Architectural Support for Address Translation on GPUs

Designing Memory Management Units for CPU/GPUs with Unified Address Spaces

Bharath Pichai* Lisa Hsu† Abhishek Bhattacharjee*
*Department of Computer Science, Rutgers University
{bsp57, abhib}@cs.rutgers.edu
†Qualcomm Research, Qualcomm, Inc.
hsul@qti.qualcomm.com

Abstract

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.

To this end, we explore GPU Memory Management Units (MMUs) consisting of Translation Lookaside Buffers (TLBs) and page table walkers (PTWs) in unified heterogeneous systems. We show the challenges posed by GPU warp schedulers on TLBs accessed in parallel with L1 caches, which provide many well-known programmability benefits. In response, we propose modest TLB and PTW augmentations that recover most of the performance lost by introducing L1-parallel TLB access. We also show that a little TLB-awareness can make other GPU performance enhancements (e.g., cache-conscious warp scheduling and dynamic warp formation on branch divergence) feasible in the face of cache-parallel address translation, bringing overheads in the range deemed acceptable for CPUs (10-15% of runtime). We presume this initial design leaves room for improvement but anticipate the bigger insight, that a little TLB-awareness goes a long way in GPUs, will spur further work in this area.

Categories and Subject Descriptors C.1.2 [*Processor Architectures*]: Multiple Data Stream Architectures

Keywords GPUs, MMUs, TLBs, Unified address space

1. Introduction

The computer systems community has recently proposed many heterogeneous systems where CPUs are aided by accelerators targeting, among others, massive multidimensional data-sets [58], common signal processing operations [50], object caching [38], and spatially-programmed architectures [47]. To ensure widespread adoption of accelerators, their programming models must be efficient and easy to use.

A key determinant of programming model efficacy is how main memory is addressed by CPUs and accelerators. Traditionally, acceleration technologies like graphics processing units (GPUs) [13, 17, 45, 46] have maintained separate virtual and physical address spaces from CPUs. Main memory may be physically shared but is usually partitioned or allows unidirectional coherence (e.g., ARM CPU/GPUs allow accelerators to snoop CPU memory partitions but not the other way around [55]). There is, however, increasing evidence that processor vendors are shifting towards unified virtual and physical address spaces between CPUs and accelerators because of their programmability benefits. Among other advantages, unified address spaces 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. As a result, vendors like Intel, AMD, ARM, Qualcomm, and Samsung are embracing integrated CPU/GPUs with fully unified address space support, as detailed in the Heterogeneous Systems Architecture (HSA) [37] specifications. AMD's upcoming Berlin processor, for example, commits fully to HSA with its heterogeneous uniform memory access (hUMA) technology [51].

A key requirement for unified address spaces is effective hardware support for GPU virtual-to-physical address translation. In the CPU world, address translation is achieved using per-core Memory Management Units (MMUs) with Translation Lookaside Buffers (TLBs) and page table walkers (PTWs) to access frequently-used address translations from operating system (OS) page tables. Commercial CPU TLBs and PTWs are usually accessed before (or in parallel with) hardware caches, making caches physically-addressed. While this organization imposes stringent access time requirements on TLBs, it also efficiently supports multiple contexts, dynamically-linked libraries, and cache coherence.

ASPLOS '14, March 1–5, 2014, Salt Lake City, Utah, USA.
Copyright © 2014 ACM 978-1-4503-2305-5/14/03...$15.00.
http://dx.doi.org/10.1145/10.1145/2541940.2541942

In this work, we study address translation for accelerators and its role in the impending unified address space programming paradigm. We focus on general-purpose programming on GPUs because of their ubiquity and relative research maturity [18, 19, 29, 39, 43, 52, 54]. To realize the same programmability benefits as for CPUs, we target designing TLBs and PTWs accessed before (or in parallel with) the GPU's hardware caches. Our high-level insight is that the GPU's warp-based execution model and its scheduler have a critical impact on MMU performance. In particular, we find that implementing a strawman CPU-like TLB and PTW degrades GPU performance severely. Furthermore, we find that previously-proposed warp scheduling enhancements like cache-conscious wavefront scheduling and dynamic warp formation lose most of their effectiveness with naively-designed CPU-like MMUs. Fortunately however, we also show that modest optimizations recover most of this lost performance. Overall, our approaches reduce GPU TLB overheads to levels deemed acceptable in the CPU world (5-15% [6, 7, 9, 10, 16]), showing that a little TLB-awareness goes a long way in GPU design. More specifically, our contributions are:

First, we show that a CPU-like design which places TLBs and PTWs before (or in parallel with) cache access degrades performance by 20-50%. This is because: (1) cache-parallel accesses mean latency (and thus sizing) is restricted; and (2) multiple warp threads execute in lock-step, meaning that a TLB miss from a single thread can stall all warp threads, magnifying miss penalties. In response, we redesign MMUs to better match GPU execution models, finding that modest changes recover most lost performance.

Second, we show the impact of TLBs on cache-conscious warp/wavefront scheduling (CCWS) [52], recently proposed to boost GPU cache hit rates. While naively adding TLBs offsets CCWS, simple modifications yield close to ideal CCWS 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) [18], 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.

Overall, this work shows that while GPUs do require careful TLB and PTW designs to ensure reasonable performance, their architecture presents non-drastic changes from the CPU world. We therefore conclude that a little TLB-awareness is highly effective in GPUs, opening up the possibility of much follow-up research in this fruitful area.

2. Background

2.1 Address Translation on CPUs

CPU TLB design and performance have long been studied by the academic community [6, 7, 9, 10, 30, 48]. 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 [14, 35]) 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) [14]. This efficiently supports multiple contexts, dynamically-linked libraries, cache coherence among multiple cores and with direct memory access (DMA) engines.

2.2 Address Translation on CPU/GPUs

Current heterogeneous systems use rigid programming models that require separate page tables, data replication, and manual data movement between the CPU and GPU. This is especially problematic for pointer-based data structures (e.g., linked lists, trees)[1]. Recent work tries to address this using various smarter memory management schemes [20, 21, 25, 26]. Furthermore, latest CUDA releases permit limited CPU/GPU virtual address sharing [57]. However, none solve the problem using as general and flexible an approach as unified address spaces.

A critical step to unified address spaces is to implement address translation in GPUs. As a first step, Intel and AMD equip today's GPUs with Input Output Memory Management Units [1, 2, 23] (IOMMUs) with their own page tables, TLBs, and PTWs. These IOMMUs have large TLBs and are placed in the memory controller, making GPU caches virtually-addressed.

3. Our Approach

Our goal is to provide to the GPU the same programmability benefits enjoyed by the CPU. This implies that GPU address translation must support physically-addressed caches. Therefore, we study GPU MMUs where TLBs are accessed in parallel with the L1 cache. Figure 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 [18, 19, 52] or using IOMMUs resident in the memory controller.

This approach has many benefits. First, cache-parallel TLB access 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 [11]. It also allows GPUs to support page faults (using traditional or non-traditional techniques that do not support precise exceptions [34, 40]) and access memory mapped files, features desired in hUMA specifications [51] (though their

[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).

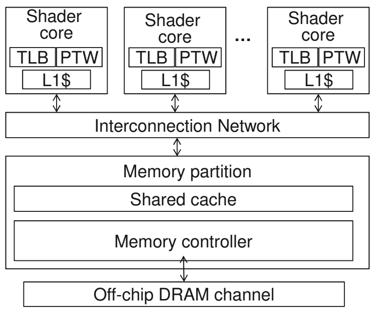

Figure 1. Our approach embeds a TLB and PTW per shader core so that all caches become physically-addressed.

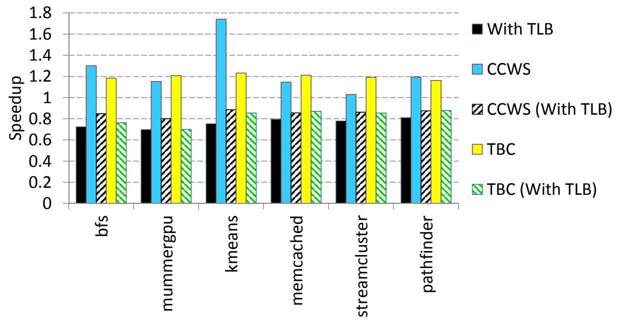

Figure 2. Compared to a baseline architecture without TLBs, speedup of naive, 3-ported TLBs per shader core, with and without cache-conscious wavefront scheduling, with and without thread block compaction. Naive TLBs degrade performance in every case.

feasibility requires a range of hardware/software studies beyond the scope of this work).

Second, physically-addressed caches support multiple contexts (one of the goals of HSA [37]) more efficiently. Virtually-addressed caches struggle due to incoherence from address synonyms [35]. 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 efficiently supports application libraries (mentioned by HSA as a design goal). Traditional virtually-addressed GPU caches suffer from address homonyms [14], which can arise when executing libraries.

Finally, cache coherence between CPU and GPU caches has long been deemed desirable [32, 37, 54]. In general, cache coherence is greatly simplified if GPU caches are physically-addressed, in tandem with CPU caches.

4. Understanding the Impact of Warp Scheduling on MMU Design

For all its programmability benefits, address translation at the L1-level is challenging because it constrains TLB access times, and hence, size. Figure 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 [18, 52]. These degradations far exceed the 5-15% overheads deemed acceptable for CPUs and may be attributed to the critical affect that warp-level execution and scheduling has on GPU MMU performance. Specifically, this occurs because:

Default GPU round robin warp scheduling interleaves the memory accesses of individual threads such that their effective temporal locality is stretched beyond the ability of the caches and TLBs to contain them. The graph on the left in Figure 3 quantifies this by plotting: (1) the number of memory references in each workload as a percentage of the total instructions; and (2) miss rates of 128-entry 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%).

Second, shader cores run warps with multiple threads in lock-step in warps/wavefronts [18, 19, 39]. Therefore, a TLB miss on one warp thread effectively stalls all warp threads, magnifying the miss penalty. Furthermore, multiple warp threads often TLB miss in tandem, stressing conventional PTWs that serialize TLB miss handling. The graph on the right in Figure 3 shows this effect by plotting page divergence (the number of distinct translations requested by a warp). We show both average page divergence and the maximum page divergence of any warp through execution. As shown, the average page divergence for some benchmarks (`bfs` and `mummergpu`) is more than 4 and 8, meaning that multiple translations are needed for a warp to progress, increasing the likelihood of multiple TLB misses. Furthermore, the maximum page divergence is consistently high, magnifying TLB miss latencies. Figure 4 quantifies these effects, comparing the latency of a GPU L1 cache miss to a TLB miss. As shown, TLB misses are about twice as expensive as L1 cache misses because they involve multiple memory references to walk the page table and because separate TLB misses are serialized using a single PTW. This, with high miss rates, explains why naive blocking GPU TLBs degrade performance.

Third, Figure 2 shows that sophisticated warp scheduling techniques beyond the traditional round robin approach lose much of their effectiveness with naively designed MMUs. Consider, for example, cache conscious wavefront scheduling [52], which introduces cache-awareness in the traditional warp scheduler to boost cache hit rates. Figure 2 compares the speedup of CCWS with and without TLBs versus a baseline setup with no TLBs. One may initially expect that warp schedulers that boost cache hit rates also improve TLB performance substantially. While CCWS with TLBs does improve performance over a baseline round-robin warp scheduler with TLBs, it still underperforms a baseline GPU with-

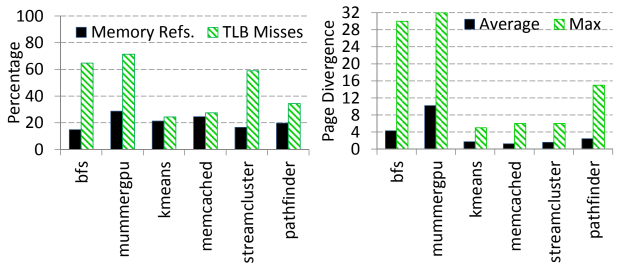

Figure 3. The left diagram shows the percentage of total instructions that are memory references and TLB miss rates; the right diagram shows the average number of distinct translations requested per warp (page divergence) and the maximum number of translations requested by any warp through the execution.

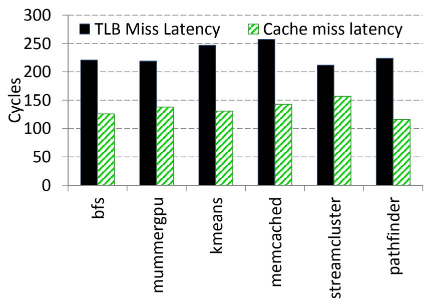

Figure 4. Average cycles per TLB miss, compared to L1 cache misses. TLB miss penalties are typically twice as long as L1 cache miss penalties.

out TLBs. The performance gap is even higher (30-50%) compared to CCWS without TLBs. As we will show, the CCWS warp scheduler must be adjusted with some notion of the relative penalties of TLB misses and cache misses to recover this lost performance.

Finally, Figure 2 also shows that warp schedulers that dynamically form warps with threads with similar control flow also lose their effectiveness with cache-parallel TLB access. Specifically, we compare thread block compaction with and without TLBs. We find that performance differences in excess of 20-25% are common between the two cases. We will show that this occurs because TBC dynamically compacts threads which access data in wildly disparate locations, increasing page divergence and hence both TLB miss rates and latencies. Specifically, we have found that TBC increases page divergence by an average of 2-4 in our workloads, and TLB miss rates by 5-10%.

The remainder of this paper refashions MMUs to match warp scheduling characteristics and improve GPU performance. Modest but informed modifications significantly reduces the degradations of Figure 2 to address translation overheads of 5-15% of system runtime (just like CPUs).

5. Methodology

5.1 Evaluation Workloads

We use server workloads from past studies on control-flow divergence and cache scheduling [18, 52]. From the Rodinia benchmarks [15], 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 [22]. Our baseline versions of these benchmarks have memory footprints greater than 1GB.

Ideally, we would run benchmarks targeted at 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) [41] will particularly benefit from unified address spaces. Our studies enable these applications and provide insights, albeit indirectly, on how GPU address translation may affect their performance.

5.2 Evaluation Infrastructure

We use GPGPU-Sim [4] with parameters similar to past work [18, 52]. 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, 7, 44]. 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.

6. Address Translation for GPUs

We now consider GPU MMU design with baseline warp scheduling. We first study the space of TLB and page table walker design options, and show their interaction with traditional round-robin warp scheduling techniques. We then present GPU-appropriate modifications to MMUs to recover much of the lost performance.

6.1 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 enjoy fast TLB lookup. 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).

TLB sizes and port counts: While larger TLBs have higher hit rates, they also require more area and access latency. Since TLB access must complete by the time the L1 cache set is selected (for virtually-indexed, physically-tagged caches), overly-lengthy hit times degrade performance. Furthermore, since TLBs use power-hungry content-addressable memories [8], their sizes must be carefully chosen. Finally, 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 walker management, 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 [27]). Hardware approaches require more area but are typically perform better [27], as they obviate the need to run an interrupt handler. 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.

Page table walk scheduling: Each page table walk requires multiple memory references (four in x86 [5]) 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 [51]. Moreover, recent work shows ways of achieving this without precise exception support [34, 40].

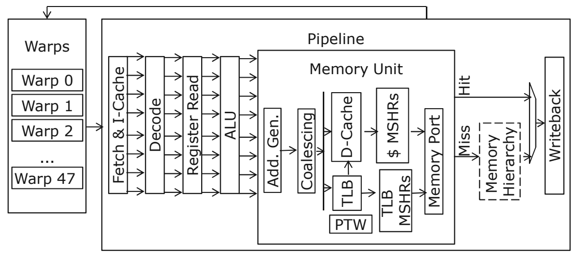

Figure 5. Shader core pipeline with address translation. We assume that L1 data caches are virtually-indexed and physically-tagged (allowing TLB lookup in parallel with cache access). All caches are physically-addressed.

6.2 Mirroring CPU Address Translation in GPUs

We begin with the natural question of how CPU-style TLBs perform in GPUs. Understanding its benefits and limitations with regard to the warp scheduler provides insights on how to better tailor MMUs for GPUs.

Figure 5 shows our initial strawman 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.

The memory unit's address generator calculates virtual addresses, which are coalesced into unique cache line references. We enhance this logic by also coalescing multiple intra-warp requests to the same virtual page (and hence PTE). This reduces 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 6.1, referring to Figure 5 when necessary.

Number and placement of TLBs: We assume 1 TLB per shader core shared among SIMD lanes, to save power and area. This is similar to the CPU approach approach of maintaining per-core TLBs.

TLB sizes and port counts: Commercial CPUs currently implement 64-512 entry TLBs per core [9, 24]. These sizes are typically picked to be substantially smaller and lower-latency than CPU L1 caches. Using a similar methodology with CACTI [42], 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 baseline GPU design, studying other sizes later.

Blocking versus non-blocking TLBs: There is evidence that that some CPU TLBs do support hits under misses because of their relatively simple hardware support [28]. However, most commercial CPUs typically use blocking TLBs [6, 48] 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 5 shows that TLBs have their own (MSHRs). We assume, like both GPU caches and past work on TLBs [39], that there is one TLB MSHR per warp thread (32 in total). MSHR allocation triggers page table walks, which inject memory requests to the shared caches and main memory.

Page table walker mechanisms, counts, and placement: Our GPU assumes hardware PTWs because: (1) they achieve higher performance than software-managed TLBs [27]; 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, while it is possible to consider multiple PTWs per GPU TLB, our baseline design mirrors CPUs (which have 1 PTW per core) and has one PTW per shader core. We investigate the suitability of this decision in subsequent sections.

Page table walk scheduling: CPU PTWs do not typically 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 [10] 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 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.

Unfortunately, we have already seen in Figure 2 that this basic design suffers from performance degradations. In response, the next section proposes a set of low-overhead optimizations to recover this lost performance.

6.3 Augmenting Address Translation for GPUs

As we have noted, the primary challenge with GPU MMUs is the nature of warp-based execution. Our baseline design assumes a warp width of 32, so it is possible for multiple threads (in the worst case, 32 threads) to demand different address translations. In the pathological case, this results in 32 simultaneous TLB misses. This section shows, however, that not only do modest TLB and PTW augmentations overcome these problems, they can actually exploit the fact that warps have threads operating in lock-step.

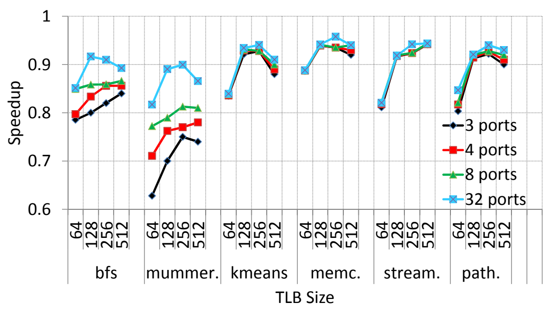

Figure 6. Performance for TLB size and port counts, assuming fixed access times. Note that TLBs larger than 128 entries and 4 ports are impractical to implement and actually have much higher access times that degrade performance.

TLB size and port counts: An ideal (but impractical) TLB is large, low-latency, and 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 6 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 with size.

Figure 6 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 6 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. This matches results from the page divergence plots of Figure 3, which showed that warps usually request far less than their maximum of 32 translations. This is because coalescing logic 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.

Blocking versus non-blocking TLBs: Using blocking TLBs, the only way to overlap miss penalties with useful work is to execute alternate warps which do not have memory references and hence do not need access to the MMU. Unfortunately however, GPU TLB miss penalties are extremely long. Therefore, GPU address translation requires more aggressive non-blocking facilities. Specifically, we investigate:

(1) Hits from one warp under misses from another warp: In this approach, the swapped-in warp executes even if it has

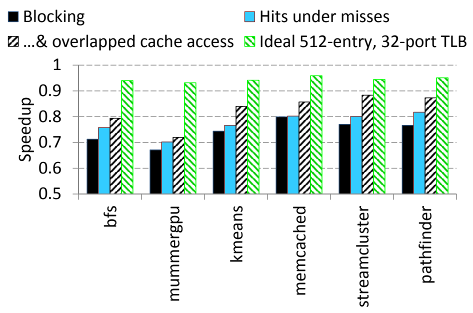

Figure 7. On a 128-entry, 4-port TLB, adding non-blocking support improves performance closer to an ideal (no increasing access latency) 32-port, 512-entry TLB.

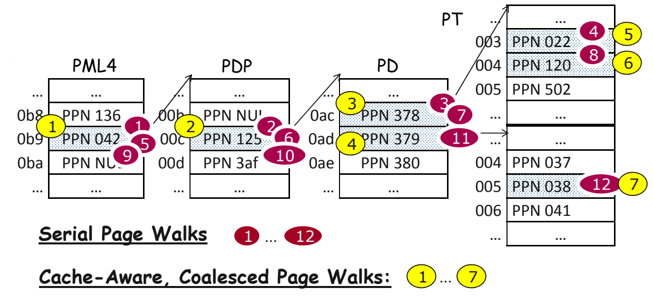

Figure 8. Three threads from a warp TLB miss on addresses `(0xb9, 0x0c, 0xac, 0x03)`, `(0xb9, 0x0c, 0xac, 0x04)`, and `(0xb9, 0x0c, 0xad, 0x05)`. A conventional page table walker carries out three serial page walks (shown with dark bubbles) , making references to (1-4), (5-8), and (9-12), a total of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

memory references, as long as they are all TLB hits. When 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 [28]. We leave more aggressive miss under miss support for future work.

(2) 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 hit in the TLB. 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, threads that hit in the TLB immediately look up the cache even if the same warp has a TLB miss on a different thread. Data found in the cache is buffered in the standard warp context state in register files (from past work [19, 52]) before the warp is swapped out. This approach requires only simple combinational logic in the PTW and MSHRs to allow TLB hits to proceed for cache lookup.

Results: Figure 7 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 access the cache without waiting for all misses from the same warp to be resolved. We compare these to an *impractical* 512-entry TLB with 32 ports *without 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, `streamcluster` gains an additional 8% performance from overlapped cache access. 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 to distinct PTEs. Consider the example in Figure 8, 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 8 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 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. We now exploit commonality in the accesses of these different page table walks in two ways to improve performance:

(1) Reducing the number of page table walk memory references: Higher order virtual address bits tend to remain unchanged 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 1GB. Since these bits index the PML4 and PDP during page table walks, multiple page table walks usually traverse similar paths. In Figure 8, all three page walks read the same PML4 and PDP locations. We therefore remember 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 read.

(2) Increasing page table walk cache hit rates: 128-byte cache lines hold 16 consecutive 8-byte PTEs. Therefore,

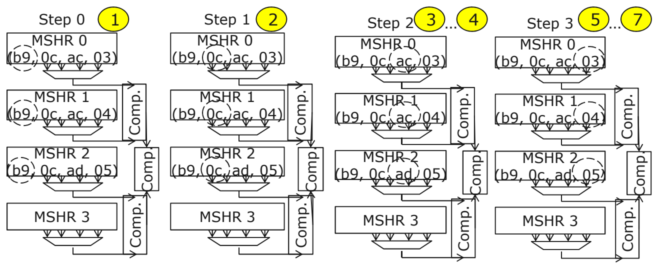

Figure 9. Our page table walk scheduler attempts to reduce the number of memory references and increase cache hit rate. The highlighted entries show what memory references in the page walks are performed and in what order. This hardware assumes a mux per MSHR, with a tree-based comparator circuit. We show 4 MSHRs though this approach generalizes to 32 MSHRs.

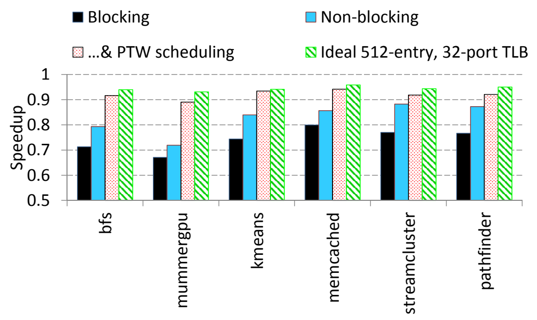

Figure 10. On a 128-entry, 4-port TLB, adding non-blocking and PTW scheduling logic achieves close to the performance of an ideal (no increasing access latency) 32-port, 512-entry TLB.

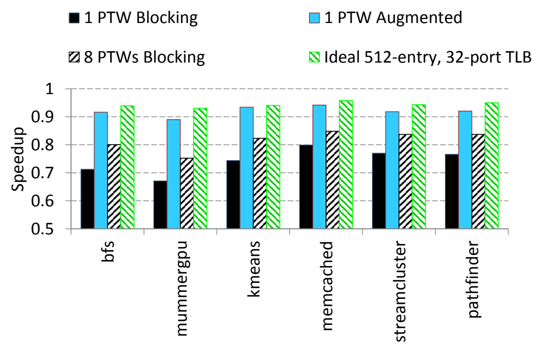

Figure 11. Our augmented TLB (with hits under misses and overlapped cache accesses), with 1 PTW outperforms 8 PTWs.

cache lines may be reused across distinct page table walks. For example, in Figure 8, two PD entries from the same cache line are used for all walks. Similarly, the PT entries for virtual pages `(0xb9, 0x0c, 0xac, 0x03)` and `(0xb9, 0x0c, 0xac, 0x04)` are on the same cache line. To exploit this, we interleave memory references from different page table walks (shown in lighter bubbles). In Figure 8, 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 9 shows PTW scheduling. 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 PT indices. Each stage checks whether the memory accesses 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 reference from Figure 8).

Figure 9 shows that a comparator tree matches indices in each stage of the algorithm. 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. 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 comparators between every pair of MSHRs (which provides maximum performance). We have found that this achieves close to pairwise comparator performance. Furthermore, we share one comparator tree with multiplexers for all page table levels. This is possible because MSHRs scans can proceed in parallel with loads from the previous step.

Results: Figure 10 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 find that PTW scheduling achieves its performance 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%, boosting performance.

Overall, Figure 10 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

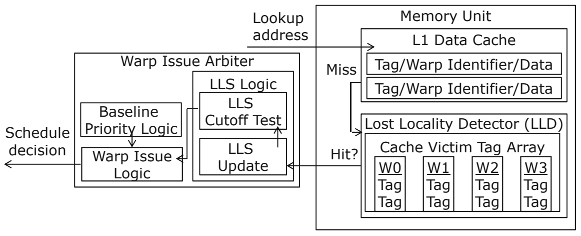

Figure 12. Conventional CCWS, with a cache victim tag array.

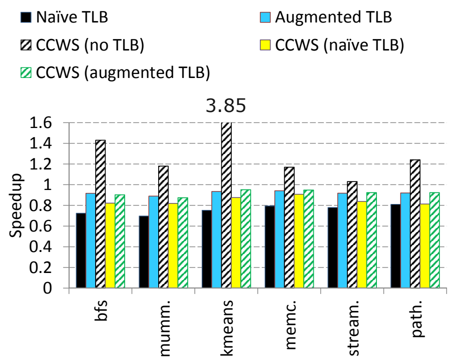

Figure 13. Compared to a baseline GPU without TLBs, speedup of naive, 4-ported TLBs per shader core, augmented TLBs and PTWs (non-blocking TLBs with PTW scheduling and cache access overlap), CCWS without TLBs, CCWS with naive TLBs and PTWs, and CCWS with augmented TLBs and PTWs.

with far higher area and power overheads. In our studies, we have found that a single PTW with augmented TLBs (4-ports with non-blocking extensions, PTW scheduling, overlapped cache access) consistently outperform naive TLBs with more PTWs. Specifically, Figure 11 shows a 10% performance gap between the augmented 1 PTW approach and 8 naive PTWs. For the rest of this paper, we therefore use a single lower-overhead PTW with non-blocking support (hit under miss and cache overlap) and PTW scheduling.

7. Cache Conscious Warp Scheduling

Our optimizations have, to this point, targeted better MMU designs for typical warp-based round-robin scheduling. There are, however, more advanced scheduling schemes that have been proposed to improve GPU cache behavior and performance on control flow divergence. In this section, we focus on the relationship between cache-conscious wavefront scheduling (CCWS) [52] and MMU design.

7.1 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 [52]. 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 13. The cache holds tags and data, but also an 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.

We now explain how CCWS operates. 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 victim tag array 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 cache misses reduced. This information is communicated to the LLD, which maintains a counter per warp. A VTA hit increments the warp counter; whenever this happens, lost locality scoring logic sums all counter values. The LLS 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 [52] for more details and sensitivity studies on the update and cutoff values, LLS, and LLD hardware overheads. The remainder of our work, like the original CCWS studies, assumes 16-entry, 8-way victim VTAs per warp.

Performance of basic approach: While baseline CCWS ignored address translation [52], one might expect that boosting cache hit rate should also increase TLB hit rates. Figure 13 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) consistently improves performance 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 (even augmented TLBs and PTWs have a 50-120% difference).

7.2 Adding Address Translation Awareness

In response to the limitations of standard CCWS, we now propose two schemes which better design MMUs to cooperate with cache aware warp scheduling.

TLB-aware CCWS (TA-CCWS): CCWS loses performance when integrating TLBs (even augmented non-blocking ones with cache overlap and PTW scheduling) because it treats all cache misses equivalently. In reality, some cache misses are

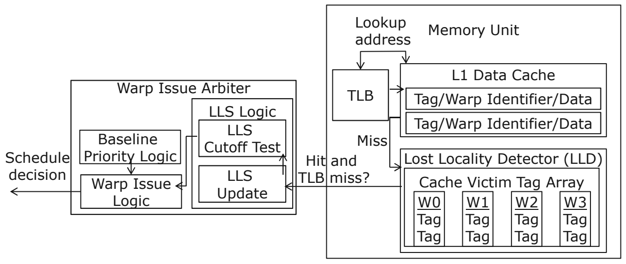

Figure 14. TLB-aware cache conscious wavefront scheduling updates locality scores with TLB misses.

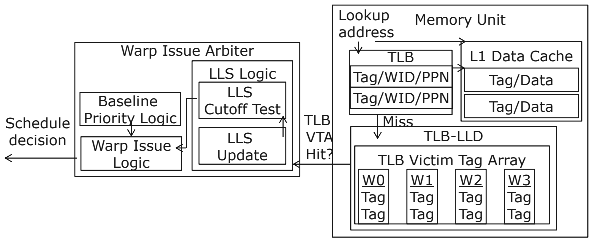

Figure 15. TLB conscious warp scheduling, which replaces cache VTAs with TLB VTAs to outperform TA-CCWS, despite using less hardware.

accompanied by TLB misses, others with TLB hits. Baseline CCWS should be modified so that lost locality scoring 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 LLS 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*.

Figure 14 shows TA-CCWS hardware, with minimal changes to CCWS. We consider only TLB weights that are multiples of 2 so that shifters perform 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 referenced long ago. 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 victim tag arrays with smaller, lower-overhead, yet more performance-efficient TLB-based VTAs.

Figure 15 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 lost locality scoring logic, where per-warp counters are updated (similar to baseline CCWS). When the sum of the counters exceeds the prefedined cutoff, warps 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.

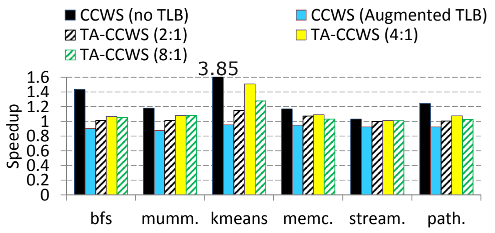

Figure 16. TLB-aware CCWS performance for varying weights of TLB misses versus cache misses. TA-CCWS (x:y) indicates that the TLB miss is weighted x times as much as y by LLS logic.

As described, we now update LLS 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 LLS 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 LLS 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.

Note that TCWS actually requires less hardware than CCWS. Since TCWS VTAs maintain tags for 4KB pages, fewer of them are necessary compared to cache line VTAs. TLB-based VTAs in TCWS require half the area overhead of cache line-based CCWS.

7.3 Performance of CCWS with TLB Information

TLB-aware cache conscious scheduling results: Figure 16 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 17 varies the number of entries per warp (EPW) in the TLB VTA, showing how it affects performance (note that this graph isolates the impact of EPWs alone without LRU depth weighting). Typically, 8 EPWs per warp VTA does best, consistently outperforming TA-CCWS.

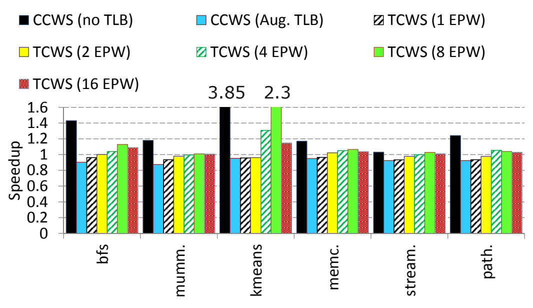

Figure 17. TLB conscious warp scheduling achieves within 5-15% of baseline CCWS without TLBs. We show TCWS as the number of entries per warp (EPW) in the VTA is varied.

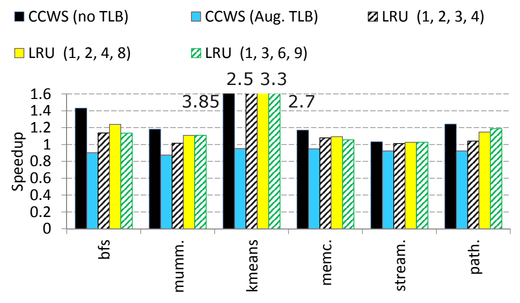

Figure 18. TLB conscious warp scheduling with LRU depth weights to added to lost locality scoring.

Figure 18 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, our graphs provide two insights. 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.

8. 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 [12, 19], significantly reducing SIMD throughput. Proposed solutions have included stream programming language extensions [31], allowing vector lanes to execute scalar codes for short durations [36], or more recently, dynamic warp formation techniques [19] 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) [18].

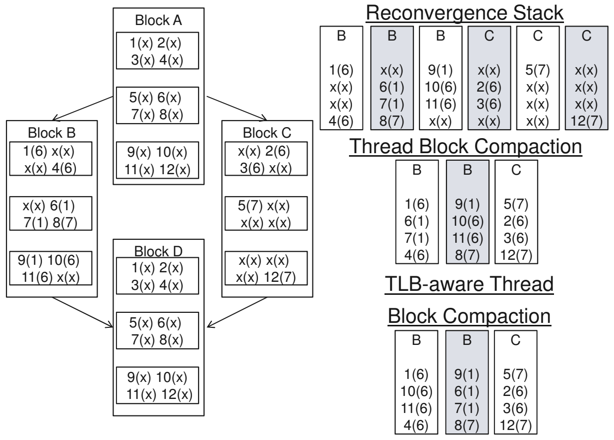

Figure 19. Comparison of warp execution when using reconvergence stacks, thread block compaction, and TLB-aware thread block compaction. While TLB-TBC may execute more warps, its higher TLB hit rate provides higher overall performance.

8.1 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 [18]. 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 [18]. 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.

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 19. 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

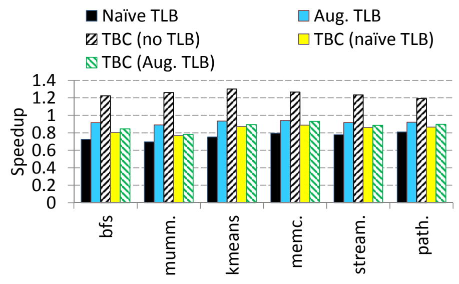

Figure 20. Performance of TBC without TLBs versus TBC when using naive 128-entry, 4-port blocking TLBs, and when augmenting TLBs with nonblocking and PTW scheduling facilities.

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 19 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 19 shows that forming TBC reduces warp fetches to just three, fully utilizing SIMD pipelines.

Address translation poses some unique problems on TBC. 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 19 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 20 quantifies the performance of baseline TBC with TLBs. 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 [18].

8.2 Address Translation Awareness

Hardware details: Figure 21 shows TBC-aware address translation with minimal additional hardware. The diagram shows the basic SIMD pipeline, with red dotted arrows

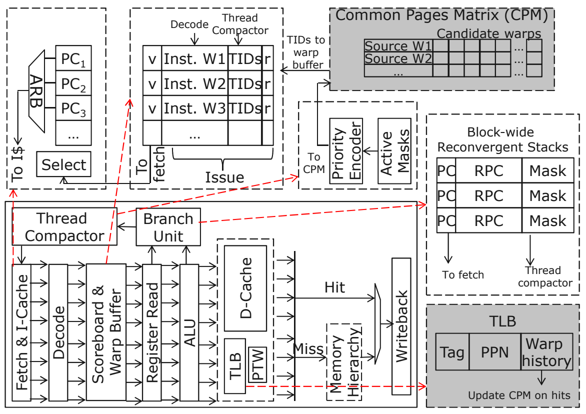

Figure 21. Hardware implementation of TLB-aware TBC. We add only the combinational logic in the common page matrix (CPM) and a warp history field per TLB entry. The red dotted arrows zoom into different hardware modules.

zooming on specific modules. We add only the shaded hardware to baseline TBC from [18].

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 [18] 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 counters are at maximum value.

Figure 21 shows that CPM counters are updated on TLB hits. Each TLB entry records warp numbers that previously

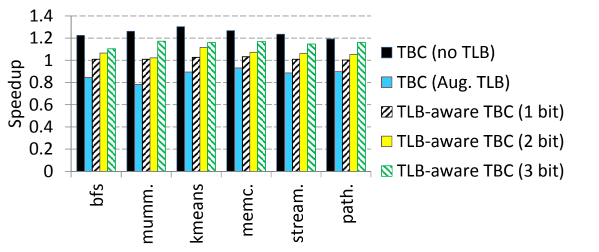

Figure 22. Performance of TLB-aware TBC, as the number of bits per CPM counter is varied. With 3-bits per counter, TLB-aware TBC achieves performance within 3-12% of TBC without TLBs.

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. 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.

8.3 Performance of TLB-Aware TBC

Figure 22 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.

9. Discussion and Future Work

Broad applicability. Our work is applicable to throughput-oriented architectures beyond GPUs like Intel's Many Integrated Core architecture [53], and accelerators like Rigel [33] and vector-thread architectures [36].

Large pages. Past work [3, 7, 44, 56] 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. 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.

Concurrent research on GPU MMUs. Concurrent with our work, Power, Hill, and Wood also study memory management unit design for GPUs [49]. Like our work, theirs shows that naively, CPU-style TLB and page table walkers can degrade GPGPU performance substantially. While we focus on the role of the warp scheduler in prudent TLB and page table walker design, Power, Hill, and Wood study the interplay between scratchpad memories, coalescing, TLBs, and eventually study schemes that provide a TLB per compute unit but then share a highly threaded page table walker and page walk cache among compute units. Our results are complementary, showing that a little TLB-awareness in GPU design can recover most of the original lost performance.

10. Conclusion

This work examines address translation in CPU/GPUs because of industry trends toward fully-coherent unified virtual address spaces in heterogeneous platforms. This unification simplifies programming models and the burden on programmers to manage memory; however, their implications on architecture remains to be studied. We find that adding address translation at the L1-level of the GPUs does degrade performance; 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. 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.

11. Acknowledgments

We thank the anonymous reviewers for their feedback on this work. This material is based upon work supported by the National Science Foundation under Grant No. 1337147 and 1253700.

References

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

[2] N. Amit, M. B. Yehuda, and B.-A. Yassour, "IOMMU: Strategies for Mitigating the IOTLB Bottleneck," *WIOSCA*, 2010.

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

[4] A. Bakhoda, G. Yuan, W. Fung, H. Wong, and T. Aamodt, "Analyzing CUDA Workloads Using a Detailed GPU Simulator," *ISPASS*, 2009.

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

[6] ——, "SpecTLB: A Mechanism for Speculative Address Translation," *ISCA*, 2011.

[7] A. Basu, J. Gandhi, J. Chang, M. Swift, and M. Hill, "Efficient Virtual Memory for Big Memory Servers," *ISCA*, 2013.

[8] A. Basu, M. Hill, and M. Swift, "Reducing Memory Reference Energy with Opportunistic Virtual Caching," *ISCA*, 2012.

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

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

[11] P. Boudier and G. Sellers, "Memory System on Fusion APUs," *Fusion Developer Summit*, 2012.

[12] W. Bouknight, S. Denenberg, D. McIntyre, J. Randall, A. Sameh, and D. Slotnick, "The Illiac IV System," *Proceedings of the IEEE*, vol. 60, no. 4, pp. 369–388, April 1972.

[13] I. Buck, T. Foley, D. Horn, J. Sugerman, K. Fatahalian, M. Houston, and P. Hanrahan, "Brook for GPUs: Stream Computing on Graphics Hardware," *SIGGRAPH*, 2004.

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

[15] S. Che, M. Boyer, J. Meng, D. Tarjan, J. Sheaffer, S. ha Lee, and K. Skadron, "Rodinia: A Benchmark Suite for Heterogeneous Computing," *IISWC*, 2009.

[16] D. Clark and J. Emer, "Performance of the VAX-11/780 Translation Buffers: Simulation and Measurement," *ACM Transactions on Computer Systems*, vol. 3, no. 1, 1985.

[17] W. Dally, P. Hanrahan, M. Erez, T. Knight, F. Labonte, J.-H. Ahn, N. Jayasena, U. Kapasi, A. Das, J. Gummaraju, and I. Buck, "Merrimac: Supercomputing with Streams," *SC*, 2003.

[18] W. Fung and T. Aamodt, "Thread Block Compaction for Efficient SIMT Control Flow," *HPCA*, 2011.

[19] W. Fung, I. Sham, G. Yuan, and T. Aamodt, "Dynamic Warp Formation and Scheduling for Efficient GPU Control Flow," *MICRO*, 2007.

[20] I. Gelado, J. Cabezas, N. Navarro, J. Stone, S. Patel, and W. mei Hwu, "An Asymmetric Distributed Shared Memory Model for Heterogeneous Parallel Systems," *ASPLOS*, 2010.

[21] B. Hechtman and D. Sorin, "Evaluating Cache Coherent Shared Virtual Memory for Heterogeneous Multicore Chips," *ISPASS*, 2013.

[22] T. Hetherington, T. Rogers, L. Hsu, M. O'Connor, and T. Aamodt, "Characterizing and Evaluating a Key-Value Store Application on Heterogeneous CPU-GPU Systems," *ISPASS*, 2012.

[23] Intel, "Intel Virtualization Technology for Directed I/O Architecture Specification," 2006.

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

[25] T. Jablin, J. Jablin, P. Prabhu, F. Liu, and D. August, "Dynamically Managed Data for CPU-GPU Architectures," *CGO*, 2012.

[26] T. Jablin, P. Prabhu, J. Jablin, N. Johnson, S. Beard, and D. August, "Automatic CPU-GPU Communication Management and Optimization," *PLDI*, 2011.

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

[28] A. Jaleel and B. Jacob, "In-Line Interrupt Handling for Software-Managed TLBs," *ICCD*, 2001.

[29] A. Jog, O. Kayiran, N. CN, A. Mishra, M. Kandemir, O. Mutlu, R. Iyer, and C. Das, "OWL: Cooperative Thread Array Aware Scheduling Techniques for Improving GPGPU Performance," *ASPLOS*, 2013.

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

[31] U. Kapasi, W. Dally, S. Rixner, P. Mattson, J. Owens, and B. Khailany, "Efficient Conditional Operations for Data-Parallel Architectures," *MICRO*, 2000.

[32] S. Kaxiras and A. Ros, "A New Perspective for Efficient Virtual-Cache Coherence," *ISCA*, 2013.

[33] J. Kelm, D. Johnson, M. Johnson, N. Crago, W. Tuohy, A. Mahesri, S. Lumetta, M. Frank, and S. Patel, "Rigel: An Architecture and Scalable Programming Interface for a 1000-core Accelerator," *ISCA*, 2008.

[34] H. Kim, "Supporting Virtual Memory in GPGPU Without Supporting Precise Exceptions," *Workshop on Memory Systems Performance and Correctness in conjunction with PLDI*, 2012.

[35] J. Kim, S. L. Min, S. Jeon, B. Ahn, D.-K. Jeong, and C. S. Kim, "U-Cache: A Cost-Effective Solution to the Synonym Problem," *HPCA*, 1995.

[36] R. Krashinsky, C. Batten, M. Hampton, S. Gerding, B. Pharris, J. Casper, and K. Asanovic, "The Vector-Thread Architecture," *ISCA*, 2004.

[37] G. Kyriazis, "Heterogeneous System Architecture: A Technical Review," *Whitepaper*, 2012.

[38] K. Lim, D. Meisner, A. Saidi, P. Ranganathan, and T. Wenisch, "Thin Servers with Smart Pipes: Designing SoC Accelerators for Memcached," *ISCA*, 2013.

[39] J. Meng, D. Tarjan, and K. Skadron, "Dynamic Warp Subdivison for Integrated Branch and Memory Divergence," *ISCA*, 2010.

[40] J. Menon, M. de Kruijf, and K. Sankaralingam, "iGPU: Exception Support and Speculative Execution on GPUs," *ISCA*, 2012.

[41] G. Morris, B. Gaster, and L. Howes, "Kite: Braided Parallelism for Heterogeneous Systems," 2012.

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

[43] V. Narasiman, M. Shebanow, C. J. Lee, R. Miftakhudinov, O. Mutlu, and Y. Patt, "Improving GPU Performance via Large Warps and Two-Level Warp Scheduling," *MICRO*, 2011.

[44] J. Navarro, S. Iyer, P. Druschel, and A. Cox, "Practical, Transparent Operating System Support for Superpages," *OSDI*, 2002.

[45] J. Owens, M. Houston, D. Luebke, S. Green, J. Stone, and J. Phillips, "GPU Computing," *IEEE*, vol. 96, no. 5, 2008.

[46] J. Owens, D. Luebke, N. Govindaraju, M. Harris, J. Kruger, A. Lefohn, and T. Purcell, "A Survey of General-Purpose Computation on Graphcis Hardware," *EUROGRAPHICS*, vol. 26, no. 1, 2007.

[47] A. Parashar, M. Pellauer, M. Adler, B. Ahsan, N. Crago, D. Lustig, V. Pavlov, A. Zhai, M. Gambhir, A. Jaleel, R. Allmon, R. Rayess, S. Maresh, and J. Emer, "Triggered Instructions: A Control Paradigm for Spatially-Programmed Architectures," *ISCA*, 2013.

[48] B. Pham, V. Vaidyanathan, A. Jaleel, and A. Bhattacharjee, "CoLT: Coalesced Large Reach TLBs," *MICRO*, 2012.

[49] J. Power, M. Hill, and D. Wood, "Supporting x86-64 Address Translation for 100s of GPU Lanes," *HPCA*, 2014.

[50] W. Qadeer, R. Hameed, O. Shacham, P. Venkatesan, C. Kozyrakis, and M. Horowitz, "Convolution Engine: Balancing Efficiency and Flexibility in Specialized Computing," *ISCA*, 2013.

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

[52] T. Rogers, M. O'Connor, and T. Aamodt, "Cache Conscious Wavefront Scheduling," *MICRO*, 2012.

[53] L. Seiler, D. Carmean, E. Sprangle, T. Forsyth, M. Abrash, P. Dubey, S. Junkins, A. Lake, J. Sugerman, R. Cavin, R. Espasa, E. Grochowski, T. Juan, and P. Hanrahan, "Larrabee: A Many-Core x86 Architecture for Visual Computing," *SIGGRAPH*, 2008.

[54] I. Singh, A. Shriraman, W. Fung, M. O'Connor, and T. Aamodt, "Cache Coherence for GPU Architecture," *HPCA*, 2013.

[55] S. Steele, "ARM GPUs Now and in the Future," 2011.

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

[57] N. Wilt, "The CUDA Handbook," 2012.

[58] L. Wu, R. Barker, M. Kim, and K. Ross, "Navigating Big Data with High-Throughput, Energy-Efficient Data Partitioning," *ISCA*, 2013.