Computer Sciences Technical Report #1842

Reducing GPU Address Translation Overhead with Virtual Caching

Hongil Yoon Jason Lowe-Power Gurindar S. Sohi

University of Wisconsin-Madison Computer Sciences Department
{ongal, powerjg, sohi}@cs.wisc.edu

ABSTRACT

Heterogeneous computing on tightly-integrated CPU-GPU systems is ubiquitous, and to increase programmability, many of these systems support *virtual address* accesses from GPU hardware. However, there is no free lunch. Supporting virtual memory entails address translations on every memory access, which greatly impacts performance (about 77% performance degradation on average).

To mitigate this overhead, we propose a *software-transparent, practical GPU virtual cache hierarchy*. We show that a virtual cache hierarchy is an *effective GPU address translation bandwidth filter*. We make several empirical observations advocating for GPU virtual caches: (1) mirroring CPU-style memory management unit in GPUs is not effective, because GPU workloads show very high Translation Lookaside Buffer (TLB) miss ratio and high miss bandwidth. (2) many requests that miss in TLBs find corresponding valid data in the GPU cache hierarchy. (3) The GPU's accelerator nature simplifies implementing a deep virtual cache hierarchy (i.e., fewer virtual address synonyms and homonyms).

We evaluate both L1-only virtual cache designs and an entire virtual cache hierarchy (private L1s and a shared L2 caches). We find that virtual caching on GPUs considerably improves performance. Our experimental evaluation shows that the proposed entire GPU virtual cache design significantly reduces the overheads of virtual address translation providing an average speedup of 1.77× over a baseline physically cached system. L1-only virtual cache designs show modest performance benefits (1.35× speedup). By using a whole GPU virtual cache hierarchy, we can obtain additional performance benefits.

1. INTRODUCTION

GPUs integrated onto the same chip as CPUs are now first-class compute devices. Many of these computational engines have full support for accessing memory via traditional virtual addresses [20]. This allows programmers to simply extend their applications to use GPUs without the need to explicitly copy data or transform pointer-based data structures. Furthermore, GPU programs can execute correctly even if they depend on specific virtual memory features, like demand paging or memory-mapped files.

However, virtual memory support does not come for free. It requires translating virtual addresses to physical addresses on every cache access. The overhead has significant performance and energy impact [4, 6, 26, 39]. Conventional CPUs mitigate the overhead by accessing translation lookaside buffers (TLBs) prior to cache lookups. However, simply reflecting the CPU-style memory management unit (MMU) in GPUs is not effective because of GPU microarchitecture differences.

GPUs have less locality—both spatial and temporal—in their address streams than CPUs and a higher rate of memory accesses for the following two reasons. First, GPUs have many more *execution contexts* than CPUs. Within one GPU compute unit, there are 10's of unique execution contexts (depending on the exact microarchitecture), each of which is capable of issuing a unique stream of memory requests. Second, each execution context has many "threads" or "lanes". A single static GPU load or store instruction can issue requests to 10's of different addresses, and these addresses could even be to different virtual memory pages! This high access rate and poor locality result in frequent TLB misses, which has significant performance and energy impact.

Preliminary studies have shown this overhead can be much worse on GPU architectures than on CPUs, with one study showing TLB misses taking 25× longer on GPUs than on CPUs [40]. Our empirical data mirrors these previous findings and shows that private GPU TLBs have very high miss ratios (average 56%) and high miss bandwidth (e.g., more than 2 misses per nanosecond in some cases). When using a practical translation implementation that has private TLBs, a large shared TLB, and multi-threaded page table walker, we see an average of 77% performance degradation over an ideal GPU MMU. This translation overhead is higher for emerging GPU applications (e.g., graph-based workloads) than traditional GPU workloads. *The major source of this overhead is the serialization delays at shared address translation hardware (e.g., a shared TLB) due to frequent private TLB misses.*

To reduce the virtual address translation overhead on GPUs, we propose a **GPU virtual cache hierarchy** that caches data based on virtual addresses instead of physical addresses. *We employ the GPU multi-level cache hierarchy as an effective bandwidth filter of TLB misses,* alleviating the bottleneck of the shared translation hardware. We take advantage of the property that no address translation is needed when valid data resides in virtual caches [45]. We empirically observe that more than 60% of references that miss in the GPU TLBs find corresponding data in the cache hierarchy (i.e., private L1s and a shared L2 cache). Filtering out the private TLB misses leads to considerable performance benefits.

Virtual caching has been proposed several times to reduce the translation overheads on CPUs [5, 13, 16, 25, 33, 41, 44]. However, to the best of our knowledge, the efficacy of virtual caching on GPUs has not been evaluated, and it is not publically known whether current GPU products use virtual caching. Furthermore, virtual caching has not been generally adopted, even for CPU architectures. The main impediment to using virtual caches for CPUs is virtual address synonyms. One recent study demonstrates that this synonym problem is exacerbated by larger caches and complex CPU workloads (e.g., server workloads) [44]. In larger caches, there is longer data residence time and thus a higher likelihood for synonymous accesses.

We leverage the fact that GPUs are most commonly used as an accelerator, not a general-purpose processor. This fact greatly simplifies virtual cache implementation, as it reduces the likelihood of virtual cache complications (i.e., synonyms and homonyms). This easily enables the scope of GPU virtual caching to be extended to the whole cache hierarchy (private L1s and a shared L2 caches), and allows us to take more advantage of virtual caching as a bandwidth filter of TLB misses. Including the shared L2 cache in the virtual cache hierarchy filters more than double the number of TLB accesses compared to virtual L1 caches (31% with L1 virtual caches and 66% with both virtual L1 and L2 caches).

These observations lead to our virtual cache hierarchy design for the GPU. In our design, all of the GPU caches—the private L1s and the shared L2—are indexed and tagged by virtual addresses. We add a new structure (called a forward-backward table, FBT) to the I/O memory management unit (IOMMU), that is fully inclusive of the GPU caches. The FBT is used to ensure correct execution of virtual caches for virtual address synonyms, TLB shootdown, and cache coherence.

We show that using virtual caching for the entire GPU cache hierarchy (both L1 and L2 caches) can be practical from perspectives of performance as well as design complexity. The experimental results show an average speedup of 1.77× over a (practical) baseline system. The proposal also shows more than 30% additional performance benefits over L1-only GPU virtual cache design.

We make the following contributions:

1. We analyze GPU memory access patterns and identify that a major source of GPU address translation overheads is the significant bandwidth demand for the shared TLB due to frequent private TLB misses.

2. We show that virtual caches filter out a significant number of private TLB misses. We evaluate both an L1-only virtual cache and an entire virtual cache hierarchy (L1 and L2 caches).

3. We propose the first pure virtual cache hierarchy for GPUs. Our design allows the flexibility for GPUs to keep their unique cache architectures (e.g., L1 caches without support for cache probes).

4. We discuss additional benefits of our design including sandboxing the system from potentially buggy software or hardware and filtering for system-GPU coherence traffic.

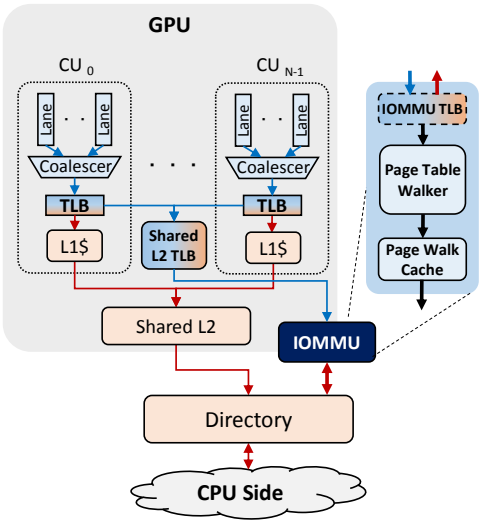

Figure 1: Overview of Baseline GPU MMU Design

The remainder of this paper is organized as follows: Section 2 discusses background on GPU address translation and virtual caching. Section 3 presents key observations of GPU memory access patterns motivating our proposal. Section 4 presents the design of the proposed GPU virtual cache hierarchy. We present our evaluation in Section 5. In Section 6, we discuss related work and conclude in Section 7.

2. BACKGROUND

In this section, we first discuss the current GPU memory system design and the GPU address translation. We then give an overview of virtual caches and design issues when using virtual caches.

2.1 GPU Address Translation

Figure 1 overviews the GPU memory system. In this paper, we consider integrated CPUs and GPUs with fully unified (shared) address space support (e.g., Heterogeneous System Architecture (HSA) specification [20]). In current systems, each computational unit (CU) has a private TLB. The TLB is consulted after the per-lane accesses have been coalesced into the minimum number of memory requests; it is not consulted for scratch-pad memory accesses. In addition to private TLBs, there is a large second-level TLB that is shared between all of the CUs. On a shared TLB miss, an address translation service request is sent to the IO Memory Management Unit (IOMMU) over the interconnection network. This interconnection network typically has a high latency. Even though integrated GPUs are not physically on the PCIe bus, IOMMU requests are still issued using the PCIe protocol, adding transfer latency to TLB miss requests [1].

The IOMMU consists of page table walker (PTW) and page walk cache; depending on microarchitecture designs, an additional IOMMU TLB may be employed for sharing across all I/O devices in a system. On a shared TLB miss, the PTW accesses the corresponding page table. The PTW is multi-threaded (e.g., supporting 16 concurrent page table walks) to reduce the high queuing delay due to frequent

shared TLB misses [1, 30, 40]. Additionally, there is a page walk cache to decrease the latency of page table walks by leveraging the locality of page table accesses. Once the address translation is successfully performed (i.e., the corresponding PTE is found in the page table), a response message is sent back to the GPU. Otherwise, a GPU page fault occurs, and the exception is handled by a CPU.

2.2 Virtual Caching

Virtual caching has been proposed many times for CPU caches over the past several decades as a way to limit the impacts of address translation [5, 13, 16, 25, 33, 41, 44]. The use of virtual caches can lower access latency and energy consumption of TLB lookups compared to physical caches, because the virtual to physical address translation is required only when a cache miss occurs. Virtual caches act to *filter TLB lookups*. Furthermore, virtual caches also *filter TLB misses* when the virtual caches hold lines for which the matching translation is not found in the TLB [45].

As the cache hit rate increases, we can expect more TLB accesses to be filtered. Accordingly, it would make sense to further extend the scope of virtual caching (i.e., multi-level virtual cache hierarchy) to take full advantage of these filtering effects, as blocks are more likely to reside longer in larger caches. However, the increase in the lifetime of data in the caches makes the occurrence of synonym accesses more likely [44], which imposes more overhead for synonym detection and management to ensure correct operation. This complicates the deployment of virtual caching for the entire cache hierarchy. However, we find that integrating virtual caching into the entire GPU cache hierarchy (L1 and L2 caches) is practical from perspectives of both performance as well as design complexity as discussed in Section 3.

3. POTENTIAL OF VIRTUAL CACHING ON GPUS

In this section, we show the potential of a GPU virtual cache hierarchy. We make the following four empirical observations:

- GPU workloads show high per-CU TLB miss ratio.
- They also show high per-CU TLB miss rates (i.e., significant bandwidth demand for the shared TLB).
- Many references that miss in per-CU TLBs find valid data in the GPU cache hierarchy (i.e., huge opportunity for caches to act as a TLB miss filter).
- Synonyms are less likely in GPUs than in CPUs since GPUs are used as an accelerator, not a general-purpose execution platform.

As we will establish below, these four key observations explain why a GPU virtual cache hierarchy is promising as a GPU address translation filter. We first substantiate these characteristics and discuss their design implications.

3.1 Evaluation Methodology

To get the empirical data, we used a full-system heterogeneous simulator, gem5-gpu [29]. We use full-system simulation to ensure that we model all system-level virtual memory operations. We consider high-performance integrated CPUs and GPUs with fully unified shared address space and full coherence among GPU and CPU caches. Details of our simulation parameters are in Table 1.

CPU	1 core, 3GHz, 64KB D$ and 32KB I$, 2MB L2$
GPU	16 CUs, 32 lanes per CU, 1.4GHz
L1 GPU Cache	per-CU 32KB, write-through no allocate
L2 GPU Cache	Shared 2MB, 8 banks, write-back, 128B lines, coherent with CPU
IOMMU	16 concurrent PTW, 8KB page-walk cache
DRAM	192 GB/s
Interconnect	Dance-hall topology within the GPU and Point-to-point network between the CPU-GPU

Table 1: Simulation Configuration

As a baseline, we use a recent work proposed by Power *et al* [30]. For the address translation hardware, we use 32 entries for the per-CU TLBs, and a large shared L2 TLB in the GPU. We assume this shared TLB can process up to one request per cycle.[1] We use 16 page table walkers to handle misses from the shared TLB, which reflects current hardware design [40]. We also have an 8 KB physical cache for the page table walkers, as prior work found this is important for high-performance translation [30]. We assume the IOMMU TLB has the same translations as GPU TLBs and do not model this structure.

We evaluate workloads from two different benchmarks suites. The Rodinia workloads [11] represent traditional GPU workloads and are mostly scientific computing algorithms. We also use Pannotia [10] to evaluate emerging GPU workloads. The Pannotia workloads are graph-based and show less locality than traditional GPU workloads. The Pannotia benchmarks suite comes with multiple versions of algorithms. We present data for each version of the algorithm separately. We run all workloads to completion. They execute from about 2 million to 2 billion cycles.

Observation 1: *GPUs show high per-CU TLB miss ratio.*

Figure 2 presents per-CU TLB miss ratio for 4KB pages for each benchmark by varying the TLB size. The results for the infinite size of TLBs indicate demand per-CU TLB misses.[2] The total height of each bar shows the average miss ratio for the entire execution of the application. We discuss the breakdown of each bar in Observation 3, below.

We notice that per-CU TLB miss ratio is very high, *much* higher than that of typical CPU workloads. Figure 2 shows about 56% (average) of misses with a 32-entry TLB per CU, and it can be above 80% for some workloads such as `pagerank`, `pagerank_spmv`, and `nw`. Even with large TLBs of 64 and 128 entries, we see frequent per-CU TLB misses.

These workloads have high TLB miss ratios for two reasons. First, GPUs are highly multi-threaded with up to 10's

[1] Power *et al* [30] considered infinite bandwidth of the shared TLB, which is unrealistic.

[2] "Demand misses", in this case, include all accesses that experience a non-zero translation latency when using infinite capacity per-CU TLBs. This includes multiple TLB misses to the same page in close temporal proximity as our TLB model does not combine multiple misses to the same page into a single request. Thus, we see demand misses that hit in L1 and L2 caches. In addition, some demand per-CU TLB misses hit in the shared L2 cache as data is shared between CUs. pathfinder experiences many such requests.

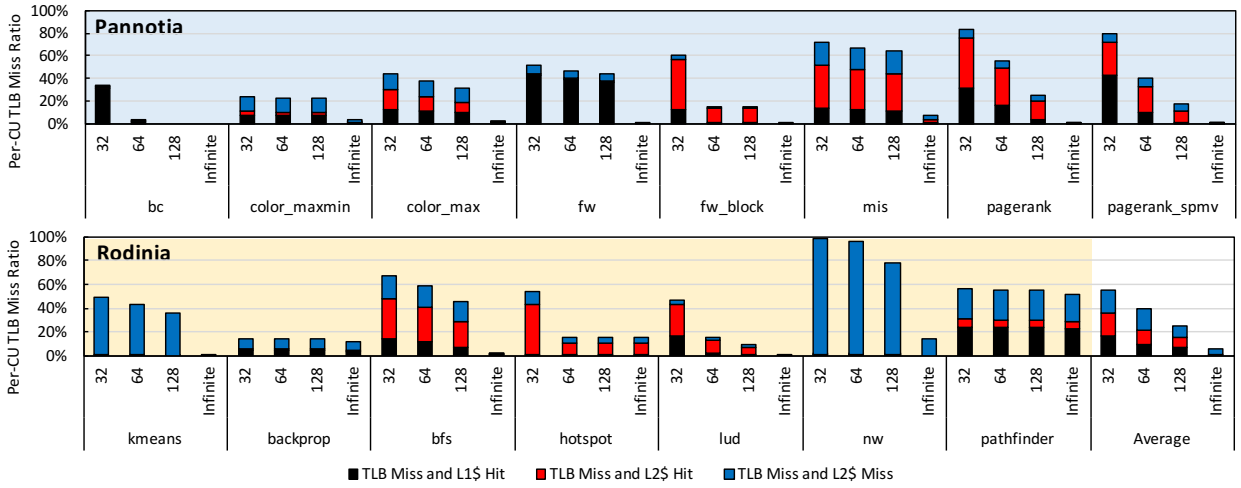

Figure 2: Breakdown of Per-CU TLB Miss Accesses

of unique execution streams per CU. Each of these streams may be accessing different pages in memory, limiting the spatial and temporal locality. Second, in addition to the many execution streams, each execution stream may result in multiple memory accesses per instruction due to memory divergence. For instance, `fw` averages 9.3 memory accesses per dynamic memory instruction. Our observations are in line with the results of a recent study on real hardware [40].

Design Implication: *The performance impact of the address translation is much higher for GPUs than CPUs.*

Observation 2: *GPU workloads show a high rate of both shared TLB accesses and misses.*

Figure 3 presents an average of events per one nanosecond for shared TLB accesses (i.e., per-CU TLB misses for all CUs) and shared TLB misses. We used periodic sampling to obtain this data. The results indicate the average of events across all sampling periods, and each bar has a one standard deviation band. For the results, 32-entry per-CU TLBs and a 512-entry shared TLB are used. The experiment for this figure assumes the shared TLB can be accessed any number of times per cycle, which is impractical.

We notice both frequent shared TLB accesses and frequent shared TLB misses. There are about 0.7 shared TLB accesses and 0.19 misses per nanosecond, on average. Some workloads (e.g., `color_max`, `fw`, `mis`, `pagerank`, `bfs`, and `lud`) show numerous sample periods with more than one shared TLB access per nanosecond. For example, `color_max` shows about 20% of sample periods with more than one shared TLB accesses per nanosecond. We also observe that, emerging GPU workloads, especially graph-based workloads like Pannotia, show much more frequent per-CU TLB misses than the traditional workloads because of high memory divergence [10]. The results show there are impractical bandwidth requirements at the shared TLB and the PTW (more than two accesses per nanosecond in some cases).

Figure 4 breaks down the performance overhead of address translation on the GPU, compared to an "IDEAL MMU" that has infinite bandwidth at the shared TLB and infinite capacity per-CU TLBs. This figure presents only the average

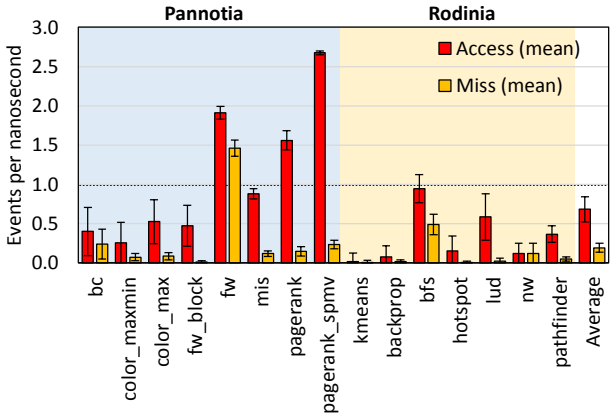

Figure 3: Analysis of Shared TLB Accesses/Misses

performance across all of the workloads evaluated. For a practical design that has 32-entry private TLBs and a 512-entry shared TLB (i.e., "BASELINE" bar), we see an average of 77% performance overhead over the "IDEAL MMU".

This overhead can be broken down into the overhead for page table walks (PTW) due to shared TLB misses, and the overhead due to serialization from limited bandwidth for the shared TLB. For the "PTW Overhead", we consider a system with an infinite capacity shared TLB, but the shared TLB bandwidth is limited to one access per cycle (1.4GHz). With an infinite capacity shared TLB, each per-CU TLB miss is immediately translated by the shared TLB, removing the PTW overhead. We can see that **performance benefits with infinite capacity of the shared TLB are small**, because the multi-threaded page table walker with a large page walk cache effectively hides shared TLB miss latency [30]. For the "Serialization Overhead", we consider a baseline system with infinite bandwidth for the shared TLB. We see that **most of the address translation overheads are due to serialization at the shared TLB, not the size of the shared TLB**.

The results suggest that the primary challenge of GPU address translation is to provide high bandwidth at the shared

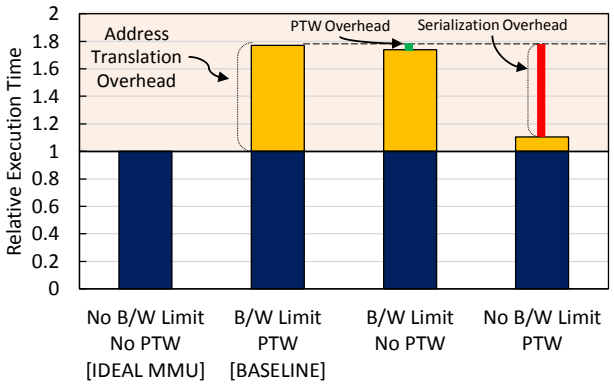

Figure 4: Analysis of GPU Address Translation Overhead

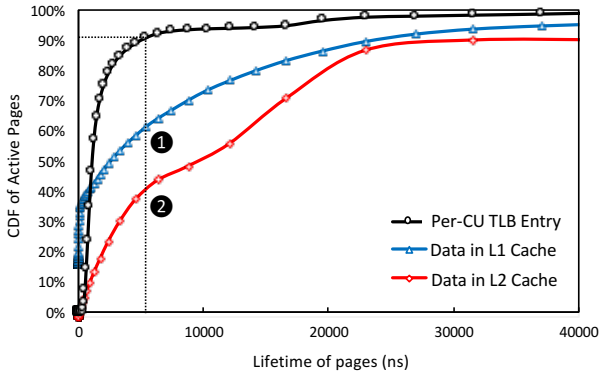

Figure 5: CDF Analysis of Active Pages: *bfs* Workload

TLB. However, it is impractical to use very large per-CU TLBs or to use highly multi-ported large shared TLB. Also, it is unrealistic to provide/manage huge buffers between the CUs and the shared TLB due to frequent shared TLB accesses.

Multi-banking the shared TLB, while costly in terms of complex pipeline and arbitration logic, could increase the bandwidth if bank conflicts are rare [3, 34]. However, bank conflicts may be more common on a banked TLB than a banked cache due to using higher order address bits for bank mapping. In fact, some of our workloads (e.g., `mis`, `color_max`, etc.) show frequent conflicts when analyzing the address stream to the shared TLB. For a 16-bank shared TLB, about 50% of shared TLB accesses see more than 4 prior requests mapped to the same bank in most recent 16 prior accesses for these workloads. The frequent bank conflicts limit the bandwidth of multi-banked designs.

The bandwidth demands of the shared TLB will continue to increase as future GPUs integrate more CUs. The recently released Playstation 4 Pro console has 36 CUs, and XBOX Scorpio is speculated to have even more 60 CUs [2]. Furthermore, we believe that the overall address translation overheads will increase with emerging applications [4, 10]. Thus, we cannot depend solely on unscalable conventional address translation optimizations, e.g., multi-banking, large per-CU TLBs (also evaluated in Figure 10), and large pages (Section 4.3). **Accordingly, we need a scalable, efficient way of filtering out the accesses to the shared TLB.**

Design Implication: *Filtering traffic to the shared TLB will improve performance.*

Observation 3: *Data resides in the caches much longer than its translation in the TLB.*

We now consider how many per-CU TLB misses can be filtered with a virtual cache hierarchy on a GPU. Figure 2 shows the breakdown of the TLB misses according to where the valid data is located in the GPU cache hierarchy. The results indicate that many references that miss in the per-CU TLB hit in the caches (black and red bars) and that only 34% of references that miss in the per-CU TLB are also L2 cache misses and access main memory (blue bars).

Many references that miss in per-CU TLBs actually hit in the caches. We notice that an average of 31% of total per-CU TLB misses find the corresponding data in private L1 caches (black bars), and an additional 35% of the total misses hit in a shared L2 virtual cache (red bars). These hits occur for two reasons. First, for L2 hits under per-CU TLB misses, data may be shared by multiple CUs. There are requests that miss in the per-CU TLB that hit in the shared L2 cache when other CUs have previously accessed the cache block. Second, for both L1 and L2 hits under per-CU TLB misses, blocks in the cache hierarchy are likely to reside longer than the lifetime of the corresponding per-CU TLB entries.

Figure 5 shows the relative lifetime of pages in each level of the cache hierarchy and the TLB. It shows three CDFs of the aspects for `bfs` workload; other workloads show similar patterns. The black line indicates the residence time of TLB entries. The other two lines show the active lifetime of data in L1 caches (blue line) and L2 shared cache (red line), respectively; the active lifetime is defined as the period between when data is cached and when it is last accessed. We notice that 90% of TLB entries are evicted after 5000 ns. However, 40% of data in L1 caches (①) and 60% of data in a shared L2 cache (②) are still actively used after that. Thus, accesses to such data hit in the cache hierarchy but are highly likely to occur misses in the TLB. In these cases, a virtual cache hierarchy is an effective TLB miss filter.

Also there is a noticeable gap between two lines for caches, suggesting that data not in L1 caches is frequently found in a larger L2 cache. This supports what we observed in Figure 2; the red bar is larger than the black bar for many workloads such as `color_max`, `fw_block`, `mis`, `pagerank`, `pagerank_spmv`, `bfs`, `hotspot`, `lud`. Thus, extending the scope of virtual caches to the L2 shared cache helps to filter out more TLB misses.

Design Implication: *A large fraction of TLB misses can be filtered by employing deeper virtual cache hierarchy (i.e., including a shared L2 cache) for GPUs.*

Observation 4: *GPU's accelerator nature makes synonyms much less likely than in CPUs.*

Others have observed a larger fraction of cache accesses to cache lines with synonyms when larger virtual caches are considered in the CPU cache hierarchy [44]. However, GPUs are not fully general-purpose compute devices. There are three key differences between CPUs and GPUs which lessen the impact of synonyms in a GPU cache hierarchy.

First, as an accelerator, current GPUs usually execute a single application at a time. Thus, there is usually one ac-

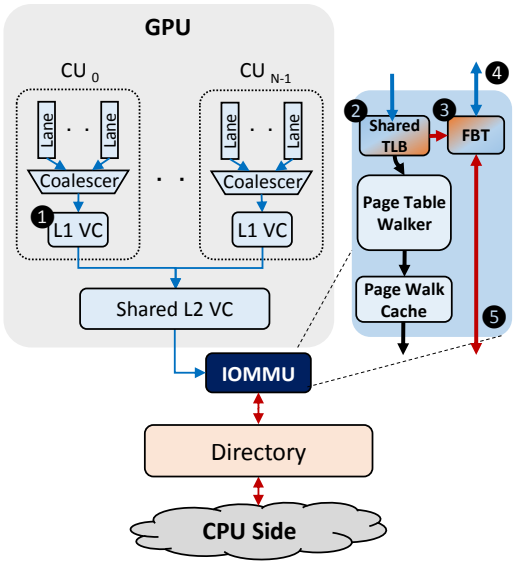

Figure 6: Schematic of Proposed GPU Virtual Cache Hierarchy

tive virtual address space. Second, GPUs rarely access I/O devices. This is likely to continue to be true since GPUs are useful for applications with data parallelism, which I/O device drivers rarely exhibit. Third, GPUs never execute OS kernel code; they only execute user-level applications. This eliminates most causes of synonym accesses, making the occurrence of active synonym accesses less likely in a GPU cache hierarchy than in a CPU cache hierarchy. For our workloads, we never observed synonyms. We discuss the implications on future system running multiple processes in Section 4.3.

Design Implication: *GPU virtual caches can be practical even for large second-level cache.*

4. DESIGN OF GPU VIRTUAL CACHE HIERARCHY

We first set two key requirements to design an efficient GPU virtual cache hierarchy:

1. The GPU virtual cache hierarchy needs to be software transparent. The proposed virtual cache design should be able to efficiently deal with potential issues from virtual memory idiosyncrasies such as potential virtual address synonyms, TLB shootdowns, etc.

2. The proposed design should be seamlessly integrated into a modern GPU cache hierarchy, which is somewhat different from a traditional CPU cache hierarchy.

Figure 6 illustrates the conceptual design of the proposed virtual cache hierarchy. Different from the baseline system (Figure 1), there are no per-CU TLBs or shared L2 TLB in the GPU, and the L1 and L2 caches at the GPU side are accessed with virtual addresses. Address translation is performed via a shared TLB in the IOMMU only when no corresponding data is found in the virtual caches. That is, the address translation point is completely decoupled from the GPU cache hierarchy.

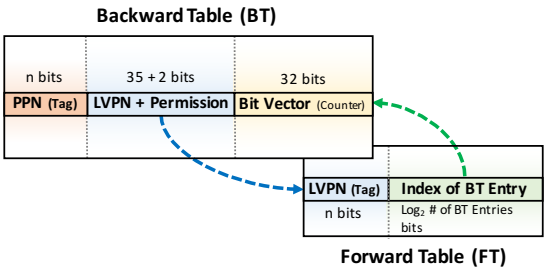

Figure 7: Overview of a Forward-Backward Table (FBT)

To correctly and efficiently support virtual memory, we add a *forward-backward table* (FBT) to the IOMMU. The FBT consists of two parts: *backward table* and *forward table* (see Figure 7). First, the backward table (BT) is primarily a reverse translation table which provides reverse translations from physical addresses to virtual addresses. We take advantage of some important features of an active synonym detection table (ASDT) from a recent virtual cache proposal for CPUs [44]. The backward table is fully inclusive of the virtual caches with an entry for every page that is currently cached. The BT tracks mappings from a physical address to a *unique* leading virtual address that is used to place the data in the virtual caches at a page level granularity. We use this information to carry out operations of virtual caches such as detection of synonymous accesses and handling TLB shootdowns. Also, the BT is employed for reverse address translations on coherence requests from outside the virtual cache using physical addresses.

The FBT also contains a small forward table (FT). The FT tracks mappings from a leading virtual address to an index of a corresponding backward table entry, which allows the FBT to be indexed by virtual address as well as physical address. Forward translation information is needed for several operations including when the virtual caches respond to coherence requests which use physical addresses, on virtual cache evictions, and TLB shootdowns. As we shall see, the extra index structure of the FT allows us to perform these operations without a shared TLB miss and subsequent page table walk. Additionally, with the forward translation information, the FBT can be used as a large second-level TLB structure.

In the following sections, we explain how the proposed design satisfies the two key requirements (Section 4.1-4.2). Then, Section 4.3 describes several design choices to improve the effectiveness not only of our proposal but also of the overall heterogeneous system by taking advantage of the forward-backward table.

4.1 Supporting Virtual Memory without OS Involvement

To handle synonym issues, some virtual cache proposals [5, 8, 9, 18, 25, 33, 45] require OS involvement (e.g., a single global virtual address space). Doing so restricts not only design flexibility of OSes but also programmability. Hence, we believe that virtual cache hierarchy for a GPU needs to be a *software transparent technique*.

Our proposal supports virtual memory requirements without any OS involvement. The proposal allows a GPU to ac-

cess any virtual address that can be accessed by a CPU. We also effectively and transparently manage the other operations (e.g., TLB shootdown, synonyms, homonyms, etc.).

Below we describe the details of operations and the extended structures to support virtual memory. For the sake of simplicity, we initially assume a system with an inclusive two-level virtual cache hierarchy. In Section 4.2, we will discuss some challenges and solutions for modern GPU cache designs that support a non-inclusive, two-level cache hierarchy.

Virtual Cache Access: For memory requests, a set of *lanes* (called shader processors or CUDA cores by NVIDIA) generate virtual addresses. After the coalescer, memory requests are sent to the L1 virtual cache without accessing a TLB (① in Figure 6). Different from a physical cache design, the permissions of a corresponding *leading* virtual page are maintained with each cache line due to deferred TLB lookups, and the permissions check is performed when the virtual cache is accessed.

On an L1 virtual cache miss, the L2 virtual cache is accessed with the given virtual address. On a hit in the virtual cache hierarchy, the valid data is provided to the corresponding CU as long as the permissions match. Thus, no address translation overhead is imposed.

Address Translation: On an L2 virtual cache miss, the request is sent to the IOMMU. The virtual to physical address translation is performed via the shared TLB (② in Figure 6). The translation is required because the rest of system is indexed with physical addresses. The permissions check for the cache miss is also performed at this point by consulting the shared TLB or the PTW.

Interestingly, we observe that, for most of shared TLB misses (e.g., 74% on average), a matching entry is found in the FBT.[3] Thus, the FBT can be a second-level TLB. We can search for the match in the FBT by consulting the FT on the misses. If there is no matching address translation, a page table walker (PTW) executes the corresponding page table walk. To reduce the long latency of consulting multiple DRAM accesses, a page walk cache that caches non-leaf nodes of the corresponding page table is accessed first. If the PTW fails to find a matching PTE, a page fault exception occurs; this exception is handled by the CPU. These actions taken on a shared TLB and FBT miss are the same as the baseline IOMMU design.

Synonym Detection and Management: To guarantee the correctness of a program, we need to check whether the cache miss occurs due to a virtual address synonym. Multiple virtual addresses (i.e., synonyms) can be mapped to the same physical address. Hence, it is possible that valid data has been cached with a different virtual address, and in this case, the data cached with the other virtual address should be provided.

To check for synonyms, the BT is consulted with the physical page number (PPN) (③ in Figure 6) obtained by the shared TLB lookup or the PTW. The BT is a set-associative cache (Figure 7). Each entry has a physical page number (PPN) as a tag and a unique (leading) virtual page number (LVPN). *Only this current leading virtual address is allowed to place and look up data from the physical page in the virtual caches as long as the entry is valid*. Thus, no data duplications are allowed in the virtual cache hierarchy.

If a valid BT entry is identified for the cache miss, it indicates that some data from the physical page resides in the virtual cache, and has been cached with the corresponding leading virtual address. Thus, if the given virtual address for the cache miss is different from the current leading virtual address, it is a synonym access. To ensure the correct data, we simply replay the virtual cache access with the current leading virtual address (④ in Figure 6). Each BT entry has a bit vector indicating which lines from a physical page are cached in the virtual caches. Thus, only addresses that will hit are replayed. If the bit in the bit vector is clear, the directory is accessed. Once the valid data is received from the memory, it is cached with the current leading virtual address, and its permissions.

For L2 virtual cache misses, if there is no match in the BT, a new entry is created for the mapping between its PPN and the given virtual page, and the given virtual page will be the leading virtual page for the physical page until it is evicted. At the same time, a corresponding FT entry is populated.

Cache Coherence between a GPU and a CPU: Cache coherence requests from a directory or CPU caches use physical addresses. Hence, a reverse translation (i.e., a physical address to a current leading virtual address) is performed via the BT (⑤ in Figure 6). Then, the request is forwarded to corresponding GPU caches with the leading virtual address. When the cache responds with a leading virtual address, it is translated to the matching physical address via the FT.

Including the BT in the IOMMU has the benefit of providing an efficient coherence filter for the GPU caches. Power *et al*. proposed using region-based coherence to filter most coherence messages sent to the GPU [28]. The BT is fully inclusive of the GPU caches. Therefore, when a coherence request is sent to the GPU, the BT can filter any requests to lines that are not cached in the GPU caches. The BT plays a similar role to the region buffer in the heterogeneous system coherence protocol [28].

TLB Shootdown and Eviction of FBT Entry: If information of a leading virtual page changes (e.g., permission or page mapping), the leading virtual page is no longer valid. Thus, the corresponding FBT entry should be invalidated. This leads to invalidations of data cached with the leading virtual address from all virtual caches for correct operations; without the invalidations, cache accesses may not respect the up-to-date permissions in the page table or may obtain stale data based on the previous mapping. Similarly, when an FBT entry is evicted (e.g., due to a conflict miss), the same operations need to be performed. By employing the information of a bit vector, only the cached data can be selectively evicted. This reduces invalidation traffic.

On a single-entry TLB shootdown, we use the FT with virtual addresses. Once a match is found, we directly access the corresponding BT entry with the index pointer. The FT filters TLB invalidation requests if no match is found.

[3] The FBT entries correspond to all of physical pages with cached data (private L1s and a shared L2 cache) We consider an over-provisioned FBT with 16K entries, which has the reach of 64MB (Section 4.3).

On an all-entry TLB shootdown, a cache flush is required.[4] The shootdowns resulting from page mapping or permission changes are infrequent [5]. Thus, the performance impact is likely minor.

Eviction of Virtual Cache Lines: When a line is evicted from the virtual cache, the bit vector information of the corresponding BT entry needs to be updated in order for the BT to maintain up-to-date inclusive information of the virtual cache. On every cache line eviction, the FT is consulted to identify the corresponding BT entry. The matching bit in the bit vector of the BT entry is unset.

4.2 Integration with Modern GPU Cache Hierarchy

We now discuss several other design aspects to be considered for a practical GPU virtual cache hierarchy.

Read-Write Synonyms: With a virtual cache hierarchy, it is possible for the sequential semantics of a program to be violated if there is a proximate read-write synonym access [33, 44]. For example, a load may not identify the corresponding older store since their virtual addresses differ, even though their corresponding physical address is the same. This load could get the stale value from a virtual cache earlier than the completion of the store.

For CPUs, this violation is detected with a leading virtual address in the load/store unit and recoverable as they provide support for precise exceptions and a recovery mechanism [44]. However, GPUs support relaxed consistency memory model without precise exceptions or precise recovery mechanisms [15, 17, 22]. Thus, resources for younger memory operations in the load/store unit can be deallocated before older memory operations are complete [37]. It could be too late to identify the violating instruction from the memory pipeline.

In practice, we believe that read-write synonyms will rarely cause problems in a GPU virtual cache hierarchy due to the following reasons:

1. GPUs do not execute any OS kernel operations that are the major source of read-write synonym accesses.
2. Although it is possible to have user-mode read-write synonyms, there are very few situations where they are used.
3. Even with read-write synonyms, correctness issues occur only if the aliases are in close temporal proximity.

Thus, we use a simple and viable solution and conservatively cause a fault when a read-write synonym access is detected at the FBT.[5] Although our design simply detects read-write synonyms and raises an error, if future GPU hardware supports recovery, it is possible to detect and recover from the violation of sequential semantics. We can use a similar mechanism as the ASDT [44] and replay the violating synonymous access with the leading virtual address obtained from the FBT. Given GPUs move toward general-purpose computing, it is possible that recovery and replay hardware may be included in future GPU architectures [22]. Regardless of the issue of read-write synonyms, we fully support the much more common read-only synonym accesses.

[4]For page mapping changes, some OSes flush whole caches (or selectively invalidate cache lines of the relevant pages depending on the ranges of target memory region). Thus, we do not have to additionally flush all caches again.

[5]Read-write synonym accesses are detected at the BT. Since the FBT forwards all coherence requests from a GPU to a directory, we track whether any writes have occurred to a cached physical page. The fault is raised when there is a synonymous access to a physical page that has previously been written, and vice versa.

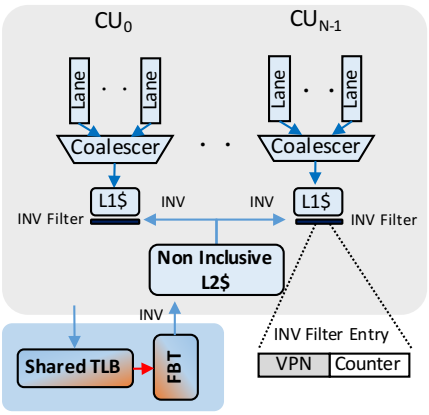

Figure 8: Supporting Non-Inclusive Cache Hierarchy

Multi-Level Cache with Non-Inclusion Property: In the previous section, we conceptually explain the operations with a two-level inclusive cache hierarchy. In practice, however, modern GPUs employ a non-inclusive hierarchy. Furthermore, the designs of modern GPU cache hierarchy show different features compared to that of CPUs with respect to writing policy, inclusion, and coherence protocols [12, 38].

The baseline GPU cache hierarchy has write-through L1 caches without write-data allocation. The shared L2 cache is non-inclusive, and it supports write-back with write-data allocation. On an L2 cache eviction, invalidations of L1 copies are not required. Thus, it is possible for private L1 caches to have data that does not reside in the shared L2 cache. This complicates tracking inclusive information of data currently residing in the private L1 virtual caches in the BT. A naive solution is that private L1 caches send messages to the BT in the IOMMU on every cache line refill and eviction to keep track of the precise information. This is not practical because of potential latency, energy and bandwidth overhead.

To keep the private L1 caches consistent, we take a conservative approach. The BT tracks the inclusive information of data currently cached only in the shared L2 cache with bit vectors. When a leading virtual page is no longer valid (e.g., an eviction of FBT entry or TLB shootdown), however, an invalidation message is sent to all L1 caches to invalidate all lines with a matching leading virtual page address. To avoid walking the L1 cache tags, we add a small invalidation filter at the L1 caches (shown in Figure 8). Each entry has the leading virtual page number as a tag and a counter tracking how many lines from the corresponding physical page currently reside in the cache. If a match is found in a filter for the invalidation request, the entire L1 cache is invalidated.

This approach does not require write backs since L1 caches have no dirty data (i.e., write-through without allocation). Additionally, this approach has only a minor impact in terms of performance. This is because first GPU L1 cache hit ratio

is usually low (less than 60% for most of our workloads), and the events triggering the flushes are highly unlikely, as TLB shootdowns are rare. Additionally, it is likely that the corresponding data are already evicted from L1 virtual caches at the point of an eviction of a corresponding FBT entry with an adequately provisioned FBT. Thus, the invalidation filter can filter the invalidation requests due to most FBT evictions.

4.3 Other Design Aspects of Proposed Design

In this section, we discuss other design aspects of the proposed virtual cache hierarchy.

Future GPU System Support: Future GPU systems will have more multiprocess support like modern CPUs, and *synonyms and homonyms* may be more common. To mitigate the overhead of handling synonym requests further, the concepts of dynamic synonym remapping [44] can be easily integrated to the proposed GPU virtual cache hierarchy. For active synonym accesses, a remapping from a non-leading virtual address to the corresponding leading virtual address can be performed prior to L1 virtual cache lookups. This reduces virtual cache misses and the latency/power overheads due to synonyms. Homonym issues can be easily managed by including address space identifier (ASID) information in a given virtual address; each cache line also need to track the corresponding ASID information. This prevents cache flushes on context switches.

Large Page Support: Using large pages is an effective way of reducing TLB miss overhead [27, 30, 40]. However, it has its own constraints and challenges [4, 23]. Increasing memory footprints will put additional pressure even on large page structures, requiring OSes and TLB organization modifications. In addition, large pages do not help workloads with poor locality (e.g., random access). Thus, we cannot depend on only large pages, but we need to support them to maintain compatibility with the conventional system.

In our design, supporting larger pages does not cause any correctness issues. However, it is impractical to use a large bit vector to track lines from a large page that are resident in the cache hierarchy. In fact, the use of a bit vector is just an optimization to selectively identify (or invalidate) lines of a physical page from virtual caches. Instead of maintaining the precise information, we can simply use a counter for the number of cached lines from a physical page. In this case, multiple invalidations could be sent until no more lines remain in the caches (i.e., the corresponding counter becomes zero). We can reduce the chances of the costly operations by deferring evicting FBT entries for a large page as much as possible until the counter becomes zero.

Area Requirements: To support the proposed virtual cache hierarchy, each cache line entry needs to be extended to track some extra information (e.g., virtual tag bits, permission bits), and we also need to support additional structures, i.e., invalidation filters for non-inclusive cache hierarchy and an FBT.

The size of the per private L1 invalidation filter is modest, relative to a private L1 cache. For instance, a 32KB L1 cache with 128B lines requires 1KB storage, which is less than 3% of the L1 cache size.

The FBT should be sufficiently provisioned to avoid potential overhead of cache line invalidations due to FBT entry evictions. We observe that there are about 6000 different 4KB pages whose data reside in a 2MB L2 cache on average for our workloads, and few workloads show more than 12K at maximum. We use an over-provisioned BT structure with 16K entries which requires about 190KB, and the relevant FT requires 80KB, totaling 270KB. This is about 7.5% overhead to the total GPU cache hierarchy. In practice, however, an adequately provisioned structure (e.g., with 8K entries) can mostly eliminate the invalidation overhead for our workloads.

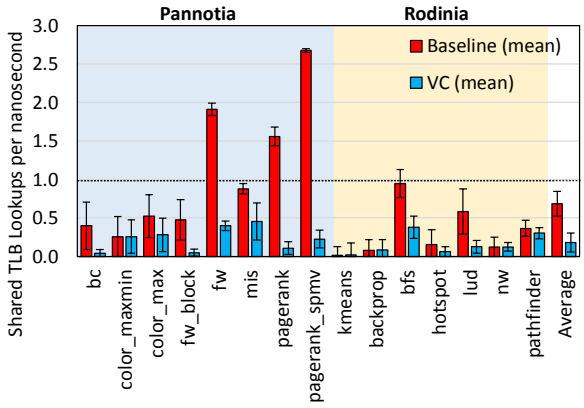

Figure 9: Bandwidth Requirements of shared TLB Lookups

Security Filter: Since the CUs are not allowed to issue physical addresses, a virtual cache hierarchy provides the same security benefits as a border control unit [24]. Similar to the border control unit, the BT tracks the permissions of each page cached by the GPUs. Additionally, every GPU memory access is checked at the IOMMU to ensure it is an allowed request. This protects the system from potentially buggy GPU hardware and buggy GPU applications.

5. EVALUATION

Our experimental evaluation presents the effectiveness of the proposed GPU virtual cache hierarchy. We run both CPU and GPU parts in the full system mode of gem5-gpu [29]. We described the details of the evaluation methodology and a set of simulated workloads earlier in Section 3.1. As a baseline, we choose a 512-entry shared L2 TLB for simulation purposes. A larger TLB would fit the entire working sets of some of our workloads, which is unrealistic for future integrated CPU-GPU systems with potentially 100's of GBs of DRAM via expandable DIMMs. Since we are focusing on the performance of the GPU address translation, we only report the time that the application executes on the GPU. We analyze the filtering effects of our proposal in Section 5.1. Then we evaluate the performance benefits in Section 5.2. Finally, we will discuss why a virtual L1 only cache design (i.e., L2 cache is physically addressed) is less effective in terms of performance and design complexity.

5.1 Virtual Cache Hierarchy's Filtering

As discussed in Section 3, a major source of GPU address translation overheads is the significant serialization at the shared TLB ("Serialization Overhead" bar in Figure 4) due

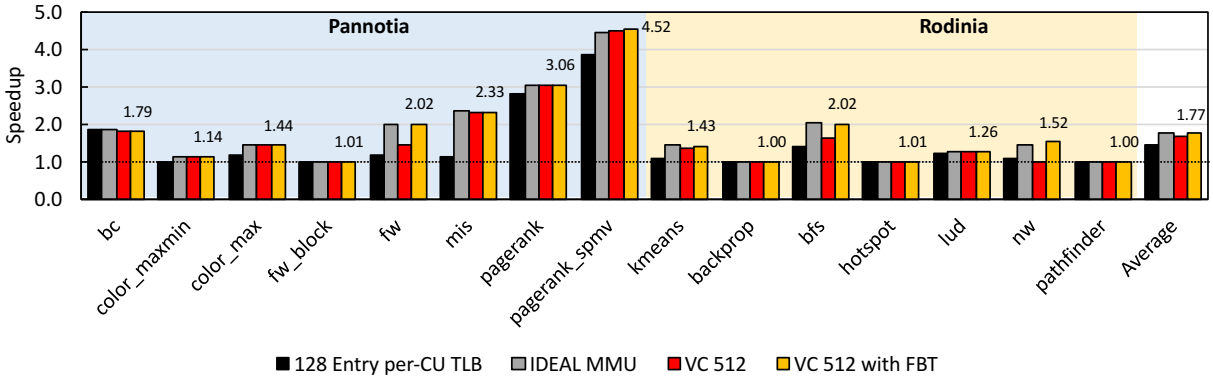

Figure 10: Performance Analysis: Speedup relative to 32-entry per-CU TLB (higher than 1.0 is better)

to high per-CU TLB miss rate (Figure 3). Accordingly, the performance benefits will be directly affected by how effectively the GPU virtual cache hierarchy filters out the shared TLB accesses. Figure 9 presents an average of shared TLB lookups per one nanosecond for the baseline and for our proposal, respectively. Each bar has a one standard deviation band for all sampling periods. The red bars for the baseline in this figure correspond to red bars in Figure 3

For most of workloads, we notice significant reductions compared to the baseline system. We observe less than 0.2 events per one nanosecond on average with our proposal. Some workloads show slightly more than one events per one nanosecond, however, they are rare events (e.g., less than 0.5% of sample periods). The results suggest that the virtual cache hierarchy is effective in reducing the load on the shared TLB causing a significant serialization overhead.

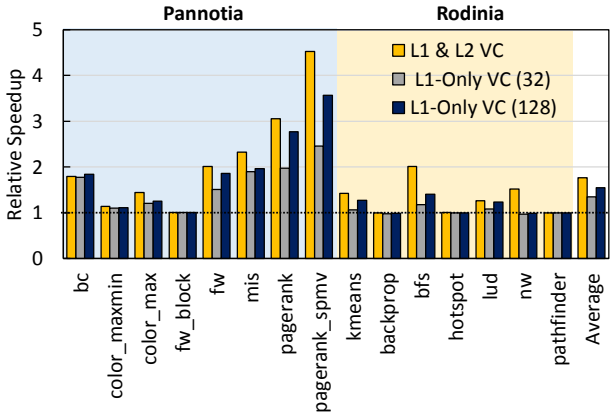

Figure 11: Comparison with L1 Virtual Cache Design

5.2 Execution Time Benefits

Figure 10 shows relative speedup compared to the baseline design with 32-entry per-CU TLB and 512-entry shared TLB, for a physical cache hierarchy with a 128-entry per-CU TLB (black bars), an IDEAL MMU with infinite capacity per-CU TLBs and infinite bandwidth at the shared TLB (gray bars), a virtual cache design (VC 512, red bars) with a 512 entry shared TLB, and a virtual cache design (VC 512 with FBT, orange bars) that employs the forward-backward table (FBT) as a second-level TLB.

Our proposal (VC 512) shows considerable speedup for most of the workloads that are sensitive to address translations (e.g., `bc`, `color_max`, `fw`, `mis`, `pagerank_spmv`, `pagerank`, `kmeans`, `bfs`, and `lud`). The average speedup is about 1.6×, with a maximum speedup of more than 4.5×. Importantly, we do not see any performance degradation.

As described in the previous sections (Section 3 and 5.1), the main benefits come from reducing the serialization overhead on the shared TLB by filtering out frequent per-CU TLB misses through virtual caches. We also find that the emerging GPU workloads from Pannotia, which is graph-based workloads, show a higher speedup than the traditional workloads from Rodinia due to high memory divergence and greater shared TLB load.

Some workloads (e.g., `fw`, `bfs`, and `nw`) do not achieve the same performance as the IDEAL MMU. This is because of the restricted reach of the 512 entry shared TLB. As described in Section 4.1 (see Address Translation), the FBT can be employed as a second-level shared TLB (VC 512 with FBT). By consulting the FBT on a shared TLB miss we can reduce the overhead and achieve the same performance as the IDEAL MMU: 1.77× speedup on average.

Black bars in Figure 10 also show the speedup using very large (128-entry) fully associative per-CU TLBs. The results show that large per-CU TLBs can hide some of the address translation overhead when using physical caches. However, using the virtual cache hierarchy still shows about 1.21× speedup over the large per-CU TLBs. In addition, virtual caches also have the benefits of reducing the power, energy, and thermal issues of consulting per-CU TLBs [31, 39].

In addition, we can expect some additional benefits by using a virtual cache hierarchy. Actually, some workloads (e.g., `pagerank` and `pagerank_spmv`) slightly outperform the IDEAL MMU. We see better cache locality in virtual caches in some cases, as performance of physical caches is sensitive to memory allocation [7, 21]. For example, `pagerank_spmv` shows more than 3% additional hits in the virtual cache hierarchy than in a physical cache hierarchy.

5.3 L1-only Virtual Caches

We also evaluate the effectiveness of a virtual cache design with only virtual L1 caches and a physical L2 cache. This system is similar to previous CPU virtual cache designs. Figure 11 shows speedup comparison between the

whole GPU virtual cache hierarchy (L1 and L2 VC) and two virtual L1 cache designs (L1-only VC). Gray bars show the speedup of the virtual L1-only design that has a 32 per-CU TLB and a 512 entry shared TLB. We also consider a larger (128-entry) per-CU TLB (black bars).

Supporting virtual caches only for GPU L1 cache can also obtain some performance benefits for the address translation sensitive workloads (more than 1.35× speedup on average). This is expected as we found many accesses that miss in per-CU TLBs but hit in the L1 cache (black bars, Figure 2).

However, we can still see noticeable gaps compared to the whole virtual cache hierarchy. Because of higher hit ratio, the larger virtual L2 cache plays a vital role as a much larger TLB miss filter; 35% more per-CU TLB misses are filtered out through the L2 virtual cache (see red bars in Figure 2). By extending the scope of virtual caches to the whole GPU cache hierarchy, we expect more latency and energy benefits. In addition, we can reduce the design complexity of the GPU cache hierarchy by completely removing per-CU TLB structures from a GPU.

6. RELATED WORK

We first elaborate on studies of virtual caching on CPUs. Some proposals take advantage of reverse mapping (physical to virtual address) table to deal with synonym accesses in virtual caches. Goodman [13] proposed dual-tags (virtual and real tags) to efficiently support reverse translations for cache coherence. Instead of using separate a reverse translation table, Wang *et al*. [41] keeps track of back-pointers with each cache line of a physical L2 cache to identify the corresponding data cached with synonyms. The similar approach is exploited to detect synonym issues for some commercial designs [35, 43]. Kaxiras and Ros [16] introduced simple virtual cache coherence. It exploits self-invalidation and downgrade to eliminates the need of reverse translations. Yoon and Sohi [44] proposed a virtual L1 cache with dynamic synonym remapping (VC-DSR) by leveraging temporal characteristics of synonyms. It dynamically decides a unique leading virtual page, which is one of synonym pages mapped to the same physical page, for all virtual cache operations. On a synonym access with a non-leading virtual page, the address is dynamically remapped to the current leading virtual address to look up the cache.

Other techniques require OS involvement to manage issues of virtual address synonyms. Some proposals advocate for eliminating or limiting sharing [8, 9, 14, 18, 42] to prevent synonyms from occurring. Enigma [45] uses an additional indirection between virtual and physical address to avoid synonym problems. The virtual to intermediate address (IA) translation is performed prior to L1 cache accesses, and the whole cache hierarchy employs the IA space. Basu *et al*. proposed an opportunistic virtual caching (OVC) for L1 virtual caches by leveraging that accesses with read-write synonyms are not prevalent [5]. The cache usually uses virtual addresses while physical addresses are used for read-write synonyms causing data consistency issues in virtual caches. This approach requires modifications of OS memory allocation. In a similar vein, Park *et al*. proposed a hybrid virtual cache [25]. This also selectively switches between virtual and physical caching like the OVC, while virtual caching is used for the entire cache hierarchy. Like the VC-DSR, this approach also performs synonym detections with Bloom filters prior to L1 cache lookups, while the information of the filters is managed by the OS. Qui and Dubois proposed a synonym lookaside buffer (SLB) [32, 33], which enforces the use of a unique primary virtual address for virtual address synonyms to avoid synonym issues. The conventional TLBs are replaced with a small scalable SLB that provides primary virtual addresses. The technique requires OS involvement to maintain mappings primary addresses and other synonyms.

We take advantage of several features of previous approaches for our proposal. However, we focus on integrating virtual caching into a GPU cache hierarchy by keeping the flexibility of the GPU's unique cache architecture without any software involvement.

There have been studies that investigate address translation for the GPU MMU. Power *et al*. [30] discussed diverse design aspects for the integration of a CPU-style MMU into a GPU. Bharath *et al.* [27] also studied how to lower the overhead of GPU address translation, but they more focused on the impact of warp schedule on the address translation. These previous works mainly focus on how to support GPU address translation, while we aim to reduce the overhead of the address translation. Others studied the overheads of virtual address translation on real GPU hardware [40]. Kumar *et al.* proposed Fusion [19], a coherent virtual cache hierarchy for specialized fixed-functional accelerators to reduce data transfer overhead between the accelerators in a tile and the host CPU.

Efficient warp schedulings [36] increase hits in a phsycal cache hierarchy. If it is applied with virtual caching, more translation hardware accesses can be filtered, which increases the effectiveness of a full virtual cache hierarchy.

7. CONCLUSION

Address translation support on GPUs is important to support flexible programmability. However, especially for emerging GPU workloads, highly divergent memory accesses put considerable pressure on address translation hardware. We show that many of these shared TLB accesses can be eliminated by using a *virtual cache hierarchy*. We present a practical and software transparent design of virtual caching for the entire GPU cache hierarchy. This virtual cache hierarchy filters most of the shared TLB accesses, which provides about 1.77× speedup on average.

Although we focused solely on GPUs in this work, we believe that other on-die accelerators that use virtual addresses may want to consider using a virtual cache hierarchy. Other accelerators will likely have similar characteristics to GPUs (e.g., unlikely to access virtual address synonyms, usually only executing a code from a single process), which makes them amenable to virtual caching. We believe these GPUs, and potentially these other co-processing units, finally provide an environment where virtual caches are both practical and provide significant benefits.

8. REFERENCES

[1] IOMMU: Virtualizing IO through IO Memory Management Unit

(IOMMU).
http://pages.cs.wisc.edu/ basu/isca_iommu_tutorial/index.htm.

[2] PS4 Pro Specs: How Does It Fare Against Xbox Project Scorpio? Which One Is Better? http://www.itechpost.com/articles/50922/20161107/ps4-pro-specs-fare-against-xbox-project-scorpio-one-better.htm.

[3] Todd M. Austin and Gurindar S. Sohi. High-bandwidth Address Translation for Multiple-issue Processors. In *Proceedings of the 23rd Annual International Symposium on Computer Architecture*, ISCA '96, pages 158–167, New York, NY, USA, 1996. ACM.

[4] Arkaprava Basu, Jayneel Gandhi, Jichuan Chang, Mark D. Hill, and Michael M. Swift. Efficient Virtual Memory for Big Memory Servers. In *Proceedings of the 40th Annual International Symposium on Computer Architecture*, ISCA '13, pages 237–248, New York, NY, USA, 2013. ACM.

[5] Arkaprava Basu, Mark D. Hill, and Michael M. Swift. Reducing Memory Reference Energy with Opportunistic Virtual Caching. In *Proceedings of the 39th Annual International Symposium on Computer Architecture*, ISCA '12, pages 297–308, Washington, DC, USA, 2012. IEEE Computer Society.

[6] Abhishek Bhattacharjee, Daniel Lustig, and Margaret Martonosi. Shared Last-level TLBs for Chip Multiprocessors. In *Proceedings of the 2011 IEEE 17th International Symposium on High Performance Computer Architecture*, HPCA '11, pages 62–63, Washington, DC, USA, 2011. IEEE Computer Society.

[7] Michel Cekleov and Michel Dubois. Virtual-Address Caches, Part 2: Multiprocessor Issues. *IEEE Micro*, 17(6):69–74, November 1997.

[8] Jeffrey S. Chase, Henry M. Levy, Michael J. Feeley, and Edward D. Lazowska. Sharing and Protection in a Single-address-space Operating System. *ACM Trans. Comput. Syst.*, 12(4):271–307, November 1994.

[9] Jeffrey S. Chase, Henry M. Levy, Edward D. Lazowska, and Miche Baker-Harvey. Lightweight Shared Objects in a 64-bit Operating System. In *Conference Proceedings on Object-oriented Programming Systems, Languages, and Applications*, OOPSLA '92, pages 397–413, New York, NY, USA, 1992. ACM.

[10] Shuai Che, Bradford M Beckmann, Steven K Reinhardt, and Kevin Skadron. Pannotia: Understanding irregular GPGPU graph applications. In *Proceedings of the IEEE International Symposium on Workload Characterization (IISWC), 2013 IEEE International Symposium on*, pages 185–195. IEEE, 2013.

[11] Shuai Che, Michael Boyer, Jiayuan Meng, David Tarjan, Jeremy W Sheaffer, Sang-Ha Lee, and Kevin Skadron. Rodinia: A benchmark suite for heterogeneous computing. In *Workload Characterization, 2009. IISWC 2009. IEEE International Symposium on*, pages 44–54. IEEE, 2009.

[12] Xuhao Chen, Li-Wen Chang, Christopher I. Rodrigues, Jie Lv, Zhiying Wang, and Wen-Mei Hwu. Adaptive Cache Management for Energy-Efficient GPU Computing. In *Proceedings of the 47th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-47, pages 343–355, Washington, DC, USA, 2014. IEEE Computer Society.

[13] James R. Goodman. Coherency for Multiprocessor Virtual Address Caches. *SIGPLAN Not.*, 22(10):72–81, October 1987.

[14] Mark Hill, Susan Eggers, Jim Larus, George Taylor, Glenn Adams, B. K. Bose, Garth Gibson, Paul Hansen, Jon Keller, Shing Kong, Corinna Lee, Daebum Lee, Joan Pendleton, Scott Ritchie, David A. Wood, Ben Zorn, Paul Hilfinger, Dave Hodges, Randy Katz, John Ousterhout, and Dave Patterson. Design Decisions in SPUR. *Computer*, 19(11):8–22, November 1986.

[15] Derek R. Hower, Blake A. Hechtman, Bradford M. Beckmann, Benedict R. Gaster, Mark D. Hill, Steven K. Reinhardt, and David A. Wood. Heterogeneous-race-free Memory Models. In *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '14, pages 427–440, New York, NY, USA, 2014. ACM.

[16] Stefanos Kaxiras and Alberto Ros. A New Perspective for Efficient Virtual-cache Coherence. In *Proceedings of the 40th Annual International Symposium on Computer Architecture*, ISCA '13, pages 535–546, New York, NY, USA, 2013. ACM.

[17] Hyesoon Kim. Supporting Virtual Memory in GPGPU Without Supporting Precise Exceptions. In *Proceedings of the 2012 ACM SIGPLAN Workshop on Memory Systems Performance and Correctness*, MSPC '12, pages 70–71, New York, NY, USA, 2012. ACM.

[18] Eric J. Koldinger, Jeffrey S. Chase, and Susan J. Eggers. Architecture Support for Single Address Space Operating Systems. In *Proceedings of the Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS V, pages 175–186, New York, NY, USA, 1992. ACM.

[19] Snehasish Kumar, Arrvindh Shriraman, and Naveen Vedula. Fusion: Design Tradeoffs in Coherent Cache Hierarchies for Accelerators. In *Proceedings of the 42Nd Annual International Symposium on Computer Architecture*, ISCA '15, pages 733–745, New York, NY, USA, 2015. ACM.

[20] George Kyriazis. Heterogeneous system architecture: A technical review. *AMD Fusion Developer Summit*, 2012.

[21] William L. Lynch. The Interaction of Virtual Memory and Cache Memory. Technical Report CSL-TR-93-587, Stanford University, 1993.

[22] Jaikrishnan Menon, Marc De Kruijf, and Karthikeyan Sankaralingam. iGPU: Exception Support and Speculative Execution on GPUs. In *Proceedings of the 39th Annual International Symposium on Computer Architecture*, ISCA '12, pages 72–83, Washington, DC, USA, 2012. IEEE Computer Society.

[23] Juan Navarro, Sitararn Iyer, Peter Druschel, and Alan Cox. Practical, Transparent Operating System Support for Superpages. *SIGOPS Oper. Syst. Rev.*, 36(SI):89–104, December 2002.

[24] Lena E. Olson, Jason Power, Mark D. Hill, and David A. Wood. Border Control: Sandboxing Accelerators. In *Proceedings of the 48th International Symposium on Microarchitecture*, MICRO-48, pages 470–481, New York, NY, USA, 2015. ACM.

[25] Chang Hyun Park, Taekyung Heo, and Jaehyuk Huh. Efficient Synonym Filtering and Scalable Delayed Translation for Hybrid Virtual Caching. In *Proceedings of the 43th Annual International Symposium on Computer Architecture*, ISCA '16, Washington, DC, USA, 2016. IEEE Computer Society.

[26] Binh Pham, Viswanathan Vaidyanathan, Aamer Jaleel, and Abhishek Bhattacharjee. CoLT: Coalesced Large-Reach TLBs. In *Proceedings of the 2012 45th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-45, pages 258–269, Washington, DC, USA, 2012. IEEE Computer Society.

[27] Bharath Pichai, Lisa Hsu, and Abhishek Bhattacharjee. Architectural Support for Address Translation on GPUs: Designing Memory Management Units for CPU/GPUs with Unified Address Spaces. In *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '14, pages 743–758, New York, NY, USA, 2014. ACM.

[28] Jason Power, Arkaprava Basu, Junli Gu, Sooraj Puthoor, Bradford M Beckmann, Mark D Hill, Steven K Reinhardt, and David A Wood. Heterogeneous system coherence for integrated CPU-GPU systems. In *Proceedings of the 46th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 457–467. ACM, 2013.

[29] Jason Power, Joel Hestness, Marc Orr, Mark Hill, and David Wood. gem5-gpu: A Heterogeneous CPU-GPU Simulator. *Computer Architecture Letters*, 13(1), Jan 2014.

[30] Jason Power, Mark D Hill, and David A Wood. Supporting x86-64 address translation for 100s of GPU lanes. In *2014 IEEE 20th International Symposium on High Performance Computer Architecture (HPCA)*, pages 568–578. IEEE, 2014.

[31] Kiran Puttaswamy and Gabriel H. Loh. Thermal Analysis of a 3D Die-stacked High-performance Microprocessor. In *Proceedings of the 16th ACM Great Lakes Symposium on VLSI*, GLSVLSI '06, pages 19–24, New York, NY, USA, 2006. ACM.

[32] Xiaogang Qiu and M Dubois. Towards virtually-addressed memory hierarchies. In *Proceedings HPCA Seventh International Symposium on High-Performance Computer Architecture*, pages 51–62. IEEE, 2001.

[33] Xiaogang Qiu and Michel Dubois. The Synonym Lookaside Buffer: A Solution to the Synonym Problem in Virtual Caches. *IEEE Trans. Comput.*, 57(12):1585–1599, December 2008.

[34] Jude A. Rivers, Gary S. Tyson, Edward S. Davidson, and Todd M. Austin. On High-bandwidth Data Cache Design for Multi-issue Processors. In *Proceedings of the 30th Annual ACM/IEEE International Symposium on Microarchitecture*, MICRO 30, pages

46–56, Washington, DC, USA, 1997. IEEE Computer Society.

[35] D. Roberts, T. Layman, and G. Taylor. An ECL RISC microprocessor designed for two level cache. In *Compcon Spring '90. Intellectual Leverage. Digest of Papers. Thirty-Fifth IEEE Computer Society International Conference.*, pages 228–231, Feb 1990.

[36] Timothy G. Rogers, Mike O'Connor, and Tor M. Aamodt. Cache-Conscious Wavefront Scheduling. In *Proceedings of the 2012 45th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-45, pages 72–83, Washington, DC, USA, 2012. IEEE Computer Society.

[37] Abhayendra Singh, Shaizeen Aga, and Satish Narayanasamy. Efficiently Enforcing Strong Memory Ordering in GPUs. In *Proceedings of the 48th International Symposium on Microarchitecture*, MICRO-48, pages 699–712, New York, NY, USA, 2015. ACM.

[38] Inderpreet Singh, Arrvindh Shriraman, Wilson W. L. Fung, Mike O'Connor, and Tor M. Aamodt. Cache Coherence for GPU Architectures. In *Proceedings of the 2013 IEEE 19th International Symposium on High Performance Computer Architecture (HPCA)*, HPCA '13, pages 578–590, Washington, DC, USA, 2013. IEEE Computer Society.

[39] Avinash Sodani. Race to Exascale: Opportunities and Challenges. MICRO 2011 Keynote talk.

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

[41] W. H. Wang, J.-L. Baer, and H. M. Levy. Organization and Performance of a Two-level Virtual-real Cache Hierarchy. In *Proceedings of the 16th Annual International Symposium on Computer Architecture*, ISCA '89, pages 140–148, New York, NY, USA, 1989. ACM.

[42] D. A. Wood, S. J. Eggers, G. Gibson, M. D. Hill, and J. M. Pendleton. An In-cache Address Translation Mechanism. In *Proceedings of the 13th Annual International Symposium on Computer Architecture*, ISCA '86, pages 358–365, Los Alamitos, CA, USA, 1986. IEEE Computer Society Press.

[43] Kenneth C. Yeager. The MIPS R10000 Superscalar Microprocessor. *IEEE Micro*, 16(2):28–40, April 1996.

[44] H. Yoon and G. S. Sohi. Revisiting virtual L1 caches: A practical design using dynamic synonym remapping. In *2016 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 212–224, March 2016.

[45] Lixin Zhang, Evan Speight, Ram Rajamony, and Jiang Lin. Enigma: Architectural and Operating System Support for Reducing the Impact of Address Translation. In *Proceedings of the 24th ACM International Conference on Supercomputing*, ICS '10, pages 159–168, New York, NY, USA, 2010. ACM.