2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture

Scheduling Page Table Walks for Irregular GPU Applications

Seunghee Shin*[1] Guilherme Cox†[1] Mark Oskin‡ Gabriel H. Loh‡
Yan Solihin* Abhishek Bhattacharjee† Arkaprava Basu§[2]

*North Carolina State University †Rutgers University ‡Advanced Micro Devices, Inc. §Indian Institute of Science
{sshin6, solihin}@ncsu.edu {guilherme.cox, abhib}@cs.rutgers.edu {mark.oskin, gabriel.loh}@amd.com arkapravab@iisc.ac.in

***Abstract*—Recent studies on commercial hardware demonstrated that irregular GPU applications can bottleneck on virtual-to-physical address translations. In this work, we explore ways to reduce address translation overheads for such applications.**

We discover that the order of servicing a GPU's address translation requests (specifically, page table walks) plays a key role in determining the amount of translation overhead experienced by an application. We find that different SIMD instructions executed by an application require vastly different amounts of *work* to service their address translation needs, primarily depending upon the number of distinct pages they access. We show that better forward progress is achieved by prioritizing translation requests from the instructions that require less work to service their address translation needs.

Further, in the GPU's Single-Instruction-Multiple-Thread (SIMT) execution paradigm, *all* threads that execute in lockstep (wavefront) need to finish operating on their respective data elements (and thus, finish their address translations) before the execution moves ahead. Thus, batching walk requests originating from the same SIMD instruction could reduce unnecessary stalls. We demonstrate that the reordering of translation requests based on the above principles improves the performance of several irregular GPU applications by 30% on average.

***Index Terms*—Computer architecture; GPU; Virtual address.**

I. Introduction

GPUs have emerged as a first-class computing platform. The massive data parallelism of GPUs had first been leveraged by highly-structured parallel tasks such as matrix multiplications. However, GPUs have more recently found use across a broader range of application such as graph analytics, deep learning, weather modeling, data analytics, computer-aided-design, oil and gas exploration, medical imaging, and computational finance [1]. Memory accesses from many of these emerging applications demonstrate a larger degree of *irregularity* – accesses are less structured and are often data dependent. Consequently, they show low spatial locality [2], [3], [4].

Irregular memory accesses can be particularly harmful to the GPU's Single-Instruction-Multi-Threaded (SIMT) execution paradigm where typically 32 to 64 threads (also called workitems) execute in a lockstep fashion (referred to as wavefronts or warps) [4], [5], [6], [7], [8], [9], [10]. When a wavefront issues a SIMD memory instruction (e.g., load/store), the instruction cannot complete until data for all workitems in the wavefront are available. This is not a problem for well-structured parallel programs with regular memory access patterns where workitems in a wavefront typically access cache lines from only one or a few unique pages. The GPU hardware exploits this to gain efficiency by coalescing multiple accesses into a single access. For irregular applications, however, memory accesses of workitems within a wavefront executing the same SIMD memory instruction can access different cache lines from different pages. This leaves little scope for coalescing and leads to *memory access divergence* – i.e., execution of a single SIMD instruction could require multiple cache accesses (when accesses fall on distinct cache lines) [7], [8], [9], [10] and multiple virtual-to-physical address translations (when accesses fall on distinct pages) [5], [11].

A recent study on real hardware demonstrated that such divergent memory accesses can slow down an irregular GPU application by up to 3.7-4× due to address translation overheads alone [5]. The study found that the negative impact of divergence could be greater on address translation than on the caches. Compared to one memory access on a cache miss, a miss in the TLB[1] triggers a page table walk that could take up to four sequential memory accesses in the prevalent x86-64 or ARM architectures. Further, cache accesses cannot start until the corresponding address translation completes as modern GPUs tend to employ physical caches.

In this work, we explore ways to reduce address translation overheads of irregular GPU applications. While previous studies in this domain primarily focused on the design of TLBs, page table walkers, and page walk caches [12], [13], [11], we show that the *order in which page table walk requests are serviced* is also critical. We demonstrate that better scheduling of page table walks can speed up applications by 30% over a baseline first-come-first-serve (FCFS) approach. In contrast, naive random scheduling can slow applications down by 26%, underscoring the need of a *good* schedule for page table walks.

We observe that page walk scheduling is particularly important for a GPU's SIMT execution. An irregular application with divergent memory accesses can generate multiple uncoalesced address translation requests while executing a single SIMD memory instruction. For a typical 32-64 wide wavefront,

[1]Authors contributed as interns in AMD Research.

[2]The author primarily contributed when he was a member of AMD Research.

[1]Translation Lookaside Buffer or TLB is a cache of address translation entries. A hit in the TLB is fast, but a miss triggers long-latency page table walk to locate the desired address translation from an in-memory page table.

2575-713X/18/$31.00 ©2018 IEEE
DOI 10.1109/ISCA.2018.00025

execution of a single SIMD memory instruction by a wavefront can generate between 1 to 32 or 64 address translation requests. Due to the lack of sufficient spatial locality in such irregular applications, these requests often miss in TLBs, each generating a page table walk request. Furthermore, servicing a page table walk requires anything between one to four sequential memory accesses. Consequently, servicing address translation needs of a single SIMD memory instruction can require between 0 to 256 memory accesses. In the presence of such a wide variance in the amount of *work* (quantified by the number of memory accesses) required to complete address translation for an instruction, we propose a SIMT-aware page walk scheduler that prioritizes walk requests from instructions that would require less work. This aids forward progress by allowing wavefronts with less address translation traffic to complete faster.

Further, page walk requests generated by a single SIMD instruction often get interleaved with requests from other concurrently executing instructions. Interleaving occurs as multiple independent streams of requests percolate through a shared TLB hierarchy. However, in a GPU's SIMT execution model, it does not help a SIMD instruction to make progress if only a subset of its page walk requests is serviced. Therefore, servicing page walk requests in a simple first-come-first-serve (FCFS) order can impede the progress of wavefronts. Our proposed scheduler thus also *batches* requests from the same SIMD instruction for them to be serviced temporally together. The SIMT-aware scheduler speeds up a set of irregular GPU applications by 30%, on average, over FCFS.

To summarize, we make two key contributions:

- We demonstrate that *the order* of servicing page table walks significantly impacts the address translation overhead experienced by irregular GPU applications.
- We then propose a SIMT-aware page table walk scheduler that speeds up applications by up to 41%.

II. Background and the Baseline

This work builds upon two aspects of a GPU's execution: the GPU's Single-Instruction-Multiple-Thread (SIMT) execution hierarchy, and the GPU's virtual-to-physical address translation mechanism.

A. Execution Hierarchy in a GPU

GPUs are designed for massive data-parallel processing that concurrently operates on hundreds to thousands of data elements. To keep this massive parallelism tractable, a GPU's hardware resources are organized in a hierarchy. The top of Figure 1 depicts the architecture of a typical GPU.

Compute Units (CUs) are the basic computational blocks of a GPU, and typically there are 8 to 64 CUs in a GPU. Each CU includes multiple Single-Instruction-Multiple-Data (SIMD) units, each of which has multiple *lanes* of execution (e.g., 16). A SIMD unit executes a single instruction across all its lanes in parallel, but each lane operates on a different data item. A GPU's memory resources are also arranged in a hierarchy. Each CU has a private L1 data cache and a scratchpad that are shared across the SIMD units only within the CU. When

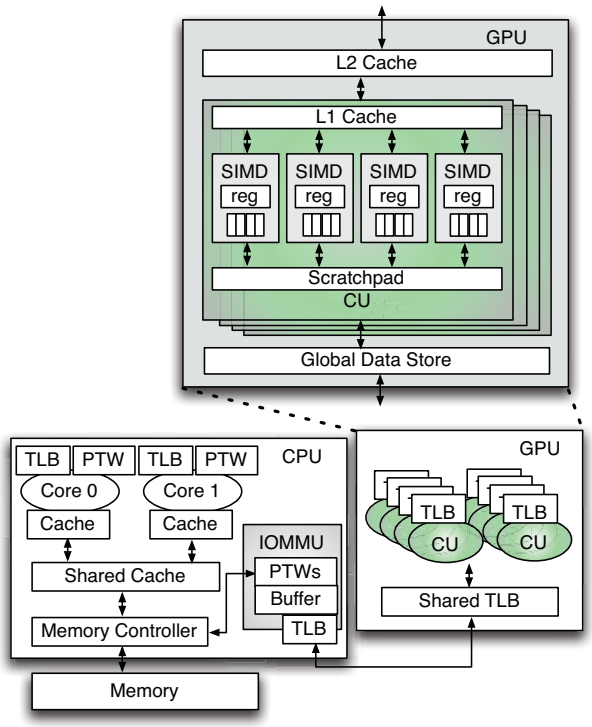


Fig. 1: Baseline system architecture.

several data elements accessed by a SIMD instruction reside in the same cache line, a hardware coalescer combines these requests into single cache access to gain efficiency. Finally, L1 caches are followed by an L2 cache that is shared across all CUs in a GPU.

GPGPU programming languages, such as OpenCL [14] and CUDA [15], expose to the programmer a hierarchy of execution groups that follows the hierarchy in the hardware resources. A workitem is akin to a CPU thread and is the smallest execution entity that runs on a single lane of a SIMD unit. A group of workitems, typically 32 to 64, forms a wavefront and is the smallest hardware-scheduled unit of work. All workitems in a wavefront execute the *same* SIMD instruction in a lockstep fashion but can operate on different data elements. An instruction completes execution *only* when *all* workitems in that wavefront finish processing their respective data elements. The next level in the hierarchy is the programmer-visible workgroup that typically comprises tens of wavefronts. Finally, work on a GPU is dispatched at the granularity of a kernel, comprised of several workgroups.

B. Virtual Address Translation in GPUs

As GPUs outgrow their traditional "co-processor" model to become first-class compute citizens, several key programmability features are making their way into mainstream GPUs. One such key feature is shared virtual memory (SVM) across the CPU and the GPU [16], [17]. For example, compliance with industry-promoted standards like the Heterogeneous System Architecture (HSA) requires SVM [17].

The bottom part of Figure 1 depicts the key hardware components of a typical SVM implementation in an HSA-enabled GPU. Conceptually, the key enabler for SVM in such a design is the GPU's ability to *walk* the same four-level x86-64 page table as the CPU. A page table is an OS-maintained data structure that maps virtual addresses to physical addresses at a page granularity (typically, 4KB). The IO Memory Management Unit (IOMMU) is the key component that enables a GPU to walk x86-64 page tables. While we focus this study on the SVM implementation mentioned above, our proposal is likely to be more broadly applicable to any GPU designs with virtual memory, not only to those supporting SVM. Next, we detail how an HSA-enabled GPU performs address translation.

GPU TLB Hierarchy: Just like in a CPU, the GPU's TLBs cache recently-used address translation entries to avoid accessing in-memory page tables on every memory access. Each CU has a private L1 TLB shared across the SIMD units. When multiple data elements accessed by a SIMD instruction reside on the same page, only a single virtual-to-physical address translation is needed. This is exploited by a hardware coalescer to lookup the TLB only once for such same page accesses. The GPU's L1 TLBs are typically backed by a larger L2 TLB that is shared across all the CUs in the GPU (bottom portion of Figure 1) [12], [11]. A translation request that misses in the GPU's TLB hierarchy is sent to the IOMMU [5]

IOMMU and Page Table Walkers: The IOMMU is the hardware component in the CPU complex that services address translation requests for accesses to the main memory (DRAM) by any device or accelerator, including that of the GPU [18], [19], [5]. The IOMMU itself has two levels of TLBs, but they are relatively small and designed to primarily serve devices that do not have their own TLBs (e.g., NIC). The IOMMU's page table walkers, however, play an essential role in servicing the GPU's address translation requests. Upon a TLB miss, a page table walker *walks* an in-memory x86-64 page table to locate the desired virtual-to-physical address mapping.

An IOMMU typically supports multiple independent page table walkers (e.g., 8-16) to concurrently service multiple page table walk requests (TLB misses) [5]. Multiple walkers are important for good performance because GPUs demand high memory bandwidth and consequently, often send many walk requests to the IOMMU.

The translation requests that miss in the TLB hierarchy queue up in the IOMMU's page walk request buffer (in short, IOMMU buffer). When a page table walker becomes free (e.g., after it finishes servicing a page walk), it could start servicing a new request from the IOMMU buffer in the order it arrived. Later in this work, we will demonstrate that such an FCFS policy for selecting page table walk requests is not well-aligned with a GPU's SIMT execution model.

Another important optimization that the IOMMU borrows from the CPU's MMU design is the page table walk caches [5], [20], [21]. Nominally, a page table walk requires four memory accesses to walk an x86-64 page table, structured as a four-level radix tree. To reduce this, the IOMMU employs small caches for the first three levels of the page tables. These specialized caches are collectively called page walk caches (PWCs). Hits in PWCs can reduce the number of memory accesses needed for a walk to anything from one to three, depending upon which intermediate level produces the hit. For example, a hit for the entire top three levels will need one memory request to complete the walk by accessing only the leaf level. In contrast, a hit for only the root level requires three memory accesses. A complete miss in PWCs however, requires four accesses.

Putting it together: Life of a GPU Address Translation Request: ① An address translation request is generated when executing a SIMD memory instruction (load/store). ② A coalescer merges multiple requests to the same page (e.g., 4KB) generated by the same SIMD instruction. ③ The coalesced translation request looks up the GPU's L1 TLB and then the GPU's shared L2 (if L1 misses). ④ On a miss in the GPU's L2 TLB, the request is sent to the IOMMU. ⑤ Upon arrival at the IOMMU, the request looks up the IOMMU's TLBs. ⑥ On a miss, the request queues up as a page walk request in the IOMMU buffer. ⑦ When an IOMMU's page table walker becomes free, it typically selects a pending request from the IOMMU buffer in FCFS order. ⑧ The page table walker first performs a PWC lookup and then completes the walk of the page table, generating one to four memory accesses. ⑨ On finishing a walk, the desired translation is returned to the TLBs and ultimately to the SIMD unit that requested it.

III. The Need for Smarter Scheduling of Page Table Walks

Irregular GPU applications often make data-dependent memory accesses with little spatial locality [2], [3]. This causes memory access divergence in the GPU's SIMT execution model where different workitems within a wavefront access data on distinct pages. The hardware coalescer is ineffective in such cases as several different address translation requests are generated by the execution of a single SIMD memory instruction. These requests then look up TLBs but often miss there owing to less locality in irregular applications. Eventually, many of these requests queue up in the IOMMU buffer to be serviced by a page table walker.

A recent study on commercial GPU hardware demonstrated that such divergent access can slowdown irregular GPU applications by up to 3.7-4× due to address translation overheads [5]. In this work, we aim to reduce address translation overheads for such irregular GPU applications.

We discover that the *order in which page table walks are serviced* can significantly impact the address translation overheads experienced by an irregular GPU application. While better page table walk scheduling (ordering) can potentially improve performance, poor scheduling (e.g., random scheduling) can be similarly detrimental. Figure 2 shows the extent by which scheduling of page table walks can impact performance on a set of representative irregular applications (methodology is detailed in Section V-A). The figure shows speedups of each application while employing naive random scheduler[2], the baseline FCFS,

[2]As its name suggests, the random policy randomly picks a pending page walk request to service from the IOMMU buffer.

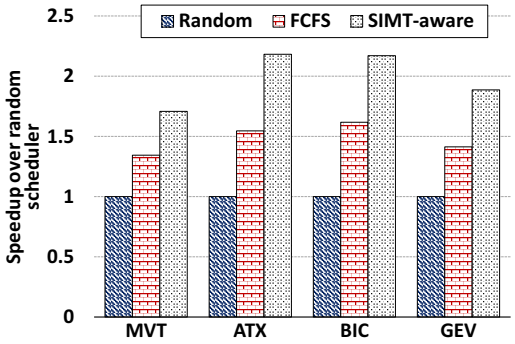


Fig. 2: Performance impact of page walk scheduling.

and the proposed SIMT-aware page walk scheduler. Each bar in the cluster shows the speedup of an application with a given scheduler, normalized to that with random scheduler. While we will detail our SIMT-aware scheduling over the next two sections, the key message conveyed by the figure is that the performance of an application can differ by more than 2.1× due to the difference in the schedule of page table walks. This underscores the importance of exploring the scheduling of a GPU's page walk requests.

A keen reader will notice the parallel between the scheduling of page table walks and the scheduling of memory (DRAM) accesses at the memory controller [22], [23], [24], [25]. The existence of a rich body of research on memory controller scheduling suggests that there exist opportunities for follow-on work to explore different flavors of page walk scheduling for both performance and QoS.

In the rest of this section, we discuss why page table walk scheduling affects performance and then provide empirical analysis to motivate better scheduling of GPU page walks.

A. Shortest-job-first Scheduling of Page Table Walks

We observe that instructions issued by a wavefront require different amounts of *work* to service their address translation needs. There are two primary reasons for this. First, the number of page table walks generated due to the execution of a single SIMD memory instruction can vary widely based on how many distinct pages the instruction accesses and the TLB hits/misses it generates. In the best case, all workitems in a wavefront access data on the same page and the perfectly coalesced translation request hits in the TLB. No page walks are necessary in that case. At the other extreme, a completely divergent SIMD instruction can generate page table walk requests equal to the number of workitems in the wavefront (here, 64). Second, each page walk may itself need anywhere between one to four memory requests to complete. This happens due to hits/misses in page walk caches (PWCs) that store recently-used upper-level entries of four-level page tables (detailed in Section II).

Figure 3 shows the distribution of the number of memory accesses required to service address translation needs of SIMD instructions for a few representative applications. The x-axis shows buckets for the number of memory accesses needed by a SIMD memory instruction to service its address translation needs. The y-axis shows the fraction of instructions issued by

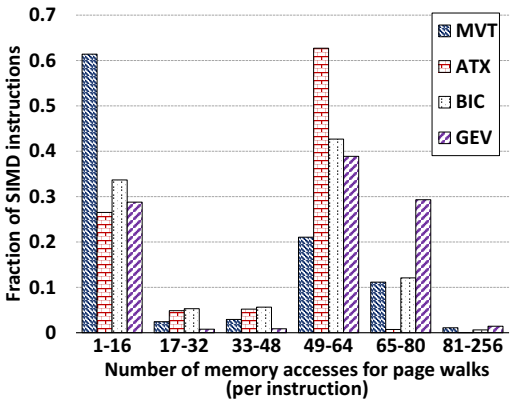


Fig. 3: Distribution of number of memory accesses (i.e.,'work') for servicing address translation needs of SIMD instructions.

the application that fall into the corresponding x-axis buckets. We excluded instructions that did not request any page table walks. We find that often between 27-61% of the instructions needed one to sixteen memory accesses to complete all the page table walks it generated. On the other hand, more than 33-70% of the instructions required forty-nine or more memory accesses. One of the applications (`GEV`) had close to 31% of instructions requiring sixty-five or more memory accesses. In summary, we observe that the amount of *work* (quantified by the number of memory accesses) required to service the address translation needs of an instruction varies significantly.

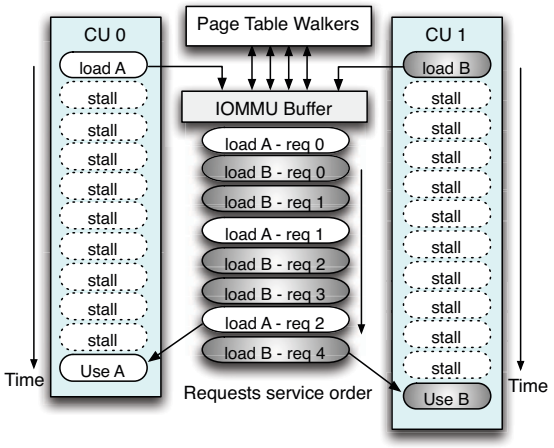
It is well studied in scheduling policies across various fields that in the presence of "jobs" of different lengths, if a longer job can delay a shorter job, then it impedes overall progress. This leads to the widely employed "shortest-job-first" (SJF) policy that prioritizes shorter jobs over longer ones [26]. By analogy, we posit that servicing all page table walks generated due to the execution of a single instruction should be treated as a single "job" because the instruction cannot complete until all those walks are serviced. Figure 3 demonstrates that the "length" of such jobs, as quantified by the number of memory accesses, vary significantly.

Key idea ①: Following the wisdom of time-tested SJF policies, we propose to prioritize the servicing of page table walk requests from instructions requiring fewer memory requests to complete their address translation needs over those requiring larger number of memory accesses.

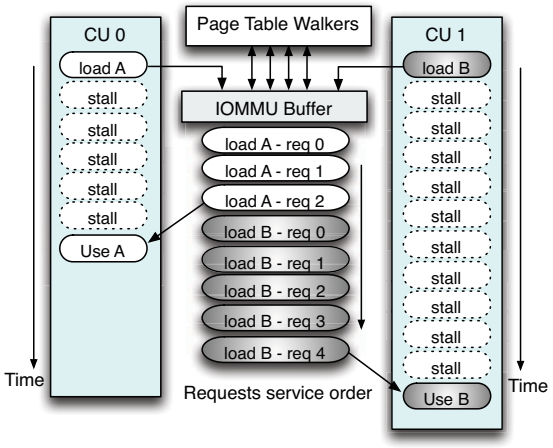
B. Batch-scheduling of Page Table Walk Requests

Owing to the GPU's SIMT execution model, all page table walks generated by a single SIMD instruction must complete before the instruction can finish execution. The performance is thus determined by when the *last* of those walk requests is serviced. Even servicing all but one walk request does not aid progress.

Figure 4a illustrates how the progress of two SIMD instructions, `load A` and `load B`, issued by two wavefronts are impaired if their page table walk requests are interleaved. Both `load A` and `load B` generate multiple walk requests and both experience stalls due to the latency to service walk



(a) Interleaving of page walk requests with hurts progress.



(b) Batching of page walk requests reduces interleaving.

Fig. 4: Impact of interleaving of a GPU's page walk requests.

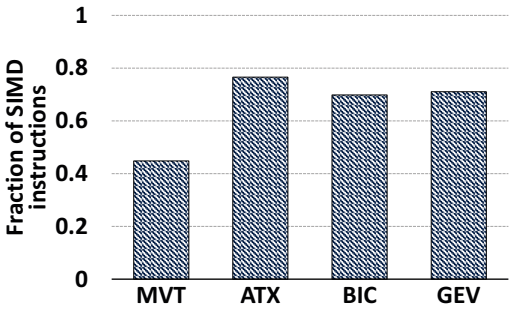


Fig. 5: Fraction of instructions whose page walk requests are interleaved with requests from other instructions.

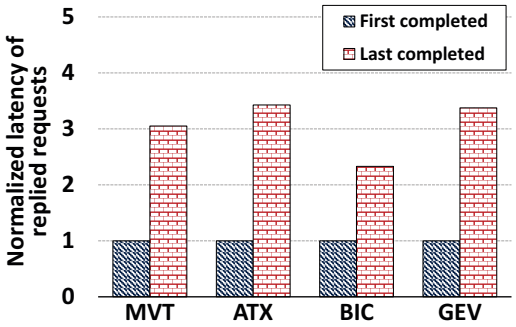


Fig. 6: Average latencies of the first- and the last-completed page walk request per instruction.

requests generated by the other. Evidently, if interleaved page walk requests are serviced in the FCFS order, then it delays completion of both `load A` and `load B` since both need all their walk requests to finish before the instruction can progress. This inefficiency exacerbates if walk requests from a larger number of distinct instructions interleave since the progress of every instruction involved in the interleaving suffer.

Unfortunately, such interleaving among page walk requests from different SIMD instructions happens fairly regularly. Figure 5 quantifies how often such interleaving happens for representative irregular GPU workloads (methodology detailed in Section V-A). The y-axis shows the fraction of executed memory instructions whose page walk requests interleave with requests from at least another instruction. We exclude any instructions that do not generate at least two page table walks as interleaving is impossible for them. We observe that 45-77% of such instructions have their walk requests interleaved.

We traced the source of this interleaving to the GPU's shared L2 TLB. The shared L2 TLB receives multiple independent streams of address translation requests generated by L1 TLB misses from concurrently executing wavefronts across different CUs. These requests can then miss in the L2 TLB and travel to the IOMMU. The IOMMU thus receives a multiplexed stream of walk requests from different wavefronts.

A reasonable question to ask is how much does this interleaving potentially impact the performance? Figure 6 shows the potential performance cost of interleaving. The figure shows the average latencies experienced by the first- and the last-completed page walk requests from the same SIMD memory instruction. The latencies are normalized to the average latency experienced by the first completed walk request. We exclude instructions that do not generate at least two page walk requests as they cannot interleave. Larger the latency gap, the more time an instruction potentially stalls for all of its page walk requests to complete. We observe that often the latency of the last completed walk is more than 2-3× that of the first completed page walk. This suggests that the interleaving of page walks can significantly impede forward progress.

Ideally, page walk requests should be scheduled to minimize such latency gaps. A smarter scheduler thus should strive to achieve a schedule as shown in Figure 4b by batching page walk requests from the same instruction. We see from the figure that `load A` can potentially complete much earlier without further delaying `load B` in Figure 4a.

Key idea ②: A smart scheduler should *batch* page walk requests from the same instruction to minimize interleaving

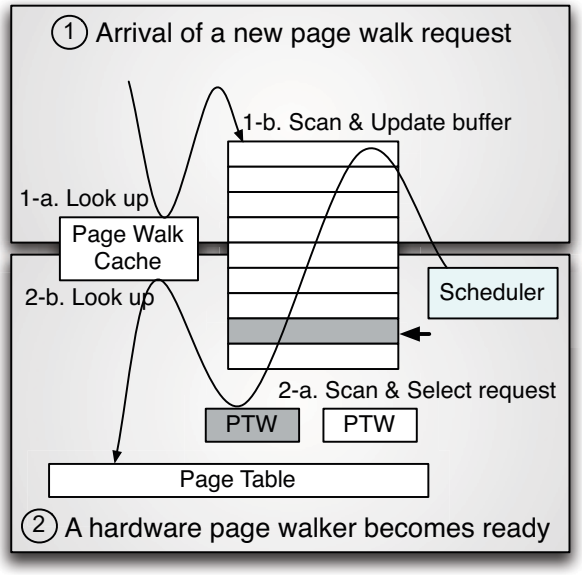


Fig. 7: Key components and actions of the SIMT-aware page table walk scheduler.

due to walk requests from other instructions.

IV. Design and implementation

Driven by the above analyses, we propose a *SIMT-aware page table walk scheduler* in the IOMMU. Figure 7 shows some of the key components and actions in the IOMMU to realize such a scheduler. When an IOMMU's page table walker becomes available to accept a new request, the scheduler selects which pending page walk request is serviced next. While we introduce a specific scheduler design, there could be several other potential designs that build upon our observations about the importance of page table walk scheduling.

Our proposed SIMT-aware scheduler follows the two key ideas mentioned in the previous section. At a high level, the scheduler first attempts to schedule a pending page walk request (in the IOMMU buffer) issued by the same SIMD instruction as the most recently scheduled page walk. If none exists, it schedules a request issued by an instruction that is expected to require the least number of memory requests (i.e., *work*) to service all its walk requests. For this purpose, we assign a score to each page walk request. This score estimates the number of memory requests that would be required to complete *all* page walk requests of the issuing instructions. The score is thus the same for all pending page walk requests generated by a given SIMD instruction. A lower value indicates fewer memory requests to service an instruction's page walk requests.

We make a few simple hardware modifications to realize the above design concept. First, each page walk request from the GPU is attached with an instruction ID (20 bits in our implementation). Correspondingly, the buffer holding the pending page walk requests at the IOMMU is extended with this ID. As shown in Figure 7, we then modify how the IOMMU behaves when ① a new page walk request arrives at the IOMMU, and when ② a hardware page walker becomes available to accept a new request. Below we detail actions taken during these two events.

① **Arrival of a new page walk request:** If there is an idle hardware page walker when a new request arrives then it starts walking immediately. Otherwise, we assign an integer score (between 1 to 256, where 256 corresponds to the maximum possible number of memory accesses required if all 64 workitems need four memory accesses each to perform their respective translation) to the newly-arrived request. The score estimates the number of memory accesses needed to complete all page walk requests of the corresponding instruction. This is done in two steps. First, the new request looks up the PWCs to estimate the number of memory requests that this request alone may need to get serviced (action `1-a` in Figure 7). This number can be between one (on a hit in all upper-levels in the PWC and thus, requiring only single memory access to the leaf-level of the page table) to four (on a complete miss in the PWC requiring the full walk of the four-level page table). Since the PWC contents could change by the time the scheduler selects the request, this number is an estimate of the actual number of memory accesses required to service the walk.

Second, we then scan all the pending page walk requests in the IOMMU buffer to find any matching page walk requests issued by the same instruction as the newly arrived one (`1-b`). All requests from the same SIMD instruction have the same score. A new score is computed by adding the PWC-based score of the newly-arrived request to the previous score of an existing request in the IOMMU buffer that is issued by the same instruction. This updated score now represents the total estimated number of memory accesses required to service *all* the translation requests from the issuing SIMD instruction. All entries in the IOMMU that match the SIMD instruction of the newly-arrived request (including the newly-arrived request) are then updated with this new score.

② **A hardware page walker becomes ready:** When a page walk finishes, the corresponding page table walker becomes available to start servicing a new request. The scheduler decides which of the pending page walk requests (if any) it should service next. First, the scheduler scans the buffer of pending page walks to find any request that matches the instruction ID of the most recently issued page walk request (`2-a`). If such a request exists, it is chosen to ensure temporal batching of page walks issued by the same SIMD instruction. If no such matching request is found, then the scheduler selects a request with the lowest score. This follows the second key idea – schedule requests from the instruction that is expected to require the fewest memory accesses. Both actions are performed during the scanning of pending page walk requests (`1-a`). Finally, the selected page walk request is serviced as usual by first looking up the PWC for partial hits and then completing the walk of the page table (`2-b`).

Putting it all together: To summarize, an address translation from the GPU flows as follows. The coalescing and lookups in the GPU TLBs happen as before (refer to Section II-B for steps). The only modification for our scheduler is that each

request now carries the ID of the instruction that generated it. As in the baseline, a translation request that misses in the GPU TLBs is sent to the IOMMU where it performs a lookup in the IOMMU's TLBs. If the request misses in all the TLBs, then it is inserted into the IOMMU buffer. If any of the page table walkers (8 in the baseline) are available, then one of them starts the page walk process. Otherwise, our scheduler calculates a score for the newly-arrived request and re-scores any of the already-pending requests from the same instruction as detailed above (actions `1-a` and `1-b`). The request then waits in the buffer until it is selected by the scheduler.

When a page table walker finishes a walk, the scheduler selects which request it should service next. The scheduler scans pending requests in the IOMMU buffer (action `2-a`) to find if there are any requests issued by the same instruction as the last-scheduled request. If so, the oldest among such requests is chosen. If not, the scheduler selects the request with the lowest score (oldest first in the case of a tie). Once a request is scheduled, the page table walker proceeds walking as usual – it looks up the PWC for partial hits before making memory accesses to the page table (`2-b`).

Design Subtleties: We now discuss a few intricacies of this design. First, note that the scanning of the pending page walk requests upon arrival of a new request is not in the critical path. The newly arrived request anyway queues up in the IOMMU buffer for the scheduler to select it. If a free page table walker is immediately available, the scheduler plays no role and no scanning is involved. However, it adds an extra latency in the critical path of servicing a new page walk request when the scheduler scans the pending request (`2-a`). Every such request in the IOMMU buffer has already suffered a long latency miss through the entire TLB hierarchy, and a walk itself requires hundreds of cycles. Therefore, the latency of scanning pending requests adds little additional delay.

As with any scheduler, the above design is susceptible to starvation. We implement an aging scheme whereby we prioritize pending walk requests that have been passed by a large number of younger requests (in our experiments, we found that setting this threshold to two million requests worked well to avoid any potential starvation).

As briefly mentioned earlier, another subtlety is that the PWC contents may change between the time a request arrives at the IOMMU and the time the scheduler selects that request. This could lead to inaccuracies in estimating the number of memory accesses needed to service a page walk since the score is calculated when the request arrives. Unfortunately, it is infeasible for the scheduler to re-calculate scores of every pending request at the time of request selection. This would have added significant latency in the critical path. Instead, we reduce this potential inaccuracy by adding 2-bit saturating counters to the entries of the PWC. Whenever a lookup for a newly-arrived request hits in the page walk cache (`1-a` in Figure 7), the counters of the corresponding entries are incremented. The counters are decremented when a selected page walk request hits in the PWC (`2-b`). Thus, a value greater than zero indicates that there exists at least one pending page walk request in the IOMMU buffer that would later hit in the page walk cache when that request is scheduled. The replacement policy in the page walk cache is then modified to avoid replacing an entry with a counter value greater than zero. If all entries in a set have value greater than zero, then a conventional pseudo-LRU policy selects a victim as usual.

TABLE I: The baseline system configuration.

GPU	2GHz, 8 CUs, 4 SIMD per CU 16 SIMD width, 64 threads per wavefront
L1 Data Cache	32KB, 16-way, 64B block
L2 Data Cache	4MB, 16-way, 64B block
L1 TLB	32 entries, Fully-associative
L2 TLB	512 entries, 16-way set associative
IOMMU	256 buffer entries, 8 page table walkers 32/256 entries for IOMMU L1/L2 TLB, FCFS scheduling of page walks
DRAM	DDR3-1600 (800MHz), 2 channel 16 banks per rank, 2 ranks per channel

TABLE II: GPU benchmarks for our study.

	Benchmark (Abbrev.)	Description	Memory Footprint
Irregular applications	Xsbench (XSB)	Monte Carlo neutronics application	212.25MB
	MVT (MVT)	Matrix vector product and transpose	128.14MB
	ATAX (ATX)	Matrix transpose and vector multiplication	64.06MB
	NW (NW)	Optimization algorithm for DNA sequence alignments	531.82MB
	BICG (BCG)	Sub kernel of BiCGStab linear solver	128.11MB
	GESUMMV (GEV)	Scalar, vector and matrix multiplication	128.06MB
Regular application	SSSP (SSP)	Shortest path search algorithm	104.32MB
	MIS (MIS)	Maximal subset search algorithm	72.38MB
	Color (CLR)	Graph coloring algorithm	26.68MB
	Back Prop. (BCK)	Machine learning algorithm	108.03MB
	K-Means (KMN)	Clustering algorithm	4.33MB
	Hotspot (HOT)	Processor thermal simulation algorithm	12.02MB

V. Evaluation

We now describe our evaluation methodology and then analyze the results in detail.

A. Methodology

We used the execution-driven gem5 simulator that models a heterogeneous system with a CPU and an integrated GPU [27]. We heavily extended the gem5 simulator to incorporate a detailed address translation model for a GPU including coalescers, the GPU's TLB hierarchy, and the IOMMU. Inside the newly-added IOMMU module, we model a two-level TLB hierarchy, multiple independent page table walkers, and page walk caches to closely mirror the real hardware. We implemented different scheduling policies for page table walks, including our novel SIMT-aware page walk scheduler inside the IOMMU module.

The simulator runs unmodified applications written in OpenCL [14] or in HCC [28]. Table I lists the relevant parameters for the GPU, the memory system, and the address

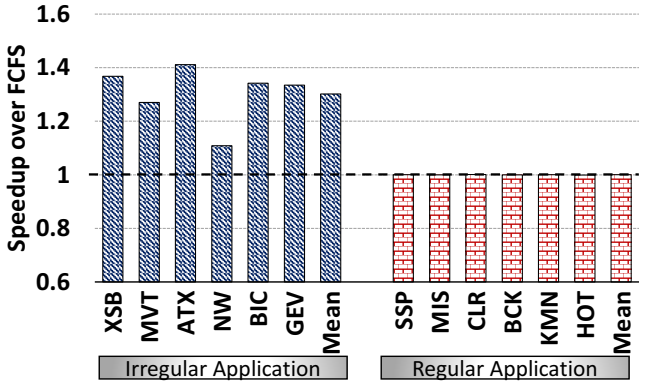


Fig. 8: Speedup with SIMT-aware page walk scheduler.

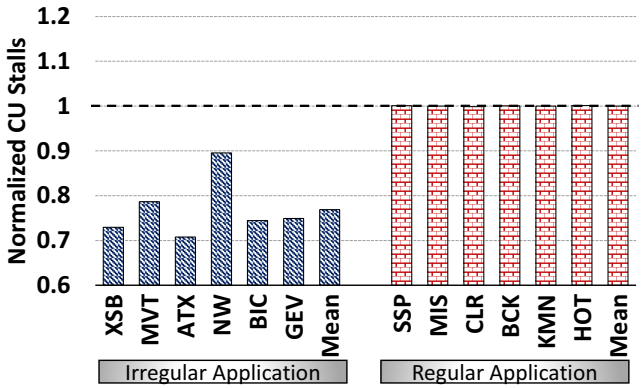


Fig. 9: GPU stall cycles in execution stage.

translation mechanism of the baseline system. Section V-B2 also presents sensitivity studies varying key parameters.

Table II lists the applications used in our study with descriptions of each workload and their respective memory footprints. We draw applications from various benchmark suites including Polybench [29] (`MVT`, `ATAX`, `BICG`, and `GESUMMV`), Rodinia [30] (`NW`, `Back propagation`, `K-Means`, and `Hotspot`), and Pannotia [31] (`SSSP`, `MIS`, and `Color`). In addition, we used a proxy-application released by the US Department of Energy (`XSBench` [32]).

In this work, we focus on emerging GPU applications with irregular memory access patterns. These applications demonstrate memory access divergence [2], [3] that can bottleneck a GPU's address translation mechanism [5]. However, not every application we studied demonstrates irregularity nor suffers from significant address translation overheads. We find that six workloads (`XSB`, `MVT`, `ATX`, `NW`, `BCG`, and `GEV`) demonstrate irregular memory access behavior while the remaining workloads (`SSP`, `MIS`, `CLR`, `BCK`, `KMN`, and `HOT`) have fairly regular memory accesses. Applications with regular memory accesses show little translation overhead to start with and thus, offer little scope for improvement. Our evaluation thus focuses on applications in the first category, but we include results for the regular applications to demonstrate that our proposed techniques do not harm workloads that are insensitive to translation overheads.

B. Results and Analysis

We evaluate the impact of page table walk scheduling and our SIMT-aware scheduler by asking the following questions: ① How much does the SIMT-aware page table walk scheduler speed up applications over the baseline FCFS scheduler? ② What are the sources of speedups (if any)? ③ How sensitive are the results to configuration parameters like the TLB size and the number of page table walkers?

Figure 8 shows the speedups of GPU applications with our SIMT-aware page walk scheduler over FCFS. The left half of the figure (dark bars) shows the speedups for irregular applications while the right half (thatched bars) shows the speedups for applications with regular memory accesses. We observe that our scheduler speeds up irregular GPU applications by up to 41%, and by 30% on average (geometric mean). On the other hand, there is little change in the performance of regular applications. This is expected; regular applications experience little address translation overhead, and thus page table walk scheduling has almost no influence on their performance. The data, however, assure that the SIMT-aware scheduling does not hurt regular workloads.

Previously in Figure 2 in Section III, we also demonstrated how naive random scheduling can significantly hurt performance. Together, these observations show that ① different scheduling of page walks can have severe performance implications, and ② the SIMT-aware scheduler can significantly speed up irregular GPU applications without hurting others.

1) Analyzing Sources of Speedup: It is important to understand the reasons behind the observed speedups. Toward this, we first present how schedulers impact GPU stall cycles, which are the cycles during which a CU cannot execute any instructions because none are ready. Figure 9 shows the normalized stall cycles for each application with our SIMT-aware page table walk scheduler. The height of each bar is normalized to the stall cycles with the FCFS scheduler. A lower number indicates better forward progress since CUs are stalled for less time on average. As before, the left half shows the results for irregular applications and the right half shows those for regular applications. We observe that the SIMT-aware scheduler reduces the stall cycles by 23% on average (up to 29%) for irregular applications. This shows how the scheduler enables instructions, and consequently, corresponding wavefronts, to make better forward progress. This ultimately leads to faster execution. As expected, the stall cycles for regular applications remain mostly unchanged. Because these applications neither alter performance nor provide any new insights, the remaining evaluations in this paper focus entirely on irregular applications.

In Figure 6 (Section III), we showed that there can be a significant gap between the latency of the first- and the last-completed page walk for a given SIMD instruction. A larger gap indicates that instructions are waiting for a large number of translation requests to be completed, or a few requests to be completed but that are delayed due to the servicing requests from other instructions, or a combination of both effects. Our

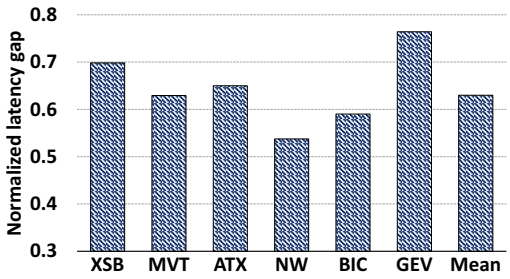


Fig. 10: Latency gap between the first and the last-completed page walk request per instruction.

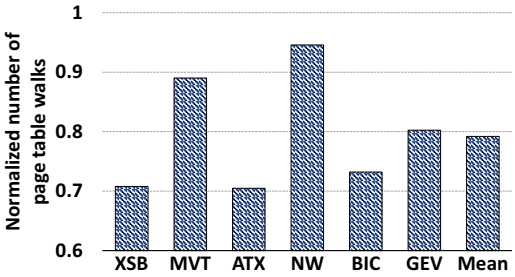


Fig. 11: Number of page walk requests with SIMT-ware scheduler normalized over FCFS.

SIMT-aware scheduler batches the servicing of page table walk requests from the same instruction to reduce this gap. Figure 10 shows the degree of effectiveness of our scheduler in reducing the gap. Each bar represents the latency gap between the first- and the last-completed page walk requests from an instruction with our scheduler. The height of each bar is normalized to the latency gap with the baseline FCFS scheduler. As before, we exclude instructions that generate less than two page table walks as they cannot interleave. We observe that the SIMT-aware scheduler reduces the latency gap by 37% over FCFS, on average. This shows the efficacy of batching page walks.

Another interesting performance impact of our scheduler is that it also reduced the total number of page table walk requests. Figure 11 shows the number of page walk requests (i.e., number of TLB misses) with our SIMT-aware scheduler, normalized to the baseline FCFS scheduler.

We observed 21% reduction (up to 30%) in the number of page table walk requests, on average. We traced the reason for this improvement to the better exploitation of intra-wavefront locality in TLBs. Our scheduler favors SIMD instructions with lower address translation needs, which in turn aids forward progress. At the same time, our scheduler also tends to delay page walk requests from instructions that generate a large amount of address translation traffic. These high-overhead instructions are anyway likely to take a long time to complete. While the translation-heavy instructions are stalled, they are kept away from polluting (thrashing) the GPU's TLBs. Consequently, the low-overhead instructions experience higher TLB hit rates as the useful TLB entries are not evicted by the high-overhead instructions. This results in a reduction in the number of TLB misses and thus, reduce the number of page walk requests.

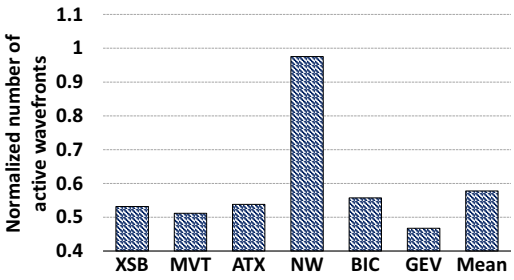


Fig. 12: Number of active wavefronts accessing the GPU's L2 TLB with SIMT-aware scheduler (normalized over FCFS).

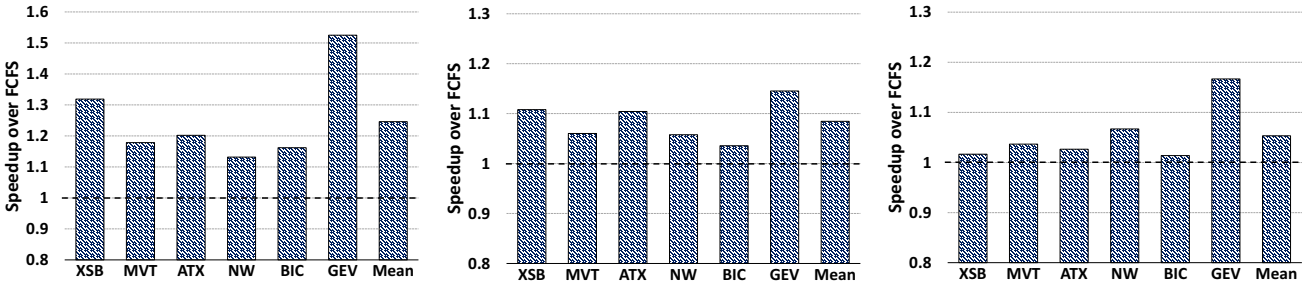
We further validated the above conjecture by counting the number of distinct wavefronts that access the GPU's L2 TLB over fixed-sized epochs (we used an epoch length of 1024 GPU L2 TLB accesses). Figure 12 presents this metric (normalized to FCFS), averaged over all epochs for the SIMT-aware scheduler. We observed a 42% reduction in the number of distinct wavefronts accessing the GPU's L2 TLB in an epoch. This shows the role of page walk scheduling in lowering the contention in the GPU's L2 TLB. Consequently, the number of page table walks decreases due to less potential thrashing in the TLB. This behavior has similarities to phenomena observed by others in the context of the GPU's caches [33].

2) Sensitivity Analysis: We measured sensitivity of the scheduler to the GPU's L2 TLB size, the number of concurrent page table walkers, and the size of IOMMU buffer holding the pending page walk requests.

Figure 13 shows the speedup achieved by our SIMT-aware scheduler with varying amounts of critical address translation resources: L2 TLB capacity and the number of page table walkers. Figure 13a shows the speedup with 1024 entries in L2 TLB and eight page table walkers. The average speedup achieved by the SIMT-aware scheduler over the FCFS scheduler is significant (on average, 25%) even with larger TLB. It is, however, slightly less than 30% speedup achieved with 512-entry L2 TLB. The larger TLB reduces the number of page walk requests and thus, the scope for improving performance by scheduling page walks diminishes.

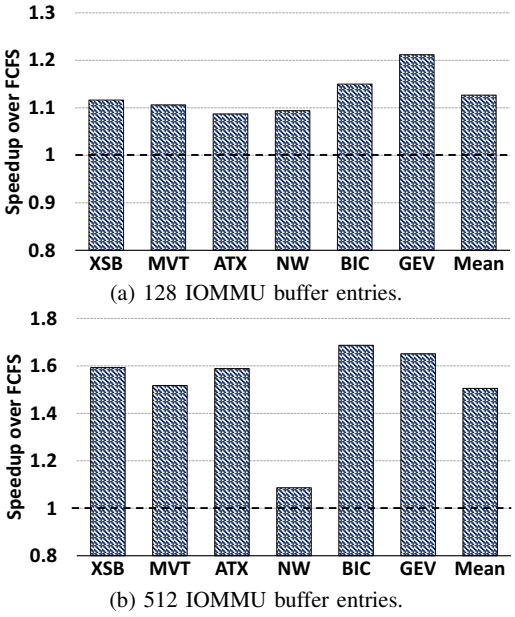
On the other hand, figure 13b shows the speedups with 16 page table walkers. Increasing the number of page table walkers reduces the number of pending page table walks as the effective address translation bandwidth increases. This also reduces headroom for the performance improvement achievable through better page walk scheduling. We observe that SIMT-aware page walk still speeds up applications by about 8.4% over the FCFS policy. Finally, figure 13c shows the combined impact of both the bigger TLB size and the increased page table walker count. In this configuration, both the increased TLB resources and the increased number of page table walkers further moderate scope for the improvement with smarter scheduling of walks. SIMT-aware scheduling speeds up applications by 5.3% in this configuration.

Overall, our SIMT-aware scheduler consistently performs better than the baseline FCFS scheduler across different configurations and different workloads, thereby demonstrating



(a) 1024 L2 TLB and 8 walkers. (b) 512 L2 TLB and 16 walkers. (c) 1024 L2 TLB and 16 walkers.

Fig. 13: Speedups with varying GPU L2 TLB size and page table walker counts.



(a) 128 IOMMU buffer entries.

(b) 512 IOMMU buffer entries.

Fig. 14: Speedups with varying IOMMU buffer size.

the robustness of our technique, although the amount of benefit depends on the severity of address translation bottleneck.

We then investigate the effect of IOMMU buffer size on our scheduler. The IOMMU buffer size determines the size of the *lookahead* for the scheduler, i.e., the maximum number of page walk requests from which it can select a request. Larger the buffer size, the larger is the lookahead potential. Figures 14a and 14b show speedups with SIMT-aware scheduler over the FCFS scheduler with 128-entry and 512-entry IOMMU buffers, respectively. All other parameters remain the same as in the baseline configuration. A smaller IOMMU buffer size reduces the opportunity for a scheduler to make smart reordering decisions, and thus, the speedups due to SIMT-aware scheduling are reduced to 13% (Figure 14a) with a 128-entry buffer. On the other hand, if the size of the buffer is increased to 512 entries, the average speedup jumps to 50% (Figure 14b). In short, the magnitude of the performance benefit from SIMT-aware scheduling varies across configurations but remains substantial across all cases.

VI. Discussion

Why not large pages? Large pages map larger ranges of contiguous virtual addresses (e.g., 2MB) to contiguous physical addresses. They can reduce the number of TLB misses by mapping more of memory with the same number of TLB entries. However, large pages are far from a panacea as decades of deployment and studies in the CPU world have demonstrated [34], [35], [36]. As memory footprints of applications continue to grow, today's large page effectively becomes tomorrow's small page. Thus, techniques that help improve performance with small (base) pages remain useful for future workloads with larger memory footprints, even with larger page sizes. Even workloads with memory footprints of a few hundred MBs (Table II) can benefit significantly from our SIMT-aware page walk scheduler, and workloads with more realistic footprints will continue to benefit from more efficient page walk scheduling, even with large pages. Unfortunately, exorbitant simulation time prevents us from evaluating such large memory footprint.

More importantly, irregular GPU applications tend to exhibit low spatial locality [2], [3] where large pages tend to have limited benefits. These applications see less benefit from large pages because the approach fundamentally relies on locality to enhance the reach of TLBs [34]. Previous works further demonstrated that large pages can even hurt performance in some cases due to the relatively lower number of entries in large page TLBs [37], [34]. Recent works on GPUs have also demonstrated that large pages can significantly increase the overhead of demand paging for GPUs [38], [13].

Interactions with Other Schedulers: In a GPU, wavefront (warp) schedulers play an important role in leveraging parallelism and impact cache behavior [39], [7]. Previous work has also shown the importance of TLB-aware wavefront scheduling [11]. Apart from the wavefront scheduler, memory controllers sport sophisticated scheduling algorithms to improve performance and fairness [2], [40]. A reasonable question to ask is how these schedulers interact with the page walk scheduler.

Page walk schedulers play an important role in reducing address translation overheads, which none of these other schedulers aim to do. Thus, even in the presence of sophisticated wavefront and memory controller schedulers, we expect that improvements to page walk scheduling will still be useful. The page walk scheduler is unlikely to have significant interactions

with the memory schedulers as the maximum amount of memory traffic from the page walk schedulers would still only consume a relatively small fraction of a GPU's total memory bandwidth. That said, there still could be opportunities for better coordination among the different schedulers, but we leave such explorations for future work.

VII. Related work

Three research domains are related to this work: TLB management, scheduling in memory controllers, and work scheduling in GPUs.

A. TLB Management

The emergence of shared virtual memory (SVM) between the CPU and the GPU as a key programmability feature in a heterogeneous system has made an efficient virtual-to-physical address translation for GPUs a necessity. Lowe-Power et al. [12] and Pichai et al. [11] were among the first to explore designs GPU MMU. Lowe-Power et al. demonstrated that coalescer, shared L2 TLB and multiple independent page walkers are essential components of an efficient GPU MMU design. Their design is similar to our baseline configuration. On the other hand, Pichai et al. showed the importance of making wavefront (warp) scheduler to be TLB-aware.

More recently, Vesely et al. demonstrated on real hardware, that a GPU's translation latencies can be much longer than that of a CPU's and GPU applications with memory access divergence may bottleneck due to address translation overheads [5]. Cong et al. proposed TLB hierarchy similar to our baseline but additionally proposed to use a CPU's page table walkers for GPUs [41]. However, accessing CPU page table walkers from a GPU could be infeasible in a real hardware due to longer latencies. Lee et al. proposed a software managed virtual memory to provide an illusion of a large memory by partitioning GPU programs to fit into the physical memory space [42]. Ausavarungnirun et al. showed that address translation overhead could be even larger in the presence of multiple concurrent applications on a GPU [13]. They selectively bypassed TLBs to avoid thrashing and prioritizing address translation over data access to reduce overheads. Yoon et al. demonstrated the significance of address translation overheads in the performance of GPU applications and proposed to employ virtual caches for GPUs to defer address translation only after a cache miss [43].

Different from these works, we demonstrate the importance of *(re-)ordering page table walk requests* and designed a SIMT-aware page table walk scheduler. Most of these works are either already part of our baseline (e.g., [12]) and/or are largely orthogonal to ours (e.g., [13]).

Address translation overheads are well studied in CPUs. To exploit page localities among threads, Bhattacharjee et al. proposed inter-core cooperative TLB prefetchers [44]. Pham et al. proposed to exploit naturally occurring contiguity to extend effective reach of TLB [45]. Bhattacharjee later proposed shared PWCs and efficient page table designs to increase PWCs hits [20]. Cox et al. have proposed `MIX TLBs` that support different page sizes in a single structure [37]. Barr et al. proposed `SpecTLB` that speculatively predicts address translations to avoid the TLB miss latency. Several others proposed to leverage segments to selectively bypass TLB and the cost of TLB misses [34], [35], [36]. While some of these techniques can be extended to GPUs, page table walk scheduling is orthogonal to them. Basu et al. and Karakostas et al. also proposed ways to reduce energy dissipation in a CPU's TLB hierarchy [46], [47].

B. Scheduling in Memory Controllers

Memory bandwidth has become a potential performance limiter with the emergence of large multi-cores and GPUs [48]. Rixner et al. introduced early memory scheduling policies to exploit memory parallelism for better performance [22]. Chatterjee et al. proposed staged reads that parallelize read and write requests through two staged read operations and scheduling of writes to take advantage of them [49]. Yoongu et al. introduced `ATLAS` that prioritizes threads with least serviced at the memory controller during an epoch [23].

A GPU's SIMT execution exacerbates memory bandwidth bottleneck [50]. Ausavarungnirun et al. proposed staged memory scheduling to exploit locality by batching row buffer hit requests [24]. Chatterjee et al. proposed a memory scheduler batching requests from the same wavefronts to solve memory access divergence [51]. Our SIMT-aware scheduler bears similarity with this work as we also batch requests but in the context of page walks. Further, Li et al. proposed to prioritize memory accesses with higher inter-core locality [25].

The fairness of resource sharing is also important in presence of multiple contenders. Mutlu et al. proposed `STFM` that estimates the slowdown of threads due to sharing the DRAM and prioritizes requests from the slowest thread [52]. `PAR-BS` provides QoS by batching and scheduling requests from the same thread [40]. Yoongu et al. proposed `TCM` that groups threads with similar memory access patterns and apply different scheduling policies for different groups [53]. Jog et al. proposed to allocate fair memory bandwidth among concurrently executing kernels on different CUs in a GPU [54]. Jeong et al. proposed a QoS-aware scheduling that prioritizes CPUs with low latency while guaranteeing QoS of GPUs [55]. Usui et al. proposed `DASH` that considers the deadline for accelerators instead of always prioritizing CPU workloads [56].

These works focus solely on memory (DRAM), and not on page table walks. However, the existence of such a rich body of work shows the potential of significant follow-on research in exploring various policies for page table walk scheduling for both performance and QoS.

C. Work Scheduling in GPUs

Smart scheduling of work in the GPU's compute units has been widely investigated, too. Rogers et al. proposed `CCWS` that limits the number of active wavefronts on computer units if it detects thrashing on L1 cache [39]. The authors then extended it considering L1 cache usage in wavefront scheduling to reduce the impact of memory access divergence [7]. Li et al. extended `CCWS` to also allow bypassing the L1 cache for selected

wavefronts when shared resources have additional headroom after limiting wavefronts [33]. Kayrian et al. dynamically throttled parallelism in CUs based on application characteristics and contention in the memory subsystem [57]. Unlike these works, we focus on page table walk scheduling. However, an interesting future study could explore interactions between page walk scheduling and scheduling at CUs.

VIII. Conclusion

We demonstrate the importance of reordering page table walk requests for GPUs. The impact of this reordering is particularly severe for irregular GPU applications that suffer from significant address translation overheads. We observed that different SIMD memory instructions executed by a GPU application could require vastly different numbers of memory accesses (*work*) to service their page table walk requests. Our SIMT-aware page table walk scheduler prioritizes page table walks from instructions that require less work to service and further batches page walk requests from the same SIMD instruction to reduce GPU-level stalls. These lead to 30% performance improvement for irregular GPU applications through improved forward progress. While we here proposed a specific SIMT-aware scheduler to demonstrate how better page table walk scheduling is valuable, we believe there exists scope for significant follow-on research on page walk scheduling akin to the rich body of work in memory controller scheduling.

IX. Acknowledgment

AMD, the AMD Arrow logo, and combinations thereof are trademarks of Advanced Micro Devices, Inc. Other product names used in this publication are for identification purposes only and may be trademarks of their respective companies. Arkaprava Basu thanks Google Inc. for providing financial aid to support his conference travel.

References

[1] NVIDIA, "Gpu-accelerated applications," 2016, http://images.nvidia.com/content/tesla/pdf/Apps-Catalog-March-2016.pdf.

[2] N. Chatterjee, M. O'Connor, G. H. Loh, N. Jayasena, and R. Balasubramonia, "Managing dram latency divergence in irregular gpgpu applications," in *SC14: International Conference for High Performance Computing, Networking, Storage and Analysis*, Nov 2014, pp. 128–139.

[3] M. Burtscher, R. Nasre, and K. Pingali, "A quantitative study of irregular programs on gpus," in *Proceedings of the 2012 IEEE International Symposium on Workload Characterization (IISWC)*, ser. IISWC '12. Washington, DC, USA: IEEE Computer Society, 2012, pp. 141–151. [Online]. Available: http://dx.doi.org/10.1109/IISWC.2012.6402918

[4] J. Meng, D. Tarjan, and K. Skadron, "Dynamic warp subdivision for integrated branch and memory divergence tolerance," in *Proceedings of the 37th Annual International Symposium on Computer Architecture*, ser. ISCA '10. New York, NY, USA: ACM, 2010, pp. 235–246. [Online]. Available: http://doi.acm.org/10.1145/1815961.1815992

[5] J. Vesely, A. Basu, M. Oskin, G. H. Loh, and A. Bhattacharjee, "Observations and opportunities in architecting shared virtual memory for heterogeneous systems," in *2016 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, April 2016, pp. 161–171.

[6] J. Sartori and R. Kumar, "Branch and data herding: Reducing control and memory divergence for error-tolerant gpu applications," in *Proceedings of the 21st International Conference on Parallel Architectures and Compilation Techniques*, ser. PACT '12. New York, NY, USA: ACM, 2012, pp. 427–428. [Online]. Available: http://doi.acm.org/10.1145/2370816.2370879

[7] T. G. Rogers, M. O'Connor, and T. M. Aamodt, "Divergence-aware warp scheduling," in *2013 46th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, Dec 2013, pp. 99–110.

[8] D. Tarjan, J. Meng, and K. Skadron, "Increasing memory miss tolerance for simd cores," in *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*, ser. SC '09. New York, NY, USA: ACM, 2009, pp. 22:1–22:11. [Online]. Available: http://doi.acm.org/10.1145/1654059.1654082

[9] B. Wang, W. Yu, X.-H. Sun, and X. Wang, "Dacache: Memory divergence-aware gpu cache management," in *Proceedings of the 29th ACM on International Conference on Supercomputing*, ser. ICS '15. New York, NY, USA: ACM, 2015, pp. 89–98. [Online]. Available: http://doi.acm.org/10.1145/2751205.2751239

[10] Wang, Bin, "Mitigating gpu memory divergence for data-intensive applications."

[11] B. Pichai, L. Hsu, and A. Bhattacharjee, "Architectural support for address translation on gpus: Designing memory management units for cpu/gpus with unified address spaces," in *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS '14. New York, NY, USA: ACM, 2014, pp. 743–758. [Online]. Available: http://doi.acm.org/10.1145/2541940.2541942

[12] J. Lowe-Power, M. Hill, and D. Wood, "Supporting x86-64 address translation for 100s of gpu lanes," in *Proceedings of International Symposium on High-Performance Computer Architecture*, ser. HPCA '14, 02 2014, pp. 568–578.

[13] R. Ausavarungnirun, V. Miller, J. Landgraf, S. Ghose, J. Gandhi, A. Jog, C. J. Rossbach, and O. Mutlu, "Mask: Redesigning the gpu memory hierarchy to support multi-application concurrency," in *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS '18. New York, NY, USA: ACM, 2018, pp. 503–518. [Online]. Available: http://doi.acm.org/10.1145/3173162.3173169

[14] K. Group, "OpenCL," 2014, https://www.khronos.org/opengl/.

[15] "Cuda c programming guide," https://docs.nvidia.com/cuda/cuda-c-programming-guide/, accessed: 2017-11-15.

[16] "Unified memory in cuda 6," https://devblogs.nvidia.com/parallelforall/unified-memory-in-cuda-6/, accessed: 2017-11-19.

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

[18] "Iommu tutorial at asplos 2016," http://pages.cs.wisc.edu/~basu/isca_iommu_tutorial/IOMMU_TUTORIAL_ASPLOS_2016.pdf, accessed: 2017-11-19.

[19] "Iommu v2 specification," https://developer.amd.com/wordpress/media/2012/10/488821.pdf, accessed: 2017-11-19.

[20] A. Bhattacharjee, "Large-reach memory management unit caches," in *Proceedings of the 46th Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO-46. New York, NY, USA: ACM, 2013, pp. 383–394. [Online]. Available: http://doi.acm.org/10.1145/2540708.2540741

[21] T. W. Barr, A. L. Cox, and S. Rixner, "Translation caching: Skip, don't walk (the page table)," in *Proceedings of the 37th Annual International Symposium on Computer Architecture*, ser. ISCA '10. New York, NY, USA: ACM, 2010, pp. 48–59. [Online]. Available: http://doi.acm.org/10.1145/1815961.1815970

[22] S. Rixner, W. J. Dally, U. J. Kapasi, P. Mattson, and J. D. Owens, "Memory access scheduling," in *Proceedings of the 27th Annual International Symposium on Computer Architecture*, ser. ISCA '00. New York, NY, USA: ACM, 2000, pp. 128–138. [Online]. Available: http://doi.acm.org/10.1145/339647.339668

[23] Y. Kim, D. Han, O. Mutlu, and M. Harchol-Balter, "Atlas: A scalable and high-performance scheduling algorithm for multiple memory controllers," in *HPCA - 16 2010 The Sixteenth International Symposium on High-Performance Computer Architecture*, Jan 2010, pp. 1–12.

[24] R. Ausavarungnirun, K. K.-W. Chang, L. Subramanian, G. H. Loh, and O. Mutlu, "Staged memory scheduling: Achieving high performance and scalability in heterogeneous systems," in *Proceedings of the 39th Annual International Symposium on Computer Architecture*, ser. ISCA '12. Washington, DC, USA: IEEE Computer Society, 2012, pp. 416–427. [Online]. Available: http://dl.acm.org/citation.cfm?id=2337159.2337207

[25] D. Li and T. M. Aamodt, "Inter-core locality aware memory scheduling," *IEEE Comput. Archit. Lett.*, vol. 15, no. 1, pp. 25–28, Jan. 2016. [Online]. Available: http://dx.doi.org/10.1109/LCA.2015.2435709

[26] "Shortest job first scheduling," http://www.geeksforgeeks.org/operating-system-shortest-job-first-scheduling-predicted-burst-time/, accessed: 2017-11-19.

[27] "The gem5 simulator," http://gem5.org/, accessed: 2017-11-19.

[28] S. Chan, "A Brief Intro to the Heterogeneous Compute Compiler," 2016, https://gpuopen.com/a-brief-intro-to-boltzmann-hcc/.

[29] L.-N. Pouchet and T. Yuki, "Polybench," 2010, http://web.cse.ohio-state.edu/~pouchet.2/software/polybench/.

[30] S. Che, M. Boyer, J. Meng, D. Tarjan, J. W. Sheaffer, S.-H. Lee, and K. Skadron, "Rodinia: A benchmark suite for heterogeneous computing," in *Proceedings of the 2009 IEEE International Symposium on Workload Characterization (IISWC)*, ser. IISWC '09. Washington, DC, USA: IEEE Computer Society, 2009, pp. 44–54. [Online]. Available: https://doi.org/10.1109/IISWC.2009.5306797

[31] S. Che, B. Beckmann, S. K. Reinhardt, and K. Skadron, "Pannotia: Understanding irregular gpgpu graph applications," in *2013 IEEE International Symposium on Workload Characterization, IISWC 2013*, 09 2013, pp. 185–195.

[32] J. R. Tramm, A. R. Siegel, T. Islam, and M. Schulz, "XSBench - The Development and Verification of a Performance Abstraction for Monte Carlo Reactor Analysis," in *PHYSOR 2014 - The Role of Reactor Physics toward a Sustainable Future*, Kyoto.

[33] D. Li, M. Rhu, D. R. Johnson, M. O'Connor, M. Erez, D. Burger, D. S. Fussell, and S. W. Keckler, "Priority-based cache allocation in throughput processors," in *2015 IEEE 21st International Symposium on High Performance Computer Architecture (HPCA)*, Feb 2015, pp. 89–100.

[34] A. Basu, J. Gandhi, J. Chang, M. D. Hill, and M. M. Swift, "Efficient virtual memory for big memory servers," in *Proceedings of the 40th Annual International Symposium on Computer Architecture*, ser. ISCA '13. New York, NY, USA: ACM, 2013, pp. 237–248. [Online]. Available: http://doi.acm.org/10.1145/2485922.2485943

[35] J. Gandhi, A. Basu, M. D. Hill, and M. M. Swift, "Efficient memory virtualization: Reducing dimensionality of nested page walks," in *Proceedings of the 47th Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO-47. Washington, DC, USA: IEEE Computer Society, 2014, pp. 178–189. [Online]. Available: http://dx.doi.org/10.1109/MICRO.2014.37

[36] V. Karakostas, J. Gandhi, F. Ayar, A. Cristal, M. D. Hill, K. S. McKinley, M. Nemirovsky, M. M. Swift, and O. Ünsal, "Redundant memory mappings for fast access to large memories," in *Proceedings of the 42Nd Annual International Symposium on Computer Architecture*, ser. ISCA '15. New York, NY, USA: ACM, 2015, pp. 66–78. [Online]. Available: http://doi.acm.org/10.1145/2749469.2749471

[37] G. Cox and A. Bhattacharjee, "Efficient address translation for architectures with multiple page sizes," in *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS '17. New York, NY, USA: ACM, 2017, pp. 435–448. [Online]. Available: http://doi.acm.org/10.1145/3037697.3037704

[38] T. Zheng, D. Nellans, A. Zulfiqar, M. Stephenson, and S. W. Keckler, "Towards high performance paged memory for gpus," in *2016 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, March 2016, pp. 345–357.

[39] T. G. Rogers, M. O'Connor, and T. M. Aamodt, "Cache-conscious wavefront scheduling," in *Proceedings of the 2012 45th Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO-45. Washington, DC, USA: IEEE Computer Society, 2012, pp. 72–83. [Online]. Available: http://dx.doi.org/10.1109/MICRO.2012.16

[40] O. Mutlu and T. Moscibroda, "Parallelism-aware batch scheduling: Enhancing both performance and fairness of shared dram systems," in *Proceedings of the 35th Annual International Symposium on Computer Architecture*, ser. ISCA '08. Washington, DC, USA: IEEE Computer Society, 2008, pp. 63–74. [Online]. Available: http://dx.doi.org/10.1109/ISCA.2008.7

[41] Y. Hao, Z. Fang, G. Reinman, and J. Cong, "Supporting address translation for accelerator-centric architectures," in *2017 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, Feb 2017, pp. 37–48.

[42] J. Lee, M. Samadi, and S. Mahlke, "Vast: The illusion of a large memory space for gpus," in *Proceedings of the 23rd International Conference on Parallel Architectures and Compilation*, ser. PACT '14. New York, NY, USA: ACM, 2014, pp. 443–454. [Online]. Available: http://doi.acm.org/10.1145/2628071.2628075

[43] H. Yoon, J. Lowe-Power, and G. S. Sohi, "Filtering translation bandwidth with virtual caching," in *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS '18. New York, NY, USA: ACM, 2018, pp. 113–127. [Online]. Available: http://doi.acm.org/10.1145/3173162.3173195

[44] A. Bhattacharjee and M. Martonosi, "Inter-core cooperative tlb for chip multiprocessors," in *Proceedings of the Fifteenth Edition of ASPLOS on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS XV. New York, NY, USA: ACM, 2010, pp. 359–370. [Online]. Available: http://doi.acm.org/10.1145/1736020.1736060

[45] B. Pham, V. Vaidyanathan, A. Jaleel, and A. Bhattacharjee, "Colt: Coalesced large-reach tlbs," in *2012 45th Annual IEEE/ACM International Symposium on Microarchitecture*, Dec 2012, pp. 258–269.

[46] A. Basu, M. D. Hill, and M. M. Swift, "Reducing memory reference energy with opportunistic virtual caching," in *Proceedings of the 39th Annual International Symposium on Computer Architecture*, ser. ISCA '12. Washington, DC, USA: IEEE Computer Society, 2012, pp. 297–308. [Online]. Available: http://dl.acm.org/citation.cfm?id=2337159.2337194

[47] V. Karakostas, J. Gandhi, A. Cristal, M. D. Hill, K. S. McKinley, M. Nemirovsky, M. M. Swift, and O. S. Unsal, "Energy-efficient address translation," in *2016 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, March 2016, pp. 631–643.

[48] B. M. Rogers, A. Krishna, G. B. Bell, K. Vu, X. Jiang, and Y. Solihin, "Scaling the bandwidth wall: Challenges in and avenues for cmp scaling," in *Proceedings of the 36th Annual International Symposium on Computer Architecture*, ser. ISCA '09. New York, NY, USA: ACM, 2009, pp. 371–382. [Online]. Available: http://doi.acm.org/10.1145/1555754.1555801

[49] N. Chatterjee, N. Muralimanohar, R. Balasubramonian, A. Davis, and N. P. Jouppi, "Staged reads: Mitigating the impact of dram writes on dram reads," in *Proceedings of the 2012 IEEE 18th International Symposium on High-Performance Computer Architecture*, ser. HPCA '12. Washington, DC, USA: IEEE Computer Society, 2012, pp. 1–12. [Online]. Available: http://dx.doi.org/10.1109/HPCA.2012.6168943

[50] E. R. Blem, M. D. Sinclair, and K. Sankaralingam, "Challenge benchmarks that must be conquered to sustain the gpu revolution," 2011.

[51] N. Chatterjee, M. O'Connor, G. H. Loh, N. Jayasena, and R. Balasubramonian, "Managing dram latency divergence in irregular gpgpu applications," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, ser. SC '14. Piscataway, NJ, USA: IEEE Press, 2014, pp. 128–139. [Online]. Available: https://doi.org/10.1109/SC.2014.16

[52] O. Mutlu and T. Moscibroda, "Stall-time fair memory access scheduling for chip multiprocessors," in *40th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO 2007)*, Dec 2007, pp. 146–160.

[53] Y. Kim, M. Papamichael, O. Mutlu, and M. Harchol-Balter, "Thread cluster memory scheduling: Exploiting differences in memory access behavior," in *Proceedings of the 2010 43rd Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO '43. Washington, DC, USA: IEEE Computer Society, 2010, pp. 65–76. [Online]. Available: http://dx.doi.org/10.1109/MICRO.2010.51

[54] A. Jog, E. Bolotin, Z. Guz, M. Parker, S. W. Keckler, M. T. Kandemir, and C. R. Das, "Application-aware memory system for fair and efficient execution of concurrent gpgpu applications," in *Proceedings of Workshop on General Purpose Processing Using GPUs*, ser. GPGPU-7. New York, NY, USA: ACM, 2014, pp. 1:1–1:8. [Online]. Available: http://doi.acm.org/10.1145/2576779.2576780

[55] M. K. Jeong, M. Erez, C. Sudanthi, and N. Paver, "A qos-aware memory controller for dynamically balancing gpu and cpu bandwidth use in an mpsoc," in *Proceedings of the 49th Annual Design Automation Conference*, ser. DAC '12. New York, NY, USA: ACM, 2012, pp. 850–855. [Online]. Available: http://doi.acm.org/10.1145/2228360.2228513

[56] H. Usui, L. Subramanian, K. K.-W. Chang, and O. Mutlu, "Dash: Deadline-aware high-performance memory scheduler for heterogeneous systems with hardware accelerators," *ACM Trans. Archit. Code Optim.*, vol. 12, no. 4, pp. 65:1–65:28, Jan. 2016. [Online]. Available: http://doi.acm.org/10.1145/2847255

[57] O. Kayiran, A. Jog, M. T. Kandemir, and C. R. Das, "Neither more nor less: Optimizing thread-level parallelism for gpgpus," in *Proceedings of the 22Nd International Conference on Parallel Architectures and Compilation Techniques*, ser. PACT '13. Piscataway, NJ, USA: IEEE Press, 2013, pp. 157–166. [Online]. Available: http://dl.acm.org/citation.cfm?id=2523721.2523745