomas Jablin, James Jablin, Prakash Prabhu, Feng Liu, and David August. Dynamically Managed Data for CPU-GPU Architectures. *CGO*, 2012.

[28] Thomas Jablin, Prakash Prabhu, James Jablin, Nick Johnson, Stephen Beard, and David August. Automatic CPU-GPU Communication Management and Optimization. *PLDI*, 2011.

[29] B. Jacob and T. Mudge. A Look at Several Memory Management Units: TLB-Refill, and Page Table Organizations. *ASPLOS*, 1998.

[30] Aamer Jaleel and Bruce Jacob. In-Line Interrupt Handling for Software-Managed TLBs. *ICCD*, 2001.

[31] Adwait Jog, Onur Kayiran, Nachiappan CN, Asit Mishra, Mahmut Kandemir, Onur Mutlu, Ravi Iyer, and Chita Das. OWL: Cooperative Thread Array Aware Scheduling Techniques for Improving GPGPU Performance. *ASPLOS*, 2013.

[32] Adwait Jog, Onur Kayiran, Asit Mishra, Mahmut Kandemir, Onur Mutlu, Ravi Iyer, and Chita Das. Orchestrated Scheduling and Prefetching for GPUs. *ISCA*, 2013.

[33] G. Kandiraju and A. Sivasubramaniam. Going the Distance for TLB Prefetching: An Application-Driven Study. *ISCA*, 2002.

[34] Ujval Kapasi, William Dally, Scott Rixner, Peter Mattson, John Owens, and Bruce Khailany. Efficient Conditional Operations for Data-Parallel Architectures. *MICRO*, 2000.

[35] Stefanox Kaxiras and Alberto Ros. A New Perspective for Efficient Virtual-Cache Coherence. *ISCA*, 2013.

[36] Onur Kayiran, Adwait Jog, Mahmut Kandemir, and Chita Das. Neither More nor Less: Optimizing Thread Level Parallelism for GPGPUs. *PACT*, 2013.

[37] Jesung Kim, Sang Lyul Min, Sanghoon Jeon, Byoungchul Ahn, Deog-Kyoon Jeong, and Chong Sang Kim. U-Cache: A Cost-Effective Solution to the Synonym Problem. *HPCA*, 1995.

[38] Ronny Krashinsky, Christopher Batten, Mark Hampton, Steve Gerding, Brian Pharris, Jared Casper, and Krste Asanovic. The Vector-Thread Architecture. *ISCA*, 2004.

[39] George Kyriazis. Heterogeneous System Architecture: A Technical Review. *Whitepaper*, 2012.

[40] Kevin Lim, David Meisner, Ali Saidi, Parthasarthy Ranganathan, and Thomas Wenisch. Thin Servers with Smart Pipes: Designing SoC Accelerators for Memcached. *ISCA*, 2013.

[41] Jiayuan Meng, David Tarjan, and Kevin Skadron. Dynamic Warp Subdivison for Integrated Branch and Memory Divergence. *ISCA*, 2010.

[42] Garret Morris, Benedict Gaster, and Lee Howes. Kite: Braided Parallelism for Heterogeneous Systems. 2012.

[43] Aftab Munshi. The OpenCL Specification. *Kronos OpenCL Working Group*, 2012.

[44] N. Muralimanohar, R. Balasubramonian, and N. Jouppi. CACTI 6.0: A Tool to Model Large Caches. *MICRO*, 2007.

[45] Venyu Narasiman, Michael Shebanow, Chang Joo Lee, Rustam Miftakhudinov, Onur Mutlu, and Yale Patt. Improving GPU Performance via Large Warps and Two-Level Warp Scheduling. *MICRO*, 2011.

[46] Juan Navarro, Sitaram Iyer, Peter Druschel, and Alan Cox. Practical, Transparent Operating System Support for Superpages. *OSDI*, 2002.

[47] NVidia. NVidia's Next Generation CUDA Compute Architecture: Kepler GK110. *NVidia Whitepaper*, 2012.

[48] John Owens, Mike Houston, David Luebke, Simon Green, John Stone, and James Phillips. GPU Computing. *IEEE*, 96(5), 2008.

[49] John Owens, David Luebke, Naga Govindaraju, Mark Harris, Jens Kruger, Aaron Lefohn, and Timothy Purcell. A Survey of General-Purpose Computation on Graphcis Hardware. *EUROGRAPHICS*, 26(1), 2007.

[50] Angshuman Parashar, Michael Pellauer, Michael Adler, Bushra Ahsan, Neal Crago, Daniel Lustig, Vladimir Pavlov, Antonia Zhai, Mohit Gambhir, Aamer Jaleel, Randy Allmon, Rachid Rayess, Stephen Maresh, and Joel Emer. Triggered Instructions: A Control Paradigm for Spatially-Programmed Architectures. *ISCA*, 2013.

[51] Binh Pham, Viswanathan Vaidyanathan, Aamer Jaleel, and Abhishek Bhattacharjee. CoLT: Coalesced Large Reach TLBs. *MICRO*, 2012.

[52] Wajahat Qadeer, Rehan Hameed, Ofer Shacham, Preethi Venkatesan, Christos Kozyrakis, and Mark Horowitz. Convolution Engine: Balancing Efficiency and Flexibility in Specialized Computing. *ISCA*, 2013.

[53] P Rogers. AMD Heterogeneous Uniform Memory Access. *AMD*, 2013.

[54] Timothy Rogers, Mike O'Connor, and Tor Aamodt. Cache Conscious Wavefront Scheduling. *MICRO*, 2012.

[55] Inderpreet Singh, Arrvindh Shriraman, Wilson Fung, Micke O'Connor, and Tor Aamodt. Cache Coherence for GPU Architecture. *HPCA*, 2013.

[56] M. Talluri and M. Hill. Surpassing the TLB Performance of Superpages with Less Operating System Support. *ASPLOS*, 1994.

[57] Michael Taylor. Is Dark Silicon Useful? *DAC*, 2012.

[58] N Wilt. The CUDA Handbook. 2012.

[59] Lisa Wu, Raymond Barker, Martha Kim, and Kenneth Ross. Navigating Big Data with High-Throughput, Energy-Efficient Data Partitioning. *ISCA*, 2013.