Increasing GPU Translation Reach by Leveraging Under-Utilized On-Chip Resources

Jagadish B. Kotra
AMD Research
USA
Jagadish.Kotra@amd.com

Michael LeBeane[*]
AMD Research
USA
mlebeane@gmail.com

Mahmut T. Kandemir
Pennsylvania State University
USA
kandemir@cse.psu.edu

Gabriel H. Loh
AMD Research
USA
Gabriel.Loh@amd.com

ABSTRACT

Many GPU applications issue irregular memory accesses to a very large memory footprint. We confirm observations from prior work that these irregular access patterns are severely bottlenecked by insufficient Translation Lookaside Buffer (TLB) reach, resulting in expensive page table walks. In this work, we investigate mechanisms to improve TLB reach without increasing the page size or the size of the TLB itself. Our work is based around the observation that a GPU's instruction cache (I-cache) and Local Data Share (LDS) scratchpad memory are under-utilized in many applications, including those that suffer from poor TLB reach. We leverage this to opportunistically utilize idle capacity and port bandwidth from the GPU's I-cache and LDS structures for address translations. We explore various potential architectural designs for each structure to optimize performance and minimize complexity. Both structures are organized as a victim cache between the L1 and L2 TLBs to boost translation reach. We find that our designs can increase performance on average by 30.1% without impacting the performance of applications that do not require additional reach.

CCS CONCEPTS

• **Computer systems organization → Heterogeneous (hybrid) systems**; **Single instruction, multiple data (GPU)**; **Virtual Memory**; **Reconfigurable computing**.

KEYWORDS

CPU+GPU Systems, Virtual Memory, Irregular Applications, Reconfigurable Systems

ACM Reference Format:
Jagadish B. Kotra, Michael LeBeane, Mahmut T. Kandemir, and Gabriel H. Loh. 2021. Increasing GPU Translation Reach by Leveraging Under-Utilized On-Chip Resources. In *MICRO '21: IEEE/ACM International Symposium on Microarchitecture, October 18–22, 2021, Virtual Event, Greece.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3466752.3480105

[*]This work was done while author was at AMD Research.

1 INTRODUCTION

GPUs are increasingly important in many domains due to their superior performance-per-watt for data parallel workloads [33]. As GPUs become more popular, their feature sets continue to improve. Today's GPUs provide many architectural and programmability-enhancing features that have typically been taken for granted on the CPU for decades. One such area is support for virtual memory. Virtual memory support on GPUs significantly eases the relatively high burden of programming and allows for easier implementation of unified virtual memory between the host and GPU.

While virtual memory is essential for many programming abstractions, it comes at a cost. Recent work has identified performance issues in a GPU's virtual memory subsystem [23, 47]. This is especially true for *irregular* applications that access a very large virtual memory footprint in an unstructured manner. There are many reasons that irregular applications impose a large translation overhead [1, 43, 44]. We observe that the biggest factor is that the TLB reach is simply inadequate to accommodate irregular accesses to a large address space, which leads to many performance-degrading page table walks [47]. While the TLB capacity is under intense capacity pressure, in this work we observe that a number of other on-chip SRAM structures are frequently underutilized. We also observe that this under-utilization can significantly vary on a kernel-by-kernel basis in an application's execution.

We use these observations to propose new mechanisms that take advantage of underutilized on-chip resources to improve the effective TLB capacity and reach for applications that demand it. The key principle of our design is to dynamically and opportunistically reallocate resources from idle on-chip SRAM structures to store additional TLB entries. Our contributions are:

- Based on real GPU measurements, we show that I-cache and LDS scratchpad in a GPU are under-utilized in terms of capacity and bandwidth and the under-utilization varies between kernels, even within the same application.
- We propose a new mechanism that effectively constructs an auxiliary victim cache from idle I-cache and/or LDS entries to opportunistically store additional translations.

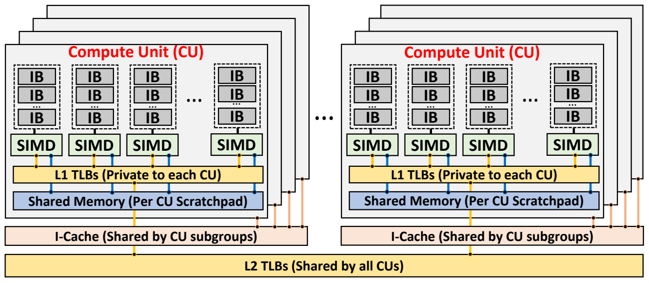

Compute Unit (CU)
IB
SIMD
L1 TLBs (Private to each CU)
Shared Memory (Per CU Scratchpad)
I-Cache (Shared by CU subgroups)
L2 TLBs (Shared by all CUs)

Figure 1: Baseline GPU architecture derived from AMD's Graphics Core Next (GCN) design [2].

- We architect designs to accommodate the unique characteristics of both the I-cache and LDS, focusing on trade-offs in implementation complexity, area overhead, and performance. For the I-cache, we explore naive and instruction-aware replacement policies as well as different packing schemes to store and decode translations in a single way of the I-cache. We also explore various designs to pack translations and tags in a monolithic LDS structure that was not originally designed as a hardware cache.
- We perform a detailed performance evaluation of our design on a CPU/GPU-based APU system. Our evaluations show that for GPU applications that suffer from insufficient TLB reach our micro-architectural solution can improve the performance by 30.1%, while not negatively impacting applications that do not require additional TLB reach.

2 BACKGROUND

Figure 1 shows the architecture for a compute optimized GPU, comprised of Compute Units (CUs), also known as Streaming Multiprocessors (SMs), each providing a collection of Single Instruction Multiple Data (SIMD) units. Groups of work-items (or threads) are dispatched on the CUs in bundles of either 32 or 64 known as wavefronts (or warps). These wavefronts are further organized into work-groups (or thread-blocks) that are guaranteed to execute on the same CU.

2.1 GPU Address Translation

Modern GPUs possess a virtual memory (VM) subsystem similar to CPUs. GPUs also employ a multi-level TLB hierarchy that caches recently-used address translations to avoid accessing page-tables for every memory (load/store) request. Each CU has a private L1 TLB shared by the SIMDs on that CU. A SIMD unit's memory accesses targeting the same page are coalesced by the hardware to minimize the number of accesses to the L1 TLB. Misses from the L1 TLBs are typically serviced by an L2 TLB that is shared across the entire GPU [37, 38]. L2 TLB misses are either handled using local GPU page-table walkers or through an IO Memory Management Unit (IOMMU) [43, 44].

2.2 Local Data Share (LDS)

Work-groups are guaranteed to execute on the same CU and can therefore make use of a per-CU, application-managed scratchpad called the LDS. Typically, the LDS is organized as 16 or 32 banks, each of which can produce a 4-byte value every cycle [2]. AMD

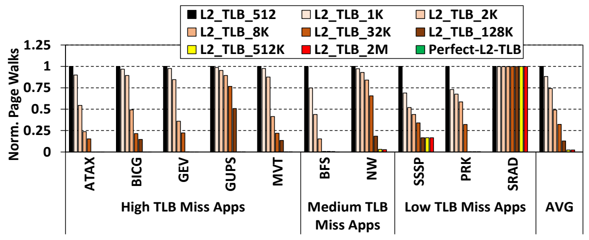


Figure 2: Page walks vs. L2 TLB capacity.

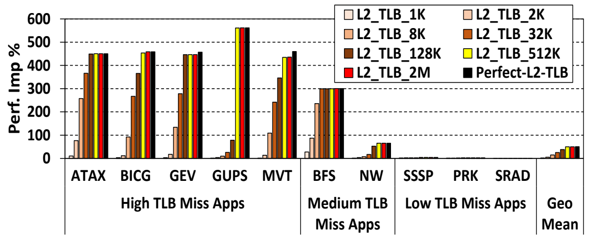


Figure 3: Performance improvement with additional L2 TLB capacity vs a 512 entry baseline.

GPUs contain 64KB LDS per CU, while NVIDIA's LDS equivalent component (called shared-memory) are similarly organized and vary in size from 64KB to 96KB per SM [35]. The LDS can provide more bandwidth and low latency compared to global memory, but comes at the cost of being managed by the application programmer.

Resources in LDS are statically reserved by the front-end scheduling unit before waves in a work-group are dispatched to the CUs. LDS allocators reserve LDS capacity in one contiguous block to satisfy the launch requirements for an entire work-group. This policy can lead to significant fragmentation and under-utilization if work-groups from different kernels with different LDS requirements are co-resident on the same CU.

2.3 I-Cache and Instruction Buffers

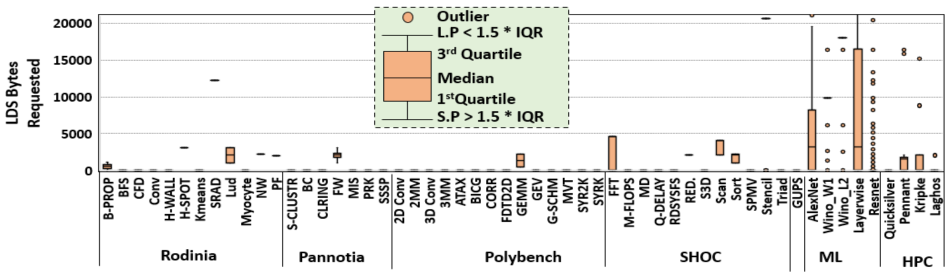
Each CU contains wavefronts that are statically assigned to SIMD units. As an example, the AMD GCN architecture supports four SIMD units that each can contain ten wavefronts [2], but these numbers can vary across vendors and between micro-architectural generations. Regardless, each wavefront has a Program Counter (PC) to hold the address of the next instruction to fetch as well as an instruction buffer (IB) that holds cache lines each containing multiple instructions.

A wavefront that cannot service the next instruction from its local IB requests access to the fetch unit. The fetch units (and their corresponding instruction caches) are shared by groups of CUs. The fetch units arbitrate requests and issue them to the L1 I-cache to fill the instruction buffer for a requesting wavefront. Misses from the I-cache are serviced from an L2 cache shared across the GPU.

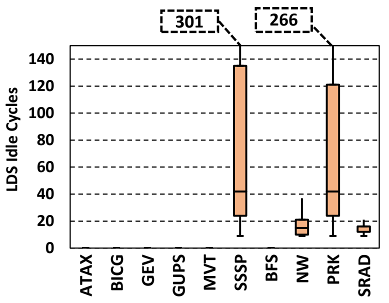
3 MOTIVATION

3.1 GPU TLB Reach Limitations

Recent work has shown that TLB reach (i.e., the amount of virtual address space that can be translated by the TLBs without incurring a page walk) can be a significant performance issue for GPU applications [8, 23]. Not only is the working set size and physical

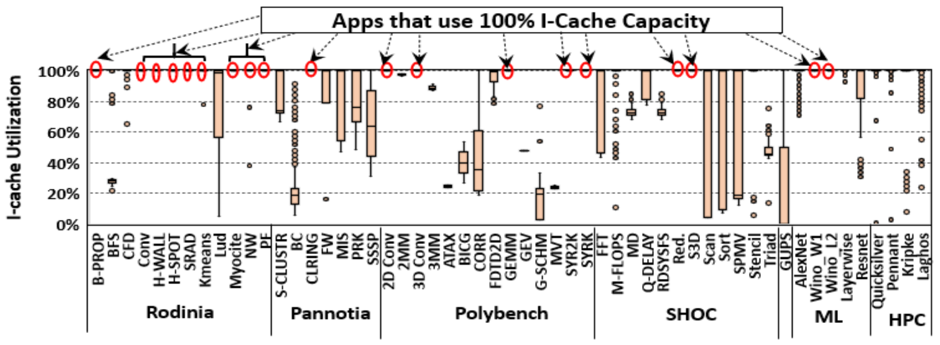


(a)

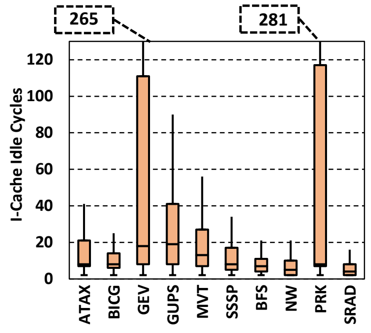


(b)

Figure 4: (a) LDS size in bytes requested by a Workgroup across benchmark suites.(IQR: Interquartile Range, L.P: Largest Point, S.P: Smallest Point), (b) Idle cycles at each LDS port. The results in Figure 4(a) are collected on a real system using performance counters while the ones in Figure 4(b) are from simulator. Please note that the Y-axis in Figure 4(a) maxes out at around 20KB, while the LDS capacity is 64KB.



(a)



(b)

Figure 5: (a) I-cache utilization across benchmark suites, (b) Idle cycles at each I-cache port. The results in Figure 5(a) are collected on a real system using performance counters while the ones in Figure 5(b) are from simulator.

memory capacity of a single GPU increasing, but new programming paradigms directly map remote GPU memory to the address space of each GPU to allow efficient data exchanges in multi-GPU workloads [34]. Additionally, host memory is mapped to a GPU's address space to allow for direct, zero-copy access from the GPU.

TLB reach can be especially problematic for a range of emerging GPU applications that exhibit patterns such as pointer chasing and random memory accesses, which generate irregular memory access patterns [43, 44]. Sporadic accesses to a large virtual memory footprint tend to miss in all levels of the TLB, generating a large number of page table walks. While frequent TLB misses negatively impact performance on any architecture, GPUs are especially sensitive. Recent work has shown that servicing TLB misses on GPUs can be an order of magnitude slower compared to CPUs [31, 47]. Also, because the Single Instruction, Multiple Thread (SIMT) execution paradigm executes threads in lockstep, a single wavefront might have to wait for many page table walks to resolve before the wavefront can make forward progress. In the worst case, each of the 64 (or 32 on depending on the architecture) threads in a wavefront can access a separate page and generate a unique page-table walk on a TLB miss.

One simple way to improve TLB reach is to increase the size of the TLBs. To quantify the performance potential of larger TLBs, we performed a study using several irregular GPU applications [14, 15, 27] by varying the size of the L2 TLB from 512 to 2M entries for a standard 4KB page size and observing the performance impact. Please refer to Section 5 for information on simulator setup and benchmarks. Additionally, we include an upper-bound Perfect-L2-TLB configuration where translations always hit in L2 TLB. The Perfect-L2-TLB configuration incurs zero page walks and hence delivers the best case performance.

Figure 2 shows how larger L2 TLBs impact the number of page table walks requested by an application. At the largest L2 TLB size considered, the number of page-table walks decreases by ~85% on average compared to a 512-entry baseline, with *SRAD, PRK and SSSP* not benefiting from the additional entries. SRAD does not see any degradation with increasing L2 TLB size as the number of page walks in the baseline is ~0.

Figure 3 shows the impact of larger L2 TLBs on relative performance over a baseline with 512 TLB entries. On average, increasing the L2 TLB size from 512 to 8K entries results in a geometric mean improvement in performance of 14.7%, and increasing the L2 TLB size to 2M entries can improve performance by up to 50.1%. We also observe that *ATAX*, *BICG*, *GEV*, *GUPS*, *MVT BFS*, and *NW* are particularly bottlenecked by TLB reach while *SRAD, PRK* and *SSSP* are not. These results show that TLB capacity is a significant performance bottleneck for some GPU applications.

3.2 Inefficient Utilization of GPU Structures

In addition to the TLBs, GPUs also dedicate significant area to other structures such as register files, LDS, and instruction and data caches. In this work, we observe that a number of GPU SRAM structures are sized *conservatively* based on the behavior of a few applications. In this section, we explore the usage characteristics of two such structures: LDS and the I-cache. Our results in this section are collected on a system with a single AMD Radeon^{TM} RX 580 GPU [4] using roc-profiler [5] to collect performance counter data. For this real-system study, we used a diverse set of benchmarks from various suites including: Rodinia [15], Pannotia [14], Polybench [27], and SHOC [17], machine learning applications [18], and high-performance computing proxy applications [11, 28–30]. The LDS capacity requested by each kernel during a kernel invocation as well as relevant performance counters (e.g., I-cache misses, prefetches) are collected at kernel granularity. I-cache utilization is calculated using Equation 1. Because I-cache misses and I-cache prefetches are the only events that result in an I-cache fill, they help us to conservatively estimate the number of I-cache cachelines filled with instructions. In scenarios where I-cache misses and prefetches are more than the total number of cache lines, the utilization is considered to be 100%.

$$\%(Icache\_Utilization) = \frac{(IC\_Misses + IC\_prefetches) * 100}{(Num.\ of\ IC\ lines)}. \quad (1)$$

Figure 4a shows the total utilization of on-chip LDS. The distribution in the box-and-whisker graph was created by sampling across all kernel launches in an application. We observe that out of the 54 total applications evaluated, 38 (or ~70%) do *not* use LDS at all. For all the applications that use LDS, none of them are able to leverage the full 64KB of space that is available per CU with their current work-group sizes. While it is reasonable to suspect that some classes of applications make use of all LDS (such as graphics workloads), our survey of compute applications indicate that these structures are significantly under-utilized.

Figure 5a makes a similar observation with the I-cache. This figure plots a distribution of I-cache utilization sampling across all kernels in a workload. I-cache utilization is computed as shown in Equation 1. In our results, we observe a wide mix of I-cache utilization across workloads. While not as consistently under-utilized as LDS, only 17 (or ~24%) of the workloads make use of the full I-cache capacity for every kernel. The remaining workloads can be broken down into two categories; 13 workloads never use the full I-cache capacity throughout their entire execution, and 24 use the full I-cache capacity only during some kernel launches.

Additionally, we observe that not only is there space available in the I-cache and LDS, but there is also bandwidth available at the ports. Figure 4b shows box-and-whisker graphs sampled whenever the ports to LDS and I-cache are accessed by the GPU wavefronts, respectively. For the I-cache, most applications have ~10-20 idle cycles between subsequent accesses, with significant upward skew indicating some periods of even more sporadic utilization. I-cache ports are not often utilized because most instructions can be serviced directly from a wavefront's instruction buffer. For the four applications that use LDS, we observe a similar trend of several tens of cycles between LDS accesses with significant upward skew. Most importantly, we observe that the applications that could significantly benefit from improved TLB reach, as discussed in Section 3.1, (i.e., *ATAX*, *BICG*, *GUPS*, *GEV*, *BFS*, and *MVT*) have idle resources in *both* LDS and I-cache in terms of capacity and port bandwidth.

3.3 Why Not Larger TLBs?

Emerging workloads are increasing demands on TLB coverage. Responding to this demand, recent commercial GPUs [6, 7, 24] have increased L2 TLB coverage over previous generations. However, the die sizes of high-performance GPUs are already nearing the lithographic reticle limit, and so blindly increasing die size via larger TLBs would be very challenging. The larger structures would also increase the difficulty of meeting circuit timing and also increases power consumption.

4 RECONFIGURABLE ARCHITECTURE

4.1 Why I-cache and LDS?

Our main goal is to improve performance for translation-sensitive workloads while not negatively impacting TLB-insensitive workloads, all while incurring minimal area overhead. To that end, the I-cache and LDS are good potential targets as the data cached in these structures do not need to track modifications with a 'dirty' bit, similar to TLB translations. This enables translations stored in I-cache and LDS to be evicted without additional writebacks.

We modify the I-cache and LDS to act as a TLB victim cache (as opposed to a prefetch buffer because the access patterns of irregular applications are hard to predict). Additionally, there is a synergy between LDS and virtual memory overheads. That is, since the LDS is looked up using virtual address, if an application uses a lot of LDS space, it will tend not to incur address translations and hence have lower TLB usage. However, if the application does not use LDS much, the spare LDS capacity can be leveraged for translations.

4.2 Reconfigurable Shared Memory (LDS)

LDS is an application-managed hardware structure (Section 2.2). The main challenge in caching address translations in LDS is that, unlike caches, the LDS does not contain separate tag and data arrays. Therefore, the translation tags that aid in detecting a hit or miss must also be stored in the LDS itself. The challenges in architecting LDS to cache translations include:

- How to identify the idle LDS capacity?
- Where to store translation tags within LDS?
- How to effectively support associativity in translations as the spare LDS capacity varies over time?
- How to efficiently enable overwriting translation tags and data in case a work-group requests LDS capacity without any extra data movement?
- How to efficiently lookup and retrieve translation tags and data from LDS without wasting LDS bandwidth?

4.2.1 Identifying LDS Idle Capacity. As mentioned in Section 2, LDS allocations are managed by the work-group scheduling unit. LDS memory is assigned in contiguous chunks to a work-group by writing a register that points to the base of a work-group's LDS allocation for each dispatched wavefront. When a work-group completes, its entire allocation is returned to the work-group scheduler for re-assignment to subsequent work-groups.

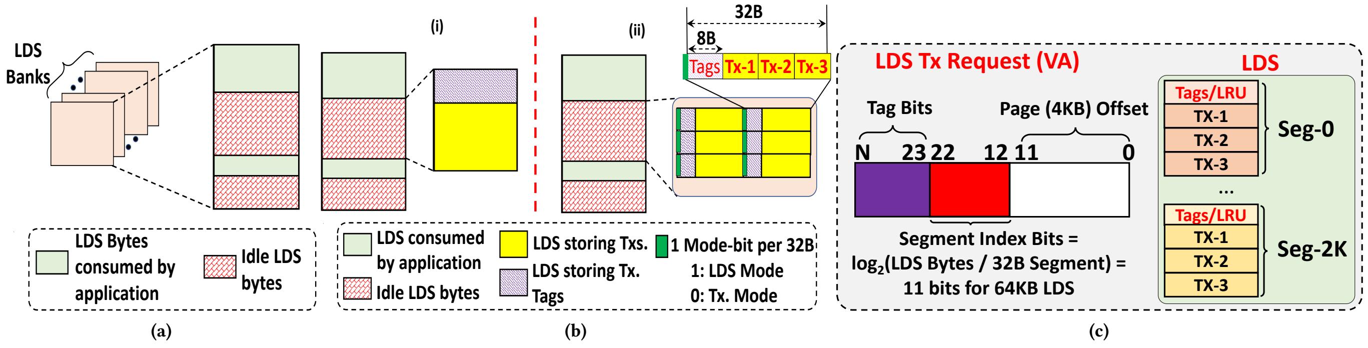


Figure 6: (a) Baseline LDS, (b) Different Reconfigurable LDS Designs, and (c) Reconfigurable LDS set indexing.

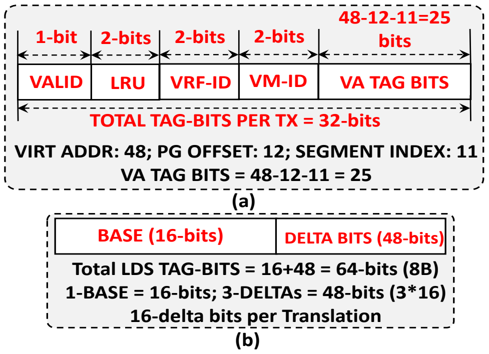

1-bit
2-bits
2-bits
2-bits
48-12-11=25 bits
VALID
LRU
VRF-ID
VM-ID
VA TAG BITS
TOTAL TAG-BITS PER TX = 32-bits
VIRT ADDR: 48; PG OFFSET: 12; SEGMENT INDEX: 11
VA TAG BITS = 48-12-11 = 25
(a)
BASE (16-bits)
DELTA BITS (48-bits)
Total LDS TAG-BITS = 16+48 = 64-bits (8B)
1-BASE = 16-bits; 3-DELTAs = 48-bits (3*16)
16-delta bits per Translation
(b)

Figure 7: (a) Per Translation Tag bits in LDS. (b) Base-delta Compressed LDS Tag-bits.

Figure 6a shows a baseline LDS system with some unused capacity. As shown in this example, the LDS baseline system is severely fragmented (see Section 2.2).

4.2.2 Isolated Translation Tags and Data. Motivated by traditional caches that isolate the tag and data arrays into two different structures, we explored dividing idle LDS capacity into tag space and data space as shown in Figure 6b-(i). In such a design, the tags need to be read out of the LDS first, and then a tag match determines whether translation data needs to be accessed. This isolated tag and data design poses several challenges especially when a new work-group is launched that requests additional LDS capacity. This design does not enable efficiently overwriting the translations. For example, in the case that the LDS capacity requested by a new work-group fits the entire translation tag capacity depicted by pattern in Figure 6b-(i), overwriting the tag space would leave the translation data orphaned and useless. In such a design, to efficiently utilize idle capacity for translations, before allocating LDS capacity to a requesting work-group, the tag and the corresponding data should be readjusted, thereby consuming additional LDS bandwidth. Hence, an isolated tx-tag and data design is not desirable as it requires additional data movement before satisfying a work-group's LDS allocation requests.

4.2.3 Co-located Translation Tags and Data. Figure 6b-(ii) presents an alternative design that co-locates translation tags and data together. The LDS is divided into smaller 32-byte segments with co-located tag and data. Figure 6b-(ii) depicts a 32-byte LDS segment storing three (eight-byte) address translations consuming 24 bytes for translation data and the remaining 8 bytes for the corresponding tags. In such a co-located design, accessing one 32-byte segment can enable streaming both translation tags and data together similar to previous DRAM cache proposals [41], allowing the immediate use of a corresponding address translation upon a tag match. The co-location design seamlessly enables overwriting tags and data when a new work-group requests additional LDS capacity without requiring remapping of tags or data.

4.2.4 Reconfigurable LDS Operation. To identify whether a segment of LDS is managed by the application or LDS controller (for translations), each LDS segment of 32-bytes requires an additional mode-bit. The bit represents whether the 32-byte segment operates in default LDS mode or Translation (Tx)-mode as shown in Figure 6b. The LDS-mode segments are managed by applications, while Tx-mode segments are managed by the LDS controller. The segment bit is set depending on who initially loads data into the segment (i.e., regular SIMD data accesses set the segment bit to LDS-mode, and translation insertions set the segment bit to Tx-mode). Hence, in an LDS using 32-byte segments, mode-bits account for a 0.4% LDS area overhead per CU (a total of 1-bit for every 32×8 = 256 bits). To preserve the primary LDS functionality as explained in the design exploration subsections above, an LDS-mode segment can overwrite a Tx-mode segment to change it back to LDS-mode, but a Tx-mode segment can never overwrite an LDS-mode segment.

Figure 6c shows how an LDS controller maps the virtual address translation request using a 4KB-based virtual page number (VPN) to a 32-byte LDS segment. An LDS structure composed of 32-byte segments (each containing 8 bytes of tags and 3×8 bytes of translations) can be architecturally considered as a 3-way set-associative structure. The LRU information for the three translation ways is maintained in the tags. Figure 7 shows how virtual address tag bits from 3 address translations are compressed using Base-Delta compression [46] in our reconfigurable LDS design. Each address translation tag, as depicted in Figure 7(a), contains a 48-bit virtual address [39] resulting in 25 VA tag-bits after removing the page offset and LDS segment index bits. Additionally, GPUs contains a 2-bit VM-ID that acts as an Address Space Identifier. Single Root I/O Virtualization (SR-IOV) enabled GPUs also contain a 2-bit VRF-ID [45] that points to the virtual function identifer when GPUs are virtualized for multiple users. Finally, there are two LRU-bits covering the 3-ways and one valid bit. Hence totally, each address translation in LDS contains 25 + 2 + 2 + 2 + 1 = 32-bits as shown

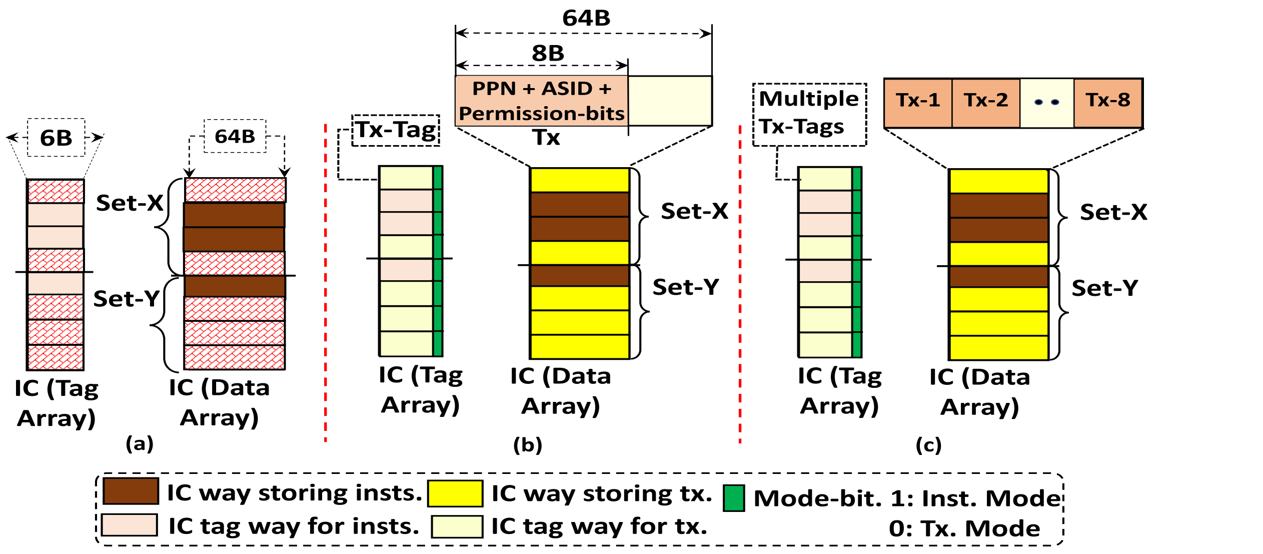


Figure 8: (a) Baseline I-cache and (b) I-cache ways storing one translation per way in Tx-mode, and (c) I-cache ways storing eight translations per way in Tx-mode.

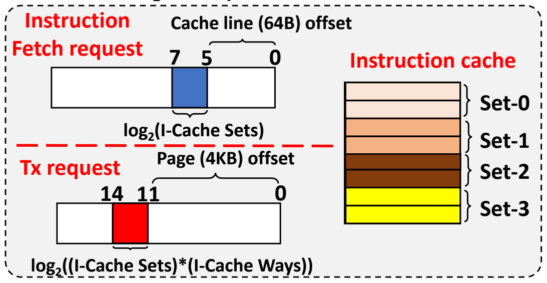


Figure 9: Reconfigurable I-cache set indexing.

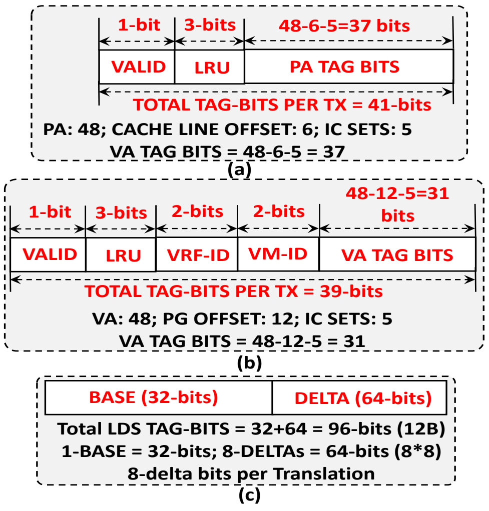

1-bit
3-bits
48-6-5=37 bits
VALID
LRU
PA TAG BITS
TOTAL TAG-BITS PER TX = 41-bits
PA: 48; CACHE LINE OFFSET: 6; IC SETS: 5
VA TAG BITS = 48-6-5 = 37
(a)
1-bit
3-bits
2-bits
2-bits
48-12-5=31 bits
VALID
LRU
VRF-ID
VM-ID
VA TAG BITS
TOTAL TAG-BITS PER TX = 39-bits
VA: 48; PG OFFSET: 12; IC SETS: 5
VA TAG BITS = 48-12-5 = 31
(b)
BASE (32-bits)
DELTA (64-bits)
Total LDS TAG-BITS = 32+64 = 96-bits (12B)
1-BASE = 32-bits; 8-DELTAs = 64-bits (8*8)
8-delta bits per Translation
(c)

Figure 10: (a) Instruction-mode I-cache Tag, (b) Tx-mode Tag in a 16KB I-cache, and, (c) Base-Delta Compressed Tx-mode Tags in I-cache.

in Figure 7(a). Hence to compress 3×32 = 96-bits from 3 address translation tag bits into a 64-bit (8B) LDS segment, we compress the tag-bits using 16-bit base and a total of 48-bits delta (Figure 7(b)).

4.3 Reconfigurable Instruction Cache

4.3.1 Reconfigurable I-cache Design. Figure 8(a) shows an example instruction cache where certain (64-byte) cache lines contain instructions while other lines (indicated by hashed patterning) are idle. Unlike the LDS, the instruction cache has all the hardware support required for normal cache operation, thereby requiring fewer changes. Figure 8(a) shows a basic reconfigurable I-cache design where each cache way stores one address translation. Similar to the reconfigurable LDS design, a reconfigurable I-cache design requires identifying whether a line stores instructions (IC-mode) or translations (Tx-mode). For that, each tag is augmented with a mode-bit. Although the basic design of storing one translation per I-cache way (Figure 8(b)) allows us to leverage I-cache idle space for translations without significant changes to the tag array, this design is inefficient as an I-cache line is 64 bytes and a translation occupies only 8 bytes, wasting the remaining 56 bytes. From our experimental results presented later in Section 6.1.2, this naive design hardly affects performance. This is because in the best case, a 16KB I-cache enables storing 256 (8-byte) address translations, which does not significantly improve TLB coverage for the large working sets of our irregular applications. To that end, we propose packing multiple translations per way. That is, eight (8-byte) translations are stored in a 64-byte I-cache line as depicted in Figure 8(c). Such a design increases the number of translations stored in an idle I-cache way, thereby improving the translation hit rate. However, this design requires widening the tag space of every way to accommodate additional tags for the multiple address translations stored per cache line. To understand the storage overhead incurred in storing multiple translation tags in I-cache, we should initially discuss the tag-bits required for storing a single instruction tag and translation tag in I-cache. Figure 10 shows the required tag-bits for storing an instruction vs eight translations in an I-cache. Each translation stored in an I-cache way would require a total of 39-bits (Figure 10(b)). Hence to store tags of all the 8-translations we first widen

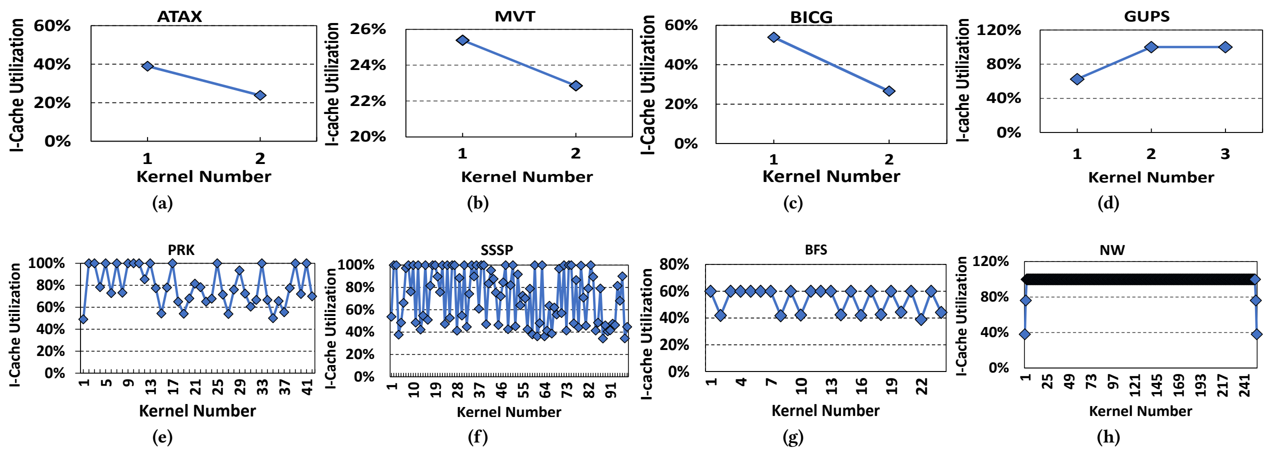


Figure 11: I-Cache utilization across kernels over time. GEV and SRAD have only one kernel each and hence not shown here. SSSP graph shows only a portion of the executed kernels as the pattern is similar across ∼10K kernels.

the tags of each I-cache from 6-Bytes to 12-Bytes. This incurs an additional overhead of $6 \times 8 \times 256 = 1.5$KB. To fit the 8-tags we employ Base-Delta compression on I-cache tags using 32-base and total of 64-delta bits as shown in 10(c). Hence our mechanism includes a total of 1.5KB per I-cache area overhead required for widening the I-cache tags. Finally, another design decision that plays a crucial role in the area overhead is the associativity of translations that must be supported. An I-cache already has some associativity; for example, an 8-way I-cache already has eight comparators for parallel instruction tag match. Now with each way storing eight more translations in Tx-mode, for lower I-cache lookup latency, in the worst case a possible (8-ways $\times$ 8-translations per way) 64 tag comparisons are required. This requires an additional 56 comparators. Also, additional logic that only considers the I-cache ways in Tx-mode for tag comparison has to be in-place. To avoid the need for all of this extra logic, we propose accessing the translations (for lookup and fill) as if the I-cache were a direct-mapped cache. This limits reusing the comparators already in-place for comparing translation tags without requiring additional comparators. This also limits the associativity of translations in the I-cache to eight. That is, a translation can only replace another translation in the way it is mapped to. Consequently, in our reconfigurable I-cache design an I-cache set index is calculated traditionally considering the I-cache ways, whereas a translation request's set index is calculated using direct-mapped logic (Figure 9). However, this design requires connecting all the ways to all the comparators to ensure we compare all the tags of translations in parallel which will increase the complexity. To avoid such complexity, similar to baseline, we allow only one I-cache way to be connected to one comparator. Hence, to compare all the translation tags in a way, we incur additional 16-cycles lookup latency along with the 4-cycle decompression latency (Table 1). Please note that it is still better to access translations from I-cache with 16-cycles than performing off-chip page walks.

4.3.2 Replacement Policy. Another design decision that plays a crucial role in architecting a set-associative reconfigurable I-cache is its replacement policy. Naively replacing the LRU candidates with the translation fill entries in a reconfigurable I-cache can lead to suboptimal performance. For example, replacing the victim instruction cache line (IC-mode) with address translations will increase I-cache misses and can be detrimental to performance as the front-end bandwidth is reduced, especially if the instruction bytes replaced contain critical instructions. To address such scenarios, we propose implementing an instruction-aware LRU replacement policy. The design principles of this replacement policy is to prioritize the residency of instructions in the I-cache. To that end, the replacement policy imposes following rules.

- An instruction line fill should choose the LRU candidate from the ways containing address translations. If none of the ways contain translations, it can replace the traditional LRU instruction cache line. Thus, a cache line operating in Tx-mode can transition to IC-mode when an I-cache line replaces translations.
- A translation fill can only replace another translation in the direct-mapped I-cache line or an idle I-cache line. That is, it can only be filled in a line that does not contain any instructions or it can only replace the LRU translation in a direct-mapped line that is operating in the translation mode (Figure 8-(c)).

4.3.3 Optimized Reconfigurable I-cache Design. One of the shortcomings of the instruction-aware I-cache design discussed in the previous subsection is that translations can never replace lines containing instructions. Such a constraint allows instruction lines to continue to reside in the I-cache across kernels as long as there is a victim line containing translations. This is important for small instruction footprint kernels.

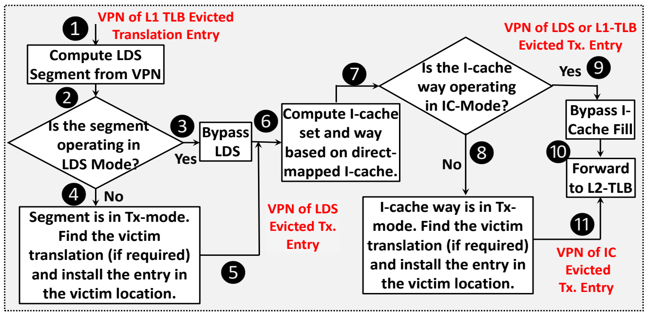
To address this problem, we propose a solution that flushes the instruction lines in the I-cache upon a kernel completion if the same kernel is not invoked back-to-back. To achieve this, the GPU runtime can place a special command packet that flushes the instruction lines into the GPU's command stream if two different kernels are enqueued back-to-back. This will allow the I-cache to start with invalid lines upon the next kernel's launch, thereby allowing the translations to be cached in the invalid lines. Please note that the GPU runtime issues these I-cache flush commands if and only if the same kernel is not going to be launched on the CU. Note that GPU runtime-issued flush commands already exist for all levels of the GPU's cache hierarchy to support coarse-grained memory model.

Figures 11a, 11b, 11c, 11d, 11e, 11f, 11g and 11h show the I-cache utilization across kernels profiled on a real GPU using rocprofiler [5] as mentioned in Section 3. In all these figures, the same kernel is not executed back-to-back, thereby allowing opportunities to flush the dead I-cache lines across kernel boundaries to leverage the I-cache capacity to cache translations. Thus, this optimization is not beneficial for applications like GEV and SRAD (Table 2) that contain only one kernel in the entire application as well as for NW which executes the same kernel ("nw_kernel1") back-to-back.

4.4 Putting It All Together

Having discussed the architectural support needed to leverage idle LDS and I-cache capacity to cache translations, we discuss the overall translation fill/lookup flows in our proposed system. Figure 12 shows how a translation evicted from the L1 TLB is filled into idle LDS/I-cache structures. Depending on whether the LDS segment corresponding to the VPN of the evicted L1-TLB entry is operating in LDS-mode or not, the translation is filled in LDS or forwarded to the I-cache. Flow ❶→❷ →❹ represents a scenario where the LDS segment corresponding to the victim VPN entry is operating in translation mode and does not incur eviction of another translation entry in LDS. However, flow ❶→❷→❹→❺→❻ corresponds to a scenario where the LDS has to evict a translation entry to accommodate the L1-TLB's victim entry. However, in a scenario where the LDS segment corresponding to the L1-TLB's victim entry is operating in LDS-mode, the entry must bypass the LDS to be inserted in the I-cache as depicted by the flow ❶→❷→❸→❻. Depending on whether the LDS could accommodate the L1-TLB's victim entry, the entry to be installed in the I-cache could be either the L1-TLB's victim entry or the LDS victim entry. Depending on the I-cache way mapped to by either of the VPN's based on the direct-mapped indexing depicted in Figure 9, if the target I-cache way operates in Tx-mode, the entry will be installed in I-cache, else not. Flow ❶→❷→❸→❻→❼→❽ depicts a scenario where the corresponding I-cache way is operating in Tx-mode and the candidate translation is installed in the target way. If a translation must be evicted from the I-cache to accommodate the candidate translation entry, the I-cache victim entry will be forwarded to the L2-TLB, as shown in flow ❶→❷→❸→❻→❼→❽→⓫. Flow ❶→❷→❸→❻→❼→❾→❿ depicts a case where the candidate translation entry at I-cache must bypass the I-cache as the corresponding way operates in IC-mode and is forwarded to L2-TLB.

The reconfigurable LDS and I-cache are looked-up in that order after a L1-TLB miss and before the L2-TLB. LDS is looked-up first since it is private to a CU and the access latency is low (2-cycles). The I-cache is looked-up if the LDS lookup results in a miss or when the segment corresponding to the virtual address is operating in LDS-mode. If the I-cache way is operating in Tx-mode and the tag in one of the sub-ways matches that of the virtual address, then the lookup results in a hit. Else, the address translation request is forwarded to the L2 TLB. Figures 6c and 9 explain the indexing



1
VPN of L1 TLB Evicted Translation Entry
Compute LDS Segment from VPN
2
Is the segment operating in LDS Mode?
3
Yes
Bypass LDS
4
No
Segment is in Tx-mode. Find the victim translation (if required) and install the entry in the victim location.
5
VPN of LDS Evicted Tx. Entry
6
Compute I-cache set and way based on direct-mapped I-cache.
7
Is the I-cache way operating in IC-Mode?
VPN of LDS or L1-TLB Evicted Tx. Entry
Yes
9
Bypass I-Cache Fill
10
Forward to L2-TLB
No
8
I-cache way is in Tx-mode. Find the victim translation (if required) and install the entry in the victim location.
11
VPN of IC Evicted Tx. Entry

Figure 12: Tx. Fill-flows in our reconfigurable design.

GPU	2 GHz, 8 CUs, 10 waves per SIMD 16 SIMD width, 4 SIMDs per CU, 64 threads per wave
L1-TLB	32-entries, Fully-associative; Access Latency: 108 cycles
L2-TLB	512-entries, 16-way associative Access Latency: 188 cycles
L1 I-Cache	16KB, 8-way, shared by 4 CUs IC-mode Tag Access Latency: 16 cycles Tx-mode Tag Access Latency: 20 cycles MUX Latency: 1 cycle Tx-mode Decompression Latency: 4 cycles
LDS	16KB, private per CU; 32-byte segments; 3-tx ways; 1-tag way LDS Tx-mode Access Latency: 35 cycles LDS-mode Access Latency: 31 cycles MUX Latency: 1 cycle Decompression Latency (Tx-mode): 4 cycles
Data Caches	L1: 32KB, 8-way; L2: 4MB, 16-way
DRAM	DDR3-1600 (800MHz), 2 channels, 16 banks per rank, 2 ranks per channel
IOMMU [38]	32 PTWs; L1/L2-TLB: 32/256 entries
IOMMU PWCs [10]	PGD/PUD/PMD Cache: 4/8/32 entries

Table 1: Simulated Setup (PTW: Page Table Walkers; PWC: Page Walk Caches)

Suite	**App**	**Kernels per App**	**B-2-B Kernels?**	**L1/L2-TLB HRs [%]**	**PTW-PKI**	**App Category**
Polybench	ATAX	2	No	(63.1/83.7)	37.68	H
	GESUMMV (GEV)	1	N/A	(27.8/75.1)	90.737	H
	MVT	2	No	(29.1/83.2)	38.76	H
	BICG	2	No	(59.1/83.5)	38.05	H
Rodinia	NW	255	Yes	(34.6/94.7)	4.92	M
	SRAD	1	N/A	(20.9/99.9)	0.04	L
	BFS	24	No	(54.8/85.4)	17.23	M
Pannotia	SSSP	10504	No	(78.8/99.8)	0.17	L
	Page Rank (PRK)	41	No	(81.3/99.9)	0.16	L
$\mu - bm$	GUPS	3	No	(25.1/46.8)	36.65	H

Table 2: Benchmarks (B-2-B: Back-to-Back Kernels; PTW-PKI: PageTable Walks per Kilo Instructions; HR: Hit Ratio percentage.)

scheme in detail for LDS and the I-cache, respectively. A translation entry that hits in these structures is promoted to the L1-TLB while the victim L1-TLB entry proceeds as explained above.

5 EXPERIMENTAL SETUP

We used gem5 [20] to simulate a system similar to an AMD A12-8870 APU [16] (Accelerating Processing Unit) containing CPU and GPU to evaluate the benefits of our approach. Our simulated system employs a shared unified virtual memory in which the CPU and GPU traverse the same four-level x86 page tables upon a TLB miss. Since we simulate an application end-to-end, we had to scale down the simulated GPU configuration significantly and simulate smaller datasets so that the simulation time would not run in to several weeks. Our gem5 model accurately models L1/L2 TLB coalescers

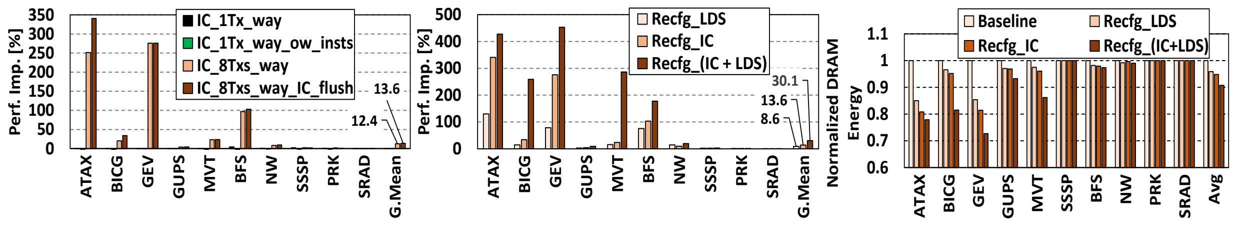


Figure 13: Reconfigurable (a) I-cache, (b) LDS + I-cache performance, and, (c) Normalized DRAM energy results.

and split page-walk caches for intermediate page table translations. We model a high-performance IOMMU that performs page walks with multiple concurrent page table walkers employing device-level L1/L2 TLBs and the split L1/L2 and L3 page walk caches that reduce the number of levels of a page table walk [10]. More details of our evaluated configuration are presented in Table 1. For the evaluation results in Section 6.1, we assume an I-cache is shared by four CUs, however, the results in Section 6.3.2 presents the results with varying number of sharers. DRAM energy results presented in Section 6.1.4 were from DRAMPower [13] tool.

We evaluated OpenCL [21], HC [12] applications from Rodinia [15], Polybench [27] and Pannotia [14] suites. Apart from these applications, we also used GUPS [22], a micro-benchmark that randomly updates a large number of pages. We categorize the applications in to High, Medium and Low based on the baseline Page Table Walks incurred Per thousand (Kilo) Instructions (PTW-PKI) executed as shown in Table 2. Applications ATAX, GEV, MVT, BICG and GUPS that incur higher than and equal to 20 are categorized as High, while, applications NW and BFS which incur a PTW-PKI greater than one but less than 20 are categorized as Medium. Finally, SRAD, SSSP and PRK that incur less than 1 PTW-PKI are categorized as Low. Though applications categorized as high and medium benefit from our schemes, we included applications from the low category to demonstrate that our proposed scheme does not degrade performance in this applications. We simulate the applications end-to-end and report the geometric mean of improvement over the baseline.

6 RESULTS

6.1 Main Results

6.1.1 Reconfigurable LDS Results. Figure 13b shows the improvement in performance as we store the translations in idle LDS space. As can be noted, the geometric mean improvement in performance across all the workloads is 8.6% with a maximum of 128.4% for ATAX. Since each 8-byte segment out of a block of 32-byte segment is consumed by the translation tags, one-fourth of the idle LDS capacity is consumed by the tags. Since applications from Polybench suite like ATAX, BICG, GEV and MVT do not use LDS (Figure 4a), out of 16KB of LDS per CU only 12KB is available for translations. Since LDS is private per CU, translations that are shared across CUs are replicated across LDS structures. Figure 14a shows the percentage of translations that can potentially be shared across all CUs. Except for GEV, NW and SRAD, a significant portion of the translations are shared across CUs thereby limiting the cumulative capacity leveraged for translations. Please note that such a duplication exists in the L1-TLBs as well. We leave optimizations to limit the translation duplication for future investigations.

6.1.2 Reconfigurable I-cache Results . Figure 13a shows the benefits of various reconfigurable I-cache designs discussed in Section 4.3. The first bar in the graph shows the improvements where only one translation is stored per I-cache way. The improvements incurred are very minimal as one translation per I-cache hardly incurs any hits. The second bar shows results which allows translations to replace instructions in the I-cache. As can be noted, the performance drops on average by 1.65% as this scheme affects instruction hit rate which thereby impacts the front-end fetch bandwidth of GPUs. To avoid such negative degradation in performance, we proposed an instruction-aware replacement policy as discussed in Section 4.3. The third bar presents a design which implements instruction-aware replacement where each I-cache operating in Tx-mode contains eight translations. Such a design can accommodate $8x$ more translations and hence the performance improvements are substantially high with a geometric mean improvement of 12.4%, with the highest improvement of 251.5% for ATAX. The last bar in the results show the benefits of the I-cache flush optimization discussed in Section 4.3.3. As shown in Table 2, applications GEV and SRAD have only one kernel and hence this optimization does not give any benefits for these applications. Also, the NW application from the Rodinia suite has the same kernel "nw_kernel1" executed back-to-back, and as a result, the I-cache is not flushed at the kernel boundary, hence there is no change in performance. The first kernel flush from ATAX makes ~6KB of free space available to store translations thereby incurring an additional improvement of 35.4%. BICG (7.4%), MVT (4.5%) and BFS (4.8%) incur significant improvements in performance as the instruction flushes increase the number of translations cached. Overall, the I-cache flush improved the performance by an additional 1.2% across all the applications.

6.1.3 Reconfigurable LDS + I-cache Results. Figure 13b shows the overall performance when both LDS and I-cache idle capacities are used to cache translations as explained in Section 4.4. Since the overall capacity available for translations is cumulative of the idle LDS and I-cache capacities, the geometric mean improvement in performance across all the applications is 30.1%, while ATAX and BICG incur improvements as high as 443.3% and 442.3%, respectively. GUPS performance is increased by 9.14%. SSSP, PRK and SRAD are not bottlenecked by virtual memory and do not see significant improvement in performance with our optimizations. Considering only the TLB miss intensive applications that are categorized

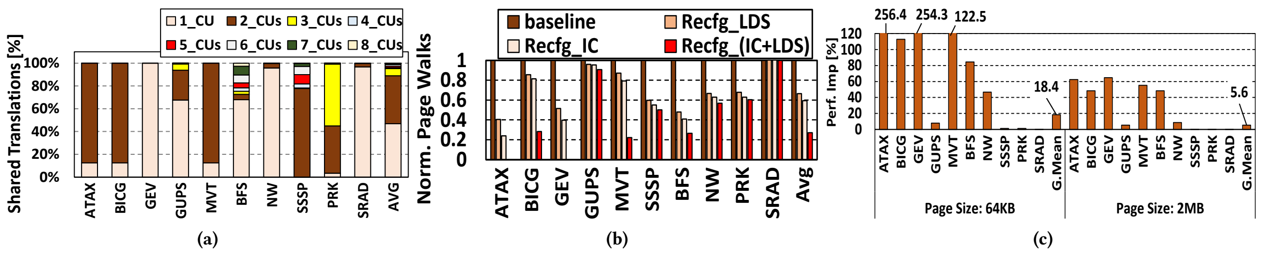


Figure 14: (a) Tx-sharing results, (b) Norm. Page Walks, and, (c) 4KB and 2MB page granularity improvements.

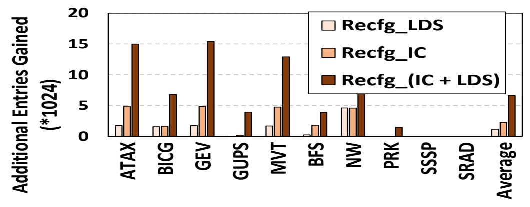


Figure 15: Additional Translation Entries Gained.

as High and Medium, the overall improvements in performance acheived by our reconfigurable LDS, I-cache and (IC+LDS) schemes are 25.9%, 36.5%, and 147.2% respectively. Considering the low TLB miss intensive applications as well, the respective improvements in performance are 8.6%, 13.6% and 30.1% as shown in Figure 13b. From Figure 14b, compared to the baseline, reconfigurable LDS, IC and IC+LDS schemes reduce the page walks by 33.5%, 40.6% and 72.9%, respectively. Please note that SRAD incurs ~0 page walks in the baseline and hence the normalized page walks do not change. Figure 15 shows the additional translation entries gained. A maximum of 16K entries can be gained in our config (12K from LDS and 4K frmom I-caches).

6.1.4 DRAM Energy Results. Figure 13c shows normalized DRAM energy consumption results. As can be noted, leveraging LDS, I-cache, and LDS + I-cache capacities for translations reduced the DRAM energy consumption on average by 4.1%, 5.2%, and 9.2%, respectively, as the number of DRAM page walk requests are reduced substantially due to on-chip translation hits in these structures, with GEV incurring the highest reduction of 27.3%.

6.2 Page Size Sensitivity Results

Page size plays a crucial role in the virtual memory traffic observed on both CPUs and GPUs. Figure 14c shows the performance improvement of our reconfigurable Icache + LDS design for page size granularities of 64KB and 2MB. 64KB page size is supported in d-GPUs and large 2MB page sizes are supported in both d-GPUs and integrated GPUs like APU. As can be observed, our reconfigurable Icache + LDS design improves performance by 18.4% for a 64KB page granularity while it improves performance by 5.6% for a large 2MB page granularity vs. an improvement of 30.1% with a 4KB page granularity. This shows that our scheme is successful in alleviating address translation overheads for applications with large footprints even with large pages.

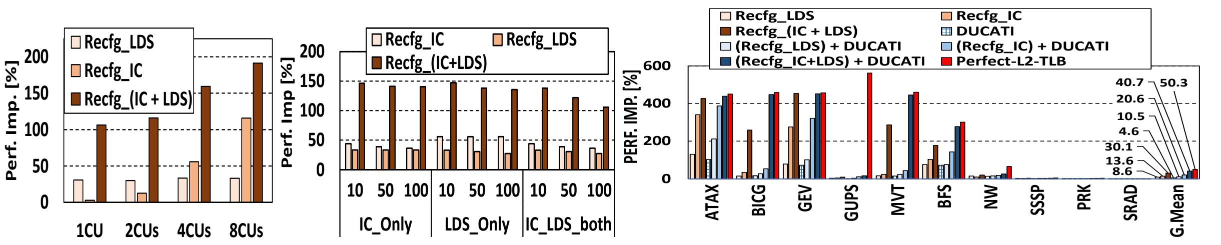
6.3 Other Results

6.3.1 LDS Segment Size Sensitivity. We have experimented with increasing the segment size of LDS from 32 bytes to 64 bytes. This increase in the segment size increases the associativity for translations from 3 to 6 but reduces the number of segments by half. With the increased 64B segment size, we did not see any improvement in performance as the translation misses are mostly capacity misses and not conflict misses. Increasing associativity without increasing capacity did not matter much for performance.

6.3.2 I-cache Sharers Sensitivity. In the baseline design, we assumed a GCN-like organization [2] where an I-cache is shared by four CUs. Figure 16a shows the improvements as the number of CUs sharing an I-cache is varied from one (private per-CU) to eight (fully-shared across all the CUs). In these results, the total I-cache capacity is kept constant and only the number of sharers vary. As the number of sharer CUs increases from 1-CU to 8-CUs, the performance increases from 17.3% to 38.4% as the duplication of translations decreases which improves the translation coverage.

6.3.3 I-cache/LDS Translation Access Latency. The routing/wire access latency to translations in I-cache and LDS is a function of the length of the additional datapaths that feed the translations from these reconfigurable hardware structures to the L1-TLBs. The wire lengths of the additional datapaths, and hence their total access latencies, are a function of the layout employed by a GPU. To understand the impact of this additional datapath wire latency, we perform sensitivity studies by adding 10, 50, and 100 cycles of additional translation wire latency to LDS/I-cache structures. Please note that these are additional latencies incurred on top of decompression, SRAM macro access and MUX latencies given in Table 1. Figure 16b presents three sets of results. "IC_only" only incurs additional wire latencies (10, 50, and 100 cycles) for I-cache translation lookup, while "LDS_only" incurs additional wire latencies for translation lookup in LDS. "IC_LDS" shows the performance as both I-cache and LDS translation lookup incurs additional wire lookup latencies. The worst-case 100-cycle wire latency for both IC and LDS show an improvement of 9.4% on average when both I-cache and LDS store victim translations. These results are not surprising as GPUs are designed to be latency-tolerant, and as a result, we still see good improvements in performance as we incur additional datapath/wire latency. This shows the efficacy of our approach in averting expensive page walks to memory by leveraging idle (though farther) structures.

6.3.4 Improvement with DUCATI [23]. Prior work DUCATI [23] proposed leveraging capacity from last-level cache and carved-out capacity from GPU device memory to store end-to-end address translations. Their scheme causes additional contention for capacity and bandwidth. Figure 16c shows the improvement in performance when our reconfigurable architecture is employed with DUCATI



(a) (b) (c)

Figure 16: Performance Results with (a) Variable number of CUs sharing an I-cache, (b) Variable access latency to LDS and I-cache due to layout constraints, and, (c) DUCATI [23].

proposal. Since DUCATI stores translations in data caches and in memory, it reduces the overall page walks. In DUCATI, address translations contend for capacity and bandwidth in the last-level cache and memory. Unlike our scheme they do not opportunistically leverage the free capacity and hence cause contention for capacity and memory bandwidth. Consequently, DUCATI improves the geometric mean performance by 4.9%. This low improvement in performance in DUCATI compared to our scheme is due to the higher number of off-chip accesses to the translations from memory. In conjunction with DUCATI, our hybrid reconfigurable Icache + LDS design improves the performance by 40.7% compared to our original design which incurred 30.1% improvement over baseline. This shows that our reconfigurable design is successful in improving the translation reach by complementing the DUCATI proposal.

7 DISCUSSION

7.1 Translation Shootdowns

GPU driver initiates a translation shootdown to invalidate stale translations in the event of a page swap (to disk) or migration (within memory) or even during the updating of permission bits. Because our proposal involves caching translations in the I-cache and LDS, the TLB shootdown process should now include these reconfigurable data structures in addition to the TLBs. PM4 command packet is used by AMD GPU driver to communicate with GPU hardware and configure it [3, 40]. TLB shootdowns are originally initiated by the GPU driver [36, 47, 49] and can use a similar command to initiate the GPU TLB shootdown on LDS/I-cache structures. The packet with a PM4-like command corresponding to TLB shootdown is enqueued by the driver into the command queue and the GPU packet processor reads this command packet from the command queue. After parsing the packet, the packet processor notifies the virtual page number of the translation to be invalidated as part of the shoot-down to I-cache/LDS controllers to invalidate the translation entry if present.

7.2 Multiple-Application Scenarios

Several authors [9, 25] have proposed executing multiple applications on a GPU concurrently. Traditionally, different applications are launched on different CUs [9, 25] as they can result in security issues [32] if they are scheduled on a same CU. Consequently, we expect our reconfigurable LDS architecture will continue to fare well as it can leverage the private per-CU LDS to store address translations of an application mapped to that particular CU. However, I-cache is a shared structure and the total amount of under-utilized capacity might reduce in the multi-application scenario. However, since our approach only opportunistically leverages idle I-cache capacity, it does not degrade the performance.

8 RELATED WORK

A number of works have explored techniques to expand TLB reach by using large pages. Ingens [26] presented techniques to improve large pages by providing primitives to manage contiguity and improve utilization and fairness. Ausavarungnirun et al. [8] specifically targeted large page support on GPUs with the Mosaic, which provides transparent support for multiple page sizes on GPUs. Yoon et al. [48] proposed virtual caching to reduce the virtual memory overheads on GPUs. These works are complimentary to our own.Power et al. [38] propose techniques for hundreds of GPU lanes to perform x86-64 address translations in a single cycle. Picchai et al. [37] proposed small modifications to both TLBs and page table walkers to improve virtual memory efficiency. Basu et al. [43, 44] proposed techniques to improve translation for GPUs by optimizing page table walks in the IOMMU. The first work [43] proposes new algorithms to schedule page table walks to optimize forward progress for SIMT execution. The second work [44] performs coalescing to take advantage of locality during the page walks themselves. Both works are complementary to our proposals, since they affect behavior after a TLB miss.

Gebhart et al. [19] proposes combining all of a GPU's on-chip storage into a single structure to avoid idle resources on the device. While such an approach could be extended for TLB entries, such a drastic design change would be prohibitively difficult to implement in practice. Finally, some researchers propose methods to improve TLB reach by storing TLB translations outside of the TLB itself. Ryoo et al. [42] expand the TLB reach of CPUs by keeping a very large TLB as a part of main memory, which they refer to as POM-TLB. As explained in Section 6.3.4, DUCATI [23] stores translations in last-level cache and memory. Our proposal focuses on increasing on-chip translation hit rate by leveraging idle LDS and I-cache on-chip SRAM structures.

9 CONCLUSION

In this work, we make a key observation that a number of statically allocated GPU SRAM structures are sized for worst-case application behavior, and are frequently underutilized. Concurrently, we observe and note from prior work [8, 23] that emerging classes of GPU applications exhibit irregular memory access patterns to an ever increasing memory space. These applications are frequently bound by insufficient TLB capacity and suffer from page table walks that have been shown to be an order of magnitude slower than their CPU counterparts [47]. To address this issue, we introduce an opportunistic mechanism that steals idle resources from underutilized I-cache and shared memory SRAMs to improve TLB reach. We show that our scheme is efficient to implement in hardware and provides a 30.1% speedup over a baseline GPU design across a variety of important GPU applications.

ACKNOWLEDGMENTS

The authors would like to thank the AMD internal reviewer, Brad Benton whose reviews helped improve the quality of this paper. Additionally, the authors would also like to thank Vignesh Adhinarayanan for loaning a GPU system that helped in profiling workloads. 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. Kandemir is supported in part by NSF grants 1763681, 1908793, 2028929, and 1931531.

REFERENCES

[1] Sriram Aananthakrishnan, Nesreen K. Ahmed, Vincent Cavé, Marcelo Cintra, Yigit Demir, Kristof Du Bois, Stijn Eyerman, Joshua B. Fryman, Ivan Ganev, Wim Heirman, Hans-Christian Hoppe, Jason Howard, Ibrahim Hur, Midhunchandra Kodiyath, Samkit Jain, Daniel S. Klowden, Marek M. Landowski, Laurent Montigny, Ankit More, Przemyslaw Ossowski, Robert Pawlowski, Nick Pepperling, Fabrizio Petrini, Mariusz Sikora, Balasubramanian Seshasayee, Shaden Smith, Sebastian Szkoda, Sanjaya Tayal, Jesmin Jahan Tithi, Yves Vandriessche, and Izajasz P. Wrosz. 2020. PIUMA: Programmable Integrated Unified Memory Architecture. *CoRR* abs/2010.06277 (2020). arXiv:2010.06277 https://arxiv.org/abs/2010.06277

[2] AMD. 2019. *AMD Graphics Cores Next (GCN) Architecture.*

[3] AMD. 2019. GPU Programming Guide. https://developer.amd.com/wordpress/media/2013/10/si_programming_guide_v2.pdf.

[4] AMD. 2019. Radeon^{TM} RX580. https://www.amd.com/en/products/graphics/radeon-rx-580.

[5] AMD. 2019. ROC Profiler Library. https://github.com/ROCm-Developer-Tools/rocprofiler.

[6] AMD. 2020. AMD MI-50 Accelerator. https://www.amd.com/system/files/documents/radeon-instinct-mi50-datasheet.pdf

[7] AMD. 2020. AMD MI-60 Accelerator. https://www.amd.com/system/files/documents/radeon-instinct-mi60-datasheet.pdf.

[8] R. Ausavarungnirun, J. Landgraf, V. Miller, S. Ghose, J. Gandhi, C. J. Rossbach, and O. Mutlu. 2017. Mosaic: A GPU Memory Manager with Application-Transparent Support for Multiple Page Sizes. In *2017 50th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO).*

[9] Rachata Ausavarungnirun, Vance Miller, Joshua Landgraf, Saugata Ghose, Jayneel Gandhi, Adwait Jog, Christopher J. Rossbach, and Onur Mutlu. 2018. 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 (ASPLOS).*

[10] Thomas W. Barr, Alan L. Cox, and Scott Rixner. 2010. Translation Caching: Skip, Don'T Walk (the Page Table). In *Proceedings of the 37th Annual International Symposium on Computer Architecture (ISCA).*

[11] Centor For Efficient Exascale Discretizations (CEED). 2019. Laghos: High-order Langrangian Hydrodynamics Miniapp. https://github.com/CEED/Laghos.

[12] S Chan. 2019. A Brief Intro to the Heterogeneous Compute Compiler. https://gpuopen.com/a-brief-intro-to-boltzmann-hcc/.

[13] K. Chandrasekar, C. Weis, Y. Li, S. Goossens, M. Jung, O. Naji, B. Akesson, N. Wehn, and K. Goossens. 2019. DRAMPower: Open-source DRAM Power and Energy Estimation Tool. www.es.ele.tue.nl/drampower/.

[14] S. Che, B. M. Beckmann, S. K. Reinhardt, and K. Skadron. 2013. Pannotia: Understanding irregular GPGPU graph applications. In *2013 IEEE International Symposium on Workload Characterization (IISWC).*

[15] S. Che, M. Boyer, J. Meng, D. Tarjan, J. W. Sheaffer, S. Lee, and K. Skadron. 2009. Rodinia: A benchmark suite for heterogeneous computing. In *2009 IEEE International Symposium on Workload Characterization (IISWC).*

[16] CPU-World. 2019. AMD A12-Series PRO A12-8870. http://www.cpu-world.com/CPUs/Bulldozer/AMD-A12-Series%20PRO%20A12-8870.html.

[17] Anthony Danalis, Gabriel Marin, Collin McCurdy, Jeremy S. Meredith, Philip C. Roth, Kyle Spafford, Vinod Tipparaju, and Jeffrey S. Vetter. [n. d.]. The Scalable Heterogeneous Computing (SHOC) Benchmark Suite.

[18] Pat Flick. 2019. MIOpen Benchmarks. https://github.com/patflick/miopen-benchmark.

[19] M. Gebhart, S. W. Keckler, B. Khailany, R. Krashinsky, and W. J. Dally. 2012. Unifying Primary Cache, Scratch, and Register File Memories in a Throughput Processor. In *2012 45th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO).*

[20] gem5. 2019. The gem5 simulator. http://gem5.org.

[21] K. Group. 2019. OpenCL. https://www.khronos.org/opengl/.

[22] GUPS. 2019. GUPS. https://icl.utk.edu/projectsfiles/hpcc/RandomAccess//.

[23] Aamer Jaleel, Eiman Ebrahimi, and Sam Duncan. 2019. DUCATI: High-performance Address Translation by Extending TLB Reach of GPU-accelerated Systems. *ACM Transactions on Architecture and Code Optimization (TACO)* (2019).

[24] Zhe Jia, Marco Maggioni, Benjamin Staiger, and Daniele Paolo Scarpazza. 2018. Dissecting the NVIDIA Volta GPU Architecture via Microbenchmarking. *CoRR* abs/1804.06826 (2018).

[25] Adwait Jog, Evgeny Bolotin, Zvika Guz, Mike Parker, Stephen W. Keckler, Mahmut T. Kandemir, and Chita R. Das. 2014. Application-Aware Memory System for Fair and Efficient Execution of Concurrent GPGPU Applications. In *Proceedings of Workshop on General Purpose Processing Using GPUs (GPGPU).*

[26] Youngjin Kwon, Hangchen Yu, Simon Peter, Christopher J. Rossbach, and Emmett Witchel. 2016. Coordinated and Efficient Huge Page Management with Ingens. In *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation (OSDI).*

[27] L.-N. Pouchet and T. Yuki. 2019. Polybench Suite. http://web.cse.ohio-state.edu/~pouchet.2/software/polybench/.

[28] Lawrence Livermore National Laboratory. 2019. Quicksilver. https://github.com/LLNL/Quicksilver.

[29] Los Alamos National Laboratory. 2019. KRIPKE. https://github.com/LLNL/Kripke.

[30] Los Alamos National Laboratory. 2019. The PENNANT Mini-App. https://github.com/lanl/PENNANT.

[31] X. Mei and X. Chu. 2017. Dissecting GPU Memory Hierarchy Through Microbenchmarking. *IEEE Transactions on Parallel and Distributed Systems* 28, 1 (2017), 72–86.

[32] H. Naghibijouybari, K. N. Khasawneh, and N. Abu-Ghazaleh. 2017. Constructing and Characterizing Covert Channels on GPGPUs. In *2017 50th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO).*

[33] Nvidia. 2019. GPU Applications. http://www.nvidia.com/object/gpu-applications-domain.html.

[34] Nvidia. 2019. NVIDIA Collective Communications Library (NCCL). https://developer.nvidia.com/nccl.

[35] Nvidia. 2019. Shared Memory. https://docs.nvidia.com/cuda/pascal-tuning-guide/index.html#shared-memory.

[36] Mark Oskin and Gabriel H. Loh. 2015. A Software-Managed Approach to Die-Stacked DRAM. In *2015 International Conference on Parallel Architecture and Compilation (PACT).*

[37] Bharath Pichai, Lisa Hsu, and Abhishek Bhattacharjee. 2014. 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 (ASPLOS).*

[38] J. Power, M. D. Hill, and D. A. Wood. 2014. Supporting x86-64 address translation for 100s of GPU lanes. In *2014 IEEE 20th International Symposium on High Performance Computer Architecture (HPCA).* 568–578. https://doi.org/10.1109/HPCA.2014.6835965

[39] Sooraj Puthoor and Mikko H. Lipasti. 2018. Compiler Assisted Coalescing. In *Proceedings of the 27th International Conference on Parallel Architectures and Compilation Techniques (PACT).* Association for Computing Machinery.

[40] Sooraj Puthoor, Xulong Tang, Joseph Gross, and Bradford M. Beckmann. 2018. Oversubscribed Command Queues in GPUs. In *Proceedings of the 11th Workshop on General Purpose GPUs (GPGPU-11.*

[41] M. K. Qureshi and G. H. Loh. 2012. Fundamental Latency Trade-off in Architecting DRAM Caches: Outperforming Impractical SRAM-Tags with a Simple and Practical Design. In *2012 45th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO).*

[42] Jee Ho Ryoo, Nagendra Gulur, Shuang Song, and Lizy K. John. 2017. Rethinking TLB Designs in Virtualized Environments: A Very Large Part-of-Memory TLB. In *Proceedings of the 44th Annual International Symposium on Computer Architecture (ISCA).*

[43] S. Shin, G. Cox, M. Oskin, G. H. Loh, Y. Solihin, A. Bhattacharjee, and A. Basu. 2018. Scheduling Page Table Walks for Irregular GPU Applications. In *2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA).*

[44] Seunghee Shin, Michael LeBeane, Yan Solihin, and Arkaprava Basu. 2018. Neighborhood-aware Address Translation for Irregular GPU Applications. In *Proceedings of the 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*.

[45] SR-IOV. [n. d.]. SR-IOV GPUs. https://www.amd.com/en/graphics/workstation-virtual-graphics. [Online; accessed February-02, 2021].

[46] Xulong Tang, Ziyu Zhang, Weizheng Xu, Mahmut Taylan Kandemir, Rami Melhem, and Jun Yang. 2020. Enhancing Address Translations in Throughput Processors via Compression. In *Proceedings of the ACM International Conference on Parallel Architectures and Compilation Techniques (PACT)*.

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

[48] Hongil Yoon, Jason Lowe-Power, and Gurindar S. Sohi. 2018. Filtering Translation Bandwidth with Virtual Caching. In *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*.

[49] Tianhao Zheng, David Nellans, Arslan Zulfiqar, Mark Stephenson, and Stephen W. Keckler. 2016. Towards high performance paged memory for GPUs. In *Proceedings of IEEE International Symposium on High Performance Computer Architecture (HPCA)*.