Orchestrated Scheduling and Partitioning for Improved Address Translation in GPUs

Bingyao Li, Yueqi Wang, Xulong Tang
Department of Computer Science, University of Pittsburgh, Pittsburgh, USA
Email: {bil35, yuw249, tax6}@pitt.edu

***Abstract*—Unified Virtual Memory (UVM) is a promising feature in CPU-GPU heterogeneous systems that allows data structures to be accessed by both CPU and GPUs through unified pointers without explicit data copying. However, the delivered performance of UVM significantly relies on the efficiency of address translation. The current GPU thread block (TB) management is not aware of the translation process and heavily thrashes the per-streaming multiprocessor (SM) private Translation Look-ahead Buffers (TLBs). In this paper, we conduct a comprehensive characterization of 10 GPU benchmarks and quantify the translation reuses among the thread blocks. Our observation reveals that there exists substantial translation reuse within TBs rather than across the TBs. Moreover, the inter-TB interference significantly enlarges the intra-TB translation reuse distances. To this end, we propose a translation-aware TB scheduling and lightweight GPU L1 TLB partitioning to effectively mitigate the contention. Experimental results show that our proposed approach improves the L1 TLB hit rate, and this improvement translates to, on average, a 12.5% execution time reduction.**

***Index Terms*—UVM, GPU, TLB**

I. Introduction

Graphics Processing Units (GPUs) are widely deployed in modern computing systems to provide acceleration for a wide range of applications, including machine learning, computer vision, social networks, and entertainment. However, traditional GPU execution requires significant programmer effort to explicitly manage the data and computation between CPUs and GPUs. Meanwhile, the limited GPU memory capacity prevents the deployment of applications with large memory footprints that exceed the GPU memory capacity. As a result, GPUs are difficult to employ and manage for complicated modern applications with large memory footprints. Recently, GPU vendors have proposed unified virtual memory (UVM) with demand paging to ease GPU programming. This feature is especially beneficial for executing complex applications whose memory footprints exceed the modern GPU memory capacity.

While UVM is promising, it introduces an expensive address translation process, which comprises hierarchical TLB lookup and multilevel page table walk. Prior works have investigated TLB optimizations from different aspects, and we summarize these related works in Table I. However, these techniques are unsuitable for GPU per-SM private L1 TLBs and fail to leverage the translation reuses across thread blocks. First, TLB clustering, range TLB, and eager paging rely on continuous and regular page accesses to improve the TLB reach. That is, these techniques rely on linear and stride memory page accesses to merge the translation requests into fewer TLB entries. However, these linear and stride memory access patterns are not always observable in general-purpose applications dealing with irregular data structures (e.g., graph processing applications). Second, while employing huge pages can enlarge the TLB reach, it suffers from severe intra-page fragmentation [1], [2]. We also evaluate our approach with huge pages in Section V. Third, speculative TLB relies on predictable access patterns to ensure prediction accuracy. Finally, while TLB probe and least-TLB work for irregular access patterns, they are not suitable for GPU L1 TLB due to L1 TLB being on the execution critical path. The overheads in the least-TLB and TLB probes can significantly degrade the execution performance. Moreover, none of the prior works have investigated the translation reuses at TB granularity and addressed the question of how to coordinate TLB designs and TB management to leverage the translation reuses.

TABLE I
Comparison with prior techniques.

Techniques	Irregular Access	No internal fragmentation	Stride Access	Suitable in GPU L1	Reuse at TB level
TLB clustering [3], [4]	✗	✓	✗	✗	✗
TLB range [5]–[7]	✗	✓	✗	✗	✗
Huge page [1], [2], [8]	✗	✗	✓	✓	✗
Eager paging [9], [10]	✗	✗	✓	✗	✗
Speculative TLB [11]	✗	✓	✓	✗	✗
TLB probe [12]	✗	✓	✓	✓	✗
Least-TLB [13]	✓	✓	✓	✗	✗
Our approach	✓	✓	✓	✓	✓

In this paper, we set to leverage the translation reuse opportunities and improve the L1 TLB hit rates. We argue that "translation locality" is even more important compared to conventional data reuse in the UVM execution. Specifically, we first conduct a comprehensive characterization to quantify the translation reuses at TB granularity. We observe that there exist substantial translation reuses within each TB. We also notice that most of the reuses show large reuse distances that exceed the L1 TLB reach. Based on our observation, we propose translation reuse-aware TB scheduling together with dynamic TLB partitioning and sharing to effectively reduce the reuse distances. It is important to emphasize that our approach does *not* increase the TLB sizes nor throttle the degree of GPU parallelism. This paper makes the following contributions:

- We conduct a thorough characterization to quantify the translation reuses at GPU TB granularity. We observe that most of the TBs show significant intra-TB reuses rather than inter-TB reuses. We further investigate the intra-TB reuse distances and find most of them exceed the GPU L1 TLB reach due to inter-TB interference, leading to poor L1 TLB performance.
- We propose two simple-yet-effective optimizations to mitigate the L1 TLB thrashing. First, we implement translation reuse-aware TB scheduling. That is, instead of baseline round-robin TB scheduling, we schedule TBs to the SMs that have low chances of thrashing. Second, we propose TLB partitioning based on TB ids to mitigate the inter-TB interference. We also implement dynamic set sharing to accommodate those TBs that have translation sharing.
- We use a cycle-accurate CPU-GPU simulator to evaluate our proposal on 10 benchmarks from various suites. The experimental results show that the proposed scheduler and TLB management effectively improve the L1 TLB hit rate without increasing the TLB capacity. This improvement translates to, on

average, 12.5% execution time reduction.

II. Background

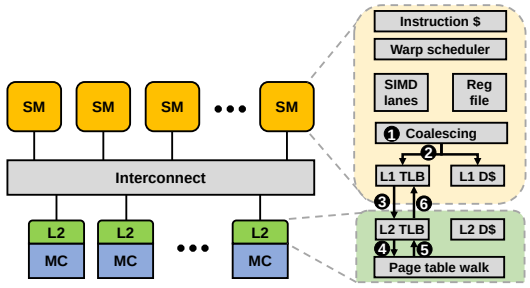

Fig. 1. Baseline GPU Architecture.

Figure 1 depicts the baseline GPU architecture comprising multiple streaming multiprocessors (SMs) and memory partitions. Each SM has its own private L1 TLB and L1 cache. L2 TLB and L2 data cache are shared across all SMs and distributed across the memory partitions. SMs are connected to memory partitions via an on-chip interconnection. The figure also illustrates the address translation process under unified virtual memory (UVM). Specifically, the memory requests generated by each thread are first coalesced by the GPU memory coalescing unit (❶). The L1 data cache lookup happens in parallel with the L1 TLB lookup in a virtually indexed physically tagged (VIPT) cache-TLB design. If the translation is present in the L1 TLB, a TLB hit is returned and the physical page number (PPN) is used to compare with the tags in the L1 cache (❷). Otherwise, if the translation misses in the L1 TLB, the request is forwarded to the shared L2 TLB (❸). If it further misses in the L2 TLB, the page table walker is triggered (❹). After the physical address is retrieved by walking the page table, the translation is populated to the shared L2 TLB (❺) and further returned to L1 TLB (❻). In a typical GPU application, a kernel is a computation block that executes on GPU. A kernel consists of multiple thread blocks (TBs). When a kernel is launched, the GPU TB scheduler selects and maps the TBs to SMs in a round-robin fashion. If an SM does not have enough resources (e.g., number of registers and number of threads) to accommodate a TB, the TB scheduler will skip that SM and find the next one with sufficient resources. It is possible that several TBs are mapped to the same SM and executed concurrently. Once scheduled, the TBs cannot migrate to other SMs. All the threads within a TB are sub-grouped into a warp with 32 threads. Threads within a warp share the program counter and execute in a lock-step fashion. In the baseline architecture, we employ the Greedy-then-Oldest (GTO) warp scheduler.

III. Motivation and Characterization

A. Benchmarks and Experimental Setup

We use 10 benchmarks from various GPU benchmark suites including Rodinia [14], poly-bench [15], and pannotia [16]. Table II lists all the benchmarks. We use the largest input sizes that are available in the benchmark suites. For `bfs`, `color`, `mis`, and `pagerank`, we use the author citation graph (`coPapersCiteseer.graph`) as inputs [17]. Table II also lists the memory footprints in the last column. It is important to mention that the benchmarks listed in the table are the versions with UVM support. We obtain these UVM-enabled versions from the gem5-gpu [18] repository.

We use a cycle-accurate simulation framework, gem5-gpu [18], to conduct our characterization as well as evaluate our proposed optimizations. Table III summarizes the detailed simulation configuration. Since the TLB configuration is not publicly available, we use the configuration parameters from prior published works [1], [19], [20].

TABLE II
List of benchmarks.

Application	Suite	Benchmark	Input	Memory footprint
Breadth-First Search	Rodinia [14]	`bfs`	citation	107.48GB
Graph coloring centrality	Pannitia [16]	`color`	citation	12.89GB
Maximal independent set	Pannitia [16]	`mis`	citation	8.44GB
Needleman-Wunsch	Rodinia [14]	`nw`	suite	0.72GB
Page rank	Pannitia [16]	`pagerank`	citation	14.70GB
3D Convolution	PolyBench [15]	`3dconv`	suite	21.32GB
Matrix Transpose and Vector Multiplication	PolyBench [15]	`atax`	suite	4.51GB
BiCG Sub Kernel of BiCGStab Linear Solver	PolyBench [15]	`bicg`	suite	3.76GB
Matrix Multiply	PolyBench [15]	`gemm`	suite	18.28GB
Matrix Vector Product and Transpose	PolyBench [15]	`mvt`	suite	4.38GB

TABLE III
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) dual warp schedule Round-Robin (RR) TB scheduler
TLB Config	L1: 64 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

B. Motivation

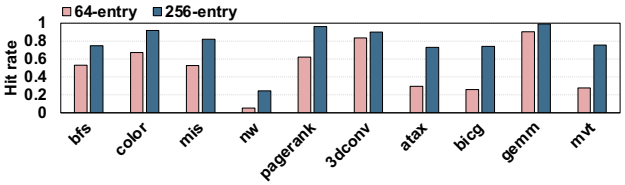

Fig. 2. L1 TLB hit rates of two different L1 TLB capacities.

We start with the GPU L1 TLB hit rates. The first bar in Figure 2 shows the L1 TLB hit rates of all benchmarks in the baseline execution. For a given benchmark, the result in the figure is the average hit rate across all SMs as the L1 TLBs are SM private. As one can observe, most of the GPU benchmarks suffer poor L1 TLB hit rates. This is because a large number of memory pages are accessed intensively due to the GPU parallel execution. Since the private L1 TLB has limited capacity (i.e., 64 entries per SM), severe contention heavily thrashes the L1 TLB. Figure 2 also shows the results when we enlarge L1 TLB capacity from 64 to 256 entries while keeping the associativity the same. It can be observed that many benchmarks benefit from larger L1 TLB capacity. Note that, `nw` has a low hit rate due to i) a significant amount of cold misses and ii) irregular data access patterns during execution. While adopting a larger TLB improves the hit rates, it is not a scalable approach and also increases both the TLB hit time (i.e., latency) and the chip die size. Therefore, we focus on optimizations that do not increase the TLB capacity.

C. Translation Reuse

In this paper, we target to leverage translation reuses to improve the "translation locality" in L1 TLBs. We first quantify the reuse potential at inter-TB and intra-TB levels. Then we study the reuse distances to reveal that inter-TB interference is the main factor that hurt the L1 TLB hit rates.

Figure 3 and Figure 4 show the quantification results. We focus on TB as it is the granularity of computations that schedule on each

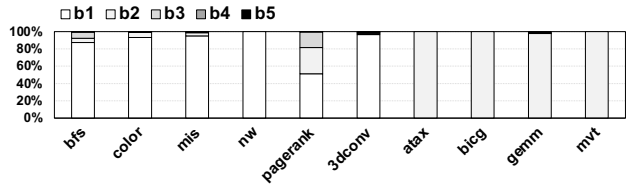

Fig. 3. Inter-TB translation reuses.

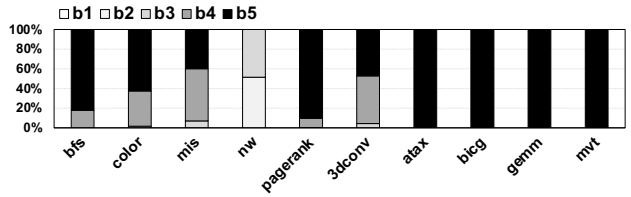

Fig. 4. Intra-TB translation reuses.

SM. The y-axis in the figures represents the percentage of the total number of TBs. We divide the reuse intensity into 5 bins ($b1$, $b2$, $\cdots$, $b5$) with 20% increment as shown in the figure legend. That is, $b1$ indicates that there are less than 20% of the total translations being reused, and $b2$ indicates there are more than 20% but less than 40% of the total translation being reused. With that notion, x% from y-axis with by in the legend indicates that there are x% of the total number of TBs that have more than (y-1)×20% but less than y×20% of their translations being reused at least once. For inter-TB reuse, a TB pair is counted in a particular bin if the translation reuse between two TBs falls into that bin. For intra-TB reuse, a TB is counted instead of a TB pair. To be more concrete, we calculate the translation reuse intensity based on the following equation:

$$R_{c1-c2} = \frac{size(x, x \in T_{c1} \mid x \in uniq(T_{c1}) \cap uniq(T_{c2}))}{size(T_{c1})}. \quad (1)$$

where $c1$ and $c2$ represent two different TBs, and T_{c1} and T_{c2} represent the total number of address translations issued from $c1$ and $c2$, respectively. For instance, if the application kernel has 10 TBs, we exhaustively calculate the intensity of all possible TB pairs and then plot them in the bins as shown in Figure 3. For intra-TB characterization, we calculate the reuse using the same equation while keeping $c1$ and $c2$ the same TB. From Figure 3 and Figure 4, one can make the following observations.

Observation 1. Comparing the inter-TB and intra-TB reuses, most benchmarks show substantial intra-TB reuses rather than inter-TB reuses. For example, `bfs` has 87% TBs in $b1$ for inter-TB while 82% TBs in $b5$ for intra-TB reuse. This is because TBs comprise several warps, and the threads in warps work on different data elements in the SIMD execution. Therefore, it is unlikely that different TBs will access the same 4KB page, unless there existed intrinsic data reuses in the application algorithm (e.g., mapping adjacent TBs to the input matrices in `gemm`).

Observation 2. For benchmarks `atax`, `bicg`, `gemm`, and `mvt`, there are sizable portions of TBs that have inter-TB translation reuse between 30% to 50%. This is because these benchmarks operate on matrices and vectors that consist of substantial data reuses. For example, the adjacent TBs holding adjacent portions of the matrices are likely to have access to common pages. Thus, the reuse of translations between TBs is also higher compared to other benchmarks.

Observation 3. For benchmarks `bfs`, `color`, `mis`, `nw`, `pagerank`, and `3dconv`, their intra-TB reuses show varied intensities. For instance, `nw` shows 51% TBs in bin $b2$ and 48% TBs in bin $b3$, whereas `bfs` shows 18% TBs in bin $b4$ and 82% TBs in bin $B5$. The reason behind this is that most of these benchmarks are dealing with irregular graphs, causing discrepancies in computation and memory accesses between TBs. Therefore, some TBs may heavily access a set of pages repetitively while other TBs do not have such characteristics.

Takeaway. From our translation reuse quantification at TB granularity, we summarize that, while for some benchmarks, there are sizable inter-TB translation reuses, the majority of TBs in the benchmarks show significant intra-TB translation reuses rather than inter-TB reuses. For some benchmarks, those reused translations are accessed multiple times during execution, bringing the potential for L1 TLB hits if the translation resides in the L1 TLB by the time it is being reused.

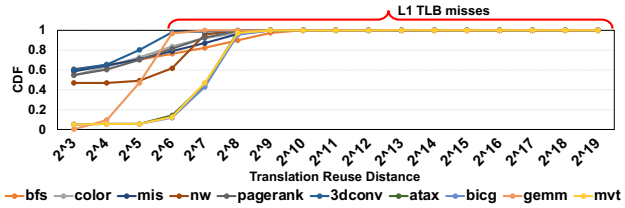

Fig. 5. Intra-TB translation reuse distance.

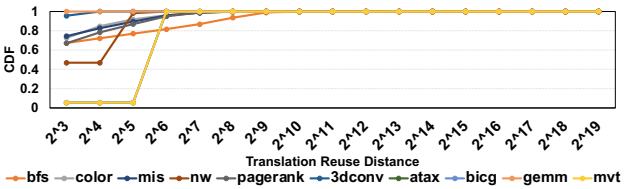

Fig. 6. Intra-TB translation reuse distance running one TB at a time (i.e., inter-TB interference has been removed).

D. Translation Reuse Distance

With substantial intra-TB translation reuse, we next conduct translation reuse distance analysis to investigate whether the L1 TLB can capture these reuses during execution. We define **translation reuse distance** as the number of unique translations between two memory accesses to the same page. We only show the distance analysis for intra-TB since a significant amount of translation reuses are from intra-TB. Figure 5 shows the cumulative distribution function (CDF) of intra-TB reuse distance. The x-axis represents the translation reuse distances in the power of 2 (starting from 3) and the y-axis represents the CDF. We also mark the capacity misses of L1 TLB. Reuses with distances larger than L1 TLB capacity will certainly miss the L1 TLB. We make two main observations:

Observation 1. The 64-entry L1 TLB fails to capture most intra-TB translation reuses. In particular, for benchmarks `bfs`, `mis`, `nw`, `atax`, `bicg`, and `mvt`, most of the intra-TB reuses have distances exceed the L1 TLB capacity (i.e., larger than 2^6), leading to poor L1 TLB hit rates as we discussed earlier in Figure 2.

Observation 2. As most TBs show a significant amount of translation reuses within TBs rather than across TBs, concurrent execution of TBs on the same SM may cause severe interference and enlarge the translation reuse distances. To verify, we show the intra-TB reuse distance when inter-TB interference is removed (Figure 6). We achieve this by allowing only one TB to run at a time. Compared with Figure 5, one can see that most of the benchmarks show reduced intra-TB reuse distances.

Takeaway. Our characterization reveals that, while there exist substantial intra-TB translation reuses, the reuse distances are generally large to yield good L1 TLB performance due to the inter-TB interference. Reducing the interference can effectively reduce the reuse distances, which can potentially translate to an improvement in the TLB hit rates.

IV. Proposed Optimization

Our goal. The goal of this paper is to improve the GPU L1 TLB hit rate by leveraging intra-TB translation reuses. To this end, we propose i) TLB thrashing-aware TB scheduling that mitigates the inter-TB interference to effectively reduce the reuse distances, ii) TLB partitioning that isolates the translations oriented from different TBs to reduce TLB thrashing, and iii) dynamic and automatic TLB set sharing that accommodates the scenarios where adjacent TBs share translations.

A. Thrashing aware TB Scheduling

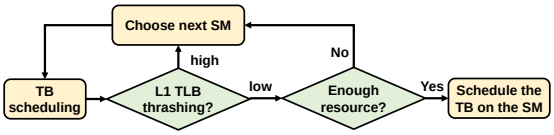

Fig. 7. TLB-aware TB scheduling.

We first propose thrashing-aware TB scheduling. Figure 7 depicts the scheduling algorithm. Specifically, we enhance the baseline round-robin TB schedule with the capability to probe the instant L1 TLB miss rate. This is based on the intuition that different SMs may have different L1 TLB hit rates because of the computation discrepancy among the TBs. This is particularly normal in graph applications where the graph structure can cause imbalanced memory accesses among TBs. We implement a hardware table in the TB scheduler, the table consists of 16 entries for the 16 SMs and each entry has two fields <TLB$_{hits}$, TLB$_{total}$>. Each SM is responsible for updating the table based on its TLB accessing statistics. The table is similar to the structure in prior work that is used to capture cache hit rates [21]. Whenever a new TB is to be scheduled, the scheduler first checks whether the candidate SM has a low TLB miss rate compared to other SMs. If so, it further checks the resource availability. Otherwise, the scheduler skips the SM and tries to find another SM with a low TLB miss rate. If the schedule cannot find an SM with low TLB miss rates, it falls back to the default scheduling. Note that our scheduling does *not* throttle the parallelism of GPU execution. That is, our approach does not limit the number of TBs scheduled on the SM if there are enough available resources to accommodate new TBs. While throttling can reduce the contention as demonstrated by prior works [22], our approach can be extended to work with TB throttling to further reduce the TLB thrashing.

B. Translation aware TLB Partitioning

Though the thrashing-aware TB scheduling tries to balance the number of translations across SMs and avoids over-subscription of particular SMs, it is still possible that the TBs scheduled on the same SM interfere with each other, enlarging intra-TB translation reuse distances. To address this problem, we propose L1 TLB partitioning based on the TB$_{id}$ instead of based on the address index bits. To be more specific, during GPU execution, each TB is labeled with a hardware TB$_{id}$ once it has been scheduled to an SM. The TB$_{ids}$ are unique for the TBs running on the same SMs. When a TB finishes execution and relinquishes the resources, the TB$_{id}$ is also freed for the next TB scheduled. In order to use TB$_{id}$ as the index to TLB sets, we also modify the TLB entry to store the entire VPN instead of just the tag. Figure 8 shows the proposed L1 TLB partitioning. Specifically, a virtual address is partitioned into VPN and page offset. Instead of using index bits to index the TLB sets, we leverage the TB$_{id}$ to perform the indexing. For instance, given an L1 TLB with 64 entries and 4-way associativity, if there are a total of 16 TBs scheduled on the SM, each TB will be associated with one set and the TB$_{id}$ is

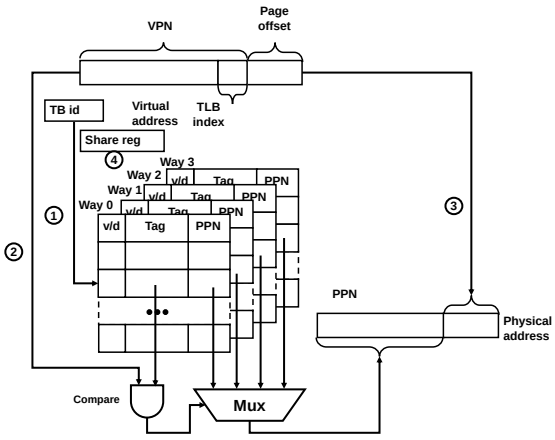

Fig. 8. Proposed L1 TLB partitioning.

used to index the corresponding set. On the other hand, if there are 4 TBs scheduled on the SM, each TB will be associated with 4 sets. In the TLB lookup process, the TB$_{id}$ is used to index the TLB sets (①). Then, the VPNs from all the TLB ways are compared with the VPN from the virtual address (②). If the comparison is successful, the corresponding PPN is retrieved from TLB and is concatenated with the page offset to form a valid physical address. Similarly, in the TLB insertion process, the TB$_{id}$ is used to index the candidate set for the translation. If the sets are full, the LRU algorithm is triggered for replacement.

We want to discuss three important points in our design. First, the TLB partitioning proposed in this work is based on the hardware TB$_{id}$ to index the TLB set, which guarantees that the concurrent running TBs on the same SM have different TB$_{ids}$. The TB$_{id}$ can be reused by subsequently scheduled TBs. This avoids expensive TLB flushing upon each TB's finish and allows us to keep the TLB entries for potential inter-TB reuse if there is any. Second, the number of TBs that can run concurrently on each SM is determined at compile time (by calculating the threads, registers, and shared memory used by the TBs). Once scheduled, the TB cannot migrate to other SMs. As a result, for different applications, the number of TBs per SM can be different. Our TB$_{id}$ guided TLB partitioning is flexible to accommodate different application scenarios because we use the TB$_{id}$ to perform indexing instead of hard partitioning the TLB into different segments. Third, using TB$_{id}$ to index the set involves overheads in the TLB lookup. Specifically, if a TB$_{id}$ is mapped with two sets, each set has to perform the lookup, doubling the lookup time if there are no additional compactors and multiplexes. In the cases when many sets are mapped to a few TBs, the lookup overhead is large. However, since the L1 TLB only has 16 sets and most of the SMs have more than 10 TBs running concurrently, the overhead is not significant. In our evaluation, we include the overheads in our results.

While TLB partitioning alleviates the inter-TB interference, it may cause two problems that can affect the TLB hit rates. First, it can introduce redundant entries. For example, if TB$_i$ and TB$_{i+1}$ both access the same memory page, the same translation might be residing in different sets if TB$_i$ and TB$_{i+1}$ are not sharing the sets due to TB$_{id}$ based set indexing. Second, it can cause imbalanced numbers of translations in different TLB sets. Specifically, some sets might get oversubscribed and experience higher miss rates, while others might be under-utilized due to fewer requests. To solve these two problems, we enhance our design with dynamic and automatic adjacent TLB set sharing. As shown in Figure 8 (④), we design sharing flags held by a sharing register in the hardware. The sharing flag is a 1-bit flag for each TB and is used to indicate whether the adjacent TBs should share

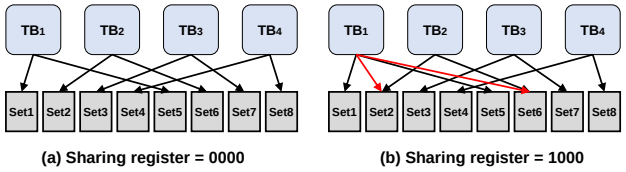

Fig. 9. Sharing of sets among adjacent TBs.

their TLB sets or not. Each TB has its own sharing flag indicating the sharing status with its neighboring TB. As the maximum number of concurrent running TB is 16 due to the hardware limitation [23], the total number of flags is 16 bits. Figure 9 shows an example of 4 TBs and 8 TLB sets. As shown in Figure 9(a), initially, the values of all sharing flags are set to 0, indicating that the TB_{id} to the TLB set mapping is exclusive and there is no sharing between TBs[1]. Figure 9(b) shows when the sharing flag of the first TB is equal to 1. As one can observe, TB_1 shares the TLB sets indexed to TB_2. Sharing the sets among multiple TBs can reduce the thrashing of particular sets and balance the number of translations across multiple sets. This is particularly useful for benchmarks where the TBs have inter-TB reuses (e.g., the adjacent TB to matrices mapping in `gemm`). Now, an important question is when to set and when to reset the sharing flag. In our current design, the initial sharing flags are set to 0. When a TLB set is oversubscribed and one entry needs to be evicted for a replacement, we check the adjacent TLB set to see whether it has an empty slot. If so, we put the evicted entry in the adjacent set. If the set is owned by another TB, the sharing flag is set to 1 in the sharing register. During TLB lookup, we also check the sharing value to determine which sets need to be searched for the requested virtual address. We reset the sharing flag of a particular TLB set when a TB that is currently indexed to that TLB set finishes its execution and relinquishes its occupied resource.

We want to emphasize two aspects of the TLB set sharing design in our approach. First, our current design uses a 1-bit flag to indicate the sharing status. One may choose to implement a counter for the sharing and explore an appropriate threshold for the sharing. This may capture the inter-TB translation sharing more accurately and perform better for particular applications. However, we found that the 1-bit flag is good enough to capture the sharing in the benchmarks we evaluated. We leave the counter and threshold exploration to our future work with other applications (e.g., machine learning and deep learning applications). Second, our current design only considers set sharing of adjacent TBs instead of other possible sharing patterns (e.g., all TB to all TB sharing). In all-to-all sharing, we will have to track the sharing TB_{ids}, which introduces additional bookkeeping hardware and TB_{id} lookup latency. We found the adjacent sharing performs well as most of the inter-TB sharing comes from adjacent TBs, as observed in our characterization. Note that, we only allow the sharing value to be 0 and 1 to control the involved lookup overheads. We found that this is good enough to capture the sharing and avoid the imbalanced number of translations among sets.

Hardware Overheads. The main hardware overheads include the TLB status table in the TB scheduler and the enlarged TLB to store the whole VPN. The TLB status table requires 136 bytes[2], which is negligible to the L1 TLB size.

[1]Note that, in our baseline setup, we have 16 L1 TLB sets and the maximum number TBs can be scheduled on each SM is 16. That is, in the initial setting, there does not exist a scenario where two TBs share sets. However, for different architectures, when the scheduled number of TBs per SM is larger than the number of TLB sets, multiple TBs can share the same sets at the initiation.

[2]The table has 16 entries, each entry consists of 4-bit SM_{id} and two 32-bit cycle counters for the TLB hits and total requests.

V. Experimental Evaluation

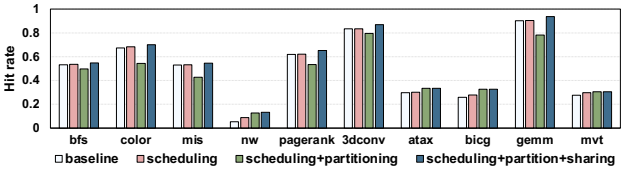

Fig. 10. L1 TLB hit rates. The higher the better.

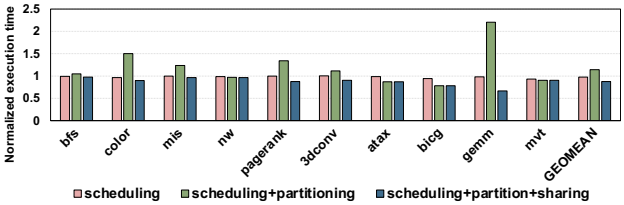

Fig. 11. Execution time normalized to the baseline. The lower the better.

In this section, we experimentally evaluate our proposed optimizations. We use the same benchmarks in Table II and the baseline simulation configuration is given in Table III. We show the results of TLB partitioning only and partitioning plus set sharing. Note that, in both results, we have enabled the translation reuse aware TB scheduling to balance the number of translations among different SMs. Figure 10 shows the L1 TLB hit rates and Figure 11 shows the execution time normalized to baseline. We make the following main observations. First, scheduling reduces the execution time by an average of 2.3%. Though the improvement is marginal, the scheduling enlarges the potential for subsequent TLB management. Second, simple partitioning of the TLB degrades the L1 TLB hit rates and performance of most benchmarks. The average (geomean) execution time increases by 14.3% with simply TLB partitioning compared to the baseline execution. This is because the number of available entries to each TB is reduced after partitioning. However, for benchmarks `atax`, `bicg`, `nw`, and `mvt`, partitioning improves the L1 TLB rates as well as the overall application performance. This is because most of these benchmarks suffer from severe inter-TB interference, and partitioning isolated the translations from different TBs, hence reducing the interference. Third, TLB set sharing effectively improves the L1 TLB hit rates as well as reduces the execution time (shown as the last bar in both figures). The average execution time reduces by 12.5% over 10 benchmarks evaluated. This indicates that the proposed set sharing is able to improve the utilization of each TLB set while keeping the inter-TB interference alleviated. Fourth, for benchmarks that already have high TLB hit rates (e.g., `gemm`), our proposed approaches do not degrade the TLB hit rates. Finally, for `nw`, the improved L1 TLB hit rate does not translate to performance improvement. This is because that `nw` is an irregular application and is compute-bound. Thus, the scheduler can effectively hide part of the translation overheads.

Large page: Our study so far employs 4KB page size in both the baseline execution and our approach. We also conduct a study employing huge pages (i.e., 2MB page) instead of 4KB pages. Results indicate that huge pages indeed significantly improve the L1 TLB hit rates, especially for matrix operation-centric applications (e.g., `gemm` and `mvt`). We also test our optimizations with huge pages, and it brings an average of 2.13% execution time reduction. This indicates that our approach can be combined with huge pages, even though the saving is less compared to 4KB pages.

Comparison with the state-of-the-art: We next compare our approach with a recent work that employs compression to enlarge

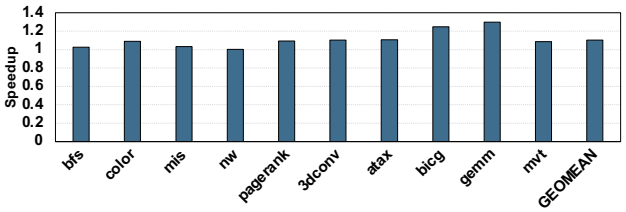

Fig. 12. Comparison against recent TLB compression work [24]. Speedup is calculated by our approach plus TLB compression normalized to the TLB compression work on L1 TLB.

the TLB reach [24]. The compression technique present in [24] relies on continuous or regular stride access patterns to compress multiple TLB entries into one entry such that more translations can be stored in the TLB. Despite the compression and decompression overheads that introduce latencies on the execution critical path on L1 TLB lookup, we implement the compression process and combine it with our approach. Figure 12 shows the speedup when our approach is combined with TLB compression normalized to the TLB compression work. As one can observe, our approach brings an additional 10.4% speedup on top of the TLB compression work. That is, our approach is complementary to the TLB compression and can bring further performance improvements.

VI. Related Work

To optimize the address translation, several prior studies [8], [11] improved TLB hit rates by speculative techniques to infer the physical page number of a translation that misses in the TLBs. Cox et al. [25] proposed MIX TLBs, which supported multiple page sizes by coalescing large pages in a single TLB entry. An OS support, named translation ranger was presented by Yan et al. [5] to enable continuity-aware TLBs in order to afford fewer translation overheads. Compared to all the prior efforts, we are the first to quantify the translation reuses in the GPU context and propose optimizations that leverage the translation reuses to effectively improve the GPU L1 TLB hit rate. Many prior works rely on data access locality (e.g., continuity) to reduce the TLB thrashing, whereas our proposed approach seeks scheduling solutions to mitigate the thrashing problem and does not rely on specific access patterns. Moreover, our approach is complementary to prior works and can be combined to further improve the TLB performance.

VII. Conclusion

Unified virtual memory is a promising feature that simplifies programming on CPU-GPU heterogeneous computing platforms. However, the incurred translation process introduces significant overheads to GPU execution, especially due to poor GPU L1 TLB hit rates. In this paper, we leverage translation reuse to alleviate the TLB thrashing. We thoroughly characterize various benchmarks and quantify the translation reuse and reuse distances among GPU TBs. We summarize several observations and insights from our characterization and propose simple yet effective TLB partitioning and sharing based on TB_{ids} to leverage the translation reuse as well as to alleviate the TLB thrashing. Experimental results have shown that our proposed approach can effectively improve the L1 TLB hit rate without compromising the degree of GPU parallelism. This TLB hit rate increase translates to an average 12.5% execution time reduction. For future work, we aim to study translation reuse at warp granularity and explore potential translation reuse-aware warp scheduling policies. We expect that the warp scheduling is complementary to the approach proposed in this paper and can be integrated to further improve the address translation performance in a UVM-based CPU-GPU heterogeneous system.

Acknowledgment

The authors would like to thank the anonymous reviewers for their constructive feedback and suggestions. This work is supported in part by NSF grants #2011146, #2154973, and #1725657.

References

[1] R. Ausavarungnirun, J. Landgraf, V. Miller, S. Ghose, J. Gandhi, C. J. Rossbach, and O. Mutlu, "Mosaic: A gpu memory manager with application-transparent support for multiple page sizes," in *MICRO*, 2017.
[2] M. Parasar, A. Bhattacharjee, and T. Krishna, "SEESAW: Using Superpages to Improve VIPT Caches," in *ISCA*, 2018.
[3] B. Pham, A. Bhattacharjee, Y. Eckert, and G. H. Loh, "Increasing tlb reach by exploiting clustering in page translations," in *HPCA*, 2014.
[4] B. Pham, V. Vaidyanathan, A. Jaleel, and A. Bhattacharjee, "Colt: Coalesced large-reach tlbs," in *MICRO*, 2012.
[5] Z. Yan, D. Lustig, D. Nellans, and A. Bhattacharjee, "Translation ranger: operating system support for contiguity-aware tlbs," in *ISCA*, 2019.
[6] V. Karakostas, J. Gandhi, F. Ayar, A. Cristal, M. D. Hill, K. S. McKinley, M. Nemirovsky, M. M. Swift, and O. Ünsal, "Redundant memory mappings for fast access to large memories," in *ISCA*, 2015.
[7] C. H. Park, T. Heo, J. Jeong, and J. Huh, "Hybrid tlb coalescing: Improving tlb translation coverage under diverse fragmented memory allocations," in *ISCA*, 2017.
[8] B. Pham, J. Veselỳ, G. H. Loh, and A. Bhattacharjee, "Large pages and lightweight memory management in virtualized environments: Can you have it both ways?" in *MICRO*, 2015.
[9] A. Basu, J. Gandhi, J. Chang, M. D. Hill, and M. M. Swift, "Efficient virtual memory for big memory servers," in *ISCA*, 2013.
[10] S. Haria, M. D. Hill, and M. M. Swift, "Devirtualizing memory in heterogeneous systems," in *ASPLOS*, 2018.
[11] T. W. Barr, A. L. Cox, and S. Rixner, "Spectlb: A mechanism for speculative address translation," in *ISCA*, 2011.
[12] T. Baruah, Y. Sun, S. A. Mojumder, J. L. Abellán, Y. Ukidave, A. Joshi, N. Rubin, J. Kim, and D. Kaeli, "Valkyrie: Leveraging inter-tlb locality to enhance gpu performance," in *PACT*, 2020.
[13] B. Li, J. Yin, Y. Zhang, and X. Tang, "Improving address translation in multi-gpus via sharing and spilling aware tlb design," in *MICRO-54*, 2021.
[14] S. Che, M. Boyer, J. Meng, D. Tarjan, J. W. Sheaffer, S. Lee, and K. Skadron, "Rodinia: A benchmark suite for heterogeneous computing," in *IISWC*, 2009.
[15] S. Grauer-Gray, L. Xu, R. Searles, S. Ayalasomayajula, and J. Cavazos, "Auto-tuning a high-level language targeted to gpu codes," in *2012 Innovative Parallel Computing (InPar)*, 2012.
[16] S. Che, B. M. Beckmann, S. K. Reinhardt, and K. Skadron, "Pannotia: Understanding irregular gpgpu graph applications," in *IISWC*, 2013.
[17] P. Sanders and C. Schulz, "10th Dimacs Implementation Challenge-Graph Partitioning and Graph Clustering," 2012.
[18] J. Power, J. Hestness, M. S. Orr, M. D. Hill, and D. A. Wood, "gem5-gpu: A heterogeneous cpu-gpu simulator," *IEEE Computer Architecture Letters*, 2015.
[19] J. Power, M. D. Hill, and D. A. Wood, "Supporting x86-64 address translation for 100s of gpu lanes," in *HPCA*, 2014.
[20] R. Ausavarungnirun, V. Miller, J. Landgraf, S. Ghose, J. Gandhi, A. Jog, C. J. Rossbach, and O. Mutlu, "Mask: Redesigning the gpu memory hierarchy to support multi-application concurrency," in *ASPLOS*, 2018.
[21] X. Tang, A. Pattnaik, O. Kayiran, A. Jog, M. T. Kandemir, and C. R. Das, "Quantifying data locality in dynamic parallelism in gpus," in *SIGMETRICS*, 2019.
[22] O. Kayıran, A. Jog, M. T. Kandemir, and C. R. Das, "Neither more nor less: optimizing thread-level parallelism for gpgpus," in *PACT*, 2013.
[23] NVIDIA, "Next Generation CUDA Compute Architecture: Kepler GK110," 2012.
[24] X. Tang, Z. Zhang, W. Xu, M. T. Kandemir, R. Melhem, and J. Yang, "Enhancing address translations in throughput processors via compression," in *PACT*, 2020.
[25] G. Cox and A. Bhattacharjee, "Efficient address translation for architectures with multiple page sizes," *ACM SIGPLAN Notices*, 2017.