Neighborhood-aware address translation for irregular GPU applications

Seunghee Shin*[1], Michael LeBeane†, Yan Solihin‡, Arkaprava Basu§

*Binghamton University, †Advanced Micro Devices, Inc., ‡North Carolina State University, §Indian Institute of Science

sshin@binghamton.edu, Michael.Lebeane@amd.com, solihin@ncsu.edu, arkapravab@iisc.ac.in

***Abstract*—Recent studies on commercial hardware demonstrated that irregular GPU workloads could bottleneck on virtual-to-physical address translations. GPU's single-instruction-multiple-thread (SIMT) execution generates many concurrent memory accesses, all of which require address translation before accesses can complete. Unfortunately, many of these address translation requests often miss in the TLB, generating many concurrent page table walks. In this work, we investigate how to reduce address translation overheads for such applications.**

We observe that many of these concurrent page walk requests, while irregular from the perspective of a single GPU wavefront, still fall on *neighboring* virtual pages. The address mappings for these neighboring pages are often stored in the same 64-byte cache line. Since cache lines are the smallest granularity of memory access, the page table walker implicitly reads address mappings (page table entries or PTEs) of many neighboring pages during the page walk of a single virtual address (VA). However, in conventional hardware, mappings not associated with the original request are simply discarded. In this work, we propose mechanisms to coalesce the address translation needs of all pending page table walks in the same neighborhood that happen to have their address mappings fall on the same cache line. This is almost *free*; the page table walker (PTW) already reads a full cache line containing address mappings of all pages in the neighborhood. We find this simple scheme can reduce the number of accesses to the in-memory page table by around 37%, on average. This speeds up irregular GPU applications by an average of $1.7\times$.

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

I. Introduction

GPUs have emerged as a first-class computing platform. The massive data parallelism of GPUs was first leveraged by highly-structured parallel tasks such as matrix multiplications. However, GPUs have more recently found uses across a broader range of application domains such as deep learning, graph analytics, 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].

A recent study on real hardware demonstrated that such irregular memory accesses could slow down irregular GPU workloads by up to $4\times$ due to address translation overheads alone [5]. Our evaluation on a simulator corroborates their finding. That study found that the negative impact of access irregularity could be greater on the address translation than on the data access itself. 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. Typically, a hardware page table walker (PTW) walks (accesses) the in-memory page table to find the desired translation.

To match a GPU's need for large memory bandwidth, hardware designers often provision multiple independent PTWs. However, TLB misses from irregular GPU workloads, and their corresponding page table walk requests, often happen in bursts. Consequently, they add significant queuing delay in the critical path of execution. A cache access cannot begin until the corresponding address translation completes, since modern GPUs tend to employ physically-tagged caches.

In this work, we aim to reduce the address translation overhead of such irregular GPU workloads. Our work relies on two key observations. First, many concurrent page table walks request address translations for virtual pages belonging to the same *neighborhood*. We define a set of page walk requests to be *concurrent* if they are pending or being serviced during the same or overlapping period of execution. We define a set of virtual page addresses to be part of the same neighborhood if their address mappings (i.e., corresponding page table entries or PTEs) are contained in the same cache line. For example, each PTE in an x86-64 system is 8-bytes long [6]. Therefore, a typical 64-byte long cache line contains PTEs of eight neighboring virtual page addresses.

Second, like any other memory access, a PTW accesses memory at the granularity of a cache line (e.g., 64 bytes) while walking the in-memory page table [7]. However, it makes use of only the 8-byte PTE needed for translating the requested virtual page address. Instead, we propose to service all pending page walk requests for any virtual page addresses in the same neighborhood (i.e., whose PTEs are contained in the same cache line) at the same time. This makes better use of other PTEs in the same cache line anyway brought in by the walker. Consequently, it obviates later memory accesses to the same cache line that would have otherwise happened when those page walk requests for virtual pages on the same neighborhood

[1]This work was performed during the author's internship at AMD Research.

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

are serviced. Ultimately, the reduction in the number of memory accesses to the in-memory page table speeds up performance by reducing page walk latency.

We extended the page walking mechanism to exploit the observations mentioned above. Specifically, whenever a PTW receives a 64-byte cache line containing a requested PTE, it also scans pending (concurrent) page walk requests to addresses in the same neighborhood (i.e., whose PTEs also fall in the just-brought cache line). All these page walk requests are then serviced immediately disregarding the order in which they may have arrived at the PTW. Effectively, this technique *coalesces* accesses to the in-memory page table for page walk requests outstanding at the same time and for pages in the same neighborhood.

We then took this mechanism a step further by observing that in a typical hierarchical page table organization (e.g., 4-level for x86) accesses to the upper levels (non-leaf) of a page table could similarly benefit from opportunistic coalescing for the entries in the same neighborhood. For example, an x86-64 page table is structured as a four-level radix tree. While the final translation is stored in the leaf level, the upper levels of the tree are traversed by the hardware page table walker to reach the correct leaf node. More importantly, like the leaf level, several entries of the upper levels of a page table are packed in a single cache line. In x86-64, the size of each entry in the upper level of the page table is also 8-byte long. Thus, eight such upper-level page table entries are stored in a single 64-byte cache line. Consequently, while accessing the upper levels of a page table, the eight entries that form a neighborhood are brought together by the walker. We leverage this to reduce memory accesses to the upper-levels of the page table as well.

Our empirical evaluation finds that such neighborhood-aware servicing of page walk requests could reduce the number of memory accesses performed by page table walkers by around 40%, on average. Our evaluation also demonstrates that coalescing of accesses to both leaf level and the upper-levels of the page table are almost equally important. Our proposed enhancement speeds up irregular GPU workloads by $1.7\times$ on average, and by up to $2.3\times$ in the best case.

II. GPU VIRTUAL MEMORY SUBSYSTEM AND IMPACT OF IRREGULAR APPLICATIONS

In this section, we first briefly discuss the baseline GPU architecture and its virtual memory subsystem. We then quantitatively discuss the impact of irregularity in memory access patterns on a GPU's address translation mechanism.

A. Execution Hierarchy in a GPU

GPUs are designed for massive data-parallel processing that operates on hundreds to thousands of data elements concurrently. GPU's hardware resources are typically organized in a hierarchy to effectively manage the massive concurrency.

The top part of Figure 1 depicts the architecture of a typical GPU. Compute Units (CUs) are the basic computational blocks of a GPU, and there are typically 8 to 64 CUs in a GPU. Each CU includes multiple Single-Instruction-Multiple-Data (SIMD)

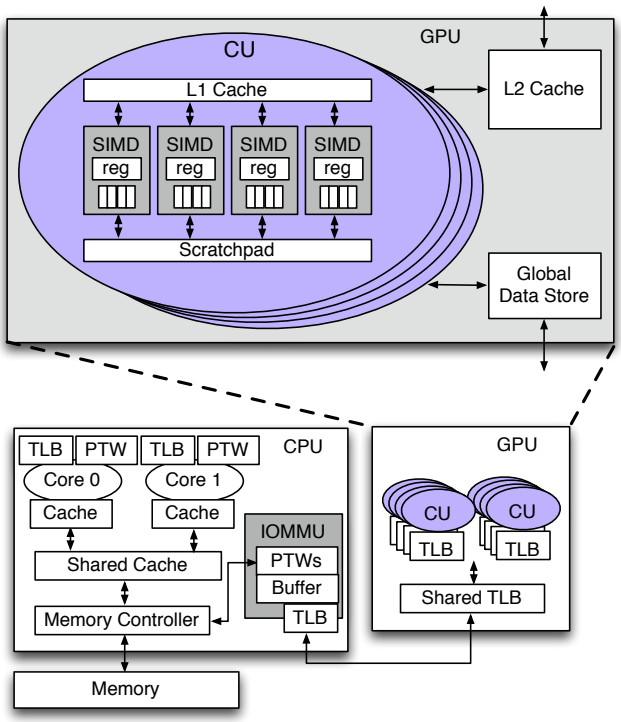

Fig. 1: Baseline Heterogeneous System Architecture.

units, each of which has multiple *lanes* of execution (e.g., 16). GPU threads are scheduled to SIMD engines in a bundle called a wavefront (or warp), which is typically composed of 32 or 64 threads. These wavefronts execute the same instruction with different data on a SIMD engine. A Single-Instruction-Multiple-Thread (SIMT) model is mapped to the SIMD engines by using execution masks in the case where GPU threads in the same wavefront follow different execution paths.

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 within the CU. When several data elements accessed by a SIMD instruction reside in the same cache line, a hardware coalescer combines these requests into a single cache access to gain efficiency. Finally, a large L2 cache is shared across all CUs in a GPU.

B. Shared Virtual Memory in GPUs

GPUs have adopted several key programmability-enhancing features as they mature to become first-class compute units. One such feature is the shared virtual memory (SVM) across the CPU and the GPU [8], [9]. For example, full compliance with industry promoted standards like the Heterogeneous System Architecture (HSA) requires GPUs to support SVM [9].

There are typically two ways to enable shared virtual memory in modern commercial GPUs – ① shared page table between CPU and GPU via the IO Memory Management Unit (IOMMU) hardware (detailed next), and ② selective mirroring of parts of page tables of CPU and GPU by an OS driver. In this work, we focus on the former without loss of generality.

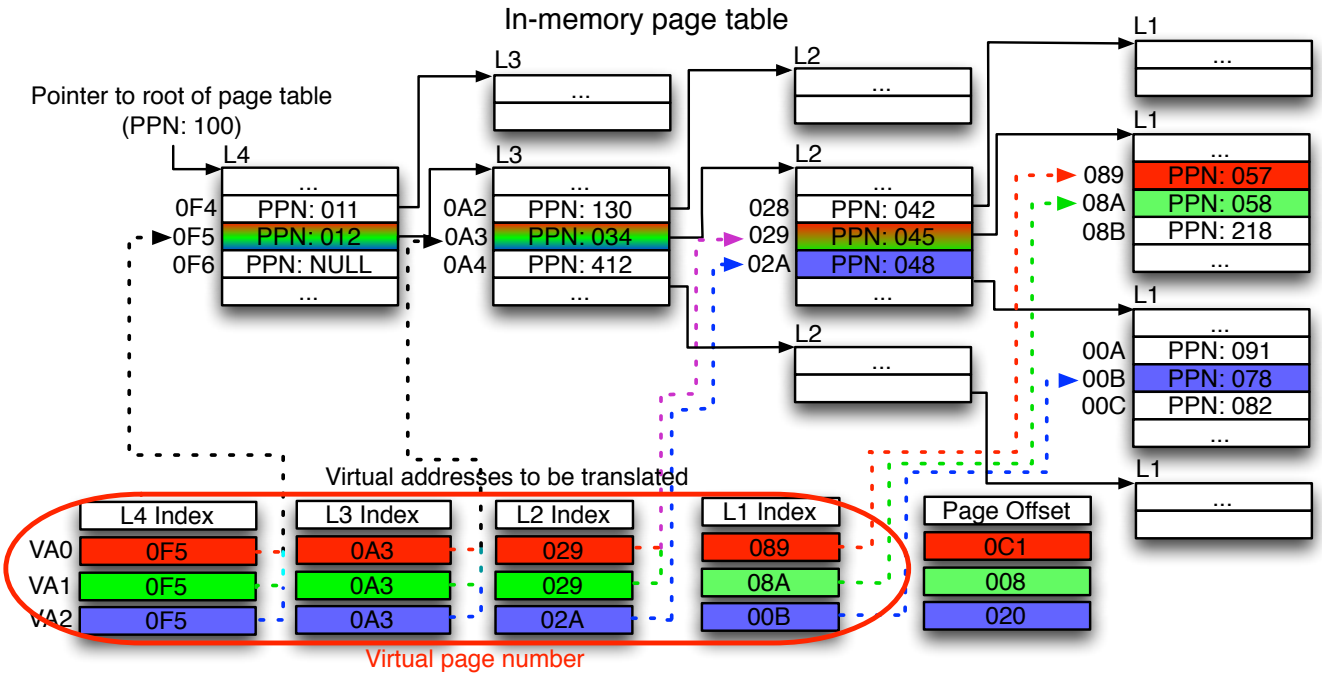

Fig. 2: Example page walks of three different virtual page addresses (colored red, green and blue).

The bottom part of Figure 1 depicts the key hardware components of SVM in a commercial GPU. Conceptually, the key enabler for SVM is the GPU's ability to *walk* the same x86-64 page table as the CPU via the IOMMU hardware [10], [11], [5]. By sharing the same page table, a process running on a GPU can observe the same virtual-to-physical address mapping as a process running on a CPU and consequently, share the same virtual memory.

Address translation using an IOMMU: Figure 2 illustrates how a hardware page table walker in an IOMMU translates three different virtual addresses (colored red, green, blue) by walking an in-memory page table. An x86-64 based processor's page table is structured as a 512-ary radix tree with four levels, and thus, requires four memory accesses to complete a page walk. As shown in the bottom part of the figure, each virtual address is broken into a virtual page number and a page offset. A virtual page number is translated to a physical page frame number by walking the page table, which is then concatenated with the page offset to generate the desired physical address.

A page walk starts by looking up the 4KB page containing the root node of a page table (also called level L4). A root node has 512 pointers [2] to nodes at next level of the tree (referred to as the L3 level). The top nine bits of a 48-bit wide virtual address (bits 47 to 39), known as the `L4 index`, are used to index into the root node to find the pointer to the appropriate L3 node. An L3 node also contains 512 pointers to nodes in the next level (referred to as the L2 level). The next nine bits of the VA (bits 38-30), known as the `L3 index`, are used to index into the L3 node to reach an appropriate node in the L2 level. Similarly, bits 29 to 21, known as the `L2 index`, are used to index into an L2 node to reach a node in the leaf level (referred to as the L1 level). An L1 node contains 512 physical page frame numbers, each corresponding to a 4KB virtual address page. Bits 20 to 12 of the VA, known as the `L1 index`, are used to index into an L1 node to find the desired physical page frame number. Henceforth, we will refer levels L4 to L2 as upper levels of the page table.

From the example, we observe that the page walk of the first two virtual addresses (`VA0` and `VA1`) shares the same entries for all upper levels of the page table (thus shaded in both red and green). Similarly, the third virtual address (`VA2`) shares the entries in the first two upper levels of the page table with the other two. This observation is exploited by hardware page walk caches (PWCs)[5], [12], [13]. PWCs store recently-used entries from the upper-levels of a page table. Hits in PWCs reduce the number memory accesses needed for a walk by up to three memory accesses depending upon which upper level (L4, L3 or L2) produces the hit. For example, a hit for the entire upper level (L4, L3 and L2) will need just one memory request to complete the walk by accessing only the leaf node (L1). In contrast, a hit for only the root level requires three memory accesses. In the worst case, a complete miss in the PWCs requires four memory accesses to complete a page walk.

An IOMMU typically houses multiple independent page table walkers (e.g., 8-16) to concurrently service several page table walk requests [5]. Multiple walkers are important since GPUs demand high memory bandwidth and, consequently, often send many concurrent walk requests. The IOMMU itself has two levels of TLBs to cache recently used address translations, but they are relatively small and designed to primarily serve devices that do not have their own TLBs (e.g., a Network Interface Controller). Page walk requests typically

[2] A pointer can be NULL if the corresponding virtual address range is unmapped.

queue up in IOMMU's page walk request buffer (a.k.a, IOMMU buffer) before triggering the walk. When a walker becomes free (e.g., after it finishes servicing a walk), it starts servicing a new request from the IOMMU buffer in the order of its arrival.

GPU TLB Hierarchy: GPUs typically have a sophisticated TLB hierarchy to reduce the number of page walks. A TLB caches recently-used address translation entries to avoid accessing in-memory page tables on every memory access. When multiple data elements accessed by a SIMD instruction reside on the same page, only a single virtual-to-physical address translation is needed. A hardware coalescer exploits this locality to look up the TLB hierarchy only once for such same-page accesses. Each CU has a private L1 TLB. Misses in the L1 TLB looks up a larger L2 TLB that is shared across all the CUs in the GPU (bottom portion of Figure 1). A translation request that misses in both levels is forwarded to the IOMMU.

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 memory instruction. ③ The coalesced request looks up the GPU's L1 TLB and then the GPU's shared L2 TLB (if it misses in the L1 TLB). ④ On a miss in the GPU's L2 TLB, the request is sent to the IOMMU. ⑤ At the IOMMU the request first 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 selects a pending request from the IOMMU buffer in first-come-first-serve 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 IOMMU and the GPU TLBs.

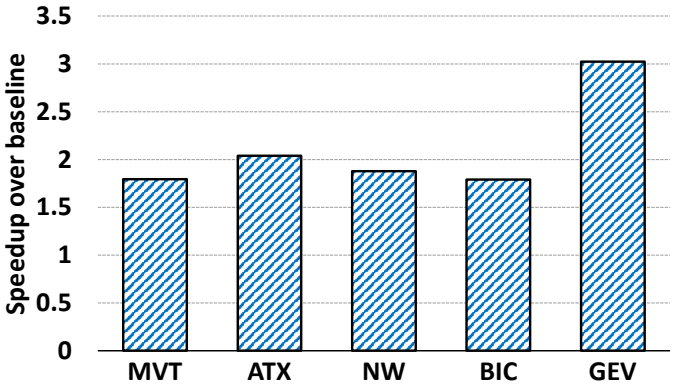

Fig. 3: Speed up possible with *ideal* address translation.

C. Address translation overheads of irregular applications

Irregular GPU workloads make data-dependent memory accesses with little spatial locality [2], [3]. Many pointer-chasing algorithms like graph manipulation show such irregularity in its memory access patterns.

Irregular memory accesses cause *memory access divergence* in a GPU's execution model. Although different work-items within a wavefront execute the same instruction, they access data from distinct pages. This renders the hardware coalescer ineffective, and consequently, many concurrent TLB accesses are generated by execution of a single SIMD load/store instruction. Furthermore, many of these requests often miss in the TLB owing to low locality of irregular applications. Eventually, these address translation requests queue up in the IOMMU buffer to be serviced by the page table walkers. The significant queuing latency at the page table walkers ultimately slows down workloads.

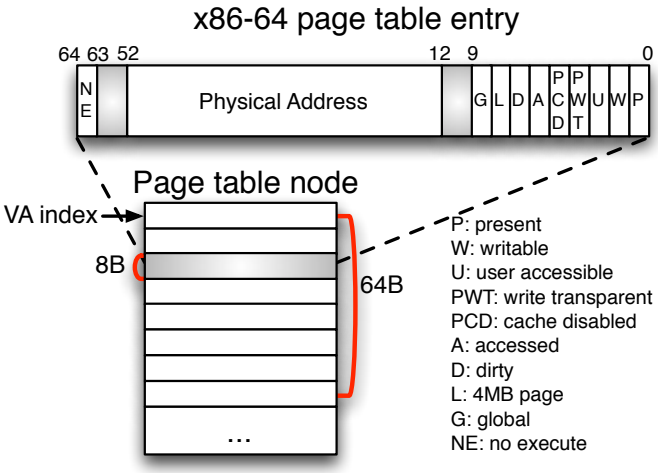

Fig. 4: x86-64 page table node.

A recent study on commercial GPU hardware demonstrated that such divergent accesses could slow down irregular GPU workloads by up to $3.7 - 4\times$ due to address translation overheads [5]. Our simulation results (detailed in Section V) corroborated these findings. Figure 3 shows the speedup achievable for five representative workloads on a system with *ideal* address translation over a typical baseline GPU with SVM. A system with an ideal address translation mechanism is an unrealistic system where all translation happens in a single cycle. Thus, the height of each bar in the figure shows performance lost due to address translation overheads. From the figure we observe that various irregular GPU workloads slowed down by $3 - 4\times$ due to address translation overheads even when using a small memory footprint of a few MBs.

In this work, we aim to reduce address translation overheads for such irregular GPU workloads.

III. The Case for Neighborhood-Aware Address Translation

We utilize the observation that multiple entries of a page table are contained within a single cache line. Each page table entry in a x86-64 page table is 8-byte long, therefore, a typical 64-byte cache line contains eight PTEs (Figure 4). These eight PTEs map eight contiguous 4KB virtual memory pages. We define such 32KB aligned virtual memory regions whose PTEs are contained in the same cache line as a *neighborhood* of the leaf level of a page table (L1 level). While we focus on x86-64's page tables, other architecture such as ARM, also have a near-identical page table format, and thus, the observation is widely applicable.

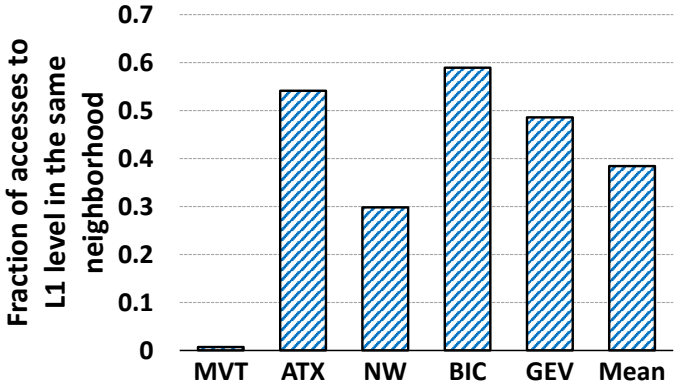

Fig. 5: Fraction of concurrent page walk requests that falls in the same leaf level (L1) neighborhood.

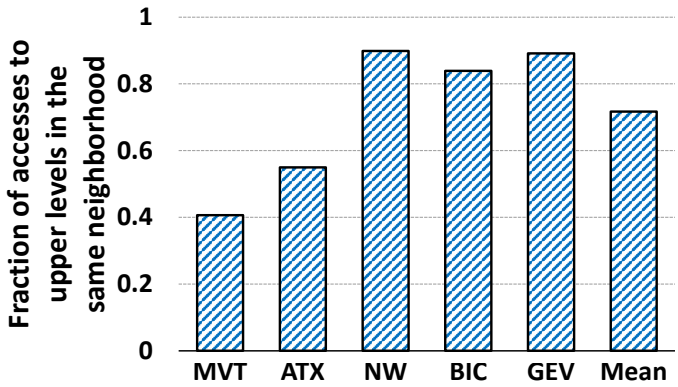

Fig. 6: Fraction of concurrent page walk requests that falls in same neighborhood of the upper-levels of the page table.

We then observe that page table walkers access memory at the granularity of a cache line, as in a CPU. Each time a walker reads an entry in the page table as part of a page walk it does so by bringing the entire cache line containing the PTEs for the entire neighborhood. However, only the desired PTE is typically used by a page table walker for calculating the physical address, and the rest of cache line is discarded.

We, however, find that many concurrent page table walks generated by execution of irregular GPU applications fall within the same neighborhood. For example, in Figure 2 the PTEs in the leaf level (L1) of the page table `PPN:057` and `PPN:058`, for virtual addresses `VA0` and `VA1`, respectively, fall in the same cache line. Therefore, address `VA0` and `VA1` are in same neighborhood of virtual address space with respect to the leaf level of the page table.

We found that concurrent access to the page table for addresses in the same neighborhood is not uncommon. The reason is that CUs concurrently execute independent work-groups, but they often run similar regions of code during the same period of execution. Consequently, the data accesses and corresponding page walk requests often fall in the same neighborhood in the virtual address space.

A smarter page table walker could exploit the neighborhood information to reduce the number of memory accesses to a page table. Access to PTEs of all concurrent page walk requests that fall in the same neighborhood can be serviced together through request coalescing. This coalescing could avoid later accesses to same cache line when servicing a different page walk request to another virtual page address in the same neighborhood.

We empirically measure the potential savings in the number of memory accesses by coalescing page walks that fall into the same neighborhood. Figure 5 shows the fraction of all memory accesses to leaf levels (L1) of the page table that are for the virtual address pages in the same neighborhood, and are concurrent (i.e., pending or being serviced at the page table walker during the same period of execution). We observe that this fraction is around 0.4, on average, with the only major outlier being `MVT`. This particular application has a nearly random memory access pattern, and thus only a small number of concurrent page walk requests fall in the same neighborhood of virtual memory.

We then observe that multiple upper-level page table entries (level L4-L2) are also contained within a single cache line. In fact, all upper-level page table entries in x86-64 are also 8-bytes long. Therefore, a typical 64-byte cache line also contains eight upper-level page table entries. For example, in Figure 2 the entry for an L2 level of the page table for the virtual address `VA2` falls in the same cache line as virtual addresses `VA0` and `VA1`. However, note that a neighborhood for the L2 level entries covers 16MB of aligned virtual address region. This is because each entry in the L2 level of the page table corresponds to 2MB of virtual address region and eight such entries are contained in a cache line. Similarly, neighborhood for the L3 and L4 levels of a page table are 8GB and 4TB aligned virtual address regions, respectively.

Figure 6 shows the fraction of memory accesses to the upper levels of the page table that falls in the same neighborhood of their respective levels and are from concurrent page walks. We exclude any walk that uses exact same upper-level page walk entries as that is typically well captured by page walk caches (PWCs). This measurement thus captures the potential of reducing memory accesses to the upper levels of the page table by exploiting neighborhood knowledge.

Summary: ① We observe that typically eight page table entries are contained in same cache line. ② We observe that the page table walkers access in-memory page tables at the granularity of a cache line. ③ We empirically find that many concurrent page table walk requests have their page table entries contained in the same cache line (a.k.a., in same neighborhood). ④ These observations can be leveraged by an enhanced page table walker to significantly reduce the number of walks it needs to perform to service a given set of page walk requests.

IV. Design and Implementation of Neighborhood-Aware Page Table Walking

Our goal is to design a page walk mechanism that opportunistically coalesces memory accesses to the page table for page walk requests that fall in the same neighborhood in the virtual address space. In Section V-B, we will demonstrate that such coalescing can significantly reduce the total number of memory accesses performed by page table walker(s).

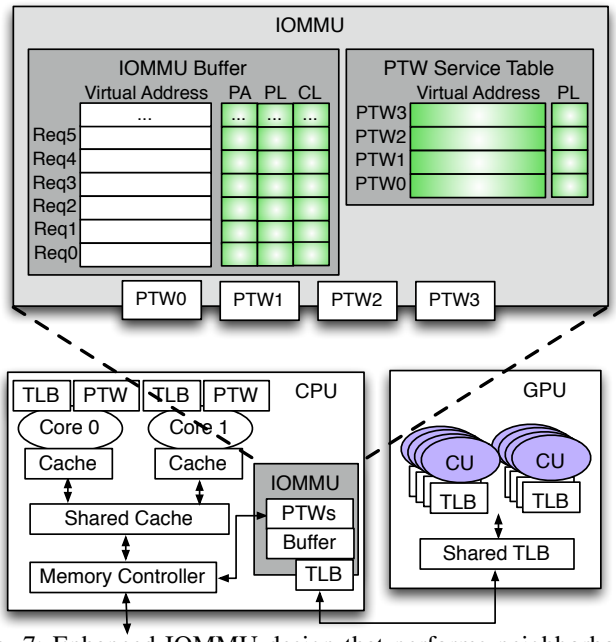

Fig. 7: Enhanced IOMMU design that performs neighborhood aware coalescing of accesses to the page table. New hardware is colored green (shaded).

We modify the IOMMU hardware that houses the page table walkers in two important ways to achieve aforementioned goals ① Whenever a page table walker completes a memory access to one of the levels of a page table, it searches the pending walk requests in the IOMMU buffer for coalescing opportunities. ② Page walk requests are selected to start their walk only if it presents no coalescing opportunity with any on-going page walks.

A. Additional IOMMU hardware state

Figure 7 shows how we propose to extend the IOMMU design with additional hardware state. We first extended each entry of the IOMMU buffer [3] with three new fields. The field `PL` contains the level of the page table up to which the given walk request has coalesced with another page table access. The value of this field could be anything between $L4$–$L1$ or `NULL`. The second new field, called `PA`, holds the physical address of the page table node in the next level (identified by the value in field `PL`) of the walk. This field is populated when the given page walk request gets coalesced for upper-level accesses to the page table. Both these fields are `NULL` if a given walk request has not coalesced. Finally, a one-bit flag (`CL`) notes if the corresponding walk request is being considered for coalescing with an on-going access to the page table. This information is used in deciding which page walk request to be serviced by a free page table walker (explained later in the Section).

We then added a small new structure called the Page Walk Service Table (PWST). It has one entry for each independent page table walker in the IOMMU (eight in our experiments). Each entry in the PWST contains the virtual page address of the page walk request currently being serviced by the corresponding walker and the level of the page table which is currently being accessed. This information is utilized to determine the opportunity for coalescing. In total, we add only around 1.5KB of additional state in these two structures.

[3] As described in Section II, the IOMMU buffer is used to hold page walk requests that are yet to start their walk.

B. Operation for coalescing accesses to the page table

We now describe how we opportunistically coalesce accesses to the page table by leveraging neighborhood information at every level of a page table walk. Whenever a memory access to a page table completes, the coalescing logic in the IOMMU first looks up the corresponding entry in the PWST structure to find out the virtual page address of the current walk and the page table level that was accessed. If the just-completed memory access was to a leaf level (L1) then the entire walk has completed, and the translated physical page frame number is returned to the TLB as is in the baseline. The coalescing logic then scans the IOMMU buffer for any pending page walk request whose virtual page address falls in the same neighborhood (i.e., in the same 32KB aligned VA region) of the virtual address of the just-completed walk. The desired physical page frame address for any such matching walk request is already available in the 64-byte cache line brought by the walker and thus, immediately returned.

If the just-completed memory access was to the L2, L3 or L4 level (upper-level) of the page table then the neighborhood is 16MB, 8GB or 4TB aligned virtual memory regions, respectively (explained in Section III). Similar to before, the coalescing logic scans the IOMMU buffer for any page walk request in the same neighborhood of the just-completed page table access. For any such matching request, the physical page frame number for the next level of the page table is available in the 64-byte cache line brought by the walker. The field `PA` in the corresponding entry in the IOMMU buffer is then updated with this address. The field `PL` is also updated to contain the corresponding level of the page table. Values of these fields are used to complete walk requests whose accesses to only upper-levels of the page table are coalesced (i.e., a partially coalesced walk).

We then modify the logic of selecting which of the pending page walk requests to be serviced next by a page table walker. Typically, the IOMMU selects requests in first-come-first-serve order from its buffer. Our coalescing logic, however, slightly modifies this to avoid servicing a walk request that could be coalesced with an on-going page table access by another page table walker. Specifically, a pending page walk request in the IOMMU buffer is not selected to start its walk if the entry has its coalescing bit (`CL` bit) set. The coalescing bit indicates that an on-going page table access could be used to service the page walk request entirely or partially. Otherwise, all valid PWST entries are scanned to compute the neighborhood of each on-going page table access using their respective virtual page address and the level of the page table being accessed. If an entry in the IOMMU buffer has a virtual page address that falls within any of these neighborhoods, the `CL` bit of the

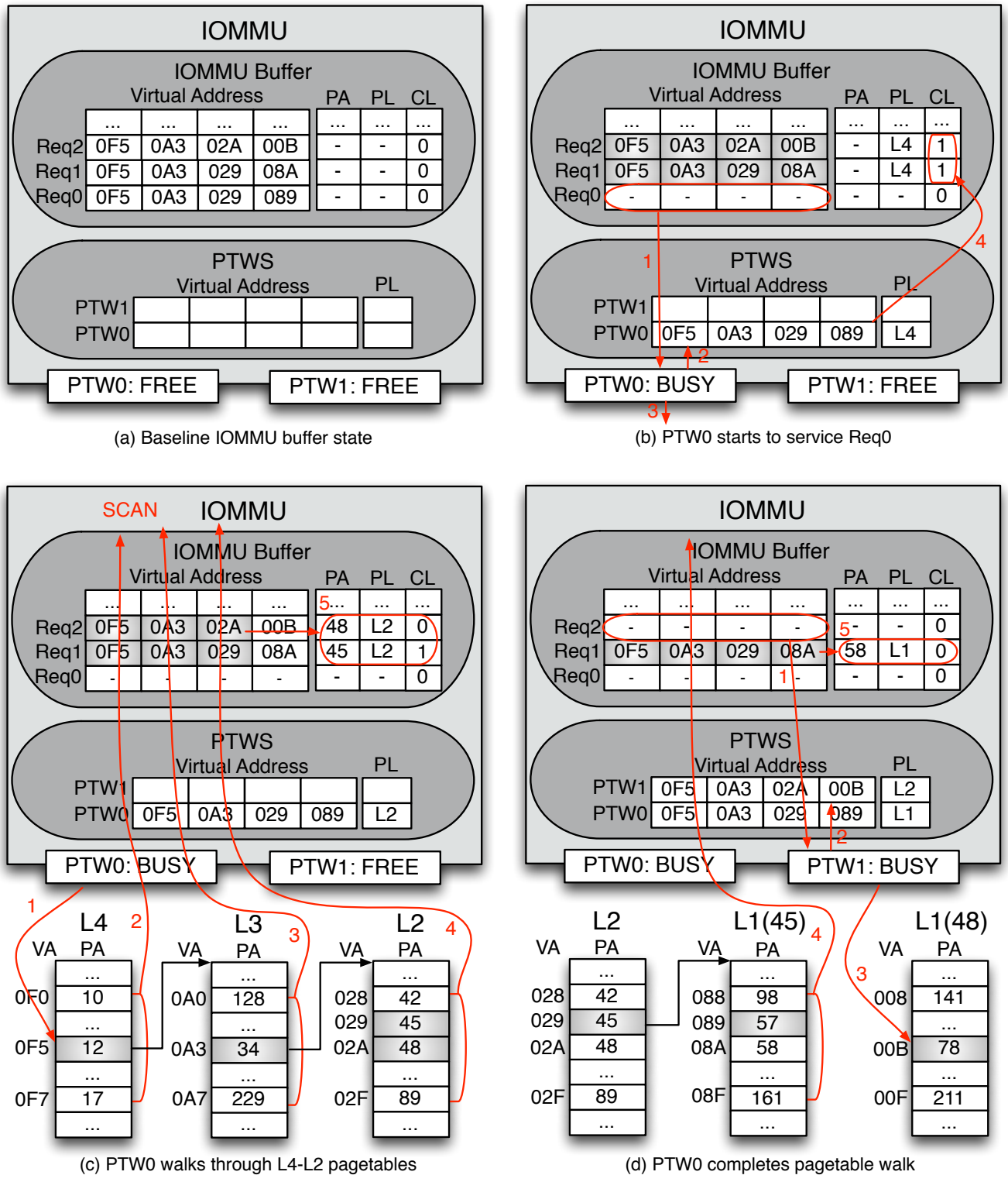

Fig. 8: Example of coalescing page table accesses. The example virtual addresses are the same as in Figure 2.

corresponding entry is set and not immediately considered to be serviced by a page table walker. If both of these conditions are false then a request from the buffer is selected on first-come-first-serve basis.

The page walk then starts by looking up the `PA` and `PL` fields in the IOMMU buffer entry of the selected request. If both these fields are unset then the walk starts from the root of the page table. Otherwise, the walk request is partially coalesced with previous walks. The walker then needs to finish the walk by starting from the level indicate in the `PL` field. The physical

address of the node in the page table that is to be accessed next is found in the PA field. Finally, the corresponding PWST entry is populated with the information of the walk request. The IOMMU buffer entry of the chosen walk request is deleted.

C. Example operation

Figure 8 shows a running example of how page table accesses are coalesced based on neighborhood information. Figure 8(a) shows the initial state with three page walk requests waiting to be serviced in the IOMMU buffer. These addresses are the same three virtual addresses (VA0, VA1 and VA2) whose page walk was depicted in Figure 2 of Section II. While Figure 2 depicts how different page walks often access entries in the same cache line (i.e., neighborhood), here we will demonstrate how this is leveraged to coalesce page table accesses in our enhanced IOMMU design. For ease of explanation, we assume there are two independent page table walkers in the IOMMU.

Figure 8(b) shows that the page walk request for Req0 (virtual address 0x0F5|0x0A3|0x029|0x089) is serviced at page table walker 0 (PTW0). Although the page table walker PTW1) is free, Req1 and Req2 do not start their walk. Instead, the coalescing logic notes the opportunity to coalesce walks of these requests with on-going walk for Req0 in the fields PL and CL of the corresponding IOMMU buffer entries.

Figure 8(c) shows the status of the IOMMU when PTW0 finishes page table accesses up to the level L2. The buffer entries corresponding to Req1 and Req2 are updated to contain the physical page address of the corresponding L1 nodes in their respective page walk. Note that the address of the L1 node in the walk for Req2's access was found in the same cache line brought by PTW0 while walking the page table for Req0. There is no more opportunity for Req2 to be coalesced with the ongoing page walk, and thus, its CL bit is unset. Removing the CL bit makes it eligible to be serviced by the other page table walker (PTW1). However, Req1's CL bit remains set as it can be coalesced with Req0 till the leaf.

Figure 8(d) shows the state after PTW0 finishes the access to the L1 (leaf) level of the page table. At this point, the address translation for Req0 is complete. Req1 can also complete its translation since its PTE falls in the same 64-byte cache line brought in by PTW0 for Req0. The page walks of Req0 and Req1 are thus completely coalesced and no access to in-memory page table is performed for Req1.

On the other hand, the page walk for Req2 could only be coalesced till level L2 with Req0's walk. Therefore, PTW1 finishes Req1's page walk by accessing the L1 node.

V. Evaluation

We describe evaluation methodology and analyze the results.

A. Methodology

We used the execution-driven gem5 simulator that models a heterogeneous system with a CPU and an integrated GPU [14]. We 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 IOMMU module, we model a two-level TLB hierarchy, multiple independent parallel page table walkers, and page walk caches to closely mirror the real hardware. In addition, we implemented our proposed logic and state for coalescing page table accesses inside the IOMMU module.

TABLE I: The baseline system configuration.

GPU	2GHz, 8 CUs, 10 waves per SIMD-16, 4 SIMDs per CU, 64 threads per wave
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 channels, 16 banks per rank, 2 ranks per channel

TABLE II: GPU benchmarks for our study.

	Benchmark (Abbrev.)	Description	Memory Footprint
Irregular applications	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 (BIC)	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
	LUD (LUD)	Lower upper decomposition	1.2MB
	Color (CLR)	Graph coloring algorithm	26.68MB
	Back Prop. (BCK)	Machine learning algorithm	108.03MB
	Hotspot (HOT)	Processor thermal simulation algorithm	12.02MB

The simulator runs unmodified applications written in OpenCL [15] or in HC [16]. Table I lists the relevant parameters for the GPU, the memory system, and the address translation mechanism of the baseline system. Section V-C 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 [17] (MVT, ATAX, BICG, and GESUMMV), Rodinia [18] (NW, Back propagation, LUD, and Hotspot), and Pannotia [19] (SSSP and Color).

In this work, we focus on emerging GPU applications with irregular memory access patterns. These applications demonstrate memory access divergence [2], [3] that bottlenecks GPU's address translation mechanism [5]. However, not every application we studied demonstrates irregularity or suffers significantly from address translation overheads. We find that five workloads (MVT, ATX, NW, BIC, and GEV) demonstrate irregular memory access behavior while the remaining (SSP, LUD, CLR, BCK, 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 therefore focuses on applications

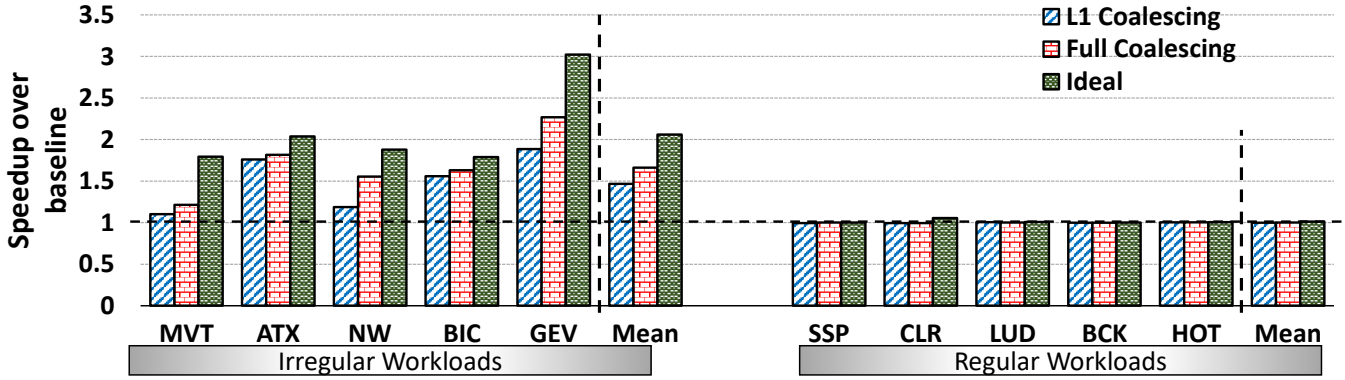

Fig. 9: Speedup with neighborhood-aware page table walk coalescing.

in the first category. However, 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

In our evaluation, we quantitatively answer the following questions: ① how coalescing of accesses to the page table can speed up applications? ② what are the factor(s) behind the speedup (or slowdown)? ③ how varying micro-architectural parameters impact the speedup (or slowdown)?

Performance analysis Figure 9 shows the speedup with neighborhood-aware coalescing of page table walks over the baseline. We divided ten applications into two groups – irregular and regular applications. Each application has three bars – the first bar (`L1 coalescing`) shows the speedup if only accesses to the `L1` (leaf) level of the page table are coalesced. The second bar shows the speedup if accesses to both the `L1` and upper-levels (`L2, L3, and L4`) are coalesced (`Full coalescing`). The third bar shows speedup achievable in an ideal system where address translations takes one cycle.

We make several observations from the Figure 9. First, full coalescing can speed up irregular applications by around 1.7×, on average. Application `GEV` speeds up by a significant 2.3×. Second, coalescing accesses to both leaf level and upper-levels of the page table are almost equally important. While a few applications benefit most from coalescing to leaf nodes (e.g., `ATX, BIC`) there are other applications (e.g.,`NW, GEV`) that benefit significantly from coalescing of accesses to the upper-level of the page table. However, the performance of most of the irregular applications are far from that of ideal address translation – indicating scope for further exploration on ways to reduce translation overheads. Applications with regular memory access patterns, however, do not suffer much overhead due to address translation as these applications observe very little speedup even with ideal address translation. However, our proposed coalescing mechanism does not hurt the performance of any of these applications. Therefore, we will focus only on irregular applications for the remainder of the evaluation.

We then analyze sources of improvement contributing to the speedup. Our neighborhood-aware coalescing of accesses to the page table reduces the number of memory accesses performed by the page table walkers. Figure 10 shows the

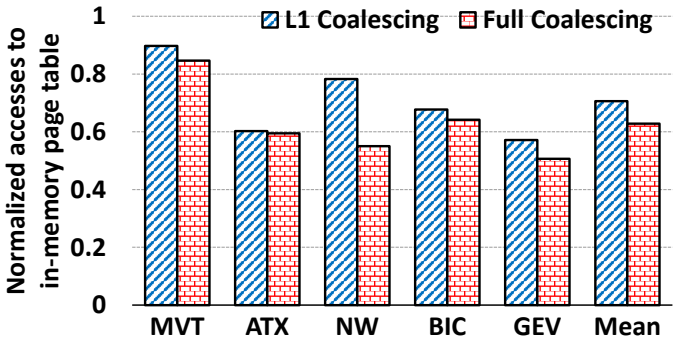

Fig. 10: Normalized number of accesses to the page table.

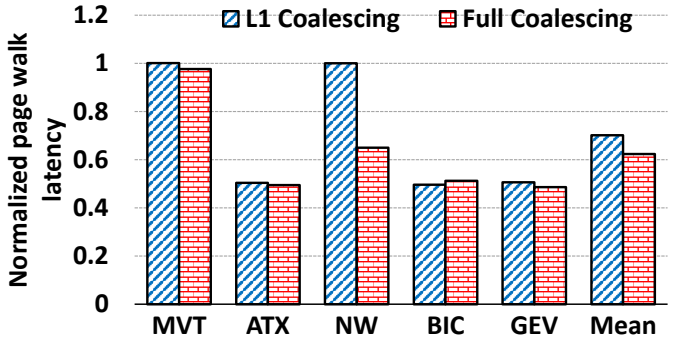

Fig. 11: Normalized average page walk latency with coalescing.

normalized number of memory accesses to the page table after coalescing (lower is better). Each irregular application has two bars corresponding to coalescing of accesses to only the L1 level and to all levels of the page table. The normalization is done over the memory accesses for page walks in the baseline. We observe that coalescing can significantly reduce the number of access to the in-memory page table – on average, full coalescing leads to about 37% reduction in page table access. We also observe that coalescing of accesses to both the leaf level and upper-levels of page table are almost equally important.

Figure 11 shows the normalized value of average page walk latencies for each application. This metric is a more direct measurement of how each page walk can leverage neighborhood information to reduce accesses to the page table. Like the previous figure, there are two bars for each application. We observe that page walk latency drops by close

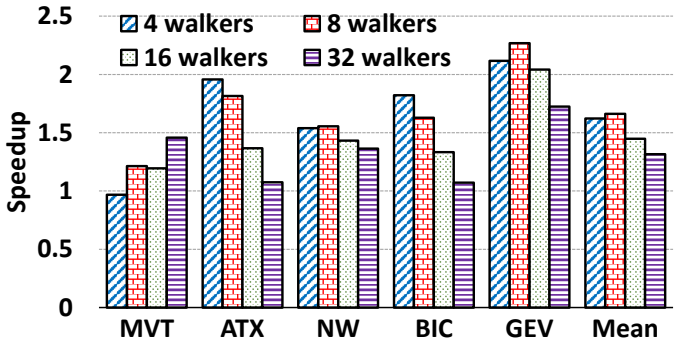

Fig. 12: Speedup of with varying number of page table walkers.

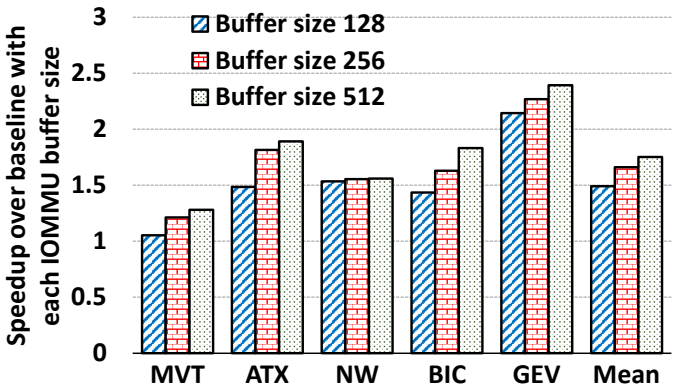

Fig. 13: Speedup of with varying IOMMU buffer size

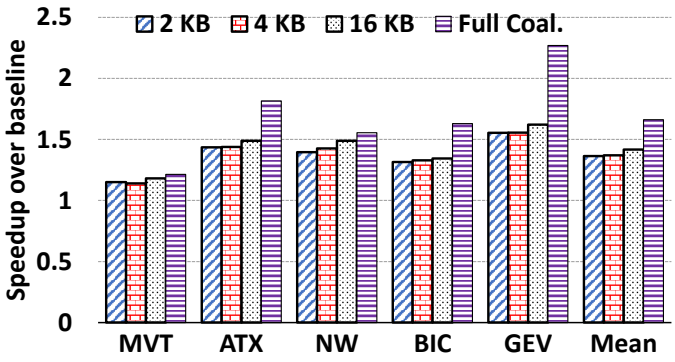

Fig. 14: Speedup comparison with cache for PTE.

to 38%. Correlating Figure 10 and Figure 11, we observe that a decrease in the number of accesses to the page table does not necessarily speed up individual page walks. This could happen since there are multiple concurrent page table walkers (eight in our baseline). The throughput of page walks could increase due to better utilization of walkers due to coalescing but without necessarily making each walk faster.

C. Sensitivity Analysis

In this section, we quantified the sensitivity of key architectural parameters towards coalescing performance.

First, we varied the number of independent page table walkers in the IOMMU. A larger number of walkers typically increases the effective of address translation bandwidth, but it could also increase congestion at the memory controller due to a larger number of concurrent page walking. We show the impact of changing the number of walkers from four to thirty-two in Figure 12. Each application has four bars where each bar represents speed up with full coalescing of page table accesses with a given number of walkers. The height of each bar is normalized to the baseline (no coalescing) for the given number of walkers. For example, the bar for `ATX` with eight page table walkers represents the speedup for `ATX` with full coalescing over the baseline; both run with eight page walkers.

We observe that speedups remain significant even with a large number of independent page table walkers (e.g., around $1.3\times$ on average even with 32 page table walkers). However, we observe that typically the improvement due to coalescing decreases with an increasing number of walkers. This is expected; a larger number of independent walkers reduces address translation overheads in the baseline by increasing the bandwidth of translation. `MVT` is an exception to this trend, though. A deeper investigation revealed an intriguing interaction between the number of page table walkers, the TLB and the coalescing of page table accesses. With the increasing number of independent walkers, the rate of completion of page walks, and consequently, the rate of allocation of new entries in the TLB goes up. This can sometimes lead to thrashing in the TLB, as is the case with `MVT`. This could allow more headroom to speed up application performance as the number of independent walkers is increased.

Next, Figure 13 shows speedups with full coalescing of accesses to page tables under different sizes of the IOMMU buffer. A larger buffer size allows coalescing of accesses to the page table across a larger number of page table walk requests. Consequently, we consistently observe a larger speedup with increasing buffer capacity.

Summary: ① We find that opportunistic coalescing of accesses to page table based on neighborhood analysis can speed up irregular GPU applications by $1.7\times$, on average. ② It is crucial to coalesce accesses to all levels of the page table. ③ Coalescing of page table accesses can reduce the number of memory accesses performed by the page table walkers by around 37%. ④ Our proposed coalescing mechanism continues to perform well under varying micro-architecture configurations like the number of page table walkers and buffer size.

VI. Discussion

Coalescing vs. Cache for PTEs: An alternative to coalescing could be to add a data cache at the IOMMU that keeps recently accessed cache lines containing PTEs. Like coalescing, such a cache could avoid memory accesses for page walks that falls in the same cache line available in the newly added cache. Figure 14 shows the speedup achieved by such a cache of varying size over the baseline. In the Figure, there are four bars for each application, corresponding to different cache sizes and our proposed full-page walk coalescing scheme. We observe that coalescing provides significantly higher speedup compared to the alternative design employing a data cache for lines containing PTEs. Note that our coalescing proposal adds only 1.5KB of additional state.

The key advantage of coalescing is that one hardware page table walker (PTW) can service walks for multiple requests (fully or partially). However, with caching, even if the required

PTE is available in the newly added IOMMU cache, a separate PTW still needs perform that walk. Often the unavailability of free PTWs leads to significant queuing delay in servicing walk requests. Coalescing reduces this queuing delay by involving fewer PTWs to perform a given number of walks, something that a cache based design cannot achieve.

Area and Power: Overall, an additional 1.5KB CAM is required for our solutions. In our analysis using FabScalar [20], we find that the design requires $0.048mm^2$ of area, and it consumes $0.041nj$ and $0.196nj$ of energy per read and write, respectively. This is about $40-200\times$ less energy consumption than DRAM read and write operations [21]. Overall, the coalescing design saves dynamic energy since it replaces costly accesses to DRAM-resident page tables with much cheaper accesses to our CAM structure. Table III shows the energy savings in page walks due to coalescing after accounting for the additional energy consumed by our hardware. We found that coalescing can reduce $103.51mW$ of power on average.

TABLE III: Saved Power.

Benchmark	MVT	ATX	NW	BIC	GEV	Mean
Power(mW)	27.69	36.86	358.19	24.81	70.00	103.51

VII. Related work

Efficient virtual-to-physical address translation for GPUs is a critical design requirement. Lowe-Power et al. [22] and Pichai et al. [23] were among the first to explore designs for a GPU's MMU. Lowe-power et al. demonstrated that a coalescer, a 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 TLB-aware. Phichai et al. also observed that multiple entries in the upper-level page table can reside in a single cache line and exploited for better page table walk scheduling. We however, opportunistically coalesce accesses to any levels of the page table that fall in same neighborhood.

Vesely et al. demonstrated that a GPU's translation latencies are much longer than that of a CPU's, and that GPU applications with memory access divergence may bottleneck due to address translation overheads, on real hardware [5]. Cong et al. proposed TLB hierarchy that is similar to our baseline but additionally proposed to use a CPU's page table walkers for GPUs [24]. However, accessing CPU page table walkers from a GPU may not be feasible in real hardware due to long latencies. Lee et al. proposed a software managed virtual memory to provide the illusion of large memory by partitioning GPU programs to fit into physical memory space [25]. Ausavarungnirun et al. showed that address translation overheads could be even larger in the presence of multiple concurrent applications on a GPU [26]. They selectively bypassed TLBs to avoid thrashing and prioritized address translation over data access to reduce overheads. Yoon et al. proposed the use of a virtual cache hierarchy in the GPU to defer address translation till a miss in the entire GPU cache hierarchy [27]. This approach could remove address translation from the critical path, but makes it harder for executing GPU kernels from multiple different process address spaces. Another recent work by Haria et al. proposed to leverage identity mapping between virtual and physical memory in accelerators to avoid overheads of walking hierarchical page tables [28]. This approach, however, requires re-defining the software-hardware interface for virtual-to-physical address mapping.

Different from most of these works, we leverage the layout of page tables and the fact that many concurrent page walks map to neighboring entries in the page tables. Our approach does not require any software or OS modification. It is orthogonal to any technique that improves the efficacy of TLBs. Many of these works are either already part of our baseline (e.g., [22]) or are mostly orthogonal to ours (e.g., [26]).

CPU's virtual memory overheads and the techniques employed to reduce them are well studied. To exploit address localities among threads, Bhattacharjee et al. proposed inter-core cooperative TLB prefetchers [29]. Pham et al. proposed to exploit naturally occurring contiguity to extend the effective reach of TLBs by enabling a single TLB entry to map multiple pages [7]. Bhattacharjee later proposed shared PWCs and efficient page table designs to increase PWC hits [12]. Cox et al. have proposed `MIX TLBs` that support different page sizes in a single structure [30]. Barr et al. proposed `SpecTLB` that speculatively predicts virtual-to-physical memory mappings to hide the TLB miss latency. Several others proposed to leverage segments to selectively bypass TLBs and avoid most TLB misses [31], [32], [33]. While some of these techniques can be extended to GPUs, most of them require OS or software changes or are orthogonal to the neighborhood-aware address translation we proposed in this work.

VIII. Conclusion

We make an important observation that during page table walking a hardware page table walker reads the in-memory page tables at the granularity of a cache line. A cache line typically contains eight page table entries. Therefore, page table entries for neighboring virtual page address are contained in the same cache line. We leverage these two observations to effectively coalesce accesses to the page table for concurrent page walk requests. We then extend the same basic mechanism to coalesce accesses to the upper levels of a hierarchical page table by observing that multiple upper-level page table entries are also brought together by a page table walker in a single cache line. Taken together, our proposed enhancement speeds up irregular GPU applications by $1.7\times$ on average, and by up to $2.3\times$ in the best case.

Acknowledgment

AMD, the AMD Arrow logo, and combinations thereof are trademarks of Advanced Micro Devices, Inc. This research was partially supported by the US Department of Energy under the PathForward program. Any opinions, findings, conclusions or recommendations expressed herein are those of the authors and do not necessarily reflect the views of the Department of Energy or other sponsors.

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.

[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. [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. [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.

[6] Intel, "Intel 64 and IA-32 Architectures Software Developers Manual: System Programming guide," 2016, https://www.intel.in/content/www/in/en/architecture-and-technology/64-ia-32-architectures-software-developer-vol-3b-part-2-manual.html.

[7] 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.

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

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

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

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

[12] 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. [Online]. Available: http://doi.acm.org/10.1145/2540708.2540741

[13] 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. [Online]. Available: http://doi.acm.org/10.1145/1815961.1815970

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

[15] Khronos Group, "OpenCL," 2014, https://www.khronos.org/opengl/.

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

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

[18] 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. [Online]. Available: https://doi.org/10.1109/IISWC.2009.5306797

[19] 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.

[20] N. K. Choudhary, S. V. Wadhavkar, T. A. Shah, H. Mayukh, J. Gandhi, B. H. Dwiel, S. Navada, H. H. Najaf-abadi, and E. Rotenberg, "FabScalar: Composing synthesizable RTL designs of arbitrary cores within a canonical superscalar template," in *Proceedings of the 38th Annual International Symposium on Computer Architecture*, ser. ISCA '11. New York, NY, USA: ACM, 2011. [Online]. Available: http://doi.acm.org/10.1145/2000064.2000067

[21] S. Volos, J. Picorel, B. Falsafi, and B. Grot, "BuMP: Bulk memory access prediction and streaming," in *2014 47th Annual IEEE/ACM International Symposium on Microarchitecture*, Dec 2014.

[22] 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.

[23] 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. [Online]. Available: http://doi.acm.org/10.1145/2541940.2541942

[24] 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.

[25] 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. [Online]. Available: http://doi.acm.org/10.1145/2628071.2628075

[26] 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. [Online]. Available: http://doi.acm.org/10.1145/3173162.3173169

[27] 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. [Online]. Available: http://doi.acm.org/10.1145/3173162.3173195

[28] S. Haria, M. D. Hill, and M. M. Swift, "Devirtualizing memory in heterogeneous systems," 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. [Online]. Available: http://doi.acm.org/10.1145/3173162.3173194

[29] 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. [Online]. Available: http://doi.acm.org/10.1145/1736020.1736060

[30] 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. [Online]. Available: http://doi.acm.org/10.1145/3037697.3037704

[31] 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. [Online]. Available: http://doi.acm.org/10.1145/2485922.2485943

[32] 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. [Online]. Available: http://dx.doi.org/10.1109/MICRO.2014.37

[33] 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. [Online]. Available: http://doi.acm.org/10.1145/2749469.2749471