Supporting x86-64 Address Translation for 100s of GPU Lanes

Jason Power Mark D. Hill David A. Wood

Department of Computer Sciences
University of Wisconsin–Madison
{powerjg,markhill,david}@cs.wisc.edu

Abstract

*Efficient memory sharing between CPU and GPU threads can greatly expand the effective set of GPGPU workloads. For increased programmability, this memory should be uniformly virtualized, necessitating compatible address translation support for GPU memory references. However, even a modest GPU might need 100s of translations per cycle (6 CUs * 64 lanes/CU) with memory access patterns designed for throughput more than locality.*

To drive GPU MMU design, we examine GPU memory reference behavior with the Rodinia benchmarks and a database sort to find: (1) the coalescer and scratchpad memory are effective TLB bandwidth filters (reducing the translation rate by 6.8x on average), (2) TLB misses occur in bursts (60 concurrently on average), and (3) post-coalescer TLBs have high miss rates (29% average).

We show how a judicious combination of extant CPU MMU ideas satisfies GPU MMU demands for 4 KB pages with minimal overheads (an average of less than 2% over ideal address translation). This proof-of-concept design uses per-compute unit TLBs, a shared highly-threaded page table walker, and a shared page walk cache.

1. Introduction

Graphics processing units (GPUs) have transformed from fixed function hardware to a far more general compute platform. Application developers exploit this programmability to expand GPU workloads from graphics-only to more general purpose (GP) applications. Compared with multi-core CPUs, GPGPU computing offers the potential for both better performance and lower energy [23]. However, there is still room for improvement; complex programming models and data movement overheads impede further expansion of the workloads that benefit from GPGPU computing [29].

Currently, processor manufacturers including AMD, Intel, and NVIDIA integrate CPUs and GPUs on the same chip. Additionally, a coalition of companies including AMD, ARM, Qualcomm, and Samsung recently formed the Heterogeneous System Architecture (HSA) Foundation to support heterogeneous computation [34].

Although physical integration is becoming widespread, the GPGPU compute platform is still widely separated from conventional CPUs in terms of its programming model. CPUs have long used virtual memory to simplify data sharing between threads, but GPUs still lag behind.

A shared virtual address space allows "pointer-is-a-pointer" semantics [30] which enable any pointer to be dereferenced on the CPU and the GPU (i.e., each data element only has a single name). This model simplifies sharing data between the CPU and GPU by removing the need for explicit copies, as well as allowing the CPU and GPU to share access to rich pointer-based data structures.

Unfortunately, there is no free lunch. Translating from virtual to physical addresses comes with overheads. Translation look-aside buffers (TLBs) consume a significant amount of power due to their high associativity [15, 16, 33], and TLB misses can significantly decrease performance [6, 17, 20]. Additionally, correctly designing the memory management unit (MMU) is tricky due to rare events such as page faults and TLB shootdown [10].

Current GPUs have limited support for virtualized addresses [11, 12, 36]. However, this support is poorly documented publicly and has not been thoroughly evaluated in the literature. Additionally, industry has implemented limited forms of shared virtual address space. NVIDIA proposed Unified Virtual Addressing (UVA) and OpenCL has similar mechanisms. However, UVA requires special allocation and pinned memory pages [25]. The HSA foundation announced heterogeneous Uniform Memory Accesses (hUMA) which will implement a shared virtual address space in future heterogeneous processors [28], but details of this support are neither published nor evaluated in public literature.

Engineering a GPU MMU appears challenging, as GPU architectures deviate significantly from traditional multi-core CPUs. Current integrated GPUs have hundreds of individual execution lanes, and this number is growing. For instance the AMD A10 APU, with 400 lanes, can require up to 400 unique translations in a single cycle! In addition, the GPU is highly multithreaded which leads to many memory requests in flight at the same time.

To drive GPU MMU design, we present an analysis of the memory access behavior of current GPGPU applications. Our workloads are taken from the Rodinia benchmark suite [9] and a database sort workload. We present three key findings and a potential MMU design motivated by each finding:

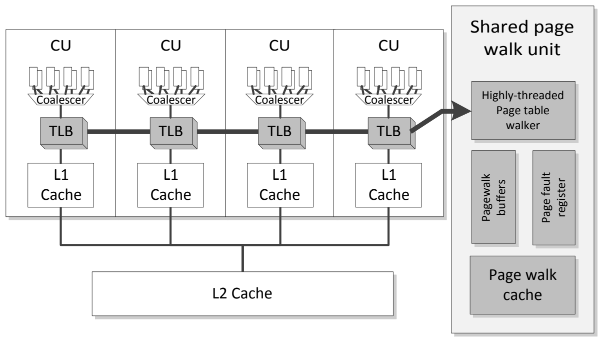

Figure 1: Proposed proof-of-concept GPU MMU.

1. The coalescing hardware and scratchpad memory effectively filter the TLB request rate. Therefore, the L1 TLB should be placed after the coalescing hardware to leverage the traffic reduction.
2. Concurrent TLB misses are common on GPUs with an average of 60 to a maximum of over 1000 concurrent page walks! This fact motivates a highly-threaded page table walker to deliver the required throughput.
3. GPU TLBs have a very high miss rate with an average of 29%. Thus, reducing TLB miss penalty is crucial to reducing the pressure on the page table walker, and thus, we employ a page walk cache.

Through this data-driven approach we develop a proof-of-concept GPU MMU design that is fully compatible with CPU page tables (x86-64 in this work). Figure 1 shows an overview of the GPU MMU evaluated in this paper. This design uses a TLB per GPU compute unit (CU) and a shared page walk unit to avoid excessive per-CU hardware. The shared page walk unit contains a highly-threaded page table walker and a page walk cache.

The simplicity of this MMU design shows that address translation can be implemented on the GPU without exotic hardware. We find that using this GPU MMU design incurs modest performance degradation (an average of less than 2% compared to an ideal MMU with an infinite sized TLB and minimal latency page walks) while simplifying the burden on the programmer.

In addition to our proof-of-concept design, we present a set of alternative designs that we also considered, but did not choose due to poor performance or increased complexity. These designs include adding a shared L2 TLB, including a TLB prefetcher, and alternative page walk cache designs. We also analyzed the impact of large pages on the GPU TLB. We find that large pages do in fact decrease the TLB miss rate. However, in order to provide compatibility with CPU page tables, and ease the burden of the programmer, we cannot rely solely on large pages for GPU MMU performance.

The contributions of this work are:

- An analysis of the GPU MMU usage characteristics for GPU applications,
- A proof-of-concept GPU MMU design which is compatible with x86-64 page tables, and

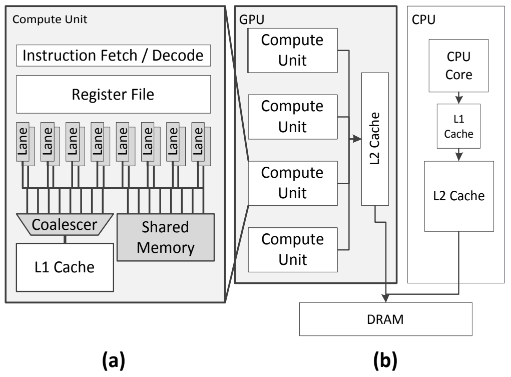

Figure 2: Overview of the heterogeneous architecture used.

- An evaluation of our GPU MMU design that shows a GPU MMU can be implemented without significant performance degradation.

This paper is organized as follows. First, Section 2 discusses background on GPU architecture, GPU virtual memory support, and CPU MMU design. Section 3 explains our simulation infrastructure and workloads. Then, Section 4 presents our three data-driven GPU MMU designs concluding with our proof-of-concept design. Section 5 discusses correctness issues. Next, Section 6 shows other possible designs we considered, and finally, Section 7 discusses related work and Section 8 concludes.

2. Background

This section first introduces the GPU architecture and current GPU virtual memory support. Then, we discuss the memory model of current GPUs and our target system. Finally, we cover background on CPU MMUs.

2.1. GPU Architecture

The important details of the GPGPU architecture are shown in Figure 2. Figure 2b shows an overview of the heterogeneous architecture used in this paper. In this architecture, the CPU and GPU share main memory. The CPU and GPU share a virtual address space and cache coherence is maintained between the CPU and GPU caches. Only a single CPU core is shown in Figure 2b; however, this architecture supports any number of CPU cores. This design is loosely based on future features announced for HSA [29].

Figure 2a shows an overview of the GPU Compute Unit (*CU*)—called a streaming multiprocessor (SM) in NVIDIA terminology. Within each CU is a set of *lanes*—called shader processors (SPs) or CUDA cores by NVIDIA and stream processors by AMD—which are functional units that can execute one lane instruction per cycle. Instructions are fetched, decoded and scheduled by the instruction fetch unit which is shared by all lanes of the CU. The lanes on each CU also share a large, banked, register file. Each lane is associated with a scalar thread, and a set of concurrently executing threads on the CU lanes is a warp. We model a 32-thread warp similar to NVIDIA GPU architecture.

Due to the parallelism available in GPUs, it is difficult to describe the performance in CPU terms. In this paper, we use ***instruction*** to refer to the static instruction that all threads execute. We use the term ***warp instruction*** to represent a dynamic instruction which is executed by all lanes on a single cycle per the Single-Instruction Multiple-Thread (SIMT) execution model. And we use ***lane instruction*** to mean a dynamic instruction executed by a single lane on a CU.

GPUs provide developers with different kinds of memory accesses. Most memory operations can be separated into two categories: Scratchpad memory—small, directly addressed software-managed caches private to each CU (called "shared memory" and "local memory" by NVIDIA and AMD, respectively)—and global memory that is shared by all CUs and addressed with *CPU virtual addresses* in our shared virtual memory system. Accesses to scratchpad memory cannot cause cache misses and use direct addresses that are not translated. All accesses to global memory are issued to the coalesce unit. This logic examines the addresses from all lanes and attempts to minimize the memory requests required. If all addresses issued by the lanes are contiguous and within a single cache block—the common case for graphics workloads—the *coalescer* takes the lane requests (which number up to the size of the warp) and generates a single memory access. After coalescing, the memory request is issued to a write-through L1 cache. Figure 2 shows that each CU's L1 cache is backed by a write-back L2 cache shared by all CUs on the GPU.

2.2. GPU TLB and Virtual Memory Support

Although current GPUs have virtual memory support, it is incompatible with the CPU virtual memory. Current GPU virtual memory, including IOMMU implementations [38], use a separate page table from the CPU process which is initialized by the GPU driver when the kernel is launched. Additionally, GPU virtual memory does not support demand paging or on the fly page table modifications by the operating system. This lack of compatibility increases the difficulty of writing truly heterogeneous applications.

Designs for TLBs specific to GPUs have been published in the form of patents [11, 36]. However, there has been no evaluation of these techniques in the public literature. Additionally, Wong et al. found evidence of TLBs implemented in NVIDIA GPUs [37]. However, the specific TLB design is not publicly documented.

2.3. GPU Programming Model

Current GPGPU programming models consider the GPU as a separate entity with its own memory, virtual address space, scheduler, etc. Programming for the GPU currently requires careful management of data between CPU and GPU memory spaces.

Figure 3 shows an example CUDA application. Figure 3a shows a simple kernel that copies from one vector (`in`) to another (`out`). Figure 3b shows the code required to use the `vectorCopy` kernel using the current separate address space paradigm. In addition to allocating the required memory on the host CPU and initializing the data, memory also is explicitly allocated on, and copied to, the GPU before running the kernel. After the kernel completes, the CPU copies the data back so the application can use the result of the GPU computation.

```
__device__ void vectorCopy(int *in, int *out) {
      out[threadId.idx] = in[threadId.idx];
}
```

(a) Simple vector copy kernel

```
void main() {
      int *d_in, *d_out;
      int *h_in, *h_out;

      // allocate input array on host
      h_in = new int[1024];
      h_in = ... // Initial host array
      // allocate output array on host
      h_out = new int[1024];

      // allocate input array on device
      d_in = cudaMalloc(sizeof(int)*1024);
      // allocate output array on device
      d_out = cudaMalloc(sizeof(int)*1024);

      // copy input array from host to device
      cudaMemcpy(d_in, h_in, sizeof(int)*1024, HtD);

      vectorCopy<<<1,1024>>>(d_in, d_out);

      // copy the output array from device to host
      cudaMemcpy(h_out, d_out, sizeof(int)*1024, DtH);

      // continue host computation with result
      ... h_out

      //Free memory
      cudaFree(d_in); cudaFree(d_out);
      delete[] h_in; delete[] h_out;
}
```

(b) Separate memory space implementation

```
int main() {
      int *h_in, h_out;

      // allocate input/output array on host
      h_in = cudaHostMalloc(sizeof(int)*1024);
      h_in = ... // Initial host array
      h_out = cudaHostMalloc(sizeof(int)*1024);

      vectorCopy <<<1,1024>>> (h_in, h_out);

      // continue host computation with result
      ... h_out

      //Free memory
      cudaHostFree(h_in); cudaFree(h_out);
}
```

(c) "Unified virtual address" implementation

```
int main() {
      int *h_in, h_out;

      // allocate input/output array on host
      h_in = new int[1024];
      h_in = ... // Initial host array
      h_out = new int[1024];

      vectorCopy <<<1,1024>>> (h_in, h_out);

      // continue host computation with result
      ... h_out

      delete[] h_in; delete[] h_out;
}
```

(d) Shared virtual address space implementation

Figure 3: Example GPGPU application

There are many drawbacks to this programming model. Although array-based data structures are straightforward to move from the CPU to the GPU memory space, pointer-based data structures, like linked-lists and trees, present complications. Also, separate virtual address spaces cause data to be replicated. Even on shared physical memory devices, like AMD Fusion, explicit memory allocation and data replication is still widespread due to separate virtual address spaces. Additionally, due to replication, only a subset of the total memory in a system is accessible to GPU programs. Finally, explicit separate allocation and data movement makes GPU applications difficult to program and understand as each logical variable has multiple names (`d_in`, `h_in` and `d_out`, `h_out` in the example).

Beginning with the Fermi architecture, NVIDIA introduced "unified virtual addressing" (UVA) [25] (OpenCL has a similar feature as well). Figure 3c shows the implementation of `vectorCopy` with UVA. The `vectorcopy` kernel is unchanged from the separate address space kernel. In the UVA example, instead of allocating two copies of the input and output vectors, only a single allocation is necessary. However, this allocation requires a special API which creates difficulties in using pointer-based data structures. Separate allocation makes composability of GPU kernels in library code difficult as well, because the allocation is a CUDA runtime library call, not a normal C or C++ allocation (e.g. `new`/`malloc`/`mmap`). Memory allocated via `cudaMallocHost` can be implemented in two different ways. Either the memory is pinned in the main memory of the host, which can lead to poor performance [24], or the data is implicitly copied to a separate virtual address space which has the previously discussed drawbacks.

Figure 3d shows the implementation with a shared virtual address space (the programming model used in this paper). In this implementation, the application programmer is free to use standard memory allocation functions. Also, there is no extra memory allocated, reducing the memory pressure. Finally, by leveraging the CPU operating system for memory allocation and management, the programming model allows the GPU to take page faults and access memory mapped files. From publicly available information, HSA hUMA seems to take this approach [28].

2.4. CPU MMU Design

The memory management unit (MMU) on CPUs translates virtual addresses to physical address as well as checks page protection. In this paper we focus on the x86-64 ISA; however, our results generalize to any multi-level hardware-walked page table structure. The CPU MMU for the x86-64 ISA consists of three major components: 1) logic to check protection and segmentation on each access, 2) a translation look-aside buffer to cache virtual to physical translations and protection information to decrease translation latency, and 3) logic to walk the page table in the case of a TLB miss. In this work, we use the term **MMU** to refer to the unit that contains the TLB and other supporting structures. Many modern CPU MMU designs contain other structures to increase TLB hit rates and decrease TLB miss latency.

The x86-64 page table is a 4-level tree structure. By default, the TLB holds page table entries, which reside in the leaves of the tree. Therefore, on a TLB miss, up to four memory accesses are required. The page table walker (PTW) traverses the tree from the root which is found in the CR3 register. The PTW issues memory requests to the page walk cache that caches data from the page table. Requests that hit in the page walk cache decrease the TLB miss penalty. Memory requests that miss in the page walk cache are issued to the memory system similar to CPU memory requests and can be cached in the data caches.

Table 1: Details of simulation parameters

CPU	1 core, 2 GHz, 64 KB L1, 2 MB L2
GPU	16 CUs, 1.4 GHz, 32 lanes
L1 cache (per-CU)	64 KB, 4-way set associative, 15 ns latency
Scratchpad memory	16 KB, 15 ns latency
GPU L2 cache	1 MB, 16-way set associative, 130 ns latency
DRAM	2GB, DDR3 timing, 8 channels, 667 MHz

The x86-64 ISA has extensions for 2 MB and 1 GB pages in addition to the default 4 KB page size. However, few applications currently take advantage of this huge page support. Additionally, it is important for an MMU to support 4 KB pages for general compatibility with all applications.

3. Simulation Methodology and Workloads

We used a cycle-level heterogeneous simulator, gem5-gpu [27], to simulate the heterogeneous system. gem5-gpu is based on the gem5 simulator [8], and integrates the GPGPU timing model from GPGPU-Sim [1]. We used gem5's *full-system mode*, running the Linux operating system. gem5-gpu models full coherence between the CPU and GPU caches as future systems have been announced supporting cache coherence [28]. Results are presented for 4 KB pages. Large page support is discussed in Section 6.4. Table 1 shows the configuration parameters used in obtaining our results.

We use a subset of the Rodinia benchmark suite [9] for our workloads. We do not use some Rodinia benchmarks as the input sizes are too large to simulate. The Rodinia benchmarks are GPU-only workloads, and we use these workloads as a proxy for the GPU portion of future heterogeneous workloads. We add one workload, sort, to this set. Sort is a database sorting kernel that sorts a set of records with 10 byte keys and 90 byte payloads. All workloads are modified to remove the memory copies, and all allocations are with general allocators (`new`/`malloc`/`mmap`). Although we are running in a simulation environment and using reduced input sized, many of our working sets are much larger than the TLB reach; thus, we expect our general findings to hold as working set size increases.

As a baseline, we use an ideal, impossible to implement MMU. We model an ideal MMU with infinite sized per-CU TLBs and minimal latency (1 cycle cache hits) for page walks. This is the minimum translation overhead in our simulation infrastructure.

Table 2: Configurations under study. Structures are sized so each configuration uses 16 KB of storage.

	Per-CU L1 TLB entries	Highly-threaded page table walker	Page walk cache size	Shared L2 TLB entries
Ideal MMU	Infinite	Infinite	Infinite	None
Section 4				
Design 0	N/A: Per-lane MMUs		None	None
Design 1	128	Per-CU walkers	None	None
Design 2	128	Yes (32-way)	None	None
Design 3	64	Yes (32-way)	8 KB	None
Section 6				
Shared L2	64	Yes (32-way)	None	1024
Shared L2 & PWC	32	Yes (32-way)	8 KB	512
Ideal PWC	64	Yes (32-way)	Infinite	None
Latency	1 cycle	20 cycles	8 cycles	20 cycles

4. Designing a GPU MMU through Analysis

We meet the challenges of designing a GPU MMU by using data to evolve through three architectures to our proof-of-concept recommendation (Design 3). Design 3 enables full compatibility with x86-64 page tables with less than 2% performance overhead, on average. Table 2 details the designs in this section as well as the additional designs from Section 6.

We start with a CPU-like MMU (Design 0) and then modify it as the GPU data demands. Design 0 follows CPU core design with a private MMU at each lane (or "core" in NVIDIA terminology). Design 0 has the same problems as a CPU MMU—high power, on the critical path, etc.—but they are multiplied by the 100s of GPU lanes. For these reasons, we do not quantitatively evaluate Design 0.

4.1. Motivating Design 1: Post-coalescer MMU

Here we show that moving the GPU MMU from before to after the coalescer (Design 0 → Design 1) reduces address translation traffic by 85%.

GPU memory referencing behavior differs from that of CPUs. For various benchmarks, Figure 4 presents operations per thousand cycles for scratchpad memory lane instructions (left bar, top, blue), pre-coalescer global memory lane instructions (left bar, bottom, green), and post-coalescer global memory accesses (right bar, brown). The "average" bars represent the statistic if each workload was run sequentially, one after the other.

Figure 4 shows that, for every thousand cycles, the benchmarks average:

- 602 total memory lane instructions,
- 268 of which are global memory lane instructions (with the other 334 to scratchpad memory), and
- Coalescing reduces global memory lane instructions to only 39 global memory accesses.

In total, the rate of memory operations is reduced from 602 to 39 per thousand cycles for an 85% reduction.

Although the coalescing hardware is effective, the benchmarks do show significant memory divergence. Perfect coalescing on 32 lanes per CU would reduce 268 global

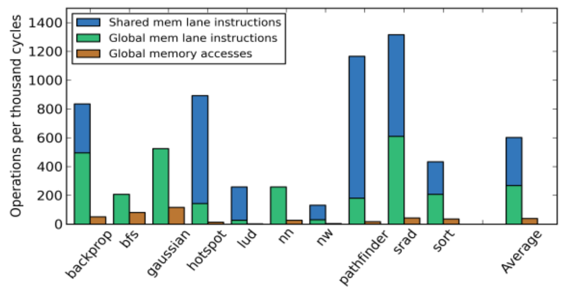

Figure 4: All memory operations global memory operations, and global memory accesses per thousand cycles.

memory lane instructions (per thousand cycles) by 32x to 9, which is much less than the 39 observed.

To benefit from this bandwidth filtering, **Design 1** includes a private per-CU L1 TLB after scratchpad memory access and after the coalescing hardware. Thus, the MMU is only accessed on global memory accesses. Figure 5 shows Design 1 in light gray and Table 2 details the configuration parameters.

4.2. Motivating Design 2: Highly-threaded page table walker

Here we show that Design 1 fails to perform well (average performance is 30% of an ideal MMU), isolate the problem to bursts of TLBs misses (60 concurrent), and advocate for a highly-threaded PTW (Design 2).

Now that we have mitigated the bandwidth issues, we might expect Design 1 to perform well; it does not. For each benchmark and the average, Figure 6 shows the performance of Design 1 (leftmost, blue) compared to an ideal MMU with an impossibly low latency and infinite sized TLBs. Performance is good when it is close to the ideal MMU's 1.0. (Designs 2 (green) and 3 (brown) will be discussed later.)

Figure 6 results show that Design 1 (blue) performs:

- Poorly on average (30% of ideal's),
- Sometimes very poorly (about 10% of ideal for backprop, bfs, and pathfinder), and
- Occasionally adequately (gaussian and lud).

These performance variations occur for various reasons. For example, bfs is memory bound—having few instructions per memory operation—making it particularly sensitive to global memory latency. On the other hand, gaussian and lud perform well, in part because the working set sizes are relatively small.

Investigating Design 1's poor performance lead to an obvious culprit: *bursts of TLB misses*. For each benchmark and the average, Figure 7 shows the average (left, blue) and maximum across CUs (right, green) occupancy of the page walk queue when each page walk begins. Note that the y-axis is logarithmic.

Figure 7 shows that when each page walk is issued:

- An average of 60 page table walks are active at that CU, and

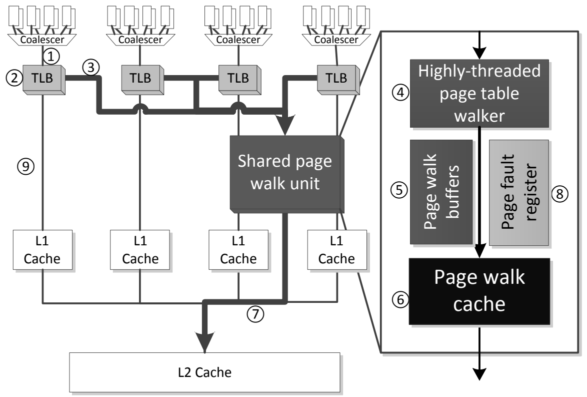

Figure 5: Overview of the GPU MMU designs. *Design 1* is shown in light gray. *Design 2* is shown in light and dark gray. And *Design 3* is shown in light and dark gray and black.

- The worst workload averages 140 concurrent page table walks.

Moreover, additional data show that, for these workloads, over 90% of page walks are issued within 500 cycles of the previous page walk, which is significantly less than the average page walk latency in this design. Also, almost all workloads use many more than 100 concurrent page walks at the maximum. Therefore, these workloads will experience high queuing delays with a conventional blocking page table walker. This also shows that the GPU MMU requires changes from the CPU-like single-threaded page-table walker of Design 1.

This high page walk traffic is primarily because GPU applications can be very bandwidth intensive. GPU hardware is built to run instructions in lock step, and because of this characteristic, many GPU threads simultaneously execute a memory instruction. This, coupled with the fact each CU supports many simultaneous warp instructions, means GPU TLB miss traffic will be high.

Therefore, **Design 2** includes a shared multi-threaded page table walker with 32 threads. Figure 5 shows how Design 2 builds on Design 1 in dark gray and Table 2 details the configuration parameters. The page walk unit is shared between all CUs on the GPU to eliminate duplicate hardware at each CU and reduce the hardware overhead. On a TLB miss in the per-CU L1 TLBs, the shared page walk unit is accessed and executes a page walk.

4.3. Motivating Design 3: Add a page walk cache

Here we show that Design 2 performs much better than Design 1 for most workloads but still falls short of an ideal MMU (30% of ideal on average). For this reason, we introduce Design 3 that adds a shared page walk cache to perform within 2% of ideal.

The second bars in Figure 6 (green) show the performance of each benchmark for Design 2. Results from this figure show that Design 2:

- Often performs much better than Design 1, but
- That these benefits are inconsistent and short of ideal

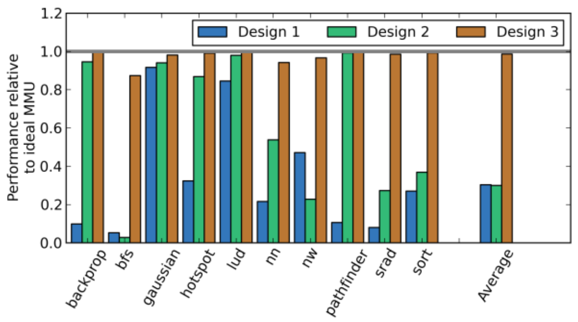

Figure 6: Performance of each design relative to an ideal MMU. See Table 2 for details of configurations.

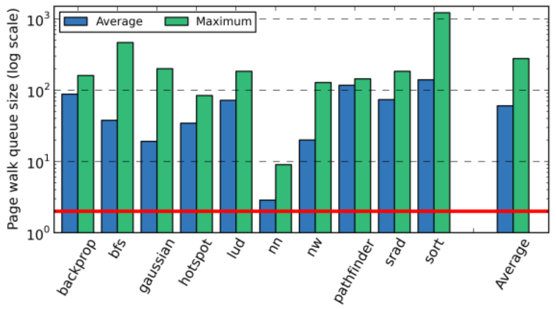

Figure 7: Average and max size of the page walk queue for each per-CU MMU in Design 1. Log scale, bold line at 2.

We find the workloads that perform best have been tuned to tolerate long-latency memory operations and the addition of the TLB miss latency is hidden by thread parallelism.

On the other hand, some workloads, e.g., bfs and nw, actually perform worse with Design 2 than Design 1. We isolated the problem to the many requests from one CU queuing in front of another CU's requests rather than being handled more round-robin as in the single-threaded page-table walker of Design 1. While this specific effect might be fixed by changing PTW queuing from first-come-first-serve, we seek a more broadly effective solution.

To better understand why Design 2 falls short of ideal, we examined the TLB miss rates. Figure 8 shows the miss rates (per-CU, 128 entry) for each benchmark and the average.

Figure 8 results show that per-CU TLB miss rates:

- Average 29% across benchmarks and
- Can be as high as 67% (nw).

Needless to say, these rates are much higher than one expects for CPU TLBs. Gaussian is a low outlier, because of high computational density (compute operations per byte of data) and small working set (so all TLB misses are compulsory misses).

This high miss rate is not surprising. With many simultaneous warps and many threads per warp, a GPU CU can issue memory requests to a very large number of pages. In addition, many GPU applications exhibit a memory access pattern with poor temporal locality reducing the effectiveness of caching translations. If an access pattern has no temporal locality (e.g. streaming), even with perfect

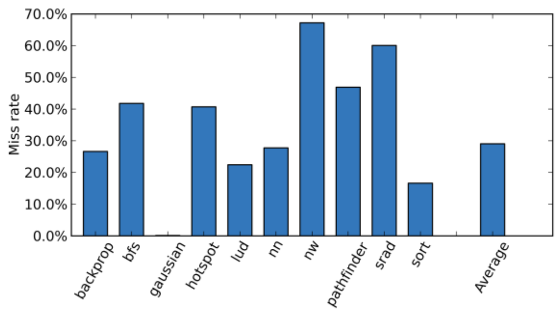

Figure 8: Miss rate for a 128 entry per-CU L1 TLB averaged across all CUs.

coalescing, each CU could potentially access 128 bytes per cycle. This translates to only 32 cycles to access an entire 4 KB page, in the worst case.

As discussed previously, the global memory access rate on GPUs is quite low (39 accesses per thousand cycles on average) and consequently the TLB request rate is small. Therefore, it's possible that even though the GPU TLB exhibits high miss rates the miss traffic (misses per cycle) could be relatively low. However, this is not the case. There is an average of 1.4 TLB misses per thousand cycles and a maximum of 13 misses per thousand cycles.

We investigated several alternatives to improve on Design 2, discussed further in Section 6, and settled on one. Our preferred **Design 3** includes a page walk cache with the shared page walk unit to decrease the TLB miss latency. Figure 5 shows how Design 3 builds on Design 2 in black and Table 2 details the configuration parameters.

Returning to Figure 6, the third bars (brown) show performance of Design 3 relative to an ideal MMU (1.0):

- Design 3 increases performance for all benchmarks over Design 2 and
- Design 3 is within 2% of the ideal MMU on average.

This increase in overall performance occurs because the page walk cache significantly reduces the average page walk time reducing the number of cycles the CU is stalled. Adding a page walk cache reduces the average latency for page table walks by over 95% and correspondingly increases the performance. The performance improvement of Design 3 is in part due to reducing the occupancy of the page walk buffers since the page walk latency decreased. This fact is most pronounced for bfs and nw, which suffered from queuing delays.

4.4. Summary of Proof-of-Concept Design (Design 3)

This section summarizes our proof-of-concept Design 3 and presents additional details to explain its operation. Figure 5 details Design 3. The numbers correspond to the order in which each structure is accessed. Design 3 is made up of three main components, the per-CU post-coalescer L1 TLBs, the highly-threaded page table walker, and a shared page walk cache, discussed below.

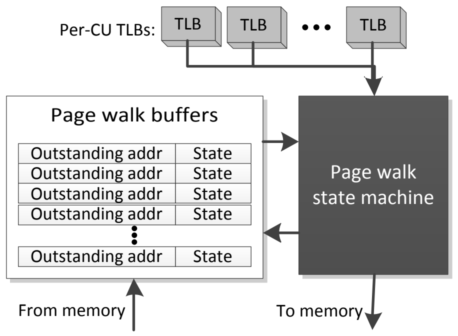

Figure 9: Details of the highly-threaded page table walker

Per-CU post-coalescer L1 TLBs—Each CU has a private TLB that is accessed after coalescing and scratchpad memory to leverage the traffic reduction. On TLB hits, the memory request is translated then forwarded to the L1 cache. On TLB misses, the warp instruction is stalled until the shared page walk unit completes the page table lookup and returns the virtual to physical address translation. Stalling at this point in the pipeline is common for memory divergent workloads, and the associated hardware is already present in the CUs.

Highly-threaded page table walker—The PTW design consists of a hardware state machine to walk the x86-64 page table and a set of page walk buffers that hold the current state of each outstanding page walk, shown in Figure 9. On a TLB miss, the page walk state machine allocates a page walk buffer entry and initializes the outstanding address to the value in the CR3 register, which holds the address of the root of the page table. Next, the page walk state machine issues the memory request corresponding to the buffer entry. When memory responds, the page walk buffers are queried for the match, and the state for that request is sent to page walk state machine, which then issues the next memory request or returns final translation.

The PTW also has supporting registers and logic to handle faults that occur when walking the page table. Concurrent page faults are serialized and handled one at a time by the operating system. Page faults are discussed in detail in Section 5.1

This design can also extend to other ISAs that have hardware page table walkers. For example, the ARM MMU also defines a hardware page table walker and this can be used in place of the x86-64 page walk state machine included in our current design.

Page walk cache—Many modern CPUs contain a small cache within the memory management unit to accelerate page table walks. Since the latency of the L2 cache of a GPU is very long (nearly 300 cycles) a cache close to the MMU decreases the page walk latency significantly. We use a page walk cache design similar to AMD [3]. It caches non-leaf levels of the page table, decreasing the number of L2 cache and DRAM accesses required for a page table walk. Other page walk cache designs are discussed in

Section 6.2. However, we do not see a significant performance improvement over Design 3 with these alternatives.

5. Correctness Issues

In this section, we discuss the issues and implementation of page faults and TLB shootdown for the GPU architecture. We expect these events to be rare. For instance, often workloads are sized to fit in main memory virtually eliminating major page faults. Nevertheless, we correctly implement page faults and TLB shootdown in Linux 2.6.22 on top of gem5 full-system simulation, to our knowledge a first in public literature.

5.1. Page fault handling

Although rare, the GPU memory management unit architecture must be able to handle page faults to have correct execution. Although there are many ways to implement handling page faults, in this GPU MMU architecture we chose to slightly modify the CPU hardware by changing the interrupt return microcode and adding hardware registers to support GPU page faults. With these changes, the GPU MMU can handle page faults with no modifications to the operating system. Our page fault handling logic leverages the operating system running on the CPU core similar to CPU MMU page fault logic. We use this design for two reasons. First, this design does not require any changes to the GPU execution hardware as it does not need to run a full-fledged operating system. Second, this design does not require switching contexts on the GPU as it can handle minor page faults by stalling the faulting instruction. Details of our page fault implementation can be found in the appendix and the gem5-gpu repository [14].

There are two different categories of page faults, major and minor. Below we give details on how each is handled.

Minor page faults—A minor page fault occurs when the operating system has already allocated virtual memory for an address, but it has not yet allocated a physical frame and written the page table. Minor page faults often occur when sharing virtual memory between processes, copy-on-write memory, and on the initial accesses after memory allocation. The last is common in our workloads as there are many large calls to `malloc` in which the memory is not touched by the CPU process. As an example, in Figure 3c, `h_out` is allocated by the CPU process, but is not accessed until the `copyVector` kernel and causes a minor page fault.

Minor page faults are low latency, about 5000 cycles on average in our workloads. The operating system only needs to allocate a physical frame and modify the page table with the new physical page number. Since the page fault is low latency, we stall the faulting warp instruction in the same way as a TLB miss.

Major page faults—For major page faults, the operating system must perform a long-latency action, such as a disk access. Stalling an application for milliseconds—while likely correct—wastes valuable GPU execution resources. To handle this case, the GPU application, or a subset thereof, could be preempted similar to a context switch on a CPU. However, this technique has drawbacks since the GPU's context is very large (e.g., 2 MB of register file on NVIDIA Fermi [25]). Another possibility is to leverage checkpointing, and restart the offending applications after the page fault has been handled. There are proposals like iGPU [21] to reduce the overhead of these events. However, none of our workloads have any major page faults so we do not focus on this case.

Page fault discussion—We implemented a hardware-based technique to handle page faults in the gem5 simulator running Linux in full-system mode. Using this implementation, the Linux kernel correctly handles minor page faults for all of our GPU applications.

For this implementation, the page fault handling logic assumes that the GPU process is still running on the CPU core. This will be true if the application can run on the CPU and GPU at the same time, or the CPU runtime puts the CPU core into a low power state and does not change contexts.

However, the CPU runtime may yield the CPU core while the GPU is running. In this situation, page faults are still handled correctly, but they are longer latency since the CPU core must context switch to the process which spawned the GPU kernel before the operating system begins handling the page fault. An alternative option is to include a separate general purpose core to handle operating system kernel execution for the GPU. This core can then handle any page faults generated by the running kernel.

5.2. TLB Flushes and shootdown

The GPU MMU design handles TLB flushes similarly to the CPU MMU. When the CR3 register is written on the CPU core that launched the GPU application, the GPU MMU is notified via inter-processor communication and all of the GPU TLBs are flushed. This is a rare event, so performance is not a first order concern. Since there is a one-to-one relation between the CPU core executing the CPU process and the GPU kernel, on TLB a shootdown to the CPU core, the GPU TLBs are also flushed. The GPU cannot initiate a shootdown, only participate. When a CPU initiates a shootdown, it sends a message to the GPU MMU which responds after it has been handled by flushing the TLBs. If the GPU runtime allows the CPU processes to be de-scheduled during GPU kernel execution, TLB flushes and shootdown become more complicated. However, this can be handled in a similar way as page faults. If TLB shootdown to GPU CUs becomes common, there are many proposals to reduce the overheads for TLB shootdown [31, 35].

6. Alternative Designs

Here, we discuss some alternative designs we considered as we developed our proof-of-concept design. These design either do not perform as well or are more complex than Design 3. We first evaluate the addition of a shared L2 TLB

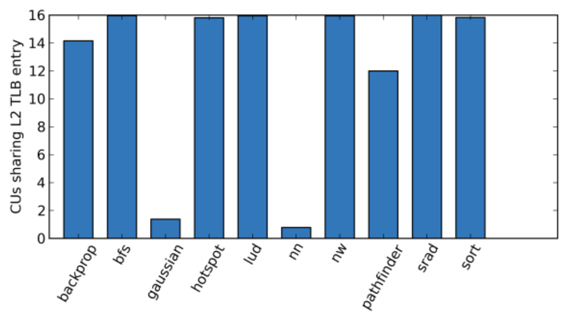

Figure 10: Sharing pattern for the 512 entry L2 TLB. 16 sharers implies all CUs sharing each entry.

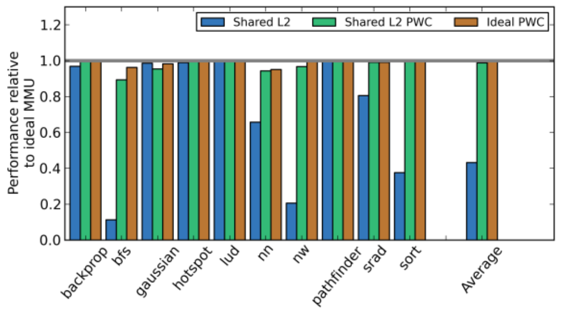

Figure 11: Performance of a shared L2 TLB and an ideal PWC relative to ideal MMU. See Table 2 for details of configurations.

and then alternative page walk cache designs. Next, we discuss adding a TLB prefetcher and the impact of large pages. Finally, we consider the impact of alternative MMU designs on area and energy.

6.1. Shared L2 TLB

A shared L2 TLB can capture multiple kinds of data sharing between execution units [5]. This cache can exploit sharing of translations between separate CUs, effectively prefetching the translation for all but the first CU to access the page. The shared L2 TLB can also exploit striding between CUs where the stride is within the page size (e.g., CU 1 accesses address 0x100, CU 2 0x200, etc.).

The shared L2 TLB is most effective when each entry is referenced by many CUs. Figure 10 shows the number of CUs that access each L2 TLB entry before eviction. 16 sharers show all CUs share each entry, and one sharer shows no overlap in the working sets. Figure 10 shows most applications share each L2 TLB entries with many CUs and some share entries with all CUs. In these cases, the shared L2 TLB can improve performance by sharing the capacity of the L2 TLB between CUs.

Due to this potential, we investigated two MMU designs with shared L2 TLBs. The first design (**Shared L2**) has private L1 TLBs with a shared L2 TLB and no page walk cache. In this design the area that is devoted to the page walk cache in Design 3 is instead used for the L2 TLB. The second design we evaluate (**Shared L2 & PWC**) contains private L1 TLBs, a shared L2 TLB, and a page walk cache. In this design each L1 TLB size is reduced to accommodate an L2 TLB. All of the designs evaluated use a total of 16 KB of storage for their implementation.

Figure 11 shows the performance of these two designs relative to an ideal MMU in the first two bars. Parameters for each configuration are in Table 2.

Shared L2*—per-CU private L1 TLBs and a shared L2 TLB:* With a shared L2 TLB, many applications perform as well as the ideal, like Design 3 (Figure 11 leftmost bars in blue). However, bfs, nw, and sort perform at least 2x worse. For these applications, decreasing the page walk latency is very important as the L1 TLBs experience a high miss rate. nn sees a slowdown when using the Shared L2 design because there is no sharing of TLB entries between CUs. Thus, area dedicated to a page walk cache is more useful for this workload. On average, there is more than a 2x slowdown when using a shared L2 TLB instead of a page walk cache.

Shared L2 & PWC*—per-CU private L1 TLBs, a shared L2 TLB, and a page walk cache:* The second bars (green) in Figure 11 show the performance with both a shared L2 TLB and a page walk cache compared to an ideal MMU. In this configuration, even though the L1 TLB size is reduced, performance does not significantly decrease; the average performance is within 0.1%. Using both a shared L2 TLB and a page walk cache achieves the benefits of both: it takes advantage of sharing between CUs and reduces the average page walk latency. We chose to not include an L2 TLB in our proof-of-concept design as it adds complexity without affecting performance.

6.2. Alternative Page Walk Cache Designs

In this work, we use a page walk cache similar to the structure implemented by AMD [3]. In this design, physical addresses are used to index the cache. Other designs for a page walk cache that index the cache based on virtual addresses, including Intel-style translation caches, have been shown to increase performance for CPUs [2]. We chose to use an AMD-style page walk cache primarily for ease of implementation in our simulator infrastructure.

To evaluate the possible effects of other page walk cache designs, we evaluated our workloads with an ideal (infinitely sized) page walk cache with a reduced latency to model a single access, the best case for the translation cache. The rightmost (brown) bars in Figure 11 show the performance with a 64 entry L1 TLB and the ideal page walk cache compared to an ideal MMU. The ideal page walk cache increases performance by an average of 1% over our proof-of-concept Design 3. For bfs the ideal page walk cache increases performance by a more significant 10% over Design 3 as this workload is sensitive to the page walk latency. From this data, a different page walk cache design may be able to increase performance, but not significantly.

6.3. TLB prefetching

We evaluated Design 3 with the addition of a one-ahead TLB prefetcher [18]. The TLB prefetcher issues a page walk for the next page on each L1 TLB miss and on each prefetch buffer hit. The TLB prefetcher does not affect the perfor-

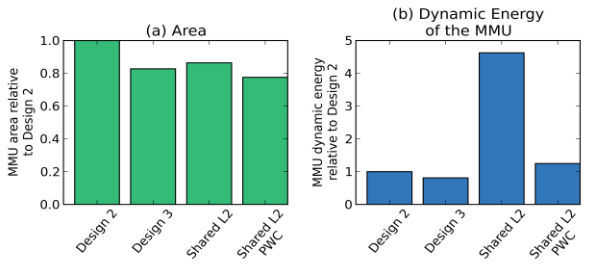

Figure 12: Energy and area of MMU configurations relative to Design 2

mance of our workloads. Prefetching, on average, has a less than 1% impact on performance. One hypothesis as to the ineffectiveness of TLB prefetching is the bursty-ness of demand misses. Other, more complicated, prefetching schemes may show more performance improvement, but are out of the scope of this paper.

6.4. Large pages

Large pages reduce the miss rate for TLBs on CPUs. On GPUs, due to the high spatial locality of accesses, large pages should also work well. When evaluated, for all of our workloads except gaussian, 2MB pages reduce the TLB miss rate by more than 99%, resulting in more than 100 times fewer TLB misses. As previously mentioned, since the working set for Gaussian fits in the TLB, large pages do not provide as much benefit, although the miss rate is reduced by over 80%.

Large pages work well for the workloads under study in this paper, but may not perform as well for future, larger memory footprint, workloads. Additionally, to maintain compatibility with today's CPU applications, the MMU requires 4 KB pages. Alternatively, requiring applications to use large pages would place a burden on the application developer to use a special allocator API for memory that is accessed by the GPU.

6.5. Energy and area

Figure 12 shows the relative area and energy of the MMU for a subset of designs. We used a combination of Cacti [22] and McPAT [19] to determine relative area and dynamic access energy of each structure.

Figure 12a shows that all configurations are less area than the L1 TLB-only Design 2. This is because Design 2 has many large and highly associative TLBs, one per CU. The other configurations that share structures can amortize the overheads and provide higher performance. The shared L2 design is much more energy hungry than the other configurations. This is because the L1 TLBs do not filter a large percentage of requests (the average miss rate is 27%) and accesses to a large associative structure are high energy.

Design 3 with only a page walk cache shows a modest energy reduction (20%) and increases performance significantly over the Design 2. This energy reduction comes from having smaller highly associative structures (L1 TLBs) and a larger lower associativity structure (the page walk cache). The design with both a page walk cache and shared L2 TLB (Shared L2 & PWC) has the smallest area, but has a modest energy increase (25%) over Design 2. Here, there is a tradeoff between energy and area reduction.

7. Related Work

In this work we build on research accelerating CPU MMUs. Page walk caches and translation caches improve MMU performance by accelerating the page table walk [3, 39]. Barr et al. explored other MMU cache structures as well and found that a unified translation cache can outperform both the page walk cache and the translation cache [2].

Sharing resources between CPU cores has been studied as a way to decrease CPU TLB miss rate. Bhattacharjee and Martonosi examine a shared last level TLB [7] and a shared page walk cache [4]. These accelerate multithreaded applications on the CPU by sharing translations between cores. These works target multithreaded applications on the CPU and apply similarly to the GPU since, in the common case, all CUs of the GPU are running the same application.

There are also several patents for industrial solutions for GPU virtual address translation [11, 36]. However, these patents have no evaluation and little implementation details.

There are multiple attempts to reduce the complexity of the GPGPU programming model through software [13, 32]. While these frameworks simplify the code for straightforward applications, like the UVA implementation of the `vectorcopy` example presented, it is still difficult to represent complex data structures.

GPUfs presents a POSIX-like API for the GPU that allows the GPU to access files mapped by the operating system. Similar to GPUfs, this paper simplifies programming the GPU by providing developers with a well-known interface (shared virtual address space). Additionally, as a consequence of correctly handling page faults on the GPU, this paper also provides a method for the GPU to access files. GPUfs accomplishes these goals on current hardware through a complicated software library whereas this paper implements solutions for future hardware similar to how conventional CPUs solve these problems.

Concurrent with our work, Bharath et al. also investigate address translation for GPGPUs [26]. Both papers show that modest hardware changes can enable low-overhead GPU address translation, but with different designs and emphasis. For example, Bharath et al. use a 4-ported TLB and PTW scheduling, while we use a single-ported TLB and highly-threaded PTW. Bharath et al. additionally explore address translation effects on GPU warp scheduling, while we explore MMU correctness issues, like page faults, in a CPU-GPU system using gem5-gpu full-system simulation.

8. Conclusions

As GPGPUs can issue 100s of per-lane instructions per cycle, supporting address translation appears formidable. Our analysis, however, shows that a non-exotic GPU MMU design performs well with commonly-used 4 KB pages: per-

CU post-coalescer TLBs, a shared 32-way highly-threaded page table walker, and a shared page walk cache. We focused on the x86-64 ISA in this work. However, our findings generalize to any ISA with a hardware walked and tree-based page table structure. The proof-of-concept GPU MMU design analyzed in this paper shows that decreasing the complexity of programming the GPU without incurring significant overheads is possible, opening the door to novel heterogeneous workloads.

9. Acknowledgements

This work is supported in part by the National Science Foundation (CCF-1017650, CNS-1117280, CCF-1218323 and CNS-1302260) and a University of Wisconsin Vilas award. The views expressed herein are not necessarily those of the NSF. Professors Hill and Wood have significant financial interests in AMD. The authors would like to acknowledge Brad Beckmann, Karu Sankaralingam, members of the Multifacet research group, and our anonymous reviewers for their comments on the paper.

References

[1] Bakhoda, A. et al. 2009. Analyzing CUDA workloads using a detailed GPU simulator. *Performance Analysis of Systems and Software, 2009. ISPASS 2009. IEEE International Symposium on* (Apr. 2009), 163–174.

[2] Barr, T.W. et al. 2010. Translation caching: Skip, Don't Walk (the Page Table). *Proceedings of the 37th annual international symposium on Computer architecture - ISCA '10* (New York, New York, USA, 2010), 48.

[3] Bhargava, R. et al. 2008. Accelerating two-dimensional page walks for virtualized systems. *Proceedings of the 13th international conference on Architectural support for programming languages and operating systems - ASPLOS XIII* (New York, New York, USA, 2008), 26.

[4] Bhattacharjee, A. 2013. Large-reach memory management unit caches. *Proceedings of the 46th Annual IEEE/ACM International Symposium on Microarchitecture - MICRO-46* (New York, New York, USA, 2013), 383–394.

[5] Bhattacharjee, A. et al. 2011. Shared last-level TLBs for chip multiprocessors. *2011 IEEE 17th International Symposium on High Performance Computer Architecture*. (Feb. 2011), 62–63.

[6] Bhattacharjee, A. and Martonosi, M. 2009. Characterizing the TLB Behavior of Emerging Parallel Workloads on Chip Multiprocessors. *2009 18th International Conference on Parallel Architectures and Compilation Techniques* (Sep. 2009), 29–40.

[7] Bhattacharjee, A. and Martonosi, M. 2010. Inter-core cooperative TLB Prefetchers for chip multiprocessors. *Proceedings of the fifteenth edition of ASPLOS on Architectural support for programming languages and operating systems - ASPLOS '10* (New York, New York, USA, 2010), 359.

[8] Binkert, N. et al. 2011. The gem5 simulator. *ACM SIGARCH Computer Architecture News*. 39, 2 (Aug. 2011), 1.

[9] Che, S. et al. 2009. Rodinia: A benchmark suite for heterogeneous computing. *2009 IEEE International Symposium on Workload Characterization (IISWC)*. 2009, (Oct. 2009), 44–54.

[10] Chip problem limits supply of quad-core Opterons: 2007. *http://techreport.com/news/13721/chip-problem-limits-supply-of-quad-core-opterons*. Accessed: 2013-03-09.

[11] Danilak, R. 2009. System and method for hardware-based GPU paging to system memory. U.S. Patent #7623134. 2009.

[12] envytools: 2013. *https://github.com/pathscale/envytools/blob/master/hwdocs/memory/nv50-vm.txt*.

[13] Gelado, I. et al. 2010. An asymmetric distributed shared memory model for heterogeneous parallel systems. *Proceedings of the fifteenth edition of ASPLOS on Architectural support for programming languages and operating systems - ASPLOS '10* (New York, New York, USA, Mar. 2010), 347.

[14] gem5-gpu: *http://gem5-gpu.cs.wisc.edu*.

[15] Juan, T. et al. 1997. Reducing TLB power requirements. *Proceedings of the 1997 international symposium on Low power electronics and design - ISLPED '97* (New York, New York, USA, 1997), 196–201.

[16] Kadayif, I. et al. 2002. Generating physical addresses directly for saving instruction TLB energy. *35th Annual IEEE/ACM International Symposium on Microarchitecture, 2002. (MICRO-35). Proceedings.* (2002), 185–196.

[17] Kandiraju, G.B. and Sivasubramaniam, A. 2002. Characterizing the d -TLB behavior of SPEC CPU2000 benchmarks. *Proceedings of the 2002 ACM SIGMETRICS international conference on Measurement and modeling of computer systems - SIGMETRICS '02* (New York, New York, USA, 2002), 129.

[18] Kandiraju, G.B. and Sivasubramaniam, A. 2002. Going the distance for TLB prefetching. *ACM SIGARCH Computer Architecture News*. 30, 2 (May. 2002), 195.

[19] Li, S. et al. 2009. McPAT : An Integrated Power , Area , and Timing Modeling Framework for Multicore and Manycore Architectures. *42nd Annual IEEE/ACM International Symposium on Microarchitecture* (2009), 469–480.

[20] McCurdy, C. et al. 2008. Investigating the TLB Behavior of High-end Scientific Applications on Commodity Microprocessors. *ISPASS 2008 - IEEE International Symposium on Performance Analysis of Systems and software* (Apr. 2008), 95–104.

[21] Menon, J. et al. 2012. iGPU. *ACM SIGARCH Computer Architecture News*. 40, 3 (Sep. 2012), 72.

[22] Muralimanohar, N. et al. 2009. *CACTI 6.0: A Tool to Model Large Caches*.

[23] Nickolls, J. and Dally, W.J. 2010. The GPU Computing Era. *IEEE Micro*. 30, 2 (Mar. 2010), 56–69.

[24] NVIDIA 2011. NVIDIA CUDA C Programming Guide Version 4.0. *Changes*.

[25] NVIDIA 2009. NVIDIA's Next Generation CUDA Compute Architecture: Fermi.

[26] Pichai, B. et al. 2013. *Architectural Support for Address Translation on GPUs*.

[27] Power, J. et al. 2014. gem5-gpu: A Heterogeneous CPU-GPU Simulator. *Computer Architecture Letters*. 13, 1 (2014).

[28] Rogers, P. et al. 2013. AMD heterogeneous Uniform Memory Access. AMD.

[29] Rogers, P. 2013. Heterogeneous System Architecture Overview. *Hot Chips 25* (2013).

[30] Rogers, P. 2011. The programmer's guide to the apu galaxy.

[31] Romanescu, B.F. et al. 2010. UNified Instruction/Translation/Data (UNITD) coherence: One protocol to rule them all. *HPCA - 16 2010 The Sixteenth International Symposium on High-Performance Computer Architecture* (Jan. 2010), 1–12.

[32] Rossbach, C.J. et al. 2011. PTask. *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles - SOSP '11* (New York, New York, USA, 2011), 233.

[33] Sodani, A. 2011. Race to Exascale: Opportunities and Challenges Intel Corporation.

[34] Stoner, G. 2012. HSA Foundation Overview. HSA Foundation.

[35] Teller, P.J. 1990. Translation-lookaside buffer consistency. *Computer*. 23, 6 (Jun. 1990), 26–36.

[36] Tong, P.C. et al. 2008. Dedicated mechanism for page mapping in a gpu. U.S. Patent #US20080028181. 2008.

[37] Wong, H. et al. 2010. Demystifying GPU microarchitecture through microbenchmarking. *2010 IEEE International Symposium on Performance Analysis of Systems & Software (ISPASS)*. (Mar. 2010), 235–246.

[38] 2011. *AMD I/O Virtualization Technology (IOMMU) Specification*.

[39] 2013. Volume 3A: System Programming Guide Part 1. *Intel® 64 and IA-32 Architectures Software Developer's Manual*. 4.35–4.38.

Appendix: GPU Page Fault Handler Implementation

Although rare, the GPU memory management unit architecture must be able to handle page faults to have correct execution. Although there are many ways to implement handling page faults, in this GPU MMU architecture we chose to slightly modify the CPU hardware and make no modifications to the operating system. Our page fault handling logic leverages the operating system running on the CPU core similar to CPU MMU page fault logic. We use this design for two reasons. First, this design does not require any changes to the GPU execution hardware as it does not need to run a full-fledged operating system. Second, this design does not require switching contexts on the GPU as it can handle minor page faults by stalling the faulting instruction.

The only CPU change necessary is modification of the microcode that implements the IRET (return from interrupt) instruction. When a page fault is detected by the page table walker logic, the address which generated the fault is written into the page fault register in the GPU page walk unit. Then, the page fault proceeds similar to a page fault generated by the CPU MMU. The faulting address is written into the CPU core's CR2 register, which hold the faulting address for CPU page faults, and a page fault interrupt is raised. Then, the page fault handler in the operating system runs on the CPU core. The operating system is responsible for writing the correct translation into the page table, or generating a signal (e.g. SEGFAULT) if the memory request is faulting. Once the page fault handler is complete, the operating system executes an IRET instruction on the CPU core to return control to the user-level code. To signal the GPU that the page fault is complete, on GPU page faults, we add a check of the GPU page fault register in the IRET microcode implementation. If the address in that register matches the CR2 address then the page fault may be complete (it is possible the operating system could have finished some other interrupt instead). To check if the page fault is complete, the page walk unit on the GPU performs a second page table walk for the faulting address. If the translation is found, then the page fault handler was successful and the page fault register on both the page walk unit and the CPU are cleared. If the second page walk was not successful then the GPU MMU continues to wait for the page fault handler to complete.

There are two different categories of page faults, major and minor. Details of each are discussed in Section 5.1.

Page fault discussion

We implemented the above technique to handle page faults in the gem5 simulator running Linux in full-system mode. Using this implementation, the Linux kernel correctly handles minor page faults for all of our GPU applications.

For this implementation, the page fault handling logic assumes that the GPU process is still running on the CPU core. This will be true if, for instance, the application can run on the CPU and GPU at the same time, or the CPU runtime puts the CPU core into a low power state and does not change contexts.

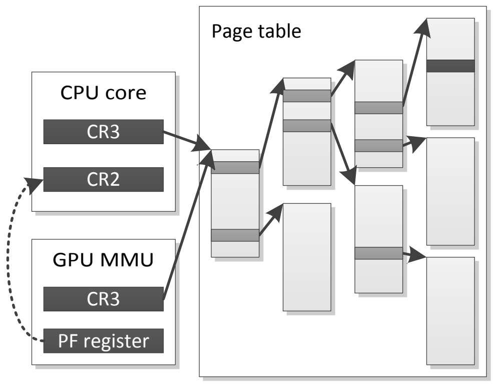

Figure 13: Page walk and page fault overview

However, the CPU runtime may yield the CPU core while the GPU is running. In this situation, page faults are still handled correctly, but they are longer latency as there is a context switch to the process which spawned the GPU work before the operating system begins handling the page fault. This is similar to what happens when other process-specific hardware interrupts are encountered. Another option is to include a separate general purpose core to handle operating system kernel execution for the GPU. This core can handle any page faults generated by the running GPU kernel in the same way as described above.