Enhancing Address Translations in Throughput Processors via Compression

Xulong Tang
University of Pittsburgh
Pittsburgh, Pennsylvania, USA
tax6@pitt.edu

Ziyu Zhang
University of Pittsburgh
Pittsburgh, Pennsylvania, USA
ziz41@pitt.edu

Weizheng Xu
University of Pittsburgh
Pittsburgh, Pennsylvania, USA
wex43@pitt.edu

Mahmut Taylan Kandemir
The Pennsylvania State University
University Park, Pennsylvania, USA
mtk2@cse.psu.edu

Rami Melhem
University of Pittsburgh
Pittsburgh, Pennsylvania, USA
melhem@cs.pitt.edu

Jun Yang
University of Pittsburgh
Pittsburgh, Pennsylvania, USA
juy9@pitt.edu

ABSTRACT

Efficient memory sharing among multiple compute engines plays an important role in shaping the overall application performance on CPU-GPU heterogeneous platforms. Unified Virtual Memory (UVM) is a promising feature that allows globally-visible data structures and pointers such that the GPU can access the physical memory space on the CPU side, and take advantage of the host OS paging mechanism without explicit programmer effort. However, a key requirement for the guaranteed performance is effective hardware support of address translation. Particularly, we observe that GPU execution suffers from high TLB miss rates in a UVM environment, especially for irregular and/or memory-intensive applications. In this paper, we propose simple yet effective compression mechanisms for address translations to improve GPU TLB hit rates. Specifically, we explore and leverage the TLB compressibility during the execution of GPU applications to design efficient address translation compression with minimal runtime overhead. Experimental results across 22 applications indicate that our proposed approach significantly improves GPU TLB hit rates, which translate to 12% average performance improvement. Particularly, for 16 irregular and/or memory-intensive applications, the performance improvements achieved reach up to 69.2%, with an average of 16.3%.

CCS CONCEPTS

• **Computer systems organization → Single instruction, multiple data**; **Heterogeneous (hybrid) systems**.

KEYWORDS

CPU-GPU heterogeneous system; unified virtual memory; TLB; performance

PACT '20, October 3–7, 2020, Virtual Event, GA, USA

ACM ISBN 978-1-4503-8075-1/20/10…$15.00
https://doi.org/10.1145/3410463.3414633

ACM Reference Format:
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 2020 International Conference on Parallel Architectures and Compilation Techniques (PACT '20), October 3–7, 2020, Virtual Event, GA, USA.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3410463.3414633

1 INTRODUCTION

The ever-increasing complexity of emerging applications has pushed for a florescence of heterogeneous computing platforms that comprise heterogeneous processing elements such as GPUs, FPGAs, and other types of accelerators. The CPU-GPU system, as a ubiquitous and widely-adopted platform, has gained momentum in various application domains such as deep learning [52, 68], high-performance scientific computing [33], bio-medical applications [13, 23], and computer vision [18].

Traditional CPU-GPU system organizes CPUs and GPUs in a "master-slave" execution model. Such organization suffers from two limitations. First, it requires significant programmer effort to explicitly manage the data transfers between the host CPU and the GPU device. Second, the limited capacity of GPU memory forbids memory-intensive applications to take full advantage of GPU execution. Unified Virtual Memory (UVM), supported by vendors such as NVIDIA [35] and AMD [4], is a promising feature that allows *globally visible* data structures and pointers so that the GPU can access the physical memory space on the host CPU side, and take advantage of the host OS paging mechanism without explicit programmer management. This feature is especially beneficial for the deployment of complex applications whose memory footprints exceed the modern GPU memory capacities [26, 71].

While UVM is certainly promising, one of the key factors that affect the delivered performance of UVM is the efficiency of the address translation support. Specifically, memory accesses need to go through a multi-step address translation process, including multi-level TLB lookups and multi-level page table walks in order to get the physical address of required data. Recent work has shown that the address translation overheads can take up to 50% of the application execution time [9, 11, 25]. These overheads are more significant in GPUs where TLB miss rates (at both L1 and L2 levels) are generally much higher compared to CPUs [46, 50]. This is due to the intensive and divergent memory requests (even after coalescing)

that originate from the GPU SIMT execution. Given the fact that a TLB miss is significantly expensive compared to a data cache miss [50], a high TLB miss rate can easily hinder the memory system from feeding the GPU computing engines with the required data in a timely fashion, leading to severe under-utilization and eventually performance degradation.

Previous works have focused on improving TLB hit rates from multiple angles, including contiguity-based range TLBs [25, 69], cluster TLBs [42, 44], employing large pages [5, 36, 45] , and eager paging [9, 22]. While these techniques effectively reduce the number of TLB misses and increase the TLB reach, they are ill-suited for GPUs. First, large pages suffer from internal memory fragmentation [30]. Besides, UVM transfers physical pages between CPU memory and GPU memory using on demand paging policy [34]. As transferring large pages incurs more data movements, they are only used cautiously in, e.g., evicting pages out of an almost full memory [9]. Second, GPU TLBs receive much higher TLB miss rates compared to CPU TLBs due to the nature of SIMT execution where intensive and divergent memory accesses are generated from multiple warps that run concurrently. Third, exploiting continuity of page accesses is difficult in GPUs. On the one hand, continuity requires daemon thread and OS support to swap physical pages and generate continuity in physical memory space [37, 69]. This is particularly inefficient in GPUs since it requires batching the page swapping requests, sending them to CPUs, and interrupting CPUs to handle the physical page swaps [3]. On the other hand, our characterization reveals that most of the GPU page accesses are *non-consecutive* accesses where stride between two consecutive requested pages usually varies during the course of execution (i.e., there does not exist a unified stride). Finally, the CPU oriented cluster TLBs [42, 44] are proposed based on the similar observation of clustered access patterns. However, such CPU-oriented approaches are not effective in handling large stride accesses, which are quite frequent in GPU execution. Therefore, these approaches result in fewer benefits and cause expensive hardware overheads to maintain the metadata in GPUs. In sum, it is important to leverage the unique GPU memory access characteristics and develop corresponding address transition optimizations in order to improve the performance of UVM in CPU-GPU platforms. Table 1 summarizes the pros and cons of the prior techniques focusing on TLB optimizations.

In this paper, we propose a simple yet effective and efficient TLB compression strategy to address the poor TLB performance in GPUs. Our approach is based on the observation that, during the execution of a GPU application, rather than showing continuity (i.e., with a stride of 1) among the accessed pages, those pages show *clustered* patterns in a periodic fashion. That is, for a given execution period, the accessed pages are close to each other in the address space. This observation also holds true for irregular applications with scattered memory access patterns. As a result, both the virtual page number (VPN) and the physical frame number (PFN) in the address translations of these accesses have a number of identical bits (i.e., the upper bits of the addresses, both virtual and physical, are identical among these accesses). Based on this observation, we propose hardware-supported address translation compression mechanisms to eliminate the redundant address bits in the TLB entries. Specifically, we adopt $\langle$base, $\delta\rangle$ compression where, instead of maintaining all bits of VPN and PFN in a TLB entry, *multiple* VPNs and PFNs are stored in *one* TLB entry in their δ format. The bases are stored separately in hardware registers. We also propose *parallel compression and decompression procedures* where the overheads of compression are effectively hidden by overlapping them with normal TLB operations. The main contributions are as follows:

- We conduct an in-depth characterization of modern GPU applications in a UVM environment. We observe that i) most GPU applications suffer from low TLB hit rates and improving TLB hit rates has significant impact on the overall application performance, ii) GPU page accesses exhibit non-consecutive but clustered access patterns during certain execution periods, and iii) there are significant number of identical bits in the VPNs (as well as PFNs) of those clustered pages, and those bits are redundantly stored in the TLB.
- We propose a TLB compression mechanism built upon the base-δ compression to improve the TLB reach and TLB hit rates by allowing multiple address translations to be stored in the same TLB entry. Our approach eliminates the identical bits (as bases) and only maintains the differences (as δs) in the TLB. Meanwhile, we propose parallel compression and decompression mechanisms where the overheads of our compression scheme can be effectively hidden by overlapping them with normal TLB operations. We also propose a partitioned TLB design to accommodate execution scenarios where the translations are non-compressible.
- We thoroughly evaluate our proposed approach using 22 application programs from various benchmark suites. Experimental results indicate that our approach effectively improves TLB hit rate. These enhanced TLB hit rates translate to average 12% performance improvements across all 22 application programs. In particular, for 16 irregular and/or memory-intensive applications, the performance improvements reach up to 69.2%, with an average of 16.3%.

2 BACKGROUND

2.1 Unified Virtual Memory (UVM)

UVM is a promising feature adopted in commercial heterogeneous platforms, especially CPU-GPU systems, to reduce the programmer burden on explicit data transfer and management [4, 35]. UVM allows globally-visible pointers and data structures such that the explicit memory copy is no longer required. As a result, it significantly simplifies GPU programming and relieves programmers from fine-grained data movement management. It also enables application kernels with a large working set to be deployed on GPUs, and allows GPU kernels to benefit from OS managed paging system. At runtime, the on-demand paging mechanism migrates memory pages from CPU to GPU and vice versa, which allows data transfers between CPU and GPU at page granularity.

However, UVM is not free. Accessing CPU memory pages from the GPU side is expensive compared to accessing the pages in the GPU local memory. The cost comprises not only the overhead of page migration, but also the overhead of address translation. Specifically, the performance of hardware support of UVM relies on the translation of virtual address to physical address for all GPU data accesses. While there are very few publicly available documents

Table 1: Comparison with prior techniques.

Techniques	No OS changes	No HW changes	Continuous accesses	Irregular accesses	Stride accesses	No internal fragmentation	Suitable in CPU-GPU
Range TLB [25, 37, 69]	✗	✓	✓	✗	✗	✓	✗
Cluster TLB [42, 44]	✓	✗	✓	✗	✗	✓	✗
Large page [5, 36, 45]	✗	✗	✓	✗	✗	✗	✗
Eager paging [9, 22]	✗	✗	✓	✗	✓	✗	✗
Speculative TLB [8]	✗	✗	✓	✗	✓	✓	✗
Our approach	✓	✗	✓	✓	✓	✓	✓

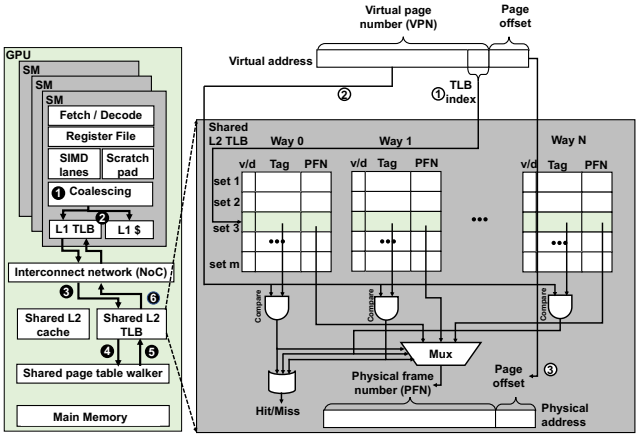

Figure 1: Baseline GPU architecture and L2 TLB lookup procedure.

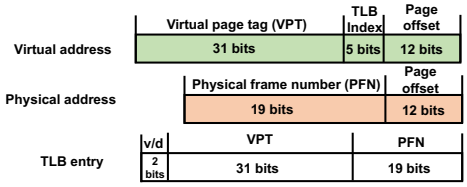

Figure 2: Address format and the content of the TLB entry in the baseline architecture.

regarding the TLB organization in commercial GPUs, a widely-adopted design is to allow the GPUs to have its own TLB hierarchy and page table walkers [50], similar to the CPU MMU design. Such design can effectively reduce the address translation overhead with a good TLB performance (hit rate). Another design employs IOMMU [3, 21], where a separate IO TLB hierarchy is maintained in IOMMU. In this design, the address translation requests that miss in the last-level GPU TLB are sent, within Address Translation Service (ATS) packages, over PCI-e, to IOMMU for IO TLB lookup and page table walk. In both the designs, address translations that miss the GPU TLB incur significant delays, leading to performance degradation.

2.2 Baseline Architecture

Figure 1 shows the targeted GPU architecture, as well as typical GPU data access flow with address translation. A GPU consists of multiple SMs that are connected through an on-chip network[1]. Each SM consists of a private L1 TLB to cache the page table entries close to execution units. In our baseline architecture, we adopt a virtually-indexed and physically-tagged (VIPT) data cache such that the TLB lookup and the data cache lookup happen in parallel. Note that, before TLB and cache lookup, the memory accesses are *coalesced* by per-SM coalescing unit to reduce the number of outstanding memory requests. L2 TLB and L2 data cache are partitioned and placed after the on-chip interconnect. Both L2 TLB and L2 data cache are *shared* across all SMs.

During execution, the data accesses generated by GPU warps are first coalesced by the coalescing engine, which combines accesses to the same cacheline (❶). After getting the coalesced address, the GPU load-store unit schedules an L1 TLB lookup to see whether the translation is cached or not (❷). Note that, the L1 cache lookup happens in parallel in VIPT data cache. If the translation lookup returns a TLB hit, the corresponding data cache lookup uses the physical page frame number (PFN) to compare with the data cache tag and determine whether the memory access is a data cache hit or not. If the lookup misses in the L1 TLB, the translation request is forwarded to the shared L2 TLB (❸). If the L2 TLB lookup also misses, the translation request invokes a *page table walker thread* to perform page table walk (❹). Since a page table is typically built as multiple levels, this page table walking process involves multiple memory accesses, which are very time consuming [5]. After the page table walk, the completed translation is sent back to the L2 TLB (❺) and further sent back into the L1 TLB in an inclusive TLB hierarchy (❻). Then, the memory access request is *replayed*, and an L1 TLB hit is returned.

Figure 1 also depicts the traditional process for L2 TLB lookup. The virtual address of memory access is partitioned into virtual page number (VPN) and page offset. Considering an N-way associative L2 TLB, the lower bits of the VPN are used as index to determine the set in the TLB (①). The upper bits of the VPN are then compared against the tag for determining TLB hit/miss (②). If it is a TLB hit, the corresponding PFN is concatenated with the page offset to form a physical address (③). Otherwise, a TLB miss occurs and the memory request is sent to page table walk queue for page table walk. Note that, although we use shared L2 TLB as a discussion example, the per-SM private L1 TLB share the same data structure and the lookup procedure as L2 TLB. The only difference between L1 and L2 is the capacity and the associativity.

Figure 2 shows the virtual address to physical address mapping as well as the contents of a particular TLB entry. The virtual address consists of a 36 bits VPN and 12 bits page offset (i.e., 4KB page size) where the lower 5 bits of the VPN (in a 512 entry, 16-way associativity TLB) are used for TLB set indexing. The upper bits of the VPN (31 bits) excluding the indexing bits are called virtual page tags (VPTs), and are used to compare with the VPTs in the TLB. Each TLB entry also consists of a valid bit and a dirty bit.

Our proposed approach in this paper is built upon the base-δ compression [41]. The key observation behind is that, for many

[1]In this paper, we use streaming multiprocessor (SM) from NVIDIA terminology and compute unit (CU) from AMD terminology interchangeably.

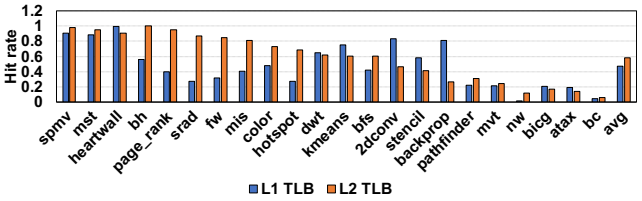

Figure 3: L1 TLB hit rate and L2 TLB hit rate in the baseline executions.

translation requests, the values in VPNs (and PFNs) stored in TLB entries have a low dynamic range: i.e., the relative difference between values is small. Therefore, base-δ compression allows us to only store the value differences in TLB entries, instead of the original values of VPNs and PFNs. As a result, the number of bits that need to be maintained in TLB is significantly reduced.

3 MOTIVATION AND CHARACTERIZATION

3.1 Application Programs

Table 2: List of benchmarks.

Name	Cat.	TLB	Name	Cat.	TLB
spmv [57]	C, I	(H,H)	mst [28]	M, I	(H,H)
heartwall [15]	C, R	(H,H)	bh [28]	M, I	(L,H)
page_rank [14]	C, I	(L,H)	srad [15]	C, R	(L,H)
fw [14]	M, I	(L,H)	mis [14]	C, I	(L,H)
color [14]	M, I	(L,H)	hotspot [15]	C, R	(L,L)
dwt [15]	C, I	(L,L)	kmeans [15]	M, I	(H,L)
bfs [15]	M, I	(L,L)	2dconv [20]	C, R	(H,L)
stencil [57]	C, R	(L,L)	backprop [15]	C, R	(H,L)
pathfinder [15]	M, R	(L,L)	mvt [20]	C, I	(L,L)
nw [15]	M, I	(L,L)	bicg [20]	C, I	(L,L)
atax [20]	M, I	(L,L)	bc [14]	C, I	(L,L)

Table 2 summarizes the important characteristics of the benchmarks we selected from various suites. Specifically, we choose 22 benchmarks from Pannotia [14], Poly [20], Rodinia [15], Lonestar [28], and Parboil [57]. These benchmarks cover different categories: (C) compute-intensive, (M) memory-intensive, (R) regular applications, and (I) irregular applications. For graph applications, we use real-world author citation network graphs [54]. For other benchmarks, We use the largest input data sets that are available in the benchmark suites. We use **speedup** as our main metric to measure the performance variations. We define speedup as the ratio of the execution time of the baseline execution to the execution time of our proposed configurations.

3.2 Baseline Configuration

Table 3: Baseline Configuration.

Module	Configuration
GPU config	16 SMs, 1400MHz, 5-stage pipeline
Resource per SM	48KB Shared Memory, 64KB Register File, Max.2048 threads (64 warps, 32 threads/warp) 16 KB, 4-way L1, 12KB 24-way Texture Cache, 8KB 2-way Constant cache, 2KB 4-way L1 I-cache, 128B cacheline
L2 unified cache	128KB/Memory Partition, 1536KB Total Size, 128B cacheline, 8-way associativity
Schedule	Greedy-Then-Oldest (GTO) [51] dual warp schedule Round-Robin (RR) thread block scheduler
TLB Config	L1: 32 entries, 4-way, 1-cycle lookup latency, SM private L2: 512 entries, 16-way, 10-cycle lookup latency, SMs shared
Page table walk	8 shared page table walker, 500-cycle latency

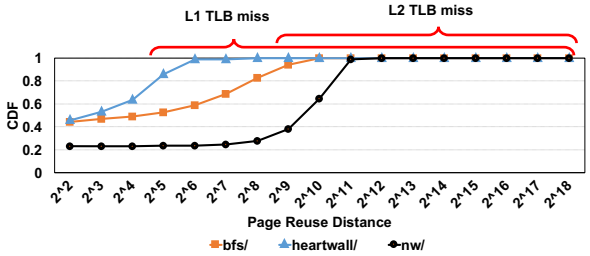

Figure 4: Cumulative distribution function (CDF) of page reuse distances. The x-axis represents the reuse distance in power of 2.

We use a cycle-accurate simulator (gem5-gpu [49]) to conduct our characterizations and later to evaluate our proposed TLB compression mechanism. gem5-gpu is a simulation environment featured with UVM between CPU host and GPU device. We modify the simulator to model the modern NVIDIA Kepler architecture. Table 3 shows the detailed baseline configuration of our simulation environment. For the organization and timing latency of TLBs and page table walker that are not publicly available, we adopt the configuration parameters from the recently published works [5, 6, 50].

3.3 GPU TLB Characterization

Figure 3 plots the TLB hit rates (both L1 and L2) in baseline execution. From these results, one can observe that memory-intensive applications with irregular execution behaviors suffer from low TLB hit rates. This is because, even after memory coalescing, a large number of distinct memory pages are being accessed by the concurrent executing warps, leading to severe TLB thrashing. We further observe that different benchmarks show different hit rates in different TLB components. For example, spmv has high hit rates in both the L1 TLB and L2 TLB, whereas atax has low hit rates in both TLBs. In comparison, bh exhibits a low hit rate at L1 TLB but a high hit rate at the L2 level. In fact, based on the results in this Figure 3, we can classify the benchmarks into four categories: 1) *high L1 and high L2*, 2) *high L1 and low L2*, 3) *low L1 and high L2*, and 4) *low L1 and low L2*. We empirically decide using 75% hit rate as the "threshold" to determine between high hit rate and low hit rate. The TLB column in Table 2 shows our classification.

Figure 4 presents the *page reuse distances* of three representative benchmarks (bfs, heartwall, and nw). Specifically, bfs is an irregular benchmark (because of the input graph), hearwall is a compute-intensive benchmark, and nw is a memory-intensive benchmark. In this paper, we define the page reuse distance as the number of distinct pages accessed between two accesses to the same page. We also draw the L1 TLB and L2 TLB compulsory misses in the figure. As one can observe, different benchmark programs have different page reuse distances. For those benchmarks with a large number of pages that are reused in large distances, the corresponding TLB hit rates are very low. For instance, most of the pages accessed by nw have large reuse distances (more than 60% of the reuses have a distance more than 512 pages), leading to poor TLB hit rates in both L1 and L2 (as shown in Figure 3). In contrast, heartwall experiences high hit rates in both L1 TLB and L2 because of its short page reuse distances (about 85% page reuses have distances less than 32 pages).

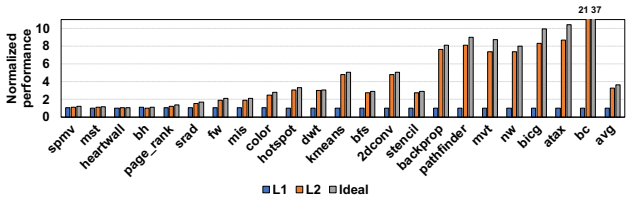

Figure 5: Impact of TLB misses on performance.

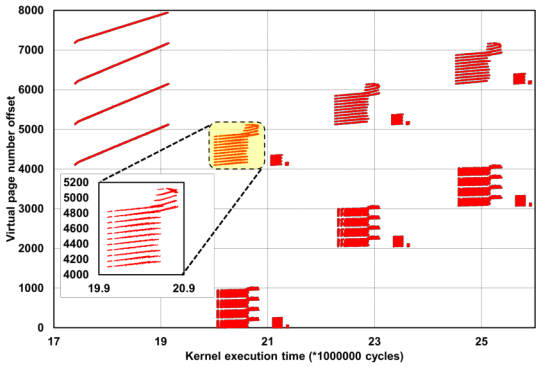

Figure 6: Virtual pages accessed during the execution of benchmark `dwt`. The X-axis shows the timeline, and the Y-axis gives the virtual page number offsets from the page number `0x2aaab6dae`.

To quantify the impact of the L1 and L2 TLB hit rates on performance, we conduct a study to measure performance gains if their miss penalties (latencies) are removed. Figure 5 shows the speedups normalized to baseline execution. Removing both L1 and L2 TLB miss penalty (denoted by 'ideal') is equivalent to having an ideal TLB hierarchy on chip. Removing only one level of TLB miss penalty ('L1' or 'L2') measures their individual impact on performance. Note that, removing the L1 miss penalty does not imply ideal L1 TLB. It just removes the latency for those translation requests that miss in L1 TLB but hit in L2 TLB. However, for those requests misses both L1 and L2 TLBs, we still keep the L2 TLB miss penalty. It can be seen from the figure that, eliminating L1 TLB miss penalties has relatively minor improvement on performance if these misses can still hit in L2 TLB. The major penalty involved in these misses is the on-chip interconnect round trip time (as the requests can hit in L2 TLB), which can be effectively hidden by the GPU warp scheduling.

On the contrary, eliminating L2 TLB miss penalties can significantly improve performance, especially for the memory-intensive and/or irregular benchmarks. We observe speedups ranging from 3.1% to 21.3×, with an average of 3.2× across all benchmark programs. Finally, the ideal TLB hierarchy achieves 6.9% to 37.5× speedup, with an average of 3.5×, demonstrating the importance of TLB behavior to the overall kernel performance.

Considering the results from this study, we can conclude that, there exists a significant potential to improve application performance by improving the TLB hit rates. In particular, optimizing the L2 TLB can significantly improve the overall performance, as the L2 TLB misses involve multi-level page table walks, which are extremely expensive. Therefore, in the rest of this paper, unless otherwise stated, our optimization focuses on the shared L2 TLB.

3.4 TLB Compressibility

Our goal in this paper is to improve TLB hit rates. To achieve the goal, we leverage GPU execution characteristics. That is, during application execution, the memory pages being accessed show clustered patterns periodically. To be more concrete, during certain execution period, the GPU kernel intensively generates memory accesses concentrated on a subset of, and often nearby memory pages. This is because, in the GPU SIMT execution model, there exist spatial data reuses at multiple granularities (e.g., among threads within a warp, among warps within a thread block, and among thread blocks within a kernel). To illustrate such a clustered access pattern, Figure 6 plots the data access pattern – over time – of `dwt` from the Rodinia benchmark suite [15] as an example. In this plot, the x-axis represents the timeline, and the y-axis represents the virtual page number in an offset format (i.e., the page offsets from the virtual page number (0x2aaab6dae)). Each data point in the figure corresponds to one page access that is generated from the GPU load-store unit. As one can observe, there exists a clustered page access pattern during different execution phases. At the beginning phase (cycles 17×10^6 to 19.1×10^6), four data structures are initialized by the initialization kernel in `dwt`. Then, computing kernels are launched sequentially after the initialization. Note that, though each kernel accesses different pages, the accessed pages by a particular kernel are clustered. We also zoom in one kernel execution and show its results. From the figure, we observe that i) at a specific execution cycle, there can be multiple accesses to different pages, and ii) pages being accessed over time show clustered patterns. It is also important to note that, though the pages being accessed are nearby from each other in the address space, the stride between subsequently accessed pages varies and is not necessarily one page. In fact, we observe that the subsequently accessed pages rarely show continuity (stride 1) in GPU execution.

Based on these observations, we leverage the clustered data access patterns to explore TLB entry compression. The key idea behind our approach is to exploit the translation *similarity* where the VPTs and PFNs in the TLB at a certain execution period have a large number of identical bits. These identical bits can be removed so that each TLB entry can accommodate more translations. To explore *TLB compressibility*, we conduct a characterization with the goal of demonstrating the similarity among translations in terms of the number of identical bits in VPNs and PFNs. We achieve this by taking a snapshot of the L2 TLB contents at every 10k cycles, during the execution of a benchmark. In each snapshot, we randomly pick up one TLB entry and use its VPT and PFN as the bases for the remaining TLB entries to calculate the "differences" (δs). For each δ, we determine the number of bits required to represent that δ, and take the average number of bits across all snapshots of that benchmark. Figure 7 shows the collected results. In this plot, the x-axis represents the number bits required to represent a δ, and the y-axis shows the CDF distribution of the number of translations in L2 TLB. In the interest of space, we choose to show 8 representative benchmarks. One can make the following main observations from these results:

Observation 1: Compared to the VPT (31 bits) and the PFN (19 bits) in the baseline case, the number of bits required by δs is significantly less for both VPTs and PFNs across all benchmark

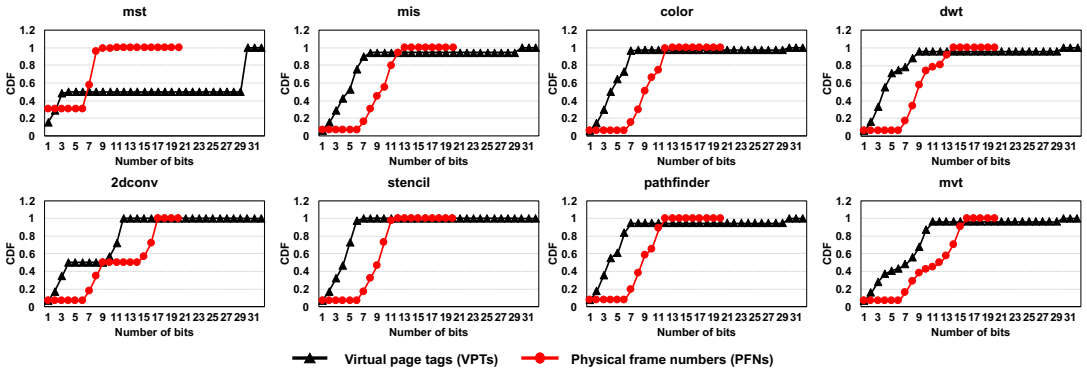

Figure 7: CDF of the average number of translations (both VPTs and PFNs) in L2 TLB (y-axis), whose differences can be represented by x number of bits (x-axis).

programs. For instance, in benchmark `color`, around 80% of VPTs and 46% of PFNs, on average, in L2 TLB have δ values that can be represented using 7 bits and 9 bits, respectively.

Observation 2: There exist uncompressible VPTs or PFNs where the δ values require a similar number of bits compared to original VPT and PFN. For instance, 6% VPTs in `mis` requires 29 bits to represent the δ values.

Observation 3: Applications such as `mst` show interesting results where 45% of VPTs can be compressed and 55% of VPTs are uncompressible. The reason behind this behavior is that multiple warps/thread blocks work on different data structures allocated from different memory segments. This interesting observation motivates us to have a partitioned TLB design, which will be discussed in detail shortly.

4 OUR APPROACH

4.1 Design Challenges

Our goal in this paper is to i) increase TLB hit rates by leveraging the aforementioned TLB compressibility, and ii) design efficient and effective compression and decompression procedures with minimal hardware overheads. To this end, we develop a hardware-supported TLB compression mechanism that takes advantage of TLB compressibility and dynamically merges multiple TLB entries into *fewer* TLB entries. Our approach divides address translations (both VPTs and PFNs) into $\langle$base, $\delta\rangle$ formats, and only maintains the δs in the TLB entries. As a result, each single TLB entry can accommodate δs from multiple translations. Compared to enlarging the TLB capacity which increases both TLB lookup time as well as its on-chip area overhead [53], our approach is more scalable and increases the TLB reach without enlarging the TLB capacity.

However, implementing an effective and efficient dynamic TLB compression mechanism is non-trivial. There are several challenges. First, it is important to select appropriate base addresses such that the compressibility can be maximized (i.e., minimized δ values). The base address should also be dynamically selected to capture the periodic execution patterns. Second, it is crucial to perform TLB compression *dynamically* without introducing significant runtime latencies to the critical paths of data accesses. That is, the compression and decompression procedures, ideally, should overlap with normal TLB operations so that the compression overheads could be hidden. Third, extra TLB misses can occur if all TLB entries

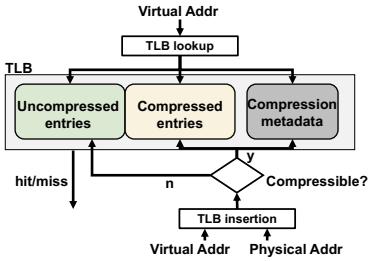

Figure 8: High level overview of our proposed approach.

are made compressible, without any accommodation for uncompressible translations. This is because there are uncompressible translations, as we observed in our characterization (Section 3.4). Therefore, the proposed mechanism needs to handle *both* compressible and uncompressible translations. Finally, the proposed TLB compression should involve minimum hardware overheads and should be much cheaper and more scalable than simply increasing the TLB size.

Motivated by these challenges, we design a hardware-assisted TLB compression mechanism to increase the TLB hit rate by enhancing the TLB reach. Figure 8 shows the high-level view of our approach. Using L2 as an example, the TLB is split by ways into uncompressed partition and compressed partition. The compressed partition is associated with structures to maintain the compression metadata. Upon a TLB lookup, both compressed entries and uncompressed entries are searched simultaneously to check for a hit. If there is a hit in the uncompressed partition, the execution is identical to the baseline. If there is a hit in compressed partition, the decompression procedure retrieves the base from metadata and the δ from TLB entry, to form a valid PFN (Section 4.4). If there is a miss, on the other hand, the request is sent for a page table walk. Upon a TLB insertion, the compression procedure inserts deltas to the compressed TLB entry if the translation is compressible (Section 4.3).

4.2 Compressed Format

Figure 9 depicts the compressed format of both virtual addresses and physical addresses. The figure also shows the contents in a compressed TLB entry. Specifically, the virtual page tag (VPT) is partitioned into an 18-bit virtual page base (VPB), and a 13-bit virtual page delta (VD). The number of bits used to represent δs are

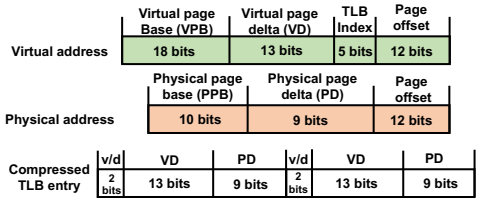

Figure 9: Compressed format of the translation and contents of a compressed TLB entry.

pre-determined by our observations from the characterization study presented earlier in Section 3.4. Specifically, we find that 13 bits are sufficient to represent the δs of clustered pages in most benchmarks. Similarly, the PFN is partitioned into a 10-bit physical page base (PPB) and a 9-bit physical page delta (PD). In the current design, each original TLB entry can hold two δs of compressed translations. That is, our compression ratio is 2. The compressed TLB entry also needs additional bits to maintain the metadata (e.g., valid/dirty bits) for each compressed translation separately. Note that, choosing between different numbers of bits for δs leads to a trade-off between compressibility and compression ratio. Specifically, a large number bits for δs can have more translations compressed but requires more bits to be stored in the TLBs, leading to a low compression ratio. On the other hand, a small number of bits for δs can increase the compression ratio but leads to fewer compressible pages. Later, we provide sensitivity results with different number bits used for δs.

4.3 Compression

Figure 10 shows the TLB compression procedure. Though we use shared L2 TLB for illustrative purposes, the compression procedure is applicable to L1 TLB as well, but we leave its discussion to Section 4.7. The compression metadata structure consists of virtual base registers (VBRs) and physical base registers (PBRs), to maintain the base VPNs and base PFNs. In our current design, *each* TLB set has one corresponding VBR and one PBR that are responsible for the compressed translations in that set. Later, in Section 4.6, we discuss different design options where multiple sets can share the same VBR and PBR. TLB compression happens during TLB insertion when the VPN to PFN mapping is retrieved through a successful page table walk. The virtual page tag and the physical page number are split into ⟨VPB, VD⟩ and ⟨PPB, PD⟩, respectively (①). The TLB index bits in the VPN are used to index the VBRs and PBRs (②a). Note that, the TLB index bits are also used to index the TLB set in parallel (②b). Then, both the VPB and PPB are compared with the corresponding base addresses in VBR and PBR (③). If the address is compressible, that is, if the VPB matches the base address in VBR and the PPB matches the base address in PBR, the translation is inserted into the compressed entry in the particular set indexed by the TLB index bits (④a). Otherwise, the translation is non-compressible and is inserted into the uncompressed entry in that set (④b), where each entry holds only one translation, as in the baseline TLB.

Note that, it can happen that a particular TLB partition (i.e., compressed or uncompressed) is full during TLB insertion, and the eviction of entry is required. To allow eviction on both compressed partition and uncompressed partition, we have separated

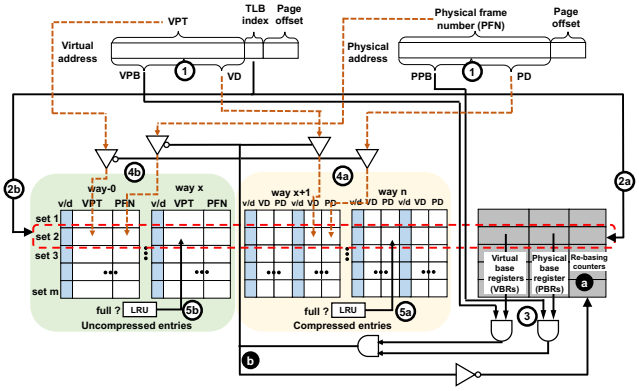

Figure 10: Compression procedure.

the eviction policy (LRU in our case) for compressed partition and uncompressed partition. That is, a compressed translation only triggers the eviction of another compressed translation in the set, and it will not affect the uncompressed partition. The eviction process of either an uncompressed or a compressed entry is similar to an eviction in the baseline, where the LRU algorithm evicts the least recently used entry and uses the slot to store the newly-arrived translation (⑤a and ⑤b).

4.4 Decompression

TLB decompression happens upon TLB lookup and is more critical compared to compression as it is on the critical path of memory access. Therefore, we need to develop an efficient and dynamic decompression mechanism such that it does not add significant extra latencies to data accesses. Figure 11 depicts the required additional hardware support as well as the decompression procedure for L2 TLB. When a translation request is received, the TLB index bits in the VPN are used to index both the TLB entries and the VBRs (⑥). When a set is determined, three lookups/comparisons are conducted in *parallel*. First, the VPB in the VPN is compared with the corresponding base in the VBR (⑦c); second, the VPT is compared with the tags in the uncompressed TLB entries (⑦a); and third, the virtual δ is compared with the δs in the compressed TLB entries (⑦b). After the parallel lookups, several scenarios may occur. First, if the VPT matches the tag in an uncompressed entry, a TLB hit is said to occur, and the uncompressed PFN is retrieved from the TLB entry (⑧a). After that, the PFN is concatenated with the page offset to form a valid physical address (⑨a). Second, if the VPT matching fails but the VPB matches with the base in the VBR, the virtual δ lookup determines whether the address translation hits in the compressed TLB set or not. As we have already looked up the virtual δ in parallel (⑦c), the TLB hit/miss status returns immediately following the VPB and VBR comparison (⑧b). If it is a TLB hit, the PPB is retrieved from the PBR and is concatenated with the physical δ retrieved from the compressed TLB entry to form a valid PFN, and the PFN is concatenated with the page offset to form the requested physical address (⑨b). Third, if the VPT matching fails and the VPB comparison fails or the virtual δ matching fails, a TLB miss occurs in the compressed entries. A TLB miss is handled the same way as in the baseline case which involves a page table walk.

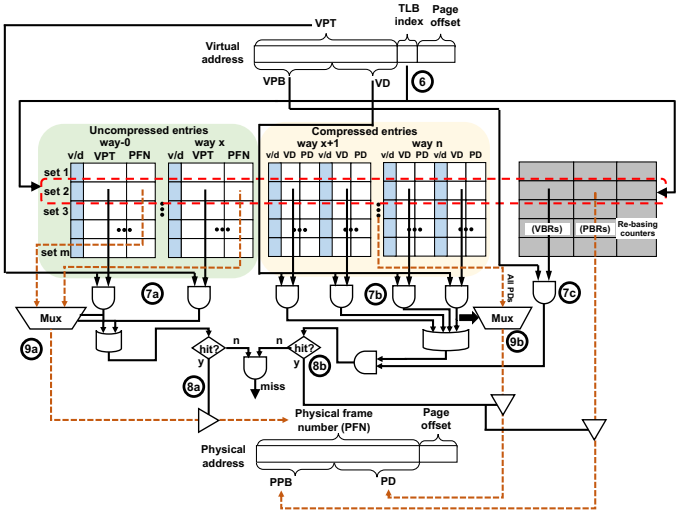

Figure 11: Decompression procedure.

4.5 Re-basing

So far, we have discussed the TLB compression and decompression procedures. However, we left an important question unaddressed: *which address translations should be selected as the base translations and maintained in the VBRs and PBRs?* While a very simple and intuitive approach is to choose the first address translation that is inserted into the TLB set in question, and keep the bases unchanged throughout the entire kernel execution, such approach would not be aware of the variance in clustered page accesses, as we pointed out in our characterization. As a result, it can happen that, after the beginning stage of execution, the majority of translations during the subsequent execution stages are uncompressible and will contend in the uncompressed partition of TLB, while leaving the compressed partition underutilized. In the worst case, this would reduce to a configuration where only half of the TLB capacity is utilized. In fact, we observe that, on average, the L2 hit rate drops 6.1% in naive compression (as shown in the Figure 15 in Section 5.1). Further, for benchmarks with irregular memory accesses, the L2 TLB hit rate drops by more than 25%.

To address this problem, we propose a simple re-basing mechanism, which dynamically changes the bases in VBRs and PBRs. Figure 10 shows the re-basing mechanism integrated into the compression process. Each (VBR, PBR) pair is associated with a re-basing counter (ⓐ), which is initially set to a positive value.[2] Whenever an uncompressed translation occurs, the re-basing counter is decremented by 1 (ⓑ). When the value reaches zero, the next translation will be used as the new base, and the corresponding VBR and PBR will be updated by the new bases. Note that this re-basing strategy will cause all the compressed entries in the TLB set to be flushed, as the new bases cannot be used with the existing δs in the set. Therefore, re-basing is not free, and incurs overheads. It is critical to control the trade-off between re-basing frequency and the associated overheads. A large initial counter value has less re-basing overheads but may incur more TLB misses, whereas a small counter value aggressively captures the variance in access pattern but may suffer from re-basing overheads.

[2]We set the re-basing counter to 16 in our experiments.

4.6 Design Space Exploration

Base selection: The very first ⟨VPB, PPB⟩s are selected using the first translations arrived at each set at the beginning of kernel execution. When re-basing happens, one can potentially i) choose the newly-received address translation, or ii) search for a different base in the uncompressed TLB partition to find the optimal base which gives the minimum number of bits needed to represent the remaining entries in the uncompressed TLB partition. While the latter chooses bases that give smaller δ values, it also introduces the overhead of base searching. In our approach, we adopt the former design for simplicity, and we also found that using the newly-received translation as the base is good enough for all the benchmarks we evaluated.

Shared VBRs and PBRs: In our current design, we employ per-set VBR and PBR. That is, each TLB set has one VPB and one PPB maintained in the VBR and PBR. As a result, the number of base registers required is the same as the number of sets in the TLB. However, one may opt to have different designs. For instance, it is possible to have one VBR and one PBR for the entire TLB compressed partition. The benefit is that it requires less bookkeeping registers, and does not require indexing. However, a uniform base will cause all the compressed entries to be flushed during re-basing. An alternative design is to have multiple sets to share several bases. For example, two TLB sets can share the same ⟨VPB, PPB⟩. As a result, the number of registers required by compression metadata is halved compared to the per-set base design. In general, if the number of the TLB indexing bits is m and x sets share the same base, the lower $m - \log x$ bits will be used to index the VBRs and PBRs (assuming an interleaved indexing). Clearly, determining the degree of sharing (i.e., number of sets that share bases) is a trade-off between the number of registers (hardware overhead) and the re-basing penalty (performance overhead).

Compression ratio: Next, we study different compression ratios. We refer to the compression ratio as the number of compressed translations that can be stored in one (original) TLB entry. In our current design, the compression ratio is two where each original TLB entry can hold two compressed translations (as shown in Figure 9). This is based on our observation in Section 3.4. However, it is possible to compress more translations into a single TLB entry (i.e., a larger compression ratio). Later, in our experiments, we also study a compression ratio of 3, and report the results as well as our observations in Section 5.2.

Size of partitions: Recalling our discussion in Section 3.4, there exist address translations that are uncompressible. If the entire TLB is made to have only compressed entries, all uncompressible translations will miss the TLB. Further, during re-basing, there would not be available translations to use for base selection. Therefore, in our design, we *partition* the TLB into compressed partition and uncompressed partition (as shown in Figure 10). A key question in such partitioning is to determine the size of each partition. We choose to have a (50%, 50%) design where half of the entries are uncompressed and half of them are compressed. The ideal partition should be decided by analyzing the data access pattern of each application, which is not possible in practice. We also study the (75%, 25%) and (25%, 75%) partitions, and the collected results are reported and analyzed in Section 5.2.

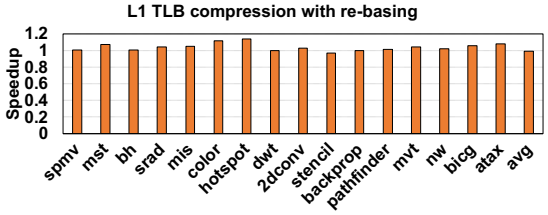

Figure 12: Speedup of L1 compression with re-basing.

4.7 Compression in L1 TLB

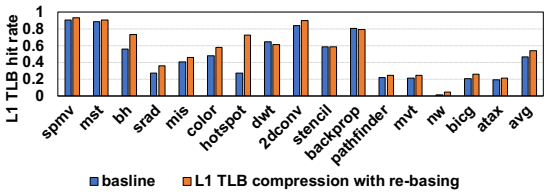

Figure 13: L1 TLB hit rate.

Intuitively, the same TLB compression and decompression mechanisms are applicable to the per-SM L1 TLBs as well, as the L1 TLBs have a similar structure with the L2 TLB. Recall our discussion in Section 3, most benchmarks have incremental improvements when the L1 TLB miss penalties are removed. This is because the interconnect latency is much less compared to the page table walk overhead that the L2 TLB misses experience. Therefore, the GPU warp scheduling is able to hide the latency of L1 TLB misses to some extent. Figure 12 and Figure 13 show performance improvements and L1 TLB hit rates, respectively, when applying L1 TLB compression. It is important to emphasize that, L1 TLBs are more time and area constrained than the L2 TLB in GPUs, as the L1 TLBs are inside each SM and are on the critical path of data accesses to the L1 data cache. Therefore, in the design of L1 TLB compression, instead of having per-set ⟨VBR, PBR⟩, each SM has one ⟨VBR, PBR⟩ for the entire L1 TLB. As one can observe from the results shown in these graphs, our compression scheme does not achieve impressive improvements as in the L2 TLB case. This is because of the small L1 TLB capacity (32 entries) and associativity (4-way). Due to this reason, we can have only 2 compressed entries per TLB set in the (50%, 50%) partitioning. Given the pre-determined compression ratio of 2, a compressed L1 TLB can hold up to 48 translations. However, our characterization of page reuse distances in Figure 4 indicates that the number of pages that has reuse distances between 32 and 48 is very low. However, we want to emphasize that, the L1 TLB compression can be more effective when the capacity and associativity increase in future generations of GPUs.

4.8 Discussion

Hardware overheads: The major hardware overhead brought by our design is the bookkeeping structures required for compression metadata (i.e., VBRs, PBRs, and re-basing counters). Specifically, each VBR is 18 bits and each PBR is 10 bits, and each re-basing counter is 4 bits (for the value of 16). In total, each entry in the compression metadata is 4 bytes (32 bits). Given an L2 TLB with 512 entries and 16-way associativity, the number of sets is 32; therefore, the size of the compression metadata is 128 bytes (4×32 bytes). Our design also requires eight 13-bit comparators to perform TLB lookup in compressed entries, and one 8-bit mux to switch between all PDs. We use shift registers to concatenate the bases with the δs. We use CACTI [64] to estimate the area and power overheads of our approach. The result shows 2.1% area overhead compared to the area of L2 TLB. The dynamic power (including both read and write) and the leakage power increase by 1.8% and 2.5%, respectively, compared to the original L2 TLB. Given the fact that the TLBs contribute to a small portion of the overall system dynamic power (less than 1%) [31], our approach brings negligible overhead to the overall system power.

5 EVALUATION

In this section, we present and discuss the results from the experimental evaluation of our proposed GPU TLB compression. We implement our TLB compression in gem5-gpu and execute it in SE mode. The system configuration (e.g., compute resource, cache capacity, and TLB capacity) is the same as the baseline listed in Table 3. It is to be noted that, all the results we present in this section have been collected from the L2 TLB, since the L1 TLB does not provide promising improvements as we discussed earlier in Section 4.7. We first report the performance improvements and L2 TLB hit rates of the baseline, naive compression without re-basing, and our defended compression with re-basing. We then thoroughly quantify the different design options for compression, as mentioned in Section 4.6, and also compare our approach to a recently published TLB optimization work.

5.1 Overall Performance

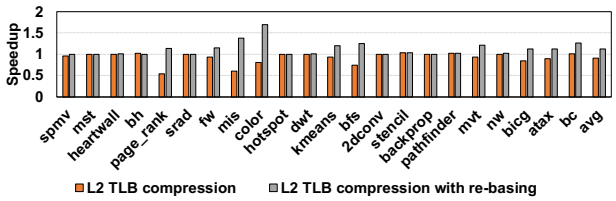

Figure 14: Normalized performance improvements of L2 compression and L2 compression with re-basing.

Figure 14 plots the overall performance improvements brought by our scheme (both with and without re-basing). We use speedup over the baseline as a measure of performance improvement. We also show the corresponding L2 TLB hit rates for all benchmarks in Figure 15. Based on the results presented in these two figures, one can make the following observations. First, the average performance improvement brought by compression with re-basing across all benchmarks is 12%. Specifically, for 16 benchmarks that are memory intensive and/or exhibit irregular memory accesses patterns, our approach improves the performance up to 69.2%, with an average improvement of 16.3%. And, for the remaining 6 benchmarks that are compute-intensive and regular, our approach does not degrade the performance. Second, these performance benefits come mainly from the enhanced L2 TLB hit rates through compression with re-basing. For example, the L2 TLB hit rate of `kmeans` improves by 14.9%, which translates to 19.6% performance improvement. A similar improvement trend can also be observed in applications

such as mis, color, bfs, mvt, and bc. Note that, our scheme benefits irregular applications significantly. This is because, in such applications, L2 TLB entries are frequently evicted before they get the chance to be reused in the baseline execution due to large page reuse distances. Our scheme effectively increases L2 TLB reach, and as a result, the TLB can accommodate more translations and also capture a larger fraction of reused translations. Third, compression without re-basing hurts performance severely, especially in the case of irregular applications. In fact, benchmarks page_rank, mis, and bfs show more that 25% performance degradation. This is because using unchanged base throughout the entire execution prevents us from capturing the variances of clustered page accesses, and causes most address translations contending for the uncompressed TLB entries while leaving the compressed entries *idle*. Fourth, our approach does not affect the applications, such as spmv, srad, and bh, which already have good L2 TLB hit rates. In other words, our TLB partition and compression do *not* degrade the hit rates of these benchmarks and do *not* affect their performance. One reason is that these benchmarks are either compute-intensive (spmv), or they have less page reuses (e.g., a streaming application (srad), or an irregular application with little data reuse (bh)). Another reason is that our design has uncompressed TLB entries for uncompressible translations, and also the parallel lookup in TLB does not introduce significant overhead to compressed translations. Finally, GPU execution can effectively hide most of the address translation latencies for those compute-intensive applications because of the lightweight context switching among warps. That is why the compression does not improve the performance of those applications since a large portion of the TLB miss penalties can be hidden by the overlapping execution with other warps (which is not possible for memory intensive and irregular applications). We also tested a 256 entry L2 TLB (i.e., half-sized TLB compared to our baseline setting), and it shows performance degradation for some of the computation-intensive applications, as more TLB misses occur and the warp scheduling is unable to hide the increasing overheads.

It is important to emphasize that simply enlarging the TLB capacity is not able to yield similar TLB hit rate improvements and performance gains. We conduct experiments with 1024 entry L2 TLB (instead of 512) in baseline execution, and the average L2 TLB hit rate improves by 3.2%, compared to 6.3% brought by our approach without enlarging the TLB capacity. Given the fact that large TLB will also increase the lookup latency, we believe our approach is more scalable for future heterogeneous platforms.

5.2 Different Design Options in TLB Compression

Base selection: As discussed earlier, one can choose to have the newly-received translation as the base or search for the optimal base. In our evaluations so far, we have employed the newly-received base for simplicity. We now discuss choosing a different base and the impact of doing so on application performance. During execution, when re-basing counter decreases to 0, all the compressed entries in the set are flushed/invalidated due to re-basing, leaving only uncompressed TLB entries in the set valid. We search the uncompressed entries for a proper base. This is because a re-basing is only triggered after a number of consecutive uncompressible address translations.

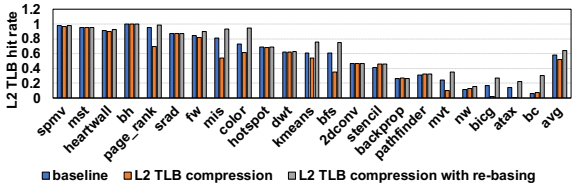

Figure 15: L2 TLB hit rates of L2 compression and L2 compression with re-basing.

These translations reside in the uncompressed TLB entries, and it is highly likely that the subsequent arriving address accesses are clustered with these translations. The bars labeled with "searched base" in Figure 16 and Figure 17 show the performance speedup and L2 hit rate of searching optimal base. As one can observe, both the TLB hit rate and the overall performance differences are negligible (less than 1%), compared to the newly-received base (our default compression scheme). We believe that this is because the newly-received translation is generally compressible to the translations that reside in the uncompressed TLB entries. As a result, the compression efficiency is not affected. Note that, an ideal base would be selected by knowing the application's future page access patterns. One can potentially build learning-models/heuristic-models for that or employ various analytic methods. A detailed exploration of these sophisticated methods is planned in our future work, and is beyond the scope of this paper.

Shared VBRs and PBRs: Next, we explore the option of having multiple TLB sets sharing the same compression metadata. (i.e., VBR, PBR, and re-basing counter). Our goal is to study the tradeoff between the hardware overheads (i.e., the number of VBRs, PBR, and re-basing counters) and the re-basing penalty (i.e., the number of entries that need to be flushed during re-basing). The bars labeled "shared by 2 sets" and "shared by 4 sets" in Figure 16 and Figure 17 plot the performance speedup and L2 hit rate, respectively. Specifically, "shared by 2 sets" represents a setup where two sets share the same base, whereas "shared by 4 sets" represents a setup where four sets share the same base. Note that, the number of registers required to maintain compression metadata is halved in the former compared to our per-set base design, whereas the number of required registers in the latter is further halved. As can be seen from these results, the overall performance of both the designs is about 1% worse, compared to the default case (no shared VBR/PBR). This is because the clustered page access pattern not only exhibits translation similarity within one particular TLB set, but across other sets. However, we want to emphasize that we choose to use per-set compression base as it does not require separate logic to index multiple sets during flushing and incurs less flushing overheads. One alternative design is to have multiple VBRs and PBRs within a given set. This is because some applications that have a small number of pages in different clusters will occupy the compressed sets quickly, and other pages will have to occupy the uncompressed entries, leading to the under-utilization of the compressed entries. Having multiple bases within a set can potentially improve the TLB utilization of applications with small clusters. However, the under-utilization is rare for most of the memory-intensive applications as i) they generally have a large number of pages in each cluster, and ii) our re-basing scheme can quickly detect and re-base the corresponding sets. Moreover, having multiple VBRs and PBRs

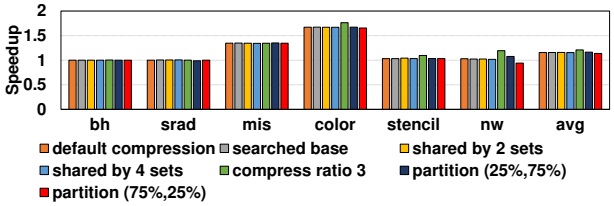

Figure 16: Normalized performance improvements of different design options

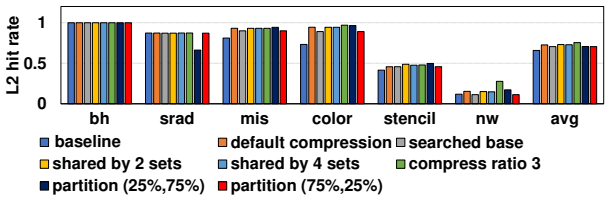

Figure 17: L2 TLB hit rate of different design options

within a set involves overhead of another level of indexing the bases within a set.

Compression ratio: Our discussion so far has employed a compression ratio of 2. That is, each original TLB entry in the compressed partition is restructured to hold two compressed translations. In this part of our experimental evaluation, we study performance and TLB hit rate when employing a compression ratio of 3, where each TLB entry in the compressed partition stores 3 compressed ⟨VD, PD⟩ pairs. The bars ("compress ratio 3") in Figure 16 and Figure 17 plot the corresponding results. We observe that the average performance improves by 5.2%, and the hit rate increases by 2.8%. In particular, for `nw`, the L2 hit rate improves by 12.3%. This is because 99% of the virtual deltas of `nw` can be represented using less than 10 bits in our characterization. Note that, one can potentially choose even larger compression ratios. However, a large compression ratio leads to less number of bits to represent the deltas without changing the TLB size. Therefore, more uncompressible translations may occur and thrash the uncompressed TLB partition.

Size of partition: We next vary the sizes of the compressed and uncompressed TLB partitions and collect performance results. Recall that, in our previous experiments, we used a (50%,50%) partition. In Figure 17, the bar labeled with "partition (75%,25%)" shows the L2 hit rate observed when 3/4 TLB entries are uncompressed. Similarly, the bar "partition (25%,75%)" indicates the hit rate observed when 1/4 TLB entries are uncompressed. Overall, we observe that, different partition sizes can affect the application L2 TLB hit rate and performance differently. For instance. the L2 hit rate of `srad` drops by 20.1% in (25%,75%) due to the fact that `srad` has large amount of uncompressible address translations. In contrast, `nw` prefers (75%,25%) configuration where the hit rate is higher compared to (25%,75%) configuration. This is because `nw` has more compressible translations than uncompressible translations during execution.

5.3 Comparison to an Alternative TLB Optimization

We quantitatively compare our proposed TLB compression mechanism with a recently published range TLB optimization [69] to

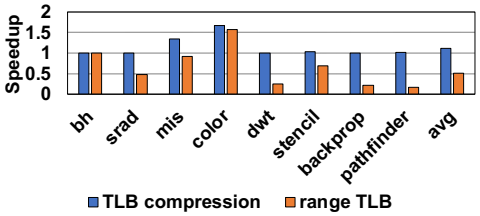

Figure 18: Comparison to an alternative TLB optimization scheme.

demonstrate that TLB compression is more effective and suitable in GPU executions. Our scheme can capture non-continuous page accesses and does not rely on the OS to generate continuous physical pages as in range TLB. Figure 18 compares the performance of our proposed approach to range TLB (the results for both the schemes are *normalized* with respect to the baseline). It can be observed that, our approach performs, on average, 49.6% than range TLB for irregular applications. The main reason is that range TLB requires the OS to generate continuous physical pages which is extremely expensive in UVM in CPU-GPU systems.

5.4 Using Large Pages

One can potentially adopt large pages to increase the TLB reach, and ultimately the TLB hit rates [9, 25]. We quantitatively evaluated the impact of adopting large pages. Figure 19 and Figure20 show the L2 TLB hit rates and the speedup for all 22 benchmarks when 2MB page is used. One can make the following observations. First, most of the benchmarks show a significant increase in L2 TLB hit rates and performance when using the 2MB page compared to the 4KB page results given in Figures 15 and 14. This indicates the benefits of adopting large pages and is consistent with a previous study [50]. Second, our proposed compression is not restricted to particular page sizes. Specifically, with the 2MB page, our compression can further improve the L2 TLB hit rate with an average of 3.3%, which translates to an average of 7.4% performance improvement. Third, for benchmarks `color` and `mst`, our approach on 4KB page has a better TLB hit rate than 2MB page without compression. The reason behind this is the large stride in data accesses caused by irregular and unstructured inputs. This leads to very poor page utilization during the period when the translation is valid in TLB. However, our compression approach can provide benefits for large stride accesses since we use different bases for different sets.

It is important to note that large pages can hurt performance in GPUs, compared to using small pages [6, 9, 16]. The reason behind this is that large pages increase the overheads tremendously in CPU-GPU unified virtual memory. Specifically, the paging traffic between CPU and GPU is severe for 2MB page compared to 4KB page [6], and a TLB miss on large page stalls many more GPU warps than a TLB miss on a small page [5]. In addition, using large pages increases intra-page fragmentation, especially for irregular GPU applications. Unfortunately, the benchmarks we evaluated in this paper do not have frequent interleaved accesses from both the CPU side and the GPU side. The GPU does most of the computations in these benchmarks.

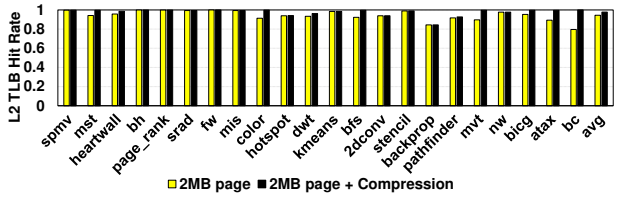

Figure 19: L2 TLB hit rates when adopting 2MB pages and our proposed compression with re-basing.

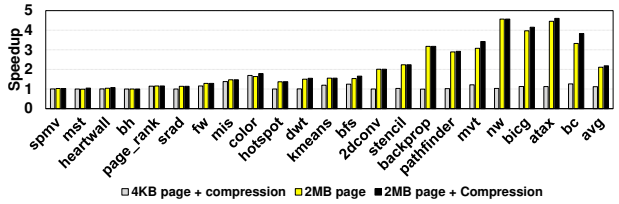

Figure 20: Speedup of 2MB page with proposed compression (normalized to 4KB page baseline execution).

6 RELATED WORK

Data access optimizations: Prior works have focused on UVM optimizations, data locality optimizations, and data parallelism optimizations [17, 19, 22, 24, 27, 38–40, 47, 48, 58–63, 70, 71, 71]. Power et al. [50] explored the GPU MMU design, and developed a x86 compatible GPU TLB hierarchy and a page table walker. Ausavarungnirun et al. [5] proposed support for multiple page sizes. Pichai et al. [46] explored TLB-aware warp scheduling to improve the L1 TLB parallel accessing performance. Hao et al. [21] proposed a two-level TLB hierarchy support UVM between the host CPU cores and customized accelerators. Agarwal et al. [1] conducted a detailed characterization of the tradeoffs between hardware cache-coherence mechanism and software page migration. Instead of hardware-supported address translation, Shahar et al. [55] implemented a software level address translation. Vesely et al. [65] characterized the performance of UVM and pointed out that the TLB misses in GPUs are significantly more expansive compared to their counterparts in CPUs. Compared with all these prior efforts on UVM, our approach is the first work to explore the translation compressibility in a CPU-GPU based UVM environment. Our approach is built upon the unique execution characteristics of GPU and effectively increases the TLB reach with minimal hardware overhead. Meanwhile, our approach is complementary to most prior works (e.g., page table walk optimization for irregular applications [56]) and can be combined with them to further improve the UVM performance.

Address translation optimizations: There exists a substantial body of research works, both from the OS community and the architecture community, focusing on address translation optimizations [2, 7, 8, 12, 29, 37, 42]. Vogel et al. [67] investigated a software-hardware codesign strategy to enable virtual memory support for accelerators in heterogeneous SoC systems. Vesely et al. [66] proposed a generic GPU system call interface for Linux, which allows GPUs to initial system calls. Bharadwaj et al. [10] studied distributed TLB slicing and corresponding network topologies, to accelerate address translation. Yan et al. [69] proposed translation ranger, an OS support, to enable continuous page accesses such that the TLB can store fewer translations. Pham et al. [43] proposed a Bloom filter-based hardware mechanism that can be used to reduce the overheads imposed by cache flushes due to virtual page remappings. Shin et al. [56] explored various critical warp-aware page table walking strategies to accelerate irregular application address translations. Margaritov et al. [32] proposed parallel translation prefetching to avoid multiple levels of sequential page table walks in CPUs. Cox et al. [16] invented MIX TLBs, which supports (concurrently) multiple page sizes by exploiting superpage allocation patterns. Compared to these prior works, our work neither requires continuity in subsequent accessed pages, nor expensive OS handling of page rearrangement. Instead, we propose a hardware-supported dynamic compression mechanism that captures the clustered page access pattern in application programs running on GPUs. Our approach is programmer-transparent and involves minimal hardware overhead. Furthermore, our approach does not require any runtime physical page rearrangement, which is very costly in UVM.

7 CONCLUDING REMARKS

Targeting unified virtual memory, which is becoming prevalent in state-of-the-art CPU-GPU systems, in this paper, we propose and experimentally evaluate simple yet efficient compression and decompression mechanisms that improve TLB hit rates in GPUs. Specifically, we explore TLB compressibility, based on the observations we make from our detailed analysis of page reuse distances and clustered page access patterns. We then enhance the GPU TLB hardware, to make it work with compressed TLB entries. Experimental results collected using benchmarks from various application domains indicate that, our proposed compression-based approach significantly improves TLB hit rates and overall application performance.

ACKNOWLEDGEMENT

The authors thank PACT reviewers and shepherd for their constructive feedback and suggestion for improving this paper. The authors also thank John Morgan Sampson and Jagadish Kotra for their involvement in early discussions relevant to this work. This material is based upon work supported by the National Science Foundation under grants #1763681, #1629129, #1931531, #1629915, and is supported by a startup grant from the University of Pittsburgh.

REFERENCES

[1] N. Agarwal, D. Nellans, M. O'Connor, S. W. Keckler, and T. F. Wenisch. 2015. Unlocking bandwidth for GPUs in CC-NUMA systems. In *2015 IEEE 21st International Symposium on High Performance Computer Architecture (HPCA)*. 354–365. https://doi.org/10.1109/HPCA.2015.7056046

[2] Hanna Alam, Tianhao Zhang, Mattan Erez, and Yoav Etsion. 2017. Do-It-Yourself Virtual Memory Translation. In *Proceedings of the 44th Annual International Symposium on Computer Architecture* (Toronto, ON, Canada) *(ISCA '17)*. ACM, New York, NY, USA, 457–468. https://doi.org/10.1145/3079856.3080209

[3] AMD Corp. 2016. *I/O Virtualization Technology(IOMMU) Specification.* https://www.amd.com/system/files/TechDocs/48882_IOMMU.pdf

[4] AMD Corp. 2017. *Radeons Next-generation Vega Architecture.* https://radeon.com/_downloads/vega-whitepaper-11.6.17.pdf

[5] 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)*. 136–150.

[6] 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* (Williamsburg, VA, USA) *(ASPLOS '18)*. ACM, New York, NY, USA, 503–518. https://doi.org/10.1145/3173162.3173169

[7] 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* (Saint-Malo, France) *(ISCA '10)*. ACM, New York, NY, USA, 48–59. https://doi.org/10.1145/1815961.1815970

[8] T. W. Barr, A. L. Cox, and S. Rixner. 2011. SpecTLB: A mechanism for speculative address translation. In *2011 38th Annual International Symposium on Computer Architecture (ISCA)*. 307–317.

[9] Arkaprava Basu, Jayneel Gandhi, Jichuan Chang, Mark D. Hill, and Michael M. Swift. 2013. Efficient Virtual Memory for Big Memory Servers. In *Proceedings of the 40th Annual International Symposium on Computer Architecture* (Tel-Aviv, Israel) *(ISCA '13)*. ACM, New York, NY, USA, 237–248. https://doi.org/10.1145/2485922.2485943

[10] S. Bharadwaj, G. Cox, T. Krishna, and A. Bhattacharjee. 2018. Scalable Distributed Last-Level TLBs Using Low-Latency Interconnects. In *2018 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. 271–284. https://doi.org/10.1109/MICRO.2018.00030

[11] Abhishek Bhattacharjee. 2013. Large-reach Memory Management Unit Caches. In *Proceedings of the 46th Annual IEEE/ACM International Symposium on Microarchitecture* (Davis, California) *(MICRO-46)*. ACM, New York, NY, USA, 383–394. https://doi.org/10.1145/2540708.2540741

[12] Abhishek Bhattacharjee. 2017. Translation-Triggered Prefetching. In *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems* (Xi'an, China) *(ASPLOS '17)*. ACM, New York, NY, USA, 63–76. https://doi.org/10.1145/3037697.3037705

[13] Luca Caucci and Lars R. Furenlid. 2015. GPU programming for biomedical imaging. In *Medical Applications of Radiation Detectors V*, H. Bradford Barber, Lars R. Furenlid, and Hans N. Roehrig (Eds.), Vol. 9594. International Society for Optics and Photonics, SPIE, 79 – 93. https://doi.org/10.1117/12.2195217

[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)*. 185–195. https://doi.org/10.1109/IISWC.2013.6704684

[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)*. 44–54. https://doi.org/10.1109/IISWC.2009.5306797

[16] Guilherme Cox and Abhishek Bhattacharjee. 2017. 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* (Xi'an, China) *(ASPLOS '17)*. ACM, New York, NY, USA, 435–448. https://doi.org/10.1145/3037697.3037704

[17] Wei Ding, Xulong Tang, Mahmut Kandemir, Yuanrui Zhang, and Emre Kultursay. 2015. Optimizing Off-chip Accesses in Multicores. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*.

[18] Z. Du, R. Fasthuber, T. Chen, P. Ienne, L. Li, T. Luo, X. Feng, Y. Chen, and O. Temam. 2015. ShiDianNao: Shifting vision processing closer to the sensor. In *2015 ACM/IEEE 42nd Annual International Symposium on Computer Architecture (ISCA)*. 92–104. https://doi.org/10.1145/2749460779.2750389

[19] Debashis Ganguly, Ziyu Zhang, Jun Yang, and Rami Melhem. 2019. Interplay Between Hardware Prefetcher and Page Eviction Policy in CPU-GPU Unified Virtual Memory. In *Proceedings of the 46th International Symposium on Computer Architecture* (Phoenix, Arizona) *(ISCA '19)*. ACM, New York, NY, USA, 224–235. https://doi.org/10.1145/3307650.3322224

[20] S. Grauer-Gray, L. Xu, R. Searles, S. Ayalasomayajula, and J. Cavazos. 2012. Auto-tuning a high-level language targeted to GPU codes. In *2012 Innovative Parallel Computing (InPar)*. 1–10. https://doi.org/10.1109/InPar.2012.6339595

[21] Y. Hao, Z. Fang, G. Reinman, and J. Cong. 2017. Supporting Address Translation for Accelerator-Centric Architectures. In *2017 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. 37–48. https://doi.org/10.1109/HPCA.2017.19

[22] Swapnil Haria, Mark D. Hill, and Michael M. Swift. 2018. Devirtualizing Memory in Heterogeneous Systems. In *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems* (Williamsburg, VA, USA) *(ASPLOS '18)*. ACM, New York, NY, USA, 637–650. https://doi.org/10.1145/3173162.3173194

[23] Timothy D.R. Hartley, Umit Catalyurek, Antonio Ruiz, Francisco Igual, Rafael Mayo, and Manuel Ujaldon. 2014. Biomedical Image Analysis on a Cooperative Cluster of GPUs and Multicores. In *ACM International Conference on Supercomputing 25th Anniversary Volume* (Munich, Germany). ACM, New York, NY, USA, 413–423. https://doi.org/10.1145/2591635.2667189

[24] Mahmut Kandemir, Hui Zhao, Xulong Tang, and Mustafa Karakoy. 2015. Memory Row Reuse Distance and Its Role in Optimizing Application Performance. In *Proceedings of the 2015 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS)*.

[25] Vasileios Karakostas, Jayneel Gandhi, Furkan Ayar, Adrián Cristal, Mark D. Hill, Kathryn S. McKinley, Mario Nemirovsky, Michael M. Swift, and Osman Ünsal. 2015. Redundant Memory Mappings for Fast Access to Large Memories. In *Proceedings of the 42Nd Annual International Symposium on Computer Architecture* (Portland, Oregon) *(ISCA '15)*. ACM, New York, NY, USA, 66–78. https://doi.org/10.1145/2749469.2749471

[26] Jens Kehne, Jonathan Metter, and Frank Bellosa. 2015. GPUswap: Enabling Oversubscription of GPU Memory Through Transparent Swapping. In *Proceedings of the 11th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments* (Istanbul, Turkey) *(VEE '15)*. ACM, New York, NY, USA, 65–77. https://doi.org/10.1145/2731186.2731192

[27] Orhan Kislal, Jagadish Kotra, Xulong Tang, Mahmut Taylan Kandemir, and Myoungsoo Jung. 2018. Enhancing Computation-to-core Assignment with Physical Location Information. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*.

[28] M. Kulkarni, M. Burtscher, C. Cascaval, and K. Pingali. 2009. Lonestar: A suite of parallel irregular programs. In *2009 IEEE International Symposium on Performance Analysis of Systems and Software*. 65–76. https://doi.org/10.1109/ISPASS.2009.4919639

[29] Mohan Kumar Kumar, Steffen Maass, Sanidhya Kashyap, Ján Veselý, Zi Yan, Taesoo Kim, Abhishek Bhattacharjee, and Tushar Krishna. 2018. LATR: Lazy Translation Coherence. In *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems* (Williamsburg, VA, USA) *(ASPLOS '18)*. ACM, New York, NY, USA, 651–664. https://doi.org/10.1145/3173162.3173198

[30] 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* (Savannah, GA, USA) *(OSDI'16)*. USENIX Association, Berkeley, CA, USA, 705–721. http://dl.acm.org/citation.cfm?id=3026877.3026931

[31] Jieun Lim, Nagesh B. Lakshminarayana, Hyesoon Kim, William Song, Sudhakar Yalamanchili, and Wonyong Sung. 2014. Power Modeling for GPU Architectures Using McPAT. *ACM Trans. Des. Autom. Electron. Syst.* 19, 3, Article 26 (June 2014), 24 pages. https://doi.org/10.1145/2611758

[32] Artemiy Margaritov, Dmitrii Ustiugov, Edouard Bugnion, and Boris Grot. 2019. Prefetched Address Translation. In *Proceedings of the 52Nd Annual IEEE/ACM International Symposium on Microarchitecture* (Columbus, OH, USA) *(MICRO '52)*. ACM, New York, NY, USA, 1023–1036. https://doi.org/10.1145/3352460.3358294

[33] Sparsh Mittal and Jeffrey S. Vetter. 2015. A Survey of CPU-GPU Heterogeneous Computing Techniques. *ACM Comput. Surv.* 47, 4, Article 69 (July 2015), 35 pages. https://doi.org/10.1145/2788396

[34] NVIDIA Corp. 2016. *NVIDIA Tesla P100*. https://images.nvidia.com/content/pdf/tesla/whitepaper/pascal-architecture-whitepaper.pdf

[35] NVIDIA Corp. 2018. *NVIDIA Pascal Architecture*. https://www.nvidia.com/en-us/data-center/pascal-gpu-architecture/

[36] M. Parasar, A. Bhattacharjee, and T. Krishna. 2018. SEESAW: Using Superpages to Improve VIPT Caches. In *2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA)*. 193–206.

[37] C. H. Park, T. Heo, J. Jeong, and J. Huh. 2017. Hybrid TLB coalescing: Improving TLB translation coverage under diverse fragmented memory allocations. In *2017 ACM/IEEE 44th Annual International Symposium on Computer Architecture (ISCA)*. 444–456. https://doi.org/10.1145/3079856.3080217

[38] E. Park, J. Ahn, S. Hong, S. Yoo, and S. Lee. 2015. Memory fast-forward: A low cost special function unit to enhance energy efficiency in GPU for big data processing. In *2015 Design, Automation Test in Europe Conference Exhibition (DATE)*. 1341–1346.

[39] Ashutosh Pattnaik, Xulong Tang, Adwait Jog, Onur Kayiran, Asit K. Mishra, Mahmut T. Kandemir, Onur Mutlu, and Chita R. Das. 2016. Scheduling Techniques for GPU Architectures with Processing-In-Memory Capabilities. In *Proceedings of the 2016 International Conference on Parallel Architectures and Compilation (PACT)*.

[40] Ashutosh Pattnaik, Xulong Tang, Onur Kayiran, Adwait Jog, Asit Mishra, Mahmut T. Kandemir, Anand Sivasubramaniam, and Chita R. Das. 2019. Opportunistic Computing in GPU Architectures. In *Proceedings of the 46th International Symposium on Computer Architecture*.

[41] Gennady Pekhimenko, Vivek Seshadri, Onur Mutlu, Michael A Kozuch, Phillip B Gibbons, and Todd C Mowry. 2012. Base-delta-immediate compression: Practical data compression for on-chip caches. In *2012 21st International Conference on Parallel Architectures and Compilation Techniques (PACT)*. IEEE, 377–388.

[42] B. Pham, A. Bhattacharjee, Y. Eckert, and G. H. Loh. 2014. Increasing TLB reach by exploiting clustering in page translations. In *2014 IEEE 20th International Symposium on High Performance Computer Architecture (HPCA)*. 558–567. https://doi.org/10.1109/HPCA.2014.6835964

[43] Binh Pham, Derek Hower, Abhishek Bhattacharjee, and Trey Cain. 2018. TLB Shootdown Mitigation for Low-Power Many-Core Servers with L1 Virtual Caches. *IEEE Comput. Archit. Lett.* 17, 1 (Jan. 2018), 17–20. https://doi.org/10.1109/LCA.

2017.2712140

[44] Binh Pham, Viswanathan Vaidyanathan, Aamer Jaleel, and Abhishek Bhattacharjee. 2012. CoLT: Coalesced Large-Reach TLBs. In *Proceedings of the 2012 45th Annual IEEE/ACM International Symposium on Microarchitecture* (Vancouver, B.C., CANADA) *(MICRO-45)*. IEEE Computer Society, USA, 258–269. https://doi.org/10.1109/MICRO.2012.32

[45] B. Pham, J. Veselý, G. H. Loh, and A. Bhattacharjee. 2015. Large pages and lightweight memory management in virtualized environments: Can you have it both ways?. In *2015 48th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. 1–12. https://doi.org/10.1145/2830772.2830773

[46] 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* (Salt Lake City, Utah, USA) *(ASPLOS '14)*. ACM, New York, NY, USA, 743–758. https://doi.org/10.1145/2541940.2541942

[47] B. Pichai, L. Hsu, and A. Bhattacharjee. 2015. Address Translation for Throughput-Oriented Accelerators. *IEEE Micro* 35, 3 (May 2015), 102–113. https://doi.org/10.1109/MM.2015.44

[48] J. Picorel, D. Jevdjic, and B. Falsafi. 2017. Near-Memory Address Translation. In *2017 26th International Conference on Parallel Architectures and Compilation Techniques (PACT)*. 303–317. https://doi.org/10.1109/PACT.2017.56

[49] J. Power, J. Hestness, M. S. Orr, M. D. Hill, and D. A. Wood. 2015. gem5-gpu: A Heterogeneous CPU-GPU Simulator. *IEEE Computer Architecture Letters* 14, 1 (Jan 2015), 34–36. https://doi.org/10.1109/LCA.2014.2299539

[50] 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

[51] Timothy G. Rogers, Mike O'Connor, and Tor M. Aamodt. 2012. Cache-Conscious Wavefront Scheduling. In *MICRO*.

[52] Jihyun Ryoo, Mengran Fan, Xulong Tang, Huaipan Jiang, Meena Arunachalam, Sharada Naveen, and Mahmut T Kandemir. 2019. Architecture-Centric Bottleneck Analysis for Deep Neural Network Applications. In *2019 IEEE 26th International Conference on High Performance Computing, Data, and Analytics (HiPC)*. IEEE, 205–214.

[53] R. Samanta, J. Surprise, and R. Mahapatr. 2008. Dynamic Aggregation of Virtual Addresses in TLB Using TCAM Cells. In *21st International Conference on VLSI Design (VLSID 2008)*. 243–248. https://doi.org/10.1109/VLSI.2008.57

[54] Peter Sanders and Christian Schulz. 2012. 10th Dimacs Implementation Challenge-Graph Partitioning and Graph Clustering. (2012).

[55] S. Shahar, S. Bergman, and M. Silberstein. 2016. ActivePointers: A Case for Software Address Translation on GPUs. In *2016 ACM/IEEE 43rd Annual International Symposium on Computer Architecture (ISCA)*. 596–608. https://doi.org/10.1109/ISCA.2016.58

[56] 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)*. 180–192. https://doi.org/10.1109/ISCA.2018.00025

[57] John A Stratton, Christopher Rodrigues, I-Jui Sung, Nady Obeid, Li-Wen Chang, Nasser Anssari, Geng Daniel Liu, and Wen-mei W Hwu. 2012. Parboil: A revised benchmark suite for scientific and commercial throughput computing. *Center for Reliable and High-Performance Computing* 127 (2012).

[58] Xulong Tang, Mahmut Kandemir, Praveen Yedlapalli, and Jagadish Kotra. 2016. Improving Bank-Level Parallelism for Irregular Applications. In *Proceedings of the 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*.

[59] Xulong Tang, Mahmut Taylan Kandemir, Hui Zhao, Myoungsoo Jung, and Mustafa Karakoy. 2019. Computing with Near Data. In *Proceedings of the 2019 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS)*.

[60] Xulong Tang, Orhan Kislal, Mahmut Kandemir, and Mustafa Karakoy. 2017. Data Movement Aware Computation Partitioning. In *Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*.

[61] Xulong Tang, Ashutosh Pattnaik, Huaipan Jiang, Onur Kayiran, Adwait Jog, Sreepathi Pai, Mohamed Ibrahim, Mahmut Kandemir, and Chita Das. 2017. Controlled Kernel Launch for Dynamic Parallelism in GPUs. In *Proceedings of the 23rd International Symposium on High-Performance Computer Architecture (HPCA)*.

[62] Xulong Tang, Ashutosh Pattnaik, Onur Kayiran, Adwait Jog, Mahmut Taylan Kandemir, and Chita R. Das. 2019. Quantifying Data Locality in Dynamic Parallelism in GPUs. In *Proceedings of the 2019 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS)*.

[63] Xulong Tang, Mahmut Taylan Kandemir, Mustafa Karakoy, and Meena Arunachalam. 2019. Co-Optimizing Memory-Level Parallelism and Cache-Level Parallelism. In *Proceedings of the 40th annual ACM SIGPLAN conference on Programming Language Design and Implementation*.

[64] S. Thoziyoor, J. H. Ahn, M. Monchiero, J. B. Brockman, and N. P. Jouppi. 2008. A Comprehensive Memory Modeling Tool and Its Application to the Design and Analysis of Future Memory Hierarchies. In *2008 International Symposium on Computer Architecture*. 51–62. https://doi.org/10.1109/ISCA.2008.16

[65] 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)*. 161–171. https://doi.org/10.1109/ISPASS.2016.7482091

[66] J. Veselý, A. Basu, A. Bhattacharjee, G. H. Loh, M. Oskin, and S. K. Reinhardt. 2018. Generic System Calls for GPUs. In *2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA)*. 843–856. https://doi.org/10.1109/ISCA.2018.00075

[67] P. Vogel, A. Marongiu, and L. Benini. 2015. Lightweight virtual memory support for many-core accelerators in heterogeneous embedded SoCs. In *2015 International Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS)*. 45–54. https://doi.org/10.1109/CODESISSS.2015.7331367

[68] Linnan Wang, Jinmian Ye, Yiyang Zhao, Wei Wu, Ang Li, Shuaiwen Leon Song, Zenglin Xu, and Tim Kraska. 2018. Superneurons: Dynamic GPU Memory Management for Training Deep Neural Networks. In *Proceedings of the 23rd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming* (Vienna, Austria) *(PPoPP '18)*. ACM, New York, NY, USA, 41–53. https://doi.org/10.1145/3178487.3178491

[69] Zi Yan, Daniel Lustig, David Nellans, and Abhishek Bhattacharjee. 2019. Translation Ranger: Operating System Support for Contiguity-aware TLBs. In *Proceedings of the 46th International Symposium on Computer Architecture* (Phoenix, Arizona) *(ISCA '19)*. ACM, New York, NY, USA, 698–710. https://doi.org/10.1145/3307650.3322223

[70] S. Zhang, Y. Yang, L. Shen, and Z. Wang. 2018. Efficient Data Communication between CPU and GPU through Transparent Partial-Page Migration. In *2018 IEEE 20th International Conference on High Performance Computing and Communications; IEEE 16th International Conference on Smart City; IEEE 4th International Conference on Data Science and Systems (HPCC/SmartCity/DSS)*. 618–625. https://doi.org/10.1109/HPCC/SmartCity/DSS.2018.00112

[71] T. Zheng, D. Nellans, A. Zulfiqar, M. Stephenson, and S. W. Keckler. 2016. Towards high performance paged memory for GPUs. In *2016 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. 345–357. https://doi.org/10.1109/HPCA.2016.7446077