Filtering Translation Bandwidth with Virtual Caching

Hongil Yoon*
Computer Sciences Department,
University of Wisconsin-Madison
ongal@cs.wisc.edu

Jason Lowe-Power
Computer Science Department,
University of California, Davis
jlowepower@ucdavis.edu

Gurindar S. Sohi
Computer Sciences Department,
University of Wisconsin-Madison
sohi@cs.wisc.edu

Abstract

Heterogeneous computing with GPUs integrated on the same chip as CPUs is ubiquitous, and to increase programmability many of these systems support virtual address accesses from GPU hardware. However, this entails address translation on every memory access. We observe that future GPUs and workloads show very high bandwidth demands (up to 4 accesses per cycle in some cases) for shared address translation hardware due to frequent private TLB misses. This greatly impacts performance (32% average performance degradation relative to an ideal MMU).

To mitigate this overhead, we propose a *software-agnostic, practical, GPU virtual cache hierarchy*. We use the virtual cache hierarchy as an *effective address translation bandwidth filter*. We observe many requests that miss in private TLBs find corresponding valid data in the GPU cache hierarchy. With a GPU virtual cache hierarchy, these TLB misses can be filtered (i.e., virtual cache hits), significantly reducing bandwidth demands for the shared address translation hardware. In addition, accelerator-specific attributes (e.g., less likelihood of synonyms) of GPUs reduce the design complexity of virtual caches, making a whole virtual cache hierarchy (including a shared L2 cache) practical for GPUs.

Our evaluation shows that the entire GPU virtual cache hierarchy effectively filters the high address translation bandwidth, achieving almost the same performance as an ideal MMU. We also evaluate L1-only virtual cache designs and show that using a whole virtual cache hierarchy obtains additional performance benefits (1.31× speedup on average).

Keywords Virtual caching, Heterogeneous computing, TLB, Virtual memory, Address translation

*Now at Google

ASPLOS '18, March 24–28, 2018, Williamsburg, VA, USA

ACM ISBN 978-1-4503-4911-6/18/03…$15.00
https://doi.org/10.1145/3173162.3173195

ACM Reference Format:
Hongil Yoon, Jason Lowe-Power, and Gurindar S. Sohi. 2018. Filtering Translation Bandwidth with Virtual Caching. In *ASPLOS '18: Architectural Support for Programming Languages and Operating Systems, March 24–28, 2018, Williamsburg, VA, USA.* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3173162.3173195

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 (e.g., Heterogeneous System Architecture (HSA) [1, 2, 28]). 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 requires translating virtual addresses to physical addresses on every cache access. The overhead has significant performance and energy impact [4, 8, 32, 46]. Conventional CPUs mitigate the overhead by accessing TLBs prior to or in parallel with cache lookups. Modern GPU architectures have reflected CPU-style memory management unit (MMU) [20, 34, 47, 50]. However, CPU-style MMUs are not effective because of GPU microarchitecture and workload differences.

There are three characteristics of GPUs which cause increased virtual memory overheads compared to CPUs. First, GPUs have many more compute units than most CPUs have cores (e.g., 40 compute units in an XBOX ONE [14]), which puts pressure on shared translation resources (e.g., IOMMU TLBs). Second, GPUs have wide SIMD units. Each compute unit has many lanes (e.g., 32), and a single static GPU load or store instruction can issue scatter/gather requests to 10's of different addresses. These requests can even be to different virtual memory pages. Furthermore, GPUs are built to tolerate memory latency by having many execution contexts. These execution contexts are like simultaneous multithreading (SMT) on CPUs, except GPUs can have up to 40 active contexts compared to 2–8 for CPU cores. Thus, GPUs can generate much greater memory-level parallelism than CPUs, putting considerable pressure on address translation resources.

This high pressure on GPU translation resources greatly impacts performance and energy consumption. When using

a practical translation implementation that has private TLBs, a large shared IOMMU TLB, and a multi-threaded page table walker, we observe that future GPUs and workloads suffer from very high *private* GPU TLB miss ratios (average 56%) and high miss rate (more than 4 misses per cycle in some cases), resulting in a high bandwidth demand of shared address translation hardware (e.g., a shared IOMMU TLB). The workloads experience an average of 32% performance degradation over an ideal GPU MMU. This translation overhead is higher for emerging GPU applications (e.g., graph-based workloads) than traditional workloads. *The major source of this overhead is the serialization delays at the shared address translation hardware due to its limited bandwidth.* The bandwidth demands on translation hardware will continue to increase as i) future GPUs integrate more compute units [6, 14] and ii) future workloads will access hundreds of GBs of data [4, 20, 38].

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 existing 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 [53]. We empirically observe that more than 60% of references that miss in the private GPU TLBs find corresponding data in the cache hierarchy (i.e., are virtual cache hits). Filtering out the TLB misses leads to considerable performance benefits.

Virtual caching has been proposed several times to reduce the translation overheads on CPUs [5, 16, 21, 31, 41, 48, 52]. However, to the best of our knowledge, the efficacy of a GPU virtual cache hierarchy as a translation bandwidth filter 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 larger caches exacerbate the synonym problem [52]. In larger caches, there is longer data residence time and there is a higher likelihood for synonymous accesses.

We leverage GPU's accelerator-specific attributes, such as few active virtual address spaces and no OS kernel execution to greatly simplify virtual cache implementation. These attributes reduce the likelihood of synonyms and homonyms which are the crux of the problems of virtual cache designs. 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 advantage of virtual caching as a bandwidth filter of TLB misses. Including the shared L2 cache filters more than double the number of TLB accesses to the shared address translation hardware, compared to virtualizing L1 caches alone (31% with L1 virtual caches and 66% with both virtual L1 and L2 caches).

These observations lead to our practical 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. This structure builds off of previous virtual cache proposals from the CPU domain and ensures correct execution of virtual caches for synonyms, TLB shootdown, cache coherence, etc.

We show that the proposed GPU cache hierarchy (both L1 and L2 caches) achieves almost the same performance as an ideal GPU MMU design. We also observe that the entire GPU virtual cache hierarchy shows more than 30% additional performance benefits over L1-only GPU virtual cache design.

In this paper:

1. We identify that a major source of GPU address translation overheads is the high bandwidth demand for the shared translation hardware (i.e., a shared TLB).
2. We show the efficacy of a GPU virtual cache hierarchy as an address translation bandwidth filter.
3. We propose a practical, software-agnostic virtual cache hierarchy for GPUs. Our proposal allows the flexibility for modern GPUs to keep their unique cache architectures (e.g., L1 caches without support for cache probes).

2 Background

In this section, we first discuss the baseline SoC package design and the baseline GPU address translation. We then give an overview of virtual caches and their design issues.

2.1 GPU Address Translation

Figure 1 overviews the baseline SoC package. In this paper, we consider fully coherent CPUs and GPUs with unified (shared) address space support (e.g., HSA specification [28]). In current systems, each computational unit (CU) has a private TLB, coalescer, and scratchpad. The TLB is consulted after the per-lane accesses have been coalesced into the minimum number of memory requests; it is not consulted for scratchpad memory accesses. When a private TLB miss occurs, an address translation service request is sent to the I/O 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, which adds transfer latency to TLB miss requests [22].

The IOMMU consists of a TLB that is shared between all of the CUs, page table walker (PTW), and page walk cache

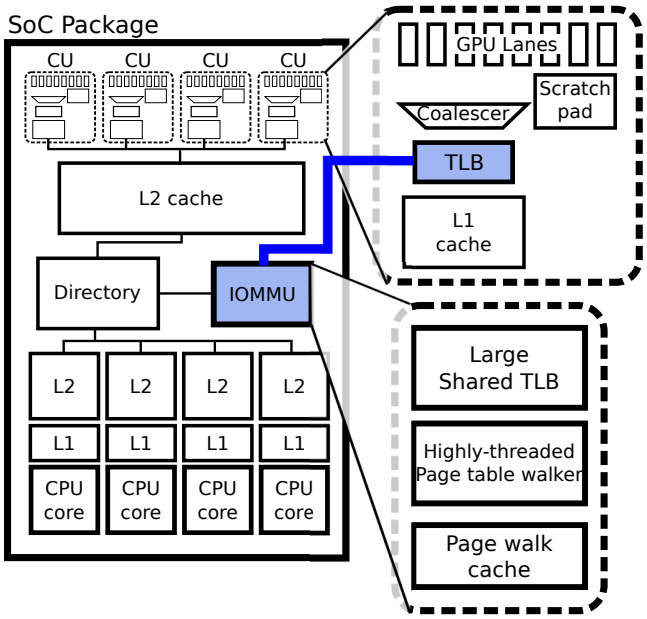

Figure 1. Overview of baseline SoC package design with a physical cache hierarchy.

(PWC).[1] On the IOMMU 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 IOMMU TLB misses [22, 37, 47]. Additionally, the PWC decreases the latency of page table walks by leveraging the locality of page table accesses (e.g., accesses to page directory entries). Once the address translation is successfully performed, 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, 16, 21, 31, 41, 48, 52]. 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 [53].

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 lower-level (larger) caches. However, the increase in the lifetime of data in the caches makes the occurrence of synonym accesses more likely [52], 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 implementing the entire GPU cache hierarchy (L1 and L2 caches) with virtual addresses is practical in terms of both performance as well as design complexity as discussed in Section 3.1.

[1] In this paper, we focus on the IOMMU TLB. However, if the GPU contains a TLB structure shared by all CUs, it will exhibit characteristics similar to that of the IOMMU TLB.

CPU	1 core, 3GHz, 64KB D$, 32KB I$, 2MB L2$
GPU	16 CUs, 32 lanes per CU, 700 MHz
L1 GPU Cache	per-CU 32KB, write-through no allocate
L2 GPU Cache	Shared 2MB, 8 banks, write-back, 128B lines
TLBs	32-entry Per-CU TLBs (4 KB pages)
IOMMU	Shared TLB (512-entry or 16K-entry), 16 concurrent PTW, and 8KB page-walk cache
DRAM, NoC	192 GB/s, Dance-hall topology in the GPU and Point-to-point network between the CPU-GPU

Table 1. Simulation configuration details.

3 Motivation

In this section, we discuss opportunities of a GPU virtual cache hierarchy. To get the empirical data, we used a full-system heterogeneous simulator, gem5-gpu [36], which models all system-level virtual memory operations. We consider a high-performance integrated CPU-GPU system with a unified shared address space and coherence among GPU and CPU caches. Table 1 contains our simulation details.

Our baseline MMU design is based on a recent work proposed by Power et al. [37], and current hardware designs [20, 47]. We evaluate a 32-entry L1 per-CU TLB as a baseline (Karnagel et al. found a current GPU design had a 16-entry L1 TLB [20]), but we also evaluate larger per-CU TLBs. We assume a large shared IOMMU TLB in the GPU that is shared between all of the CUs; the IOMMU TLB can also be considered a second-level shared GPU TLB. We assume this IOMMU TLB can process up to one request per cycle.[2] We use 16 page table walkers to handle misses from the IOMMU TLB [47]. We also have an 8 KB physical cache for the page table walkers, as prior work found this is important for high-performance translation [37].

We evaluate workloads from two different benchmark suites. The Rodinia workloads [12] represent traditional GPU workloads and are mostly scientific computing algorithms. We also use Pannotia [11] to evaluate emerging GPU workloads. The Pannotia workloads are graph-based and show less locality than traditional GPU workloads. The Pannotia benchmark suite comes with multiple versions of algorithms. We present data for each version of the algorithm separately. We run all workloads to completion.

[2] Power *et al.* [37] considered infinite bandwidth of the shared IOMMU TLB, which is unrealistic. They attacked the bandwidth issue of the PTW on the shared TLB misses.

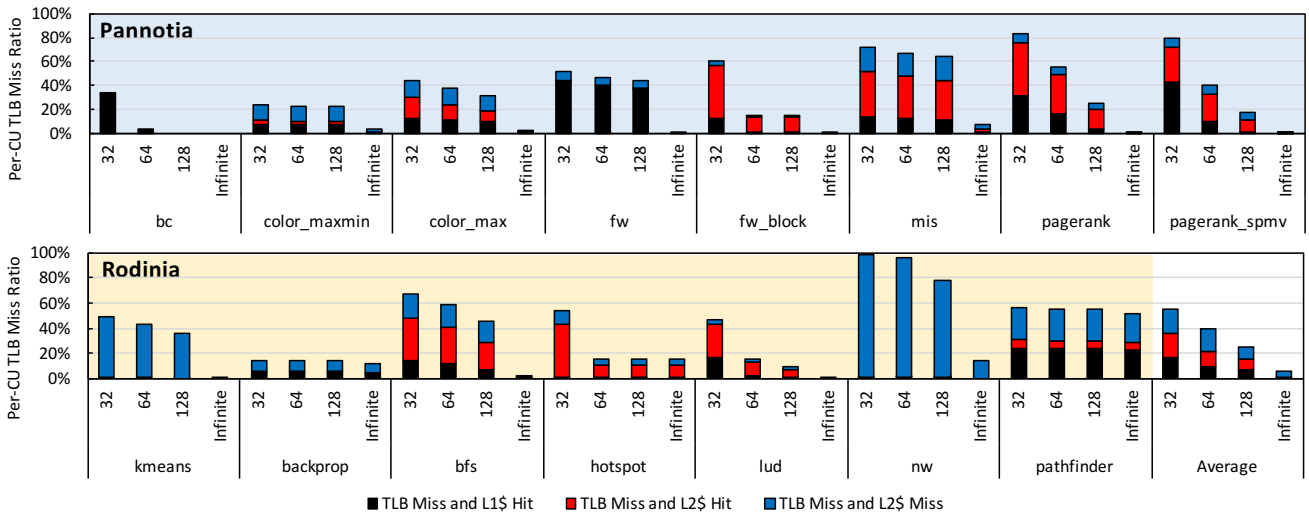

Figure 2. Breakdown of per-CU TLB miss accesses. Top shows the results of workloads from Pannotia suite (irregular graph applications). Bottom shows the results of seven workloads from Rodinia suite and the average across all simulated workloads.

3.1 Virtual Cache Hierarchy Opportunities

Observation 1: *A large fraction of TLB misses find data in the GPU data caches.*

Figure 2 presents private 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. The total height of each bar shows the average miss ratio for the entire execution of the application. Figure 2 also 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 bars in the L1 cache and red bars in the L2 cache).

We notice that the per-CU TLB miss ratio is high; however, many TLB misses hit in the GPU caches. Only 34% of references that miss in the 32-entry per-CU L1 TLB are also L2 cache misses and access main memory (blue bars). 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 because blocks in the cache hierarchy are likely to reside longer than the lifetime of the corresponding per-CU TLB entries (see Appendix), and these blocks are from numerous pages (e.g., 6000 different 4KB pages on average for our workloads). Thus, using a virtual cache hierarchy considerably increases address translation reach when considering locality in the cache hierarchy.

Using virtual caches eliminates TLB misses that find valid data in the cache hierarchy (66% of TLB misses) since virtual cache hits do not access the address translation hardware. Figure 2 shows that even for large per-CU TLBs (128 entries) about the same percentage (65%) of TLB misses can be filtered by virtual caches. This behavior is more pronounced for the emerging graph workloads in the Pannotia suite, and future workloads with similar access patterns will also benefit from virtual caches. Additionally, as cache sizes increase more accesses will hit in the caches, improving the efficacy of a virtual cache design.

There is a large opportunity for virtual caches to filter TLB misses as high per-CU L1 TLB miss ratios are common in GPU applications. One factor contributing to high TLB miss ratios is memory divergence (i.e., scatter/gather). Even with large TLBs of 64 and 128 entries, we see frequent per-CU TLB misses due to high memory divergence (e.g., `color_max`, `fw`, `mis`, and `bfs`). For instance, `fw` averages 9.3 memory accesses per dynamic memory instruction.

Some applications (e.g., `pathfinder` and `nw`) have a high infinite private TLB miss ratio. Most of these two applications' memory accesses are to scratchpad memory which does not access the TLB (not shown in Figure 2). When these applications read data from main-memory, there is high memory divergence due to scatter/gather requests. Thus, we see a large burst of TLB misses at the beginning and end of each GPU kernel when loading data into and storing data from the scratchpad cache. These bursts of TLB misses cause a high TLB miss ratio, but do not significantly affect performance due to the GPU's ability to hide the extra memory access latency for these accesses. Future applications (e.g., Pannotia) are less likely to use the scratchpad cache given their input-dependent access pattern.

Observation 2: *GPU workloads very frequently access the IOMMU TLB.*

Figure 3 presents IOMMU TLB accesses per cycle (i.e., per-CU TLB misses for all CUs). We track the number of accesses in each microsecond interval. Blue bars indicate the average of events across all sampling periods. Each bar has a one standard deviation band, and red dots indicate the maximum of accesses per cycle among samples. For the results, 32-entry per-CU TLBs are used. The experiment for this figure assumes that the IOMMU TLB can be accessed any number of times per cycle, which is impractical. The results are sorted by the frequency of the accesses.

We observe about one IOMMU TLB access per cycle, on average. Most workloads show bursts of accesses during which the IOMMU TLB is accessed more than once per cycle (refer to the red dots for the maximum accesses). For example, `color_max` shows about 25% of sample periods with more than one IOMMU TLB accesses per cycle. We also observe that graph-based workloads like Pannotia show much more frequent per-CU TLB misses than the traditional workloads because of high memory divergence [11]. The results show there are impractical bandwidth requirements at the IOMMU TLB (e.g., more than two accesses per cycle in some cases).

Observation 3: *The GPU shows high serialization overhead at the IOMMU TLB due to its limited bandwidth.*

Figure 4 presents the performance overhead of address translation on the GPU for all simulated workloads, compared to an "IDEAL MMU" that has infinite capacity per-CU and IOMMU TLBs, minimal latency, and infinite IOMMU TLB bandwidth. The overhead is classified according to two sources: 1) the page table walk (PTW) overhead on IOMMU TLB misses and 2) the serialization delay due to request queuing at the IOMMU TLB. This figure presents the average performance across all of the workloads evaluated.

For a baseline design (Small IOMMU TLB) that has 32-entry per-CU TLBs and a 512-entry IOMMU TLB, we see an average of 1.77× runtime overhead over the "IDEAL MMU." We also consider a design with a large (16K-entry) IOMMU TLB (Large IOMMU TLB) to isolate the impact of the serialization delay. We observe that the performance benefits with a large capacity IOMMU TLB are small, indicating that the PTW overhead is not a significant factor. This is because the multi-threaded page table walker with a large page walk cache effectively hides IOMMU TLB miss latency [37]. The results suggest that **most of the address translation overheads are due to serialization at the IOMMU TLB, not its capacity**.

Observation 4: *GPU requires impractically high bandwidth to alleviate the serialization overhead at the IOMMU TLB.*

Figure 5 breaks down the performance overheads, compared to the "IDEAL MMU." This figure presents only the average performance across the workloads showing high

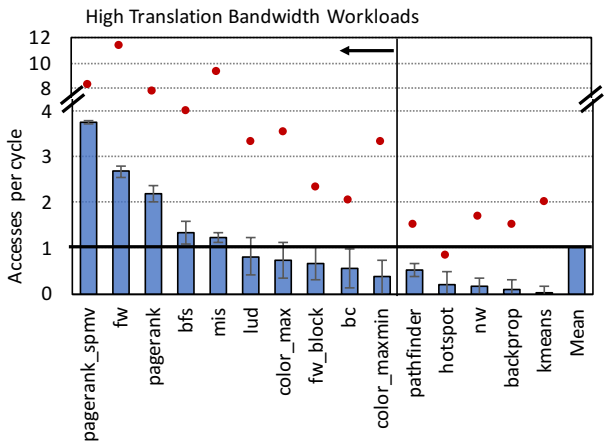

Figure 3. Analysis of IOMMU TLB access rate. Red dots indicate the maximum of accesses per cycle among samples.

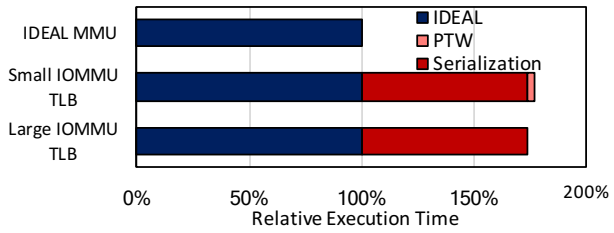

Figure 4. GPU address translation overheads for all simulated workloads.

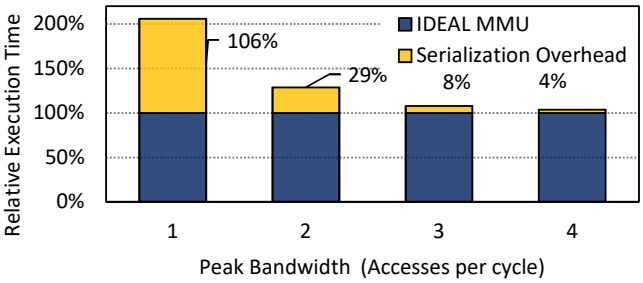

Figure 5. Impact of the IOMMU TLB bandwidth limit on the serialization overhead for high translation bandwidth workloads.

translation bandwidth demands in Figure 3.[3] We consider a large capacity (16K-entry) IOMMU TLB to focus solely on the serialization overheads due to the bandwidth limit. The performance overheads reduce when the bandwidth of the IOMMU TLB is increased. However, to limit this overhead a very high bandwidth TLB is required. It is costly in terms of both area and power to implement a large high-bandwidth associative structure like a TLB. Thus, the empirical results suggest that *the primary challenge of GPU address translation is to provide high bandwidth at the IOMMU TLB.*

[3] Other workloads are not considered here because their performance is less affected by the bandwidth limit of the IOMMU TLB. We discuss performance benefits of the workloads with low bandwidth demands in Section 5.2.

Observation 5: *GPUs' accelerator usage pattern reduces the likelihood of virtual cache synonym issues.*

Others have observed more synonym accesses with larger virtual caches in CPU cache hierarchies [52]. However, GPUs are not fully general-purpose compute devices, and there are three key differences compared to CPUs which reduce the impact of synonyms in GPU cache hierarchies.

First, as an accelerator, GPUs execute a small number of applications at a time. Thus, there is usually few active virtual address spaces, reducing the likelihood of data sharing among them. 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 offload OS function calls to CPUs [22, 24, 43]. 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.

3.2 Discussion of Conventional Mechanisms

Before discussing the design details of our virtual cache hierarchy proposal, we discuss possible questions related to conventional high-bandwidth translation mechanisms.

Larger (or multi-level) per-CU TLBs: A larger per-CU TLB would fit working sets of some of our simulated workloads. However, even with large TLBs of 128 entries (Figure 2), some workloads show frequent per-CU TLB misses. Additionally, even for large TLBs we find a significant fraction of TLB misses can be filtered by a virtual cache hierarchy (65% for 128-entry TLBs).

Keeping address translation overheads in check when using a physical cache hierarchy requires adding more TLBs as the number of CUs increases and increasing the size of the TLB to cover larger working sets. Future integrated GPUs will have more CUs [6, 14], and we believe that future workloads will access hundreds of GBs of data [4, 20, 38]. These two trends put increased pressure on GPU address translation hardware.

Large (or multi-level) per-CU TLB designs for physical caches increase the power and area overheads of address translation [7] although they reduce the per-CU TLB miss rate. For example, consulting highly associative large TLBs causes hotspots due to high power dissipation [39]. Thus, the scalability of TLB reach is restricted [19].

Multi-banked (or multi-ported) large IOMMU TLB: Highly multi-banking the IOMMU TLB, while costly in terms of complex interconnection and arbitration logic, could increase the bandwidth if bank conflicts are rare [3, 42]. 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 high translation bandwidth workloads (e.g., `mis`, `color_max`, etc.) show frequent conflicts. Frequent bank conflicts limit the bandwidth of multi-banked designs.

Multi-ported designs increase the bandwidth by allowing multiple simultaneous accesses. However, to support high bandwidth for a large IOMMU TLB (e.g., 16K entries) with long access latency, numerous ports are required. More ports lead to longer access latency [49], which requires extra ports to satisfy particular high bandwidth demands. In addition, the design suffers from area overhead due to additional wires for the ports [3].

Large Pages: Using large pages can effectively reduce TLB miss overhead [33, 37, 47]. However, they are not a panacea [4, 30]. Basu et al. show that, for big-memory CPU workloads (e.g., working sets of about 100 GB), large pages only slightly reduce the TLB miss overhead [4]. In addition, large pages do not help workloads with poor locality. Karnagel et al. show that data-intensive workloads with irregular access patterns lead to high address translation overheads on GPUs [20].

We may selectively combine these mechanisms. However, the bandwidth demands will continue to increase as future GPUs integrate more CUs. The recently released PlayStation 4 Pro console has 36 CUs [6], and the XBOX ONE has 40 CUs [14]. *Thus, we need a scalable, efficient way of filtering the accesses to the shared IOMMU TLB.*

3.3 Summary and Rationale of Our Approach

We find the address translation overhead on GPUs is mostly due to the limited bandwidth of the shared IOMMU TLB. Our goal is to reduce the number of accesses to this centralized shared structure. We find that many TLB misses hit in the GPU caches, and, thus, a virtual cache hierarchy can be an *effective address translation bandwidth filter*.

Specifically, we make the following observations:

- Many per-CU L1 TLB misses find valid data in the GPU cache hierarchy (L1 or L2 cache).
- There are many accesses per cycle to the shared translation structure, which causes significant serialization delays and hurts performance.
- The complexities of managing deep virtual cache hierarchies (e.g., synonyms) are unlikely for GPU workloads which allows us to extend the GPU virtual cache hierarchy to large caches increasing its benefits.

These empirical observations suggest that a GPU virtual cache hierarchy is a promising and effective GPU address translation filter. As we will establish below, our approach is a scalable mechanism. Our hardware overhead scales with capacity of the *existing* caches, not workload size (which is scaling much faster [15, 19]).

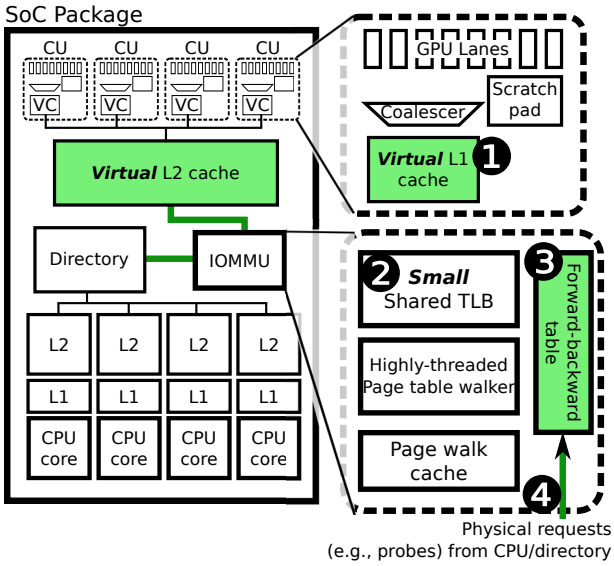

Figure 6. Proposed GPU virtual cache hierarchy.

4 Practical GPU Virtual Cache Hierarchy

We first set two key requirements to design a practical GPU virtual cache hierarchy:

1. The proposed GPU virtual cache hierarchy should be able to efficiently deal with potential issues from virtual memory idiosyncrasies, such as potential virtual address synonyms, TLB shootdowns, etc., without *OS involvement*.
2. The proposed design should be seamlessly integrated into a modern GPU cache hierarchy, which is different from a traditional CPU cache hierarchy.

To satisfy the requirements, we base our GPU virtual cache design on a recent CPU L1 virtual cache design [52] as an example. Further, we extend the approach for a whole GPU cache hierarchy (including a shared L2) by changing the design of the IOMMU. However, we expect that other CPU virtual cache proposals (Section 6) will achieve similar benefits on GPUs.

Overview of our proposal: 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, and the GPU L1 and L2 caches 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.

To correctly and efficiently support virtual memory, we add a *forward-backward table* (FBT) to the IOMMU. The FBT is fully inclusive of the virtual caches with an entry for every page that is currently cached. The FBT consists of two parts: *backward table* and *forward table* (see Figure 7). First, the backward table (BT) is primarily a reverse translation

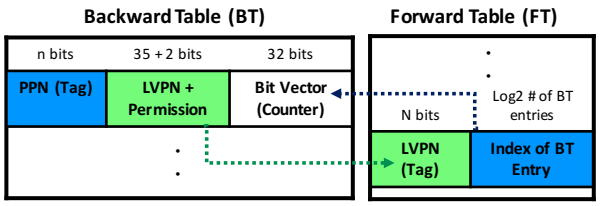

Figure 7. Overview of a Forward-Backward Table (FBT). PPN denotes physical page number, and LVPN denotes leading virtual page number.

table which provides reverse translations from physical addresses to virtual addresses. The BT tracks mappings from a physical address to a *unique* leading virtual address [52]. The leading virtual address is the first virtual address used to reference a given physical address and is used to place the data during the data residence in the virtual caches. We use this information to detect synonymous accesses and handle 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 forward table (FT). The FT tracks mappings from a leading virtual address to an index of a corresponding backward table entry. The FT allows the FBT to be indexed by both virtual and physical addresses. Forward translation information is needed for several operations including when the virtual caches respond to coherence requests, 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.

4.1 Supporting Virtual Memory without OS Involvement

To handle synonym issues, some virtual cache proposals [5, 9, 10, 25, 31, 41, 53] 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 GPUs need to be a *software transparent technique*.

Our proposal supports virtual memory requirements without any OS involvement. The proposal allows GPUs to access 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 supporting a non-inclusive 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, the requests are sent to the L1 virtual cache without accessing a TLB (① in Figure 6). Unlike a physical cache design, the permissions of the virtual page are maintained with each cache line, and the permission check is performed on virtual cache accesses.

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 if the permissions match. On L1 or L2 virtual cache hits, no address translation overhead is imposed.

Address Translation: When an L2 virtual cache miss occurs, the request 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.

We observe that, for most shared TLB misses (e.g., 74% on average), a matching entry is found in the FBT.[4] 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. The actions taken on a shared TLB and FBT miss are the same as the baseline IOMMU design. If there is no matching address translation, a page table walker (PTW) executes the corresponding page table walk. If the PTW fails to find a matching PTE, a page fault exception occurs; this exception is handled by the CPU.

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) obtained by the shared TLB lookup or the PTW (③ in Figure 6). Each entry has a physical page number (PPN) as a tag and stores the unique leading virtual page number (LVPN). *Only this unique 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 replay the virtual cache access with the leading virtual address. 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.

Synonym accesses with non-leading virtual addresses always lead to cache misses in the virtual cache hierarchy. If the same address is accessed multiple times with a non-leading virtual address, it will miss in the cache and will be replayed on every access. We do not believe this significantly affects performance because GPUs' accelerator usage pattern reduces the likelihood of virtual cache synonym issues (Observation 5). Future GPU systems may show more synonym accesses, but the overhead can be mitigated by integrating the concepts of dynamic synonym remapping [52] to our proposal (Section 4.3).

Cache Coherence between GPUs and CPUs: 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 (④ 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. 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 [35].

TLB Shootdown and Eviction of FBT Entry: If information of a virtual page changes (e.g., permission or page mapping), the FBT and cache entries corresponding to that virtual page are no longer valid. In this case, the corresponding FBT entry should be invalidated. This leads to invalidations of data cached with the virtual address from all virtual caches. While the invalidation is in progress, the FBT entry is locked and no new requests can be initiated for the virtual page. All L1 caches are checked and all outstanding L2 cache misses for the page must be completed before acknowledging the TLB shootdown. 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 the bit vector, we can selectively evict only the cached data, reducing invalidation traffic.

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

On a single-entry TLB shootdown, we use the FT with virtual addresses. Once a match is found, we directly access a matching BT entry with the index pointer. The FT filters TLB invalidation requests if no match is found. On an all-entry TLB shootdown, a cache flush is required. The shootdowns resulting from page mapping or permission changes are infrequent [5]. Thus, the performance impact is minor.

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

4.2 Integration with a 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 [41, 52]. For example, a load may not identify the corresponding older store since their virtual addresses differ, even though their corresponding physical addresses are same. This load could get the stale value from a virtual cache earlier than the completion of the store.

Read-write synonyms can result in correctness issues for GPUs with a virtual cache hierarchy. GPUs do not support precise exceptions or recovery mechanisms [18, 23, 29, 44]. Therefore, when we detect a read-write synonym at the FBT, it may be too late to identify and roll back the violating instruction from the processor pipeline.

We detect all synonymous accesses at the FBT, and conservatively cause a fault when a read-write synonym access is detected.[5] 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. It is possible to have user-mode read-write synonyms, but 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.

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 [52] 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 [29]. Regardless of the issue of read-write synonyms, we fully support the much more common read-only synonym accesses.

Multi-Level Cache with Non-Inclusion Property: In the previous sections we conceptually explain the operations with a two-level inclusive cache hierarchy. In practice, however, modern GPUs employ a non-inclusive hierarchy [13, 45]. 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.

In practice, modern GPU L1 caches are not coherent, and thus we do not have to track the precise information. Instead, 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 virtual page is no longer valid (e.g., an eviction of FBT entry or TLB shootdown), an invalidation message is sent to all L1 caches to invalidate all lines with a matching virtual page address. To avoid walking the L1 cache tags, we add a small invalidation filter at the L1 caches. Each entry has a 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 eliminate most 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 multi-process 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 [52] 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

[5] Read-write synonym accesses are detected at the BT. Since the FBT forwards all coherence requests from GPUs 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.

due to synonyms. Homonym issues can be easily managed by employing address space identifier (ASID) information; each cache line needs to track the corresponding ASID information. This prevents cache flushes on context switches.

Large Page Support: In our design, supporting larger pages does not cause any correctness issues. But, it is impractical to allocate FBT entries for large pages since it requires a 16,384-bit vector to track cached lines from a 2MB page. A bit vector is simply an optimization to track precise information about cached lines, enabling selective cache line invalidation. Thus, instead of using a bit vector, an associated counter can be employed without any correctness issues. Using a counter requires walking through lines in the virtual caches until all of the lines from the page are invalidated.

As an optimization, we can optionally break the large page into 4KB subpages and use normal FBT entries to track the large page at a subpage granularity. In this case, there is no need to preallocate FBT entries for the entire large page. Instead, we only need to allocate an entry when the subpage is accessed, saving FBT space on sparsely accessed large pages.

Area Requirements: To support the proposed design, each cache line entry needs to be extended to track some extra information (e.g., virtual tags, permissions), 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 extra line-level information (e.g., extra bits for virtual tags and permissions) is about 1% overhead to the total GPU cache hierarchy when all components (e.g., 128B line, physical tags, LRU information, etc.) are considered. 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 pages at maximum. We model a 16K-entry BT structure which is large enough to cover a unique page for every L2 cache entry. This structure 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 eliminate most of the invalidation overhead for our workloads. Additionally, the FBT is not significantly larger than the large shared IOMMU TLB in current GPUs (1000s of entries). The FBT is combined with a small (512-entry) IOMMU TLB (see Figure 6).

5 Evaluation

In this section, we present the effectiveness of our proposal. We described the details of the evaluation methodology and simulated workloads earlier in Section 3. We model 10 cycle interconnect latency between a GPU L2 cache and FBT, and 5 cycles for FBT lookups. We run both CPU and GPU parts, but we only report the time that the application executes on the GPU since we are focusing on the performance of the GPU address translation.

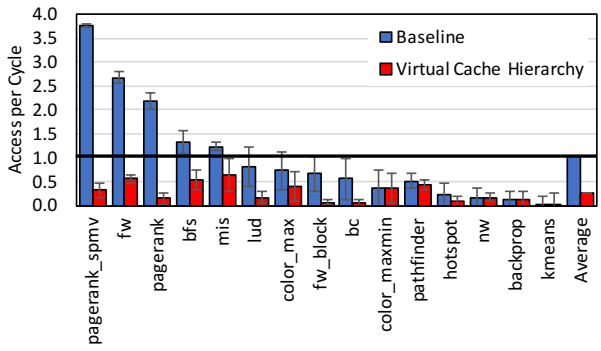

Figure 8. Bandwidth reduction of IOMMU TLB.

5.1 Virtual Cache Hierarchy's Filtering

As discussed in Section 3, a major source of GPU address translation overhead is the significant serialization at the shared IOMMU TLB (Figure 4) due 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 IOMMU TLB accesses. Figure 8 presents an average of shared IOMMU TLB lookups per cycle for the baseline and for our proposal, respectively. Each bar has a one standard deviation band for all sampling periods. The blue bars for the baseline in this figure correspond to blue bars in Figure 3.

For most of workloads, we notice significant reductions compared to the baseline system. We observe less than 0.3 events per cycle on average with our proposal. Some workloads show slightly more than one event per cycle; 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 IOMMU TLB.

Takeaway 1: A GPU virtual cache hierarchy is an efficient TLB miss bandwidth filter.

5.2 Execution Time Benefits

We classify tested workloads into two groups according to the translation bandwidth requirement.

High translation bandwidth workloads

Figure 9 shows relative performance compared to an IDEAL MMU design. We consider two baseline designs by varying the size of a shared IOMMU TLB (i.e., Baseline 512 and 16K). We also consider a virtual cache design (VC W/O OPT, orange bars) with a 512-entry shared IOMMU TLB, and a virtual

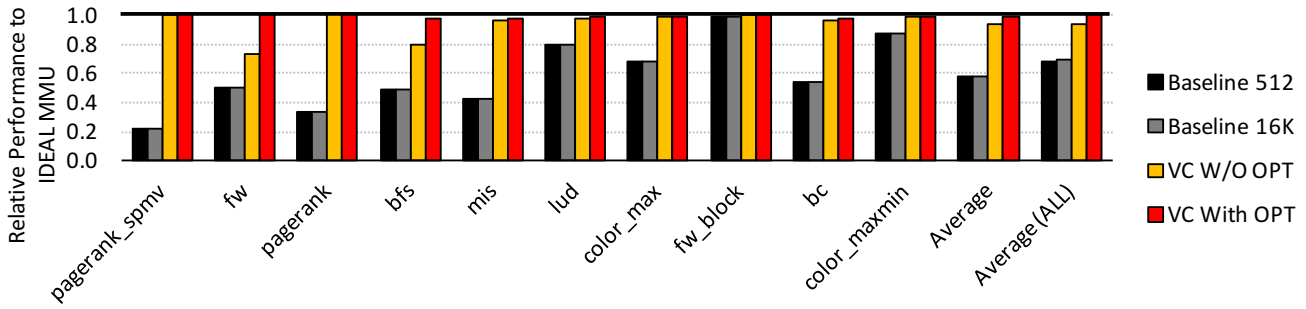

Figure 9. Performance of high translation bandwidth workloads to an IDEAL MMU (closer to 1.0 is better).

Design	Per-CU TLB	IOMMU TLB	B/W Limit
IDEAL MMU	Infinite size	Infinite size	Infinite
Baseline 512	32-entry	512-entry	1 Access/Cycle
Baseline 16K	32-entry	16K-entry	1 Access/Cycle
VC W/O OPT	-	512-entry	1 Access/Cycle
VC With OPT	-	+16K-entry FBT	1 Access/Cycle

Table 2. Evaluated MMU design configurations.

cache design (VC With OPT, red bars) that employs the FBT as a second-level TLB. Except for the ideal MMU design, the shared IOMMU TLB bandwidth is limited to one access per cycle. Table 2 summarizes the MMU designs.

The baseline with a small shared IOMMU TLB (black bars) shows an average 42% performance degradation relative to an IDEAL MMU for high translation bandwidth workloads. We model a very large shared IOMMU TLB to analyze the impact of the IOMMU miss overhead.[6] We notice that doing so does not alleviate the overhead (gray bars). This suggests that most performance overheads result from the serialization at the shared IOMMU TLB due to its limited bandwidth.

A virtual cache hierarchy uses *existing* caches to filter translation bandwidth. Our proposal (VC W/O OPT, orange bars) achieves performance of the IDEAL MMU for most of these workloads. With a virtual cache hierarchy, the serialization overhead on the shared IOMMU TLB is removed even with a small shared IOMMU TLB.

Filtering the high bandwidth exposes some page table walk overhead for two workloads (`fw` and `bfs`). Our observations are in line with the results of a recent study [37]. As described in Section 4.1 (see Address Translation), the FBT can be employed as a second-level shared IOMMU TLB (VC With OPT). By consulting the FBT on a small shared IOMMU TLB miss, we can reduce the page table walk overhead and achieve almost the same performance as the IDEAL MMU without any extra area overhead. With similar area overhead to the large IOMMU TLB, we get much better performance because the virtual cache hierarchy filters the high translation bandwidth.

[6]Designers could take this approach because structures in the IOMMU are less latency constrained.

Figure 10 shows relative speedup of our proposal compared to the baseline design with large (128-entry) fully associative per-CU TLBs and a large (16K-entry) shared IOMMU TLB. Using larger per-CU TLBs reduces bandwidth demands for the shared IOMMU TLB, showing comparable performance for some workloads (e.g., `bc`, `fw_block`, and `lud`). However, using the virtual cache hierarchy shows an average of about 1.2× speedup over the large per-CU TLBs; large private TLBs can filter some accesses, but the virtual cache hierarchy filters more for high translation bandwidth workloads. In addition, virtual caches also have the benefits of reducing the power, energy, and thermal issues of consulting per-CU TLBs on every memory access [39, 46].

Low translation bandwidth workloads

Our technique shows less benefits for low translation bandwidth workloads (`kmeans`, `backprop`, `hotspot`, `nw`, and `pathfinder`) due to low shared IOMMU TLB bandwidth demands. However, there is no performance degradation.

The rightmost bars (Average(ALL)) in Figure 9 show the relative performance of all 15 workloads. This shows that there is high performance degradation on average across all workloads (about 32%). However, virtual caches achieve nearly ideal performance.

Takeaway 2: We can achieve considerable performance benefits by filtering accesses to the shared IOMMU TLB through a GPU virtual cache hierarchy.

5.3 Power and Energy Benefits

We can expect power reduction by not consulting TLBs on every cache access. In addition, due to the filtering effect of the virtual cache hierarchy, the IOMMU is less frequently consulted. We can also reduce GPU L2 lookups by using the FBT as a coherence filter [35]. Our design increases performance as well, leading to proportional energy benefits. These potential benefits are not quantified in this paper.

Takeaway 3: We expect considerable energy benefits by reducing the serialization delay at the shared IOMMU TLB.

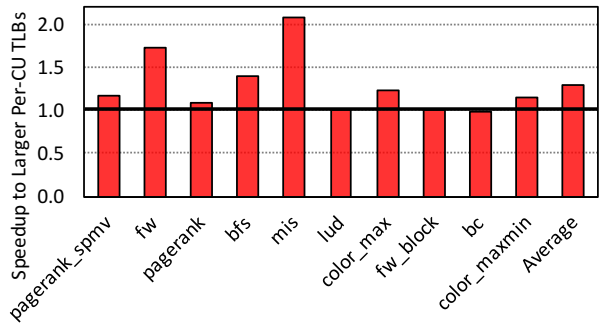

Figure 10. Comparison with larger per-CU TLBs.

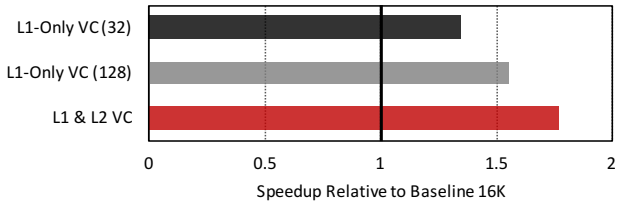

Figure 11. Comparison with L1 virtual cache design.

5.4 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 relative speedup, compared to the baseline physical cache design ("Baseline 16K") for the whole GPU virtual cache hierarchy (L1 and L2 VC) and two virtual L1 cache designs (L1-only VC). The black bar shows the speedup of the virtual L1-only design that has a 32 per-CU TLB. We also consider a larger (128-entry) per-CU TLB (gray bar).

Virtualizing L1 caches alone can also obtain some performance benefits (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). By extending the scope of virtual caches to the whole GPU cache hierarchy, however, we show more latency and energy benefits. Thirty-five percent more per-CU TLB misses are filtered out through the L2 virtual cache (red bars in Figure 2). This leads to fewer TLB and FBT accesses compared to L1-only designs, which reduces power consumption. In addition, we can reduce the design complexity of the GPU cache hierarchy by completely removing per-CU TLB structures from GPUs.

Takeaway 4: A whole GPU virtual cache hierarchy is compelling relative to L1-only virtual caches in terms of performance, power, energy, and design complexity.

6 Related Work

GPU Address Translation: There are some studies that investigate address translation for the GPU MMU. Power *et al.* [37] discussed diverse design aspects for the integration of a CPU-style MMU into GPUs. Bharath *et al.* [33] also studied how to lower the overheads 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 [20, 47]. Kumar *et al.* proposed Fusion [27], 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. Koukos *et al.* consider an L1-Only VC design for the GPU [26], but do not specifically analyze the performance benefits of virtual caching, focusing instead on reducing coherence traffic overheads like the prior work [21].

CPU Virtual Caching: Some proposals take advantage of reverse mapping (physical to virtual address) table to deal with synonym accesses in virtual caches. Goodman [16] proposed dual-tags (virtual and real tags) to efficiently support reverse translations for cache coherence. Instead of using a separate reverse translation table, Wang *et al.* [48] keep track of back-pointers with each cache line of a physical L2 cache to identify the corresponding data cached with synonyms. Kaxiras and Ros [21] introduced simple virtual cache coherence. It exploits self-invalidation and downgrade to eliminate the need of reverse translations.

Other techniques require OS involvement to manage virtual address synonyms. Some proposals advocate for eliminating or limiting sharing [9, 10, 17, 25, 51] to prevent synonyms from occurring. Enigma [53] uses virtual to intermediate address (IA) translation to avoid synonym problems, 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. In a similar vein, Park *et al.* proposed a hybrid virtual cache [31]. This also selectively switches between virtual and physical caching like the OVC, while virtual caching is used for the entire cache hierarchy. Qui and Dubois proposed a synonym lookaside buffer (SLB) [40, 41], 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 CPU virtual cache designs 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.

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, achieving almost the same performance as an ideal MMU.

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., less likelihood of synonyms and offloading OS function to CPUs), making them amenable to virtual caching. We believe these GPUs, and potentially other co-processing units, finally provide an environment where virtual caches are both practical and beneficial.

Acknowledgements

This work was supported in part by funding from the William F. Vilas Trust Estate (Vilas Research Professor), the University of Wisconsin Foundation (John P. Morgridge Professor), the Wisconsin Alumni Research Foundation, and the National Science Foundation (grants CCF-1617824 and CCF-1533885). The authors thank the anonymous reviewers for their insightful feedback, and Gagan Gupta, Mark D. Hill, David A. Wood, Mikko H. Lipasti, Karu Sankaralingam, Marc Orr, Arkaprava Basu, Abhishek Bhattacharjee, and Jan Vesely for their helpful comments and discussions.

Appendix

Figure 12 shows that blocks in the cache hierarchy are likely to reside longer than the lifetime of the corresponding per-CU TLB entries. This figure shows the relative lifetime of pages in each level of the cache hierarchy and the TLB. The `bfs` workload is considered for this analysis, but 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); 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. Thus, accesses to such data hit in the cache hierarchy but are highly likely to miss 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 our observation 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`, and `lud`. Thus, extending the scope of virtual caches to the L2 shared cache helps to filter out more TLB misses.

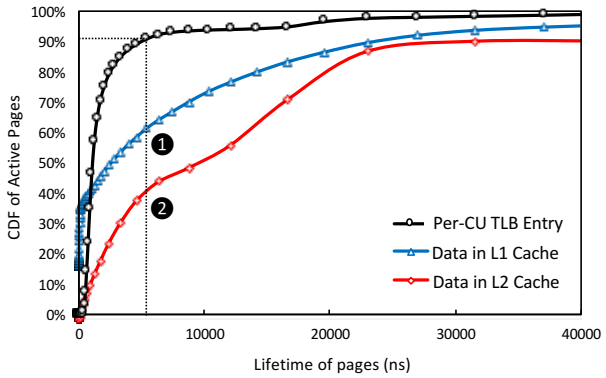

Figure 12. Relative lifetime of active pages in each level of the cache hierarchy and the per-CU TLB: *bfs* workload.

References

[1] [n. d.]. AMD and HSA. ([n. d.]). Retrieved Accessed: 2017-12-09 from http://www.amd.com/en-us/innovations/software-technologies/hsa

[2] [n. d.]. The ARM CoreLink CCI-550 Cache Coherent Interconnect. ([n. d.]). Retrieved Accessed: 2017-12-09 from https://developer.arm.com/products/system-ip/corelink-interconnect/corelink-cache-coherent-interconnect-family/corelink-cci-550

[3] Todd M. Austin and Gurindar S. Sohi. 1996. High-bandwidth Address Translation for Multiple-issue Processors. In *Proceedings of the 23rd Annual International Symposium on Computer Architecture (ISCA '96)*. ACM, New York, NY, USA, 158–167. https://doi.org/10.1145/232973.232990

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

[5] Arkaprava Basu, Mark D. Hill, and Michael M. Swift. 2012. Reducing Memory Reference Energy with Opportunistic Virtual Caching. In *Proceedings of the 39th Annual International Symposium on Computer Architecture (ISCA '12)*. IEEE Computer Society, Washington, DC, USA, 297–308. http://dl.acm.org/citation.cfm?id=2337159.2337194

[6] Benjie Batanes. 2016. PS4 Pro Specs: How Does It Fare Against Xbox Project Scorpio? Which One Is Better? (November 2016). Retrieved Accessed: 2017-12-09 from http://www.itechpost.com/articles/50922/20161107/ps4-pro-specs-fare-against-xbox-project-scorpio-one-better.htm

[7] A. Bhattacharjee. 2017. Preserving Virtual Memory by Mitigating the Address Translation Wall. *IEEE Micro* 37, 5 (September 2017), 6–10. https://doi.org/10.1109/MM.2017.3711640

[8] Abhishek Bhattacharjee, Daniel Lustig, and Margaret Martonosi. 2011. Shared Last-level TLBs for Chip Multiprocessors. In *Proceedings of the 2011 IEEE 17th International Symposium on High Performance Computer Architecture (HPCA '11)*. IEEE Computer Society, Washington, DC, USA, 62–63. http://dl.acm.org/citation.cfm?id=2014698.2014896

[9] Jeffrey S. Chase, Henry M. Levy, Michael J. Feeley, and Edward D. Lazowska. 1994. Sharing and Protection in a Single-address-space Operating System. *ACM Trans. Comput. Syst.* 12, 4 (Nov. 1994), 271–307. https://doi.org/10.1145/195792.195795

[10] Jeffrey S. Chase, Henry M. Levy, Edward D. Lazowska, and Miche Baker-Harvey. 1992. Lightweight Shared Objects in a 64-bit Operating System. In *Conference Proceedings on Object-oriented Programming Systems, Languages, and Applications (OOPSLA '92)*. ACM, New York, NY, USA, 397–413. https://doi.org/10.1145/141936.141969

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

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

[13] Xuhao Chen, Li-Wen Chang, Christopher I. Rodrigues, Jie Lv, Zhiying Wang, and Wen-Mei Hwu. 2014. Adaptive Cache Management for Energy-Efficient GPU Computing. In *Proceedings of the 47th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-47)*. IEEE Computer Society, Washington, DC, USA, 343–355. https://doi.org/10.1109/MICRO.2014.11

[14] Ian Cutress. 2017. Hot Chips: Microsoft Xbox One X Scorpio Engine Live Blog. (August 2017). Retrieved Accessed: 2017-12-13 from https://www.anandtech.com/show/11740/hot-chips-microsoft-xbox-one-x-scorpio-engine-live-blog-930am-pt-430pm-utc

[15] Koen De Bosschere, Albert Cohen, Jonas Maebe, and Harm Munk. 2015. HiPEAC Vision. (2015).

[16] James R. Goodman. 1987. Coherency for Multiprocessor Virtual Address Caches. *SIGPLAN Not.* 22, 10 (Oct. 1987), 72–81. https://doi.org/10.1145/36205.36186

[17] 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. 1986. Design Decisions in SPUR. *Computer* 19, 11 (Nov. 1986), 8–22. https://doi.org/10.1109/MC.1986.1663096

[18] Derek R. Hower, Blake A. Hechtman, Bradford M. Beckmann, Benedict R. Gaster, Mark D. Hill, Steven K. Reinhardt, and David A. Wood. 2014. Heterogeneous-race-free Memory Models. In *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '14)*. ACM, New York, NY, USA, 427–440. https://doi.org/10.1145/2541940.2541981

[19] Bruce Jacob. 2009. *The Memory System: You Can't Avoid It, You Can't Ignore It, You Can't Fake It.* Morgan and Claypool Publishers.

[20] Tomas Karnagel, Tal Ben-Nun, Matthias Werner, Dirk Habich, and Wolfgang Lehner. 2017. Big Data Causing Big (TLB) Problems: Taming Random Memory Accesses on the GPU. In *Proceedings of the 13th International Workshop on Data Management on New Hardware (DAMON '17)*. ACM, New York, NY, USA, Article 6, 10 pages. https://doi.org/10.1145/3076113.3076115

[21] Stefanos Kaxiras and Alberto Ros. 2013. A New Perspective for Efficient Virtual-cache Coherence. In *Proceedings of the 40th Annual International Symposium on Computer Architecture (ISCA '13)*. ACM, New York, NY, USA, 535–546. https://doi.org/10.1145/2485922.2485968

[22] Andy Kegel, Paul Blinzer, Arka Basu, and Maggie Chan. 2016. Virtualizing IO through IO Memory Management Unit. (2016). Retrieved Accessed: 2017-12-09 from http://pages.cs.wisc.edu/~basu/isca$_{i}$ommu$_{t}$utorial/IOMMU$_{T}$UTORIAL$_{A}$SPLOS$_{2}$016.pdf

[23] Hyesoon Kim. 2012. 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)*. ACM, New York, NY, USA, 70–71. https://doi.org/10.1145/2247684.2247698

[24] Sangman Kim, Seonggu Huh, Yige Hu, Xinya Zhang, Emmett Witchel, Amir Wated, and Mark Silberstein. 2014. GPUnet: Networking Abstractions for GPU Programs. In *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation (OSDI'14)*. USENIX Association, Berkeley, CA, USA, 201–216. http://dl.acm.org/citation.cfm?id=2685048.2685065

[25] Eric J. Koldinger, Jeffrey S. Chase, and Susan J. Eggers. 1992. 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)*. ACM, New York, NY, USA, 175–186. https://doi.org/10.1145/143365.143508

[26] Konstantinos Koukos, Alberto Ros, Erik Hagersten, and Stefanos Kaxiras. 2016. Building Heterogeneous Unified Virtual Memories (UVMs) Without the Overhead. *ACM Trans. Archit. Code Optim.* 13, 1, Article 1 (March 2016), 22 pages. https://doi.org/10.1145/2889488

[27] Snehasish Kumar, Arrvindh Shriraman, and Naveen Vedula. 2015. Fusion: Design Tradeoffs in Coherent Cache Hierarchies for Accelerators. In *Proceedings of the 42nd Annual International Symposium on Computer Architecture (ISCA '15)*. ACM, New York, NY, USA, 733–745. https://doi.org/10.1145/2749469.2750421

[28] George Kyriazis. 2012. Heterogeneous system architecture: A technical review. *AMD Fusion Developer Summit* (2012).

[29] Jaikrishnan Menon, Marc De Kruijf, and Karthikeyan Sankaralingam. 2012. iGPU: Exception Support and Speculative Execution on GPUs. In *Proceedings of the 39th Annual International Symposium on Computer Architecture (ISCA '12)*. IEEE Computer Society, Washington, DC, USA, 72–83. http://dl.acm.org/citation.cfm?id=2337159.2337168

[30] Juan Navarro, Sitararn Iyer, Peter Druschel, and Alan Cox. 2002. Practical, Transparent Operating System Support for Superpages. *SIGOPS Oper. Syst. Rev.* 36, SI (Dec. 2002), 89–104. https://doi.org/10.1145/844128.844138

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

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

[33] Bharath Pichai, Lisa Hsu, and Abhishek Bhattacharjee. 2014. Architectural Support for Address Translation on GPUs: Designing Memory Management Units for CPU/GPUs with Unified Address Spaces. In *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '14)*. ACM, New York, NY, USA, 743–758. https://doi.org/10.1145/2541940.2541942

[34] Jason Power. 2017. Inferring Kaveri's Shared Virtual Memory Implementation. (July 2017). Retrieved Accessed: 2017-12-09 from http://www.lowepower.com/jason/inferring-kaveris-shared-virtual-memory-implementation.html

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

[36] Jason Power, Joel Hestness, Marc Orr, Mark Hill, and David Wood. 2014. gem5-gpu: A Heterogeneous CPU-GPU Simulator. *Computer Architecture Letters* 13, 1 (Jan 2014). https://doi.org/10.1109/LCA.2014.2299539

[37] Jason Power, Mark D. Hill, and David A. Wood. 2014. Supporting x86-64 address translation for 100s of GPU lanes. In *Proceedings of the*

2014 IEEE 19th International Symposium on High Performance Computer Architecture (HPCA '14). IEEE, 568–578.

[38] Jason Power, Yinan Li, Mark D. Hill, Jignesh M. Patel, and David A. Wood. 2015. Toward GPUs Being Mainstream in Analytic Processing: An Initial Argument Using Simple Scan-aggregate Queries. In *Proceedings of the 11th International Workshop on Data Management on New Hardware (DaMoN'15)*. ACM, New York, NY, USA, Article 11, 8 pages. https://doi.org/10.1145/2771937.2771941

[39] Kiran Puttaswamy and Gabriel H. Loh. 2006. Thermal Analysis of a 3D Die-stacked High-performance Microprocessor. In *Proceedings of the 16th ACM Great Lakes Symposium on VLSI (GLSVLSI '06)*. ACM, New York, NY, USA, 19–24. https://doi.org/10.1145/1127908.1127915

[40] Xiaogang Qiu and Michel Dubois. 2001. Towards virtually-addressed memory hierarchies. In *Proceedings of the 2001 IEEE 7th International Symposium on High Performance Computer Architecture (HPCA '01)*. 51–62. https://doi.org/10.1109/HPCA.2001.903251

[41] Xiaogang Qiu and Michel Dubois. 2008. The Synonym Lookaside Buffer: A Solution to the Synonym Problem in Virtual Caches. *IEEE Trans. Comput.* 57, 12 (Dec. 2008), 1585–1599. https://doi.org/10.1109/TC.2008.108

[42] Jude A. Rivers, Gary S. Tyson, Edward S. Davidson, and Todd M. Austin. 1997. On High-bandwidth Data Cache Design for Multi-issue Processors. In *Proceedings of the 30th Annual ACM/IEEE International Symposium on Microarchitecture (MICRO-30)*. IEEE Computer Society, Washington, DC, USA, 46–56. http://dl.acm.org/citation.cfm?id=266800.266805

[43] Mark Silberstein, Bryan Ford, Idit Keidar, and Emmett Witchel. 2013. GPUfs: Integrating a File System with GPUs. In *Proceedings of the 18th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '13)*. ACM, New York, NY, USA, 485–498. https://doi.org/10.1145/2451116.2451169

[44] Abhayendra Singh, Shaizeen Aga, and Satish Narayanasamy. 2015. Efficiently Enforcing Strong Memory Ordering in GPUs. In *Proceedings of the 48th International Symposium on Microarchitecture (MICRO-48)*. ACM, New York, NY, USA, 699–712. https://doi.org/10.1145/2830772.2830778

[45] Inderpreet Singh, Arrvindh Shriraman, Wilson W. L. Fung, Mike O'Connor, and Tor M. Aamodt. 2013. Cache Coherence for GPU Architectures. In *Proceedings of the 2013 IEEE 19th International Symposium on High Performance Computer Architecture (HPCA '13)*. IEEE Computer Society, Washington, DC, USA, 578–590. https://doi.org/10.1109/HPCA.2013.6522351

[46] Avinash Sodani. 2011. Race to Exascale: Opportunities and Challenges *(MICRO 2011 Keynote talk)*.

[47] J. Vesely, A. Basu, M. Oskin, G. H. Loh, and A. Bhattacharjee. 2016. Observations and opportunities in architecting shared virtual memory for heterogeneous systems. In *2016 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*. 161–171. https://doi.org/10.1109/ISPASS.2016.7482091

[48] W. H. Wang, J.-L. Baer, and H. M. Levy. 1989. Organization and Performance of a Two-level Virtual-real Cache Hierarchy. In *Proceedings of the 16th Annual International Symposium on Computer Architecture (ISCA '89)*. ACM, New York, NY, USA, 140–148. https://doi.org/10.1145/74925.74942

[49] Neil H. E. Weste and Kamran Eshraghian. 1985. *Principles of CMOS VLSI Design: A Systems Perspective.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA.

[50] H. Wong, M. M. Papadopoulou, M. Sadooghi-Alvandi, and A. Moshovos. 2010. Demystifying GPU microarchitecture through microbenchmarking. In *2010 IEEE International Symposium on Performance Analysis of Systems Software (ISPASS)*. 235–246. https://doi.org/10.1109/ISPASS.2010.5452013

[51] D. A. Wood, S. J. Eggers, G. Gibson, M. D. Hill, and J. M. Pendleton. 1986. An In-cache Address Translation Mechanism. In *Proceedings of the 13th Annual International Symposium on Computer Architecture (ISCA '86)*. IEEE Computer Society Press, Los Alamitos, CA, USA, 358–365. http://dl.acm.org/citation.cfm?id=17407.17398

[52] H. Yoon and G. S. Sohi. 2016. Revisiting virtual L1 caches: A practical design using dynamic synonym remapping. In *Proceedings of the 2016 IEEE 21st International Symposium on High Performance Computer Architecture (HPCA '16)*. 212–224. https://doi.org/10.1109/HPCA.2016.7446066

[53] Lixin Zhang, Evan Speight, Ram Rajamony, and Jiang Lin. 2010. 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)*. ACM, New York, NY, USA, 159–168. https://doi.org/10.1145/1810085.1810109