Address Translation for Throughput-Oriented Accelerators

With processor vendors embracing hardware heterogeneity, providing low-overhead hardware and software abstractions to support easy-to-use programming models is a critical problem. In this context, this work sets the foundation for designing memory management units (MMUs) for GPUs in CPU/GPU systems, the key mechanism necessary to support the increasingly important unified address space paradigm in heterogeneous systems.

Bharath Pichai
Rutgers University

Lisa Hsu
Qualcomm Research

Abhishek Bhattacharjee
Rutgers University

To ensure widespread adoption of hardware accelerators, their programming models must be efficient and easy to use. A key determinant of programming model efficacy is how main memory is addressed. As such, processor vendors are shifting away from the traditional approach of separating CPU and accelerator virtual and physical address spaces (or variants in which physical memory is shared but physically partitioned) to systems with unified virtual and physical addresses. Among other advantages, unified address spaces make data structures and pointers globally visible among computing units, which obviates the need for expensive memory copies between CPUs and accelerators. They also unburden CPUs from pinning data pages for accelerators in main memory, which improves memory efficiency. As a result, vendors like Intel, AMD, ARM, Qualcomm, and Samsung are embracing integrated CPUs and GPUs with fully unified address space support, consistent with Heterogeneous Systems Architecture[1] (HSA) specifications. AMD's upcoming Carrizo processor, for example, commits fully to HSA, with its heterogeneous uniform memory access (hUMA) technology.[2]

Unified address spaces require effective hardware memory management units (MMUs) for virtual-to-physical address translation in accelerators. This article highlights and extends our recent studies on address translation hardware for accelerators and its role in the unified address space paradigm.[3,4] We focus on general-purpose programming on GPUs because of their ubiquity and relative research maturity.[5–7] To realize the same programmability benefits as CPUs, we target designing translation look-aside buffers (TLBs) and page table walkers (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 such as cache-conscious wavefront scheduling and dynamic warp formation lose most of their effectiveness with naively designed CPU-like MMUs. Fortunately, however, modest

Published by the IEEE Computer Society

optimizations recover most of this lost performance. Overall, we reduce GPU TLB overheads to levels deemed acceptable in the CPU world (5 to 15 percent),[8–10] which shows that a little TLB awareness goes a long way in GPU design.

The ideas we present in this article are complemented by an excellent, informative treatment of GPU MMUs published in parallel with our prior work.[11] Although the authors propose architectural alternatives to our design, many of their insights correlate with ours. We therefore point interested readers to their paper.

Our approach

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 (such as linked lists and trees). Recent academic and industry efforts try to address this using smarter memory management schemes.[12] However, full support for unified address spaces is likely to provide the most general means of solving these problems. A critical step toward unified address spaces is to implement address translation in GPUs. As a first step, Intel and AMD equip GPUs with I/O MMUs[13–15] that manage their own page tables, TLBs, and PTWs. These I/O MMUs have large TLBs and are placed in the memory controller, which makes GPU caches virtually addressed.

Our goal is to provide GPUs with the same programmability benefits enjoyed by CPUs. This implies that GPU address translation must support physically addressed caches. Figure 1 shows our approach, with per-shader-core TLBs and PTWs. As with CPUs, we assume that L1 caches are virtually indexed and physically tagged, which allows TLB access to overlap with L1 cache access. Cache-parallel TLB access eliminates the current need for CPUs to initialize, copy, pin data pages (in main memory), duplicate page tables for GPU I/O MMUs, and set up I/O MMU TLB entries,[16] which have recently been shown to be expensive operations.[17] It lets GPUs support page faults (using traditional or nontraditional techniques that do not support precise exceptions[18,19]) and access memory-mapped files, features desired in hUMA specifications[2] (although their feasibility requires a range of hardware/software studies beyond this article's scope). Furthermore, physically addressed caches support multiple contexts efficiently, without address-synonym issues (although there are other workarounds to synonyms like cache flushing on context switches, these have a relatively high performance cost). Finally, cache coherence between CPU and GPU caches has long been deemed desirable.[1,7] In general, cache coherence is greatly simplified if GPU caches are physically addressed, in tandem with CPU caches.

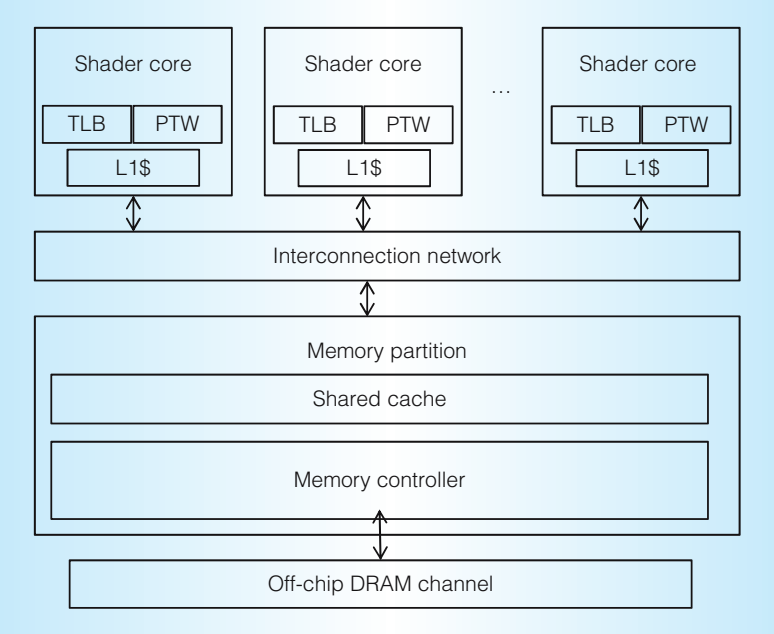

Figure 1. Our approach embeds a memory management unit (MMU) with a translation look-aside buffer (TLB) and page table walker (PTW) per shader core so that all caches become physically addressed. Adding MMUs is a necessary first-step to support unified address spaces.

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 MMU designs can severely degrade GPU performance. The plots show speedups (values higher and lower than 1 are improvements and degradations) of general-purpose GPU benchmarks using naive 128-entry, three-port TLBs with one PTW per shader

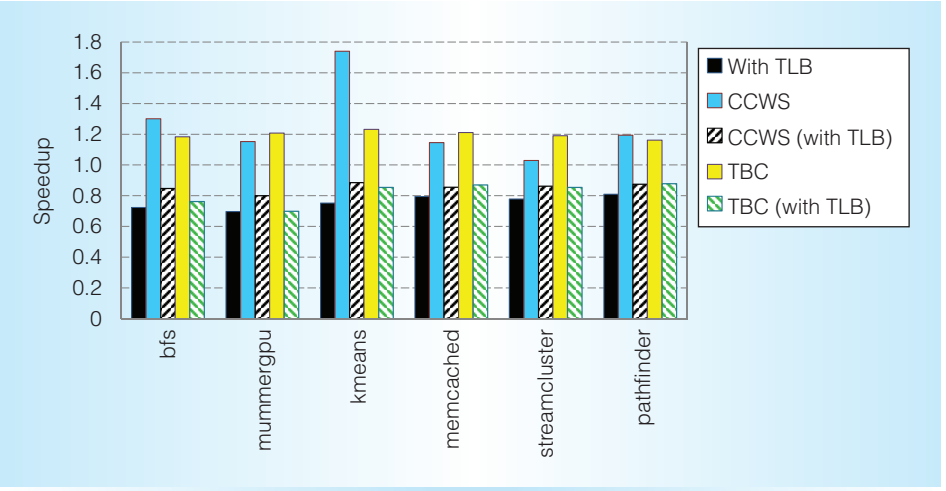

Figure 2. Compared to a baseline architecture without TLBs, speedup of naive three-ported TLBs per shader core, with and without cache-conscious wavefront scheduling (CCWS), with and without thread block compaction (TBC). Naive TLBs degrade performance in every case.

core (with TLB). Not only do naive TLBs degrade performance, they also lose 30 to 50 percent performance versus conventional cache-conscious wavefront scheduling and thread block compaction (TBC),[5,6] which far exceeds the 5 to 15 percent overheads deemed acceptable for CPUs. There are several reasons for this.

First, 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 in Figure 3a quantifies this by plotting the number of memory references in each workload as a percentage of the total instructions, and miss rates of 128-entry TLBs. Although there are fewer memory references compared to CPUs, TLB miss rates are high (ranging from 22 to 70 percent).

Second, shader cores run warps with multiple threads in lockstep in warps/wavefronts.[5] Therefore, a TLB miss on one warp thread effectively stalls all warp threads, which magnifies the miss penalty. Furthermore, multiple warp threads often TLB miss in tandem, stressing conventional PTWs that serialize TLB miss handling. The graph in Figure 3b shows this effect by plotting page divergence (the number of distinct translations requested by a warp). We show both the average page divergence and the maximum page divergence of any warp through execution. The average page divergence for some benchmarks (`bfs` and `mummergpu`) is more than four and eight, which means that multiple translations are needed for a warp to progress, and places incredible bandwidth pressure on MMUs. This, combined with the fact that TLB misses are long-latency events (we've measured TLB misses to be roughly twice as long as L1 cache misses), degrades 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 (CCWS),[6] 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 might initially expect that warp schedulers that boost cache hit rates also improve TLB performance substantially. Although CCWS with TLBs does improve performance over a baseline round robin warp scheduler with TLBs, it still underperforms a baseline GPU without TLBs. The performance gap is even higher (30 to 50 percent) compared to CCWS without TLBs.

Finally, Figure 2 shows that warp schedulers that dynamically form warps of threads of similar control flow also lose their

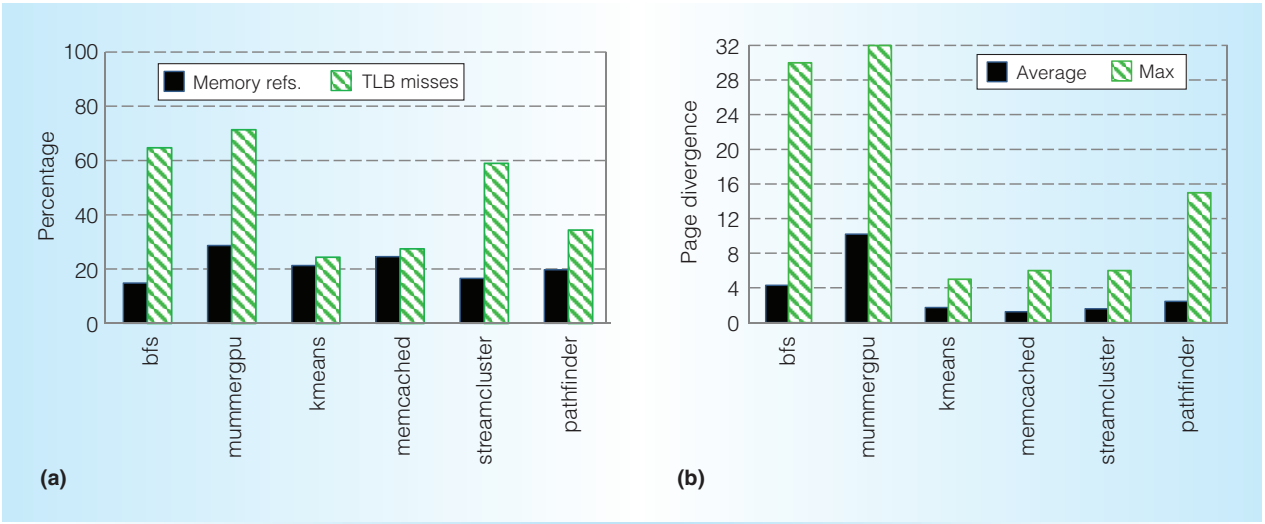

Figure 3. GPU memory reference characterization data. (a) The percentage of total instructions that are memory references and TLB miss rates. (b) The average number of distinct translations requested per warp (page divergence) and the maximum number of translations requested by any warp through the execution. Despite relatively infrequent memory references, TLB misses are extremely common.

effectiveness with cache-parallel TLB access. Specifically, we compare TBC with and without TLBs. We find that performance differences in excess of 20 to 25 percent are common between the two cases. We will show that this occurs because TBC dynamically compacts threads that access data in wildly disparate locations, which increases page divergence and hence, both TLB miss rates and latencies.

Methodology

We conducted detailed experiments using Rodinia workloads[20] and `memcached`[21] (simulated with Wikipedia traces) from past studies on control-flow divergence and cache scheduling.[5,6] We used GPGPU-Sim[22] to study the impact of MMUs on these workloads.

We assumed 30 single-instruction, multiple-thread cores; 32-thread warps; and a pipeline width of eight. We had 1,024 threads, 16 Kbytes of shared memory, and 32-Kbyte L1 data caches (with 128 byte lines and least-recently used (LRU) per core. We also used eight memory channels with 128 Kbytes of unified L2 cache space per channel. Most of our results focused on 4-Kbyte pages because of the additional challenge imposed by small page size; however, we will also present initial results for large 2-Mbyte pages. Note that large pages, while effective, aren't free and can have their own overheads in certain situations.[8,23] As a result, many applications are restricted to 4-Kbyte page sizes; it is important for our GPUs with address translation to be compatible with them.

Address translation for GPUs

In this section, we detail key architectural decisions that influence effective GPU MMU design. Our studies indicate the challenges and opportunities presented by warp-based execution scheduling on TLBs and PTWs.

Mirroring CPU address translation in GPUs

We begin with the natural question of how CPU-style TLBs perform in GPUs. Understanding their benefits and limitations with regard to the warp scheduler provides insights on how to better tailor MMUs for GPUs. Figure 4 shows a strawman design. Each shader core maintains a TLB and a PTW and has 48 warps (the minimum scheduling unit) per shader core. A warp's threads execute the same instruction in lockstep. Instructions are fetched from an instruction cache and operands are read from a banked register file. Loads and stores access the memory unit.

The memory unit's address generation unit calculates virtual addresses, which are coalesced into unique cache line references.

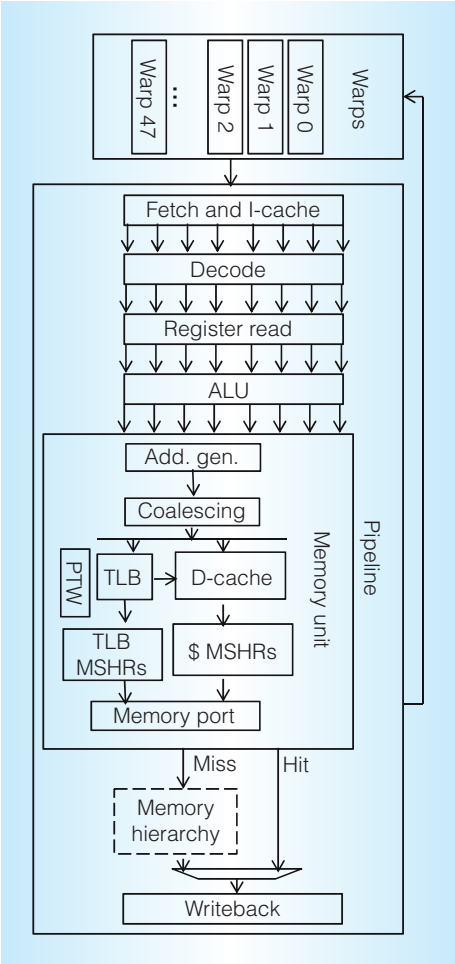

Figure 4. Shader core pipeline with address translation. We assume that L1 data caches are virtually indexed, physically tagged (allowing TLB lookup in parallel with cache access). All caches (L1 and shared caches, which are not shown) are physically addressed.

We enhance this logic by also coalescing multiple intrawarp requests to the same virtual page (and hence page table entry (PTE). This is crucial for reducing TLB access traffic and port counts. Two sets of accesses are now available: unique cache accesses and unique PTE accesses. These are presented in parallel to the TLB and the virtually indexed, physically tagged data cache.

Number and placement of TLBs. We assume that a single TLB per shader core is shared among single-instruction, multiple-data (SIMD) lanes, to save power and area. This is similar to the CPU approach of maintaining per-core TLBs. Alternate approaches exist; for example, it is possible to implement one TLB per SIMD lane. These approaches, however, typically consume more power and area.

TLB sizes and port counts. Commercial CPUs currently implement 64- to 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,[25] we found that 128-entry TLBs are the largest possible structures that do not increase the access time of 32-Kbyte GPU L1 data caches.

Blocking versus nonblocking TLBs. One way to reduce the impact of TLB misses is to overlap them with useful work. To this end, some CPU TLBs support hits under misses.[26] However, most commercial CPUs typically use blocking TLBs because of their high hit rates.[27,28] We assume blocking TLBs in our baseline GPU design; therefore, TLB misses prompt the scheduler to swap another warp into the SIMD pipeline. Swapped-in warps executing nonmemory instructions proceed unhindered (until the original warp's PTWs finish and it completes the writeback stage). Swapped-in warps with memory references, however, stall in this design because they require nonblocking support.

Figure 4 shows that TLBs have their own miss status holding registers (MSHRs). We assume, as with both GPU caches, 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.

PTW mechanisms, counts, and placement. Our GPU, like most CPUs today, assumes hardware PTWs because they achieve higher performance than software-managed TLBs and do not need to run OS code (which GPUs cannot currently execute). CPUs usually place page table walks close to each TLB so that page table walks can be initiated quickly on misses. We use the same logic to place hardware PTWs next to TLBs. Finally, CPUs maintain one PTW per TLB. Although it's possible to consider multiple PTWs per GPU TLB, our baseline design mirrors CPUs

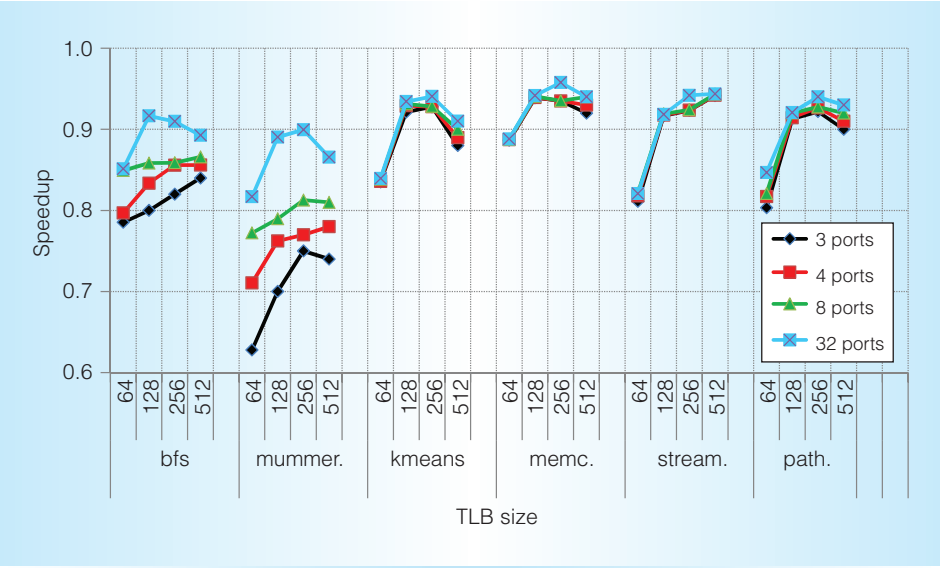

Figure 5. Performance for TLB size and port counts, assuming fixed access times. Excessively large and multiported TLBs increase access time despite improving miss rates and available bandwidth.

and similarly has one PTW per shader core. We investigate the suitability of this decision in subsequent sections.

System-level issues (shootdowns and page faults). A CPU usually shoots down TLBs on remote cores when its own TLB updates an entry, using software interprocessor interrupts (IPIs). We assume the same approach for GPUs (that is, 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 might be able to run dedicated OS code for shootdowns and page faults without interrupting CPUs. We leave this for future work.

Unfortunately, we already saw 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.

Augmenting address translation for GPUs

We now detail how MMU designs must be redesigned to suit warp-based execution.

TLB size and port counts. An ideal (but impractical) TLB is large and low latency. Furthermore, it is heavily multiported, with one port per warp thread (32 in total). Although having more ports facilitate quick lookups and miss detection, they also significantly increase area and power. Figure 5 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 three (like L1 CPU TLBs) to an ideal number of 32. We present speedups against the no-TLB case (speedups are under one because adding TLBs degrades performance).We use CACTI to assess access time increases with size.

Figure 5 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 5 also shows that while port counts do impact performance (particularly for `mummergpu` and `bfs`), modestly increasing from three ports (in our naive baseline) to four 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 fewer than their maximum of 32 translations.

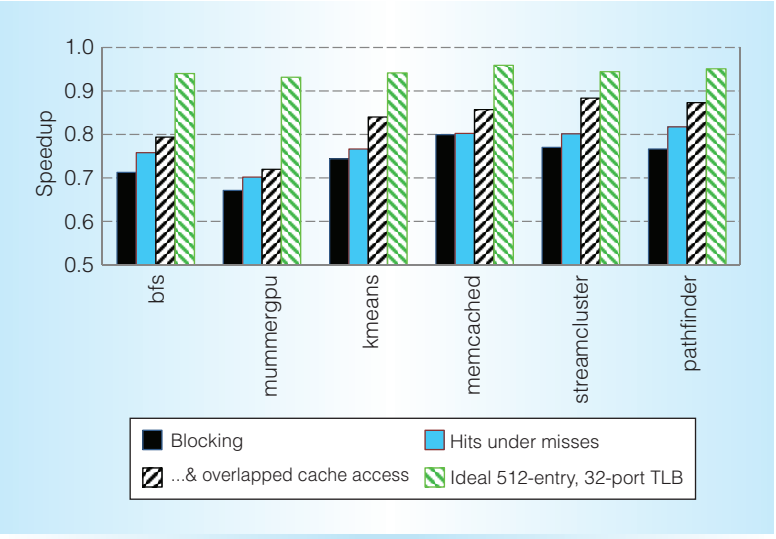

Figure 6. On a 128-entry, four-port TLB, adding nonblocking support improves performance closer to an ideal (no increasing access latency) 32-port, 512-entry TLB. Of the nonblocking strategies, more intelligently scheduling the memory references of page table walks provides the highest benefits.

This is because coalescing logic before TLBs reduces requests to the same PTE into a single lookup.

Blocking versus nonblocking TLBs. We investigated two mechanisms to overlap TLB misses with useful work.

- *Hits under misses.* 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; we leave more aggressive miss-under-miss support for future work.
- *Overlapping TLB misses with cache accesses.* It is also possible to overlap TLB misses with work from the warp that originally missed. Because warps have multiple threads, even when some miss, others may hit in the TLB. Since hits immediately yield physical addresses, it's 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 because this warp's data is more likely 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[6]) 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.

Figure 6 quantifies the benefits of nonblocking TLBs, normalized to a baseline without TLBs. We first permit hits under misses, then allow TLB hits to proceed to the cache without waiting for all misses from the same warp to be resolved. We compare these to an ideal and impractical 512-entry TLB with 32 ports without increased access latencies.

Although hits under misses improve performance, it is even more effective to immediately look up the cache for threads that TLB hit. For example, `streamcluster` gains an additional 8 percent performance from overlapped cache access.

PTW 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 7, which shows three concurrent x86 page walks, each of which traverses a four-level radix tree page table. We assume that the three threads miss 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

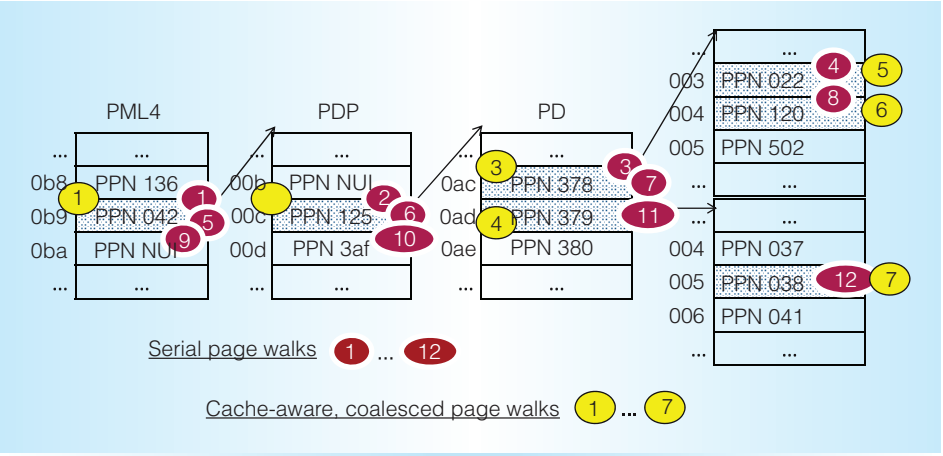

Figure 7. Three threads from a warp TLB miss on addresses (0xb9, 0x0c, 0xac, 0x03), (0xb9, 0x0c, 0xac, 0x04), and (0xb9, 0x0c, 0xad, 0x05). A conventional PTW carries out three serial page walks (shown with dark bubbles), making references to 1–4, 5–8, and 9–12—a total of 12 loads. Our cache-aware coalesced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

shared cache (several tens of cycles) or main memory. However, we augment this basic PTW by noting that scheduling the memory references of the distinct walks in an interleaved fashion can eliminate redundant memory accesses and boost cache hit rates.

Specifically, note that higher-order virtual address bits tend to remain unchanged across memory accesses as they cover large memory regions (for example, consider bits 47–39 and 38–30, which cover 1-Gbyte spaces). Therefore, we can replace the corresponding redundant memory reads with single reads. In addition, note that 128-byte cache lines hold 16 consecutive 8-byte PTEs, which means that there is potential for cache line reuse across different page table walks. For example, in Figure 7, 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 7, 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), which boosts hit rates. Figure 8 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 to 20 percent of the PTW memory references and boosting PTW cache hit rates by 5 to 8 percent across the workloads. Consequently, the number of idle cycles (due in large part to TLB misses) reduces from 5 to 15 percent to 4 to 6 percent, which boosts performance.

Overall, Figure 8 shows that thoughtful nonblocking and PTW scheduling extensions to naive baseline GPUs boost performance to the extent that it is within 1 percent 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 to less than 10 percent for all benchmarks, which is well within the 5 to 15 percent range considered acceptable on CPUs.

Integration with advanced warp schedulers

Having detailed basic GPU MMU design, we now briefly investigate the relationship between TLBs and advanced warp-scheduling schemes such as CCWS and TBC.

Cache-conscious wavefront scheduling

CCWS is one of several recent warp-scheduling techniques that boosts cache hit rates.[6]

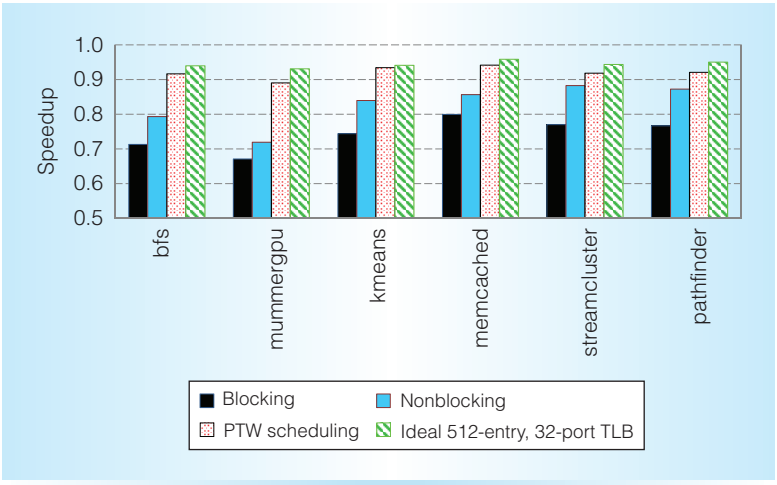

Figure 8. On a 128-entry, four-port TLB, adding nonblocking and PTW scheduling logic achieves close to the performance of an ideal (no increasing access latency) 32-port, 512-entry TLB. Note that these schemes achieve performance benefits with low-overhead hardware enhancements.

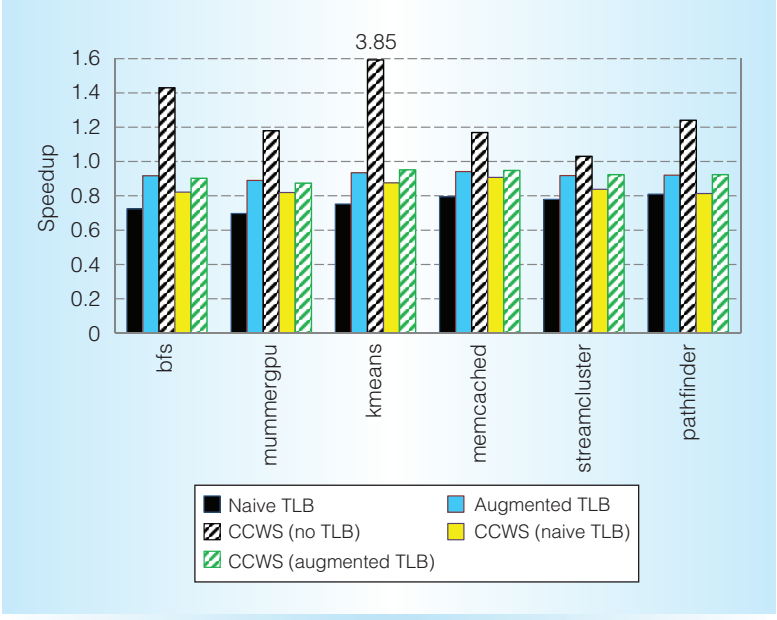

Figure 9. Compared to a baseline architecture without TLBs, speedup of naive, four-ported TLBs per shader core, augmented TLBs and PTWs (nonblocking TLBs with PTW scheduling and cache access overlap), CCWS without TLBs, CCWS with naive TLBs and PTWs, and CCWS with augmented TLBs and PTWs.

Its key intuition is that round robin warp scheduling is oblivious to intrawarp data locality and thrashes L1 caches by switching between too many warps too aggressively. By carefully limiting which warps overlap with one another, CCWS promotes cache reuse and boosts performance.

We might expect, at first blush, that boosting cache-hit rate would naturally mitigate TLB miss overheads. Figure 9 quantifies the speedup (against a baseline without TLBs) of

- naive blocking 128-entry, four-port TLBs with one PTW (no nonblocking or PTW scheduling);
- augmented TLBs that overlap misses with cache access and allow hits under misses (nonblocking), with PTW scheduling;
- CCWS without TLBs;
- CCWS with naive TLBs; and
- CCWS with augmented TLBs.

Baseline CCWS (without TLBs) improves performance for all benchmarks by at least 20 percent. However, adding CCWS to naive TLBs and augmented TLBs outperforms vanilla naive and augmented versions by only 5 to 10 percent. Also, the gap between CCWS with and without TLBs remains large (even augmented TLBs and PTWs have a 50 to 120 percent difference).

Fortunately, our recent work identifies the reason for this and shows that extremely simple and low-overhead hardware can resolve these problems.[3,4] Intuitively, we find that CCWS loses performance when integrating TLBs because it treats all cache misses equivalently. In reality, some cache misses are accompanied by TLB misses and others with TLB hits. In practice, we show that adding this information cuts TLB overheads to less than 10 percent while requiring almost negligible hardware changes, and, in some cases, even less hardware. We use two techniques for this.

In the first approach, we note that CCWS uses logic to track which cache misses occur because of more frequent warp switches by incrementing dedicated counters when this happens. Separate counters are maintained per warp, and when their sum is higher than a threshold, the CCWS scheduler backs off from switching between multiple warps too aggressively. We leverage this approach by merely requiring that cache misses with prior TLB misses further increment these counters. Just this simple fix almost entirely eliminates TLB overheads.

In our second approach, we go a step further by noting that CCWS uses additional

logic at the cache-line granularity to detect those cache misses which would have been hits with better warp scheduling. We observe that since TLB and cache misses are highly correlated, we can replace this logic with page-level information instead of cache line-level information. This TLB-aware CCWS scheme comes within 10 percent of the performance of baseline CCWS but uses only half the hardware resources.

Thread block compaction

We also studied the relationship between control flow warp schedulers and intelligent MMUs. Specifically, we studied TBC,[5] although our insights broadly apply to other approaches. 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.[5] 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 in Figure 10. Each thread block contains three warps of four 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 executing a nonmemory 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 because of a branch divergence at the end of A (the rest execute block B). Blocks B and C comprise a memory operation. Figure 10 shows the order in which warps execute blocks B and C using conventional stack reconvergence. Because there is no dynamic warp formation, it takes six distinct warp fetches to execute both branch paths. Instead, Figure 10 shows that forming TBC reduces warp fetches to just three, fully utilizing SIMD pipelines.

Address translation poses unique problems on TBC. For example, the first dynamic warp now requires virtual pages 1 and 6. If we consider a one-entry TLB that's initially empty, the first warp takes two TLB misses, the second takes three, and the third takes two. Instead, Figure 10 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 two TLB misses as opposed to five (for baseline, TLB-agnostic TBC) without sacrificing SIMD utilization. The overall effect is a performance loss of 20 to 25 percent.

Fortunately, our recent work shows that these problems easily can be overcome with modest hardware.[3,4] At a high level, TLB-aware TBC tracks the history of past dynamic warps to identify which threads tend to access similar regions in memory. It then uses this information to assess which threads to dynamically assimilate into warps. Our detailed experiments reveal that this approach alone (which requires an additional area budget of less than 1 percent of the shader core) drives TLB miss overheads to less than 10 percent of runtime for every single workload.

Discussion

Our initial GPU MMU design presents several possibilities for further improvement. We list a brief subset of these possibilities below.

Shared last-level TLBs

Recent work has shown the benefits of last-level TLBs shared among multiple CPU cores.[9] Similarly, we have gone beyond our initial work[3] to study the benefits of architecting L2 TLBs shared among shader cores. In practice, we found that L1 TLB misses occur so often that huge bandwidth requirements are placed on shared L2 TLBs.

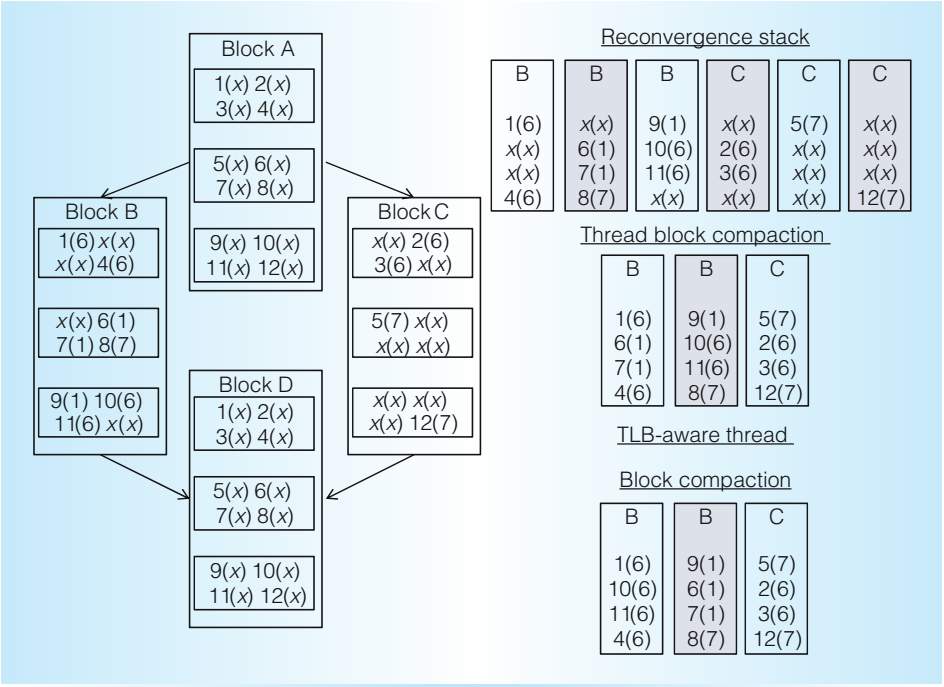

Figure 10. Comparison of warp execution when using reconvergence stacks, TBC, and TLB-aware TBC. Note the difference in schedules for TLB-aware behavior.

Consequently, it is difficult for the shared TLB to support enough ports while retaining high capacity. In practice, we have therefore found this approach to have limited utility.

MMU caches

MMU caches are used by x86 cores to store frequently used PTEs from upper levels of the page table tree.[29] These structures accelerate page table walks; we've found that, in general, shared MMU caches provide additional performance boosts of 2 to 3 percent across our workloads. In general, these are a better alternative to shared last-level TLBs because their capacity is implicitly higher as they cache upper page table levels; hence, they can more easily support the greater bandwidth demands of many shader cores.

This work examines address translation in CPUs and GPUs. This unification simplifies programming models and the burden on programmers to manage memory; however, its implications on architecture remain to be studied. Adding address translation at the L1-level of the GPU does degrade performance; GPU address translation should not be naively borrowed from CPUs because the overheads are untenable. Overall, mindful implementation of TLB-awareness in GPUs is not complicated, which enables manageable performance degradation in exchange for the desire for enhanced programmability. We expect there is a body of low-hanging fruit for enhancing address translation in heterogeneous systems.

MICRO

References

1. G. Kyriazis, *Heterogeneous System Architecture: A Technical Review*, white paper, 2012.
2. P. Rogers, "AMD Heterogeneous Uniform Memory Access," AMD, 2013.
3. B. Pichai, L. Hsu, and A. Bhattacharjee, "Architectural Support for Address Translation on GPUs," ASPLOS, 2014.
4. B. Pichai, L. Hsu, and A. Bhattacharjee, *Architectural Support for Address Translation on GPUs*, tech. report DCS-TR-703, Dept. of Computer Science, Rutgers Univ., 2014.
5. W. Fung and T. Aamodt, "Thread Block Compaction for Efficient SIMT Control Flow," HPCA, 2011.

6. T. Rogers, M. O'Connor, and T. Aamodt, "Cache Conscious Wavefront Scheduling," MICRO, 2012.
7. I. Singh et al., "Cache Coherence for GPU Architecture," HPCA, 2013.
8. A. Basu et al., "Efficient Virtual Memory for Big Memory Servers," ISCA, 2013.
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. J. Power, M. Hill, and D. Wood, "Supporting x86-64 Address Translation for 100s of GPU Lanes," HPCA, 2014.
12. N. Wilt, "The CUDA Handbook," 2012.
13. AMD, "AMD I/O Virtualization Technology (IOMMU) Specification," 2006.
14. N. Amit, M.B. Yehuda, and B.-A. Yassour, "IOMMU: Strategies for Mitigating the IOTLB Bottleneck," WIOSCA, 2010.
15. Intel, "Intel Virtualization Technology for Directed I/O Architecture Specification," 2006.
16. P. Boudier and G. Sellers, "Memory System on Fusion APUs," Fusion Developer Summit, 2012.
17. M. Malka et al., "rIOMMU: Efficient IOMMU for I/O Devices that Employ Ring Buffers," ASPLOS, 2015.
18. H. Kim, "Supporting Virtual Memory in GPGPU without Supporting Precise Exceptions," Workshop on Memory Systems Performance and Correctness in conjunction with PLDI, 2012.
19. J. Menon, M. de Kruijf, and K. Sankaralingam, "iGPU: Exception Support and Speculative Execution on GPUs," ISCA, 2012.
20. S. Che et al., "Rodinia: A Benchmark Suite for Heterogeneous Computing," IISWC, 2009.
21. T. Hetherington et al., "Characterizing and Evaluating a Key-Value Store Application on Heterogeneous CPU-GPU Systems," ISPASS, 2012.
22. A. Bakhoda et al., "Analyzing CUDA Workloads Using a Detailed GPU Simulator," ISPASS, 2009.
23. J. Navarro et al., "Practical, Transparent Operating System Support for Superpages," OSDI, 2002.
24. *TLBs, Paging-Structure Caches and their Invalidation*, tech. report, Intel, 2008.
25. N. Muralimanohar, R. Balasubramonian, and N. Jouppi, "CACTI 6.0: A Tool to Model Large Caches," MICRO, 2007.
26. A. Jaleel and B. Jacob, "In-Line Interrupt Handling for Software-Managed TLBs," ICCD, 2001.
27. T. Barr, A. Cox, and S. Rixner, "SpecTLB: A Mechanism for Speculative Address Translation," ISCA, 2011.
28. B. Pham et al., "CoLT: Coalesced Large Reach TLBs," MICRO, 2012.
29. A. Bhattacharjee, "Large-Reach Memory Management Unit Caches," MICRO, 2013.

Bharath Pichai is a software development engineer at Amazon, where he works on the Amazon Web Services infrastructure. His research interests include compilers, programming languages, architecture, and databases. Pichai has an MS in computer science from Rutgers University, where he completed the research for this article. Contact him at bsp57@cs.rutgers.edu.

Lisa Hsu is a staff engineer in the R&D wing of Qualcomm. Her research interests include architecture, memory system design, and performance modeling. Hsu has a PhD in computer science from the University of Michigan, Ann Arbor. Contact her at hsul@qti.qualcomm.com.

Abhishek Bhattacharjee is an assistant professor in the Department of Computer Science at Rutgers University. His research interests include the intersection of architecture and operating systems, memory systems architecture, and virtualization. Bhattacharjee has a PhD in electrical engineering from Princeton University. Contact him at abhib@cs.rutgers.edu.