Towards High Performance Paged Memory for GPUs

Tianhao Zheng[†‡], David Nellans[†], Arslan Zulfiqar[†], Mark Stephenson[†], Stephen W. Keckler[†]
[†] NVIDIA and [‡]The University of Texas at Austin

{dnellans,azulfiqar,mstephenson,skeckler}@nvidia.com, thzheng@utexas.edu

ABSTRACT

Despite industrial investment in both on-die GPUs and next generation interconnects, the highest performing parallel accelerators shipping today continue to be discrete GPUs. Connected via PCIe, these GPUs utilize their own privately managed physical memory that is optimized for high bandwidth. These separate memories force GPU programmers to manage the movement of data between the CPU and GPU, in addition to the on-chip GPU memory hierarchy. To simplify this process, GPU vendors are developing software runtimes that automatically page memory in and out of the GPU on-demand, reducing programmer effort and enabling computation across datasets that exceed the GPU memory capacity. Because this memory migration occurs over a high latency and low bandwidth link (compared to GPU memory), these software runtimes may result in significant performance penalties. In this work, we explore the features needed in GPU hardware and software to close the performance gap of GPU paged memory versus legacy programmer directed memory management. Without modifying the GPU execution pipeline, we show it is possible to largely hide the performance overheads of GPU paged memory, converting an average 2× slowdown into a 12% speedup when compared to programmer directed transfers. Additionally, we examine the performance impact that GPU memory oversubscription has on application run times, enabling application designers to make informed decisions on how to shard their datasets across hosts and GPU instances.

1. INTRODUCTION

Discrete PCIe attached GPUs combined with x86 processors dominate the marketplace for GPU computing environments today. This union of high thermal design power (TDP) processors offers significant flexibility, meeting a variety of application needs. Serial code sections benefit from ILP-optimized CPU memory systems and processors, while parallel code regions can run efficiently on the attached discrete GPUs. However, the GPU's constantly growing demand for memory bandwidth is putting significant pressure on the industry standard CPU–GPU PCIe interconnect, with GPUs having local GDDR or HBM bandwidth that is 30–50× higher than the bandwidth available via PCIe.

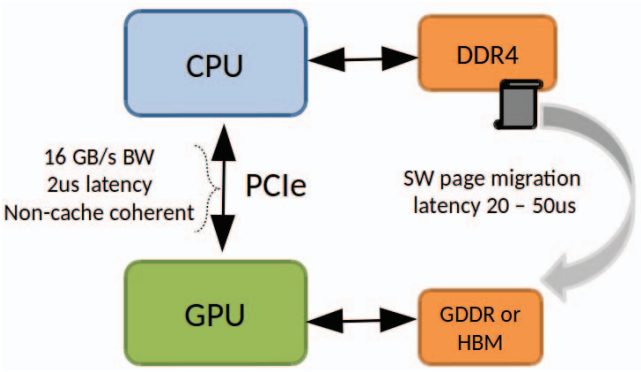

Figure 1: PCIe attached CPUs and GPUs implementing on-demand memory migration to the GPU.

Until now, the onus of utilizing this small straw between the GPU and CPU has fallen squarely on the application programmer. To manage this memory movement, GPU programmers have historically copied data to the GPU up-front, only then launching their GPU kernels. In large part, because GPU kernels have only been able to reference memory physically located in GPU memory. While this restriction has recently been relaxed, performance-conscious programmers still explicitly manage data movement, often by painstakingly tiling their data and using asynchronous constructs to overlap computation with data migration. The challenges of this complex data management is one of the largest barriers to efficiently porting applications to GPUs.

Recently, AMD and NVIDIA have released software-managed runtimes that can provide programmers the illusion of unified CPU and GPU memory by automatically migrating data in and out of the GPU memory [20, 34]. This automated paging of GPU memory is a significant advance in programmer convenience, but initial reports on real hardware indicate that the performance overhead of GPU paged memory may be significant, even for highly optimized microbenchmarks [25]. While up-front bulk transfer of data from the CPU to GPU maximizes PCIe bandwidth, the GPU is left idle until the transfer is complete. With paged GPU memory, the GPU can now begin execution before any data transfers have occurred. This execution will now stall how-

This research was supported in part by the United States Department of Energy. The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of the U.S. Government.

978-1-4673-9211-2/16/$31.00 ©2016 IEEE

ever and incur the page-fetch latency to transfer pages from CPU to GPU memory before returning data to the compute unit (CU) which performed the first load or store to a previously unaccessed page.

Despite PCIe latency being just several microseconds [30] the page fault and DMA-process (called far-faults henceforth) requires several PCIe round trips and significant interaction with the host CPU. Because GPUs do not have the capability to handle page faults within the execution pipeline themselves (as CPUs typically do today), the GPU offloads GPU page table manipulation and transfer of pages to a GPU software runtime executing on the CPU. The end-to-end cost of this process may range from as much as 50 microseconds to as little as 20 microseconds depending on the exact implementation, as shown in Figure 1. For a simple scheme in which the faulting process merely DMAs data to the GPU, the overhead may be on the low end of this range. A more complex scheme where the software fault handler performs significant CPU and GPU page table manipulations on the host, before invalidating and updating the GPU resident page table, will be more expensive.

While GPUs are noted for being memory-latency tolerant, access latencies extending into 10's of microseconds, blocked behind far-fault handling, are orders of magnitude larger than latencies to the local memory system. Because of this, paged memory GPU implementations are likely to see significant performance degradation as a byproduct of this improved programming convenience. However, if these performance overheads can be hidden, GPU paged memory has the potential to improve both programmer productivity and performance, as well as enable GPU computation across datasets that exceed the physical memory capacity of the GPU. In this work we attempt to quantify the performance impact of GPU paged memory implementations and explore both hardware and software improvements that can mitigate the impact of these high-latency paged memory transfers. The contributions of this work are the following:

1. We show that under the existing GPU compute unit (CU) model, blocking paged memory fault handling will not be competitive with programmer controlled memory management, even if far-fault latency is reduced to an unrealistic 5 microseconds.

2. We propose a compute unit agnostic page fault handling mechanism (called replayable far-faults) that treats far-faults as long latency memory operations rather than execution exceptions. This allows CUs to continue issuing instructions from available warps as long as they are not also blocked on pending far-faults. We show that allowing just a small number of simultaneous outstanding far-faults per CU can reduce the impact of GPU paged memory from a 3.6× to 2× slowdown.

3. We show that even with replayable far-faults, PCIe bandwidth is often underutilized and intelligent prefetching is required to maximize CPU–GPU memory transfer bandwidth. We provide a simple software prefetcher implementation that performs within 3% of oracular prefetching and is able to improve the performance of GPU paged memory from a 2× *slowdown* to a 12% speedup when compared to legacy programmer directed transfers.

4. We evaluate the impact of GPU memory oversubscription on workload performance and show that for some workloads oversubscription is likely to be utilized, but for others oversubscription may be untenable, regardless of the paging policies employed.

2. MOTIVATION AND BACKGROUND

GPUs have become the de facto standard for parallel program acceleration because of their highly-threaded, throughput-oriented designs. Whereas traditional CPUs have relied on caches and low latency memory systems to help improve single-threaded execution, GPUs typically forgo large caches and instead rely on multithreading to hide memory system latency. As a consequence of exposed long latency memory operations, GPU programmers try to structure their algorithms to eliminate memory dependencies between threads. To scale performance, GPUs have historically increased both the numbers of threads per compute unit and added additional compute units (CU).

As the number of GPU compute units and threads increases, memory bandwidth must also be scaled commensurately to keep these units fed. Despite GPU memory footprints and bandwidths continuously growing, the memory capacity of GPUs remains relatively small compared to the capacity of the CPU-attached memory system. Because of the separate physical memories the GPU and GPU operate within, data must be copied from CPU memory into GPU memory before the GPU is able to access it.[1] GPU programmers explicitly control this transfer and typically front-load the transfer of memory from host to device before GPU kernel execution to maximize PCIe bus efficiency.

Current GPU programming models do provide the functionality to overlap data transfer and kernel execution, as in the CUDA streams programming model. However the CUDA streams model restricts GPU execution from accessing memory that is currently undergoing transfers. Restrictions like this increase the difficulty for programmers to try and efficiently overlap memory transfers and kernel execution on GPUs. As a result, the dominant "copy then execute" model employed by most GPU programs effectively restricts the effective working set of the application to be GPU memory resident. On kernel completion, programmers must also copy the return data from the GPU to CPU, but the returned data often has a significantly smaller footprint when compared to the pre-execution transfer.

Figure 2 shows the effect of using a BW optimized front-loaded memory transfer, before kernel execution, for workloads from the Rodinia [12], Parboil [42], and DoE HPC suites [11, 19, 32, 45]. As others have observed, this memory transfer overhead is an important aspect to consider when quantifying GPU performance [18]. For many kernels, data migration overhead may match or even dominate kernel execution time. As GPU compute capabilities improve the relative overhead of memory transfers to execution time will grow. Simultaneously, improved CPU–GPU interconnects

[1]While it is possible to access memory directly over PCIe this is rarely done in practice.

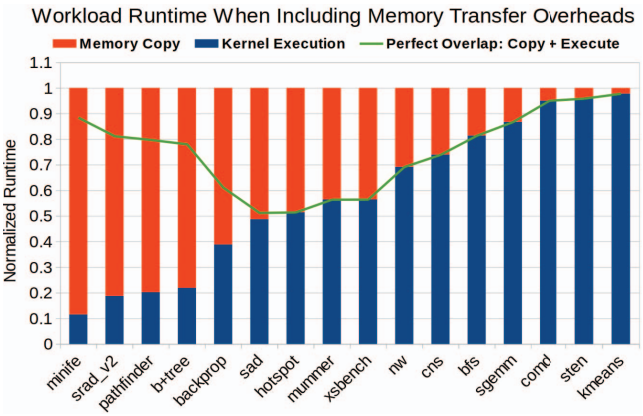

Figure 2: Workload execution time breakdown including data transfer overhead, kernel execution time, and potential speedup if copy and execution could be perfectly overlapped in time.

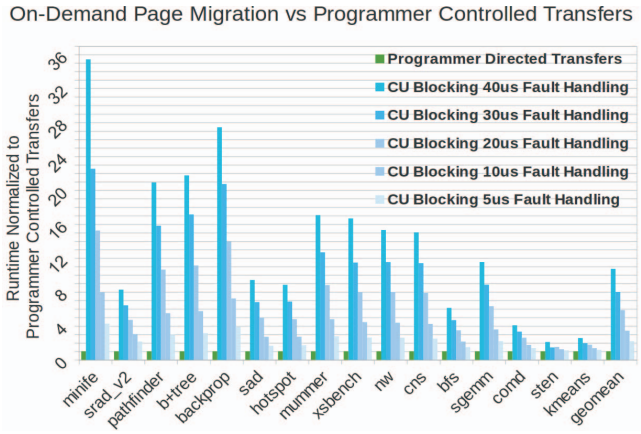

Figure 3: The effect of improving the page-handling latency of far-faults on a GPU that blocks compute units on page-fault handling.

such as faster versions of PCIe and next generation interconnects like NVIDIA's NVLink [35] will reduce memory transfer times. As a result, workloads are likely to continue having a variety of balance points between kernel execution and memory transfer overheads.

In the commonly used programmer directed memory transfer model, the transfer and execute phases of a GPU application are typically serialized, resulting in execution time that is the sum of these costs. The paged memory models recently provided by GPU vendors, not only improved programming convenience, but they also have the potential to improve performance by overlapping these application phases. Programmatically, these runtimes allow application programmers to simply allocate memory as needed and the software runtime will actively transition pages between the CPU and GPU when required for execution to proceed. Figure 2 shows, in green, the theoretical performance improvement achievable if the memory transfer and kernel execution phases could be perfectly overlapped with no dependence. For applications in which either kernel execution or memory transfer dominates the run time, there is little room for performance improvement through overlapping. But, for applications with balanced execution and transfer, efficient on-demand paged memory implementations may be able to actually improve performance beyond that of legacy programmer directed transfers by as much as 49%.

2.1 Supporting On-Demand GPU Memory

Despite using an identical interconnect, on-demand paged GPU memory can improve performance over up-front bulk memory transfer by overlapping concurrent GPU execution with memory transfers. However, piecemeal migration of memory pages to the GPU results in significant overheads being incurred on each transfer rather than amortized across many pages in an efficient bulk transfer. Because pageable memory is a new feature in GPUs the concept of a valid physical page, which is not already present in GPU memory, is a new addition to the GPU's microarchitectural memory model.

GPUs do not support context switching to operating system service routines, thus page-faults that can be resolved by migrating a physical page from the host to the device cannot be handled in-line by the GPU compute units, as they would be on a CPU today. Instead the GPU's MMU (GMMU) must handle this outside of the compute unit, returning either a successful page translation request or a fatal exception. Because the GMMU handling of this page-fault actually invokes a software runtime on the host CPU, the latency of completing this handling is both long (10's us) and non-deterministic. As such, GPUs may choose to implement page-fault handling by having the GMMU stop the GPU TLB from taking new translation requests until the SW runtime has performed the page migration and the GMMU can successfully return a page translation. Under such a scenario, each individual CU could be blocked for many microseconds while its page fault is handled, but other non-faulting compute units can continue making progress, enabling some overlap between GPU kernel execution and on-demand memory migration.

Figure 3 shows the application run time of using such a paged memory design compared to programmer directed transfer under a sweep of possible page-fault handling latencies. Though compute units are now able to continuously execute stalling only for their own memory dependencies, this improvement appears to be subsumed by the additional overhead of performing many small on-demand memory transfers rather than a single efficient up-front transfer. On real GPUs, the load latency for the first access to a page when using paged GPU memory implementations ranges from 20μs (NVIDIA Maxwell) to 50μs (AMD Kaveri), though the exact steps occurring that contribute to this latency are not documented.

Thus using 20μs as an optimistic lower bound on SW controlled page fault latency, applications employing paged GPU memory may see slowdowns of up to 15$\times$ compared to programmer directed memory transfers, with an average slowdown of 6$\times$. Even if page fault latencies could be reduced to 5us, the performance impact of on-demand paging would still result in nearly a 2$\times$ average slowdown ver-

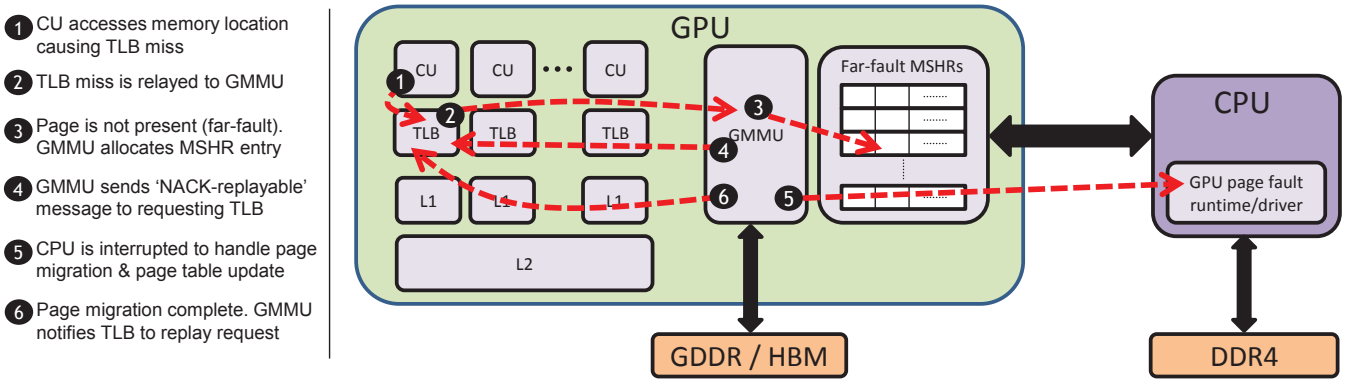

Figure 4: Architectural view of GPU MMU and TLBs implementing compute unit (CU) transparent far page faults.

sus programmer controlled memory transfers. While paged memory handling routines undoubtedly will continue to improve, PCIe's $\sim 2\mu$s latency combined with multiple round trips for CPU–GPU communication makes it extremely unlikely that CPU based GPU page fault resolution will ever become faster than 10μs indicating that alternatives to simply speeding up page-fault latencies need to be examined to enable high performance GPU paged-memory.

This paper explores two techniques that *hide* on-demand GPU page fault latencies rather than trying to reduce them. For example, we can potentially hide page fault latency by not just decoupling GPU CUs from each other under page faults, but by allowing each CU itself to continue executing in the presence of a page-fault. GPUs are efficient in part because their pipelines are drastically simplified and do not typically support restartable instructions, precise exceptions, nor the machinery required to replay a faulting instruction without side effects [29]. While replayable instructions are a common technique for supporting long latency paging operations on CPUs, this would be an exceptionally invasive modification to current GPU designs. Instead, we explore the option of augmenting the GPU memory system, which already supports long latency memory operations, to gracefully handle occasional *ultra-long* latency memory operations (those requiring page fault resolution) in Section 3. In Section 4, we show that in addition to improving CU execution and memory transfer overlap, aggressive page-prefetching can build upon this concurrent execution model and eliminate the latency penalty associated with the first touch to a physical page.

3. ENABLING COMPUTE UNIT EXECUTION UNDER PAGE-FAULTS

In Section 2 we showed that allowing GPU compute units to execute independently, stalling execution only on their own page faults, was insufficient to hide the effects of long latency page fault handling. Because the GPU compute units are not capable of resolving these page faults locally, the GMMU must interface with a software driver executing on the CPU to resolve these faults, as shown in Figure 4. Because this fault handling occurs outside the GPU CU, they are oblivious that a page-fault is even occurring. To prevent overflowing the GMMU with requests while a page-fault is being resolved, the GMMU may choose to pause the CU TLB from accepting any new memory requests, effectively blocking the CU. Alternatively, to enable the CU to continue executing in the presence of a page-fault (also called far-fault to distinguish it from a GMMU translation request that can be resolved in local-memory), both the CU TLB and GMMU structures need to be extended with new capabilities to track and replay far-faulting page translation requests once they have been handled by the software runtime, a capability we refer to as "replayable faults".

Figure 4 shows a simplified architecture of a GPU that supports 'replayable' page faults. ❶ Upon first access to a page it expects to be available in GPU memory, a TLB miss will occur in the CU's local TLB structure. ❷ This translation miss will be forwarded to the GMMU which performs a local page table lookup, without any indication yet that the page may be valid but not-present in GPU memory. Upon discovering that this page is not physically present, the GMMU would normally return an exception to the CU or block the TLB from issuing additional requests. To enable the CU to continue computation under a page fault, our proposed GPU's GMMU employs a bookkeeping structure called 'far-fault MSHRs' (see Figure 4) to track potentially multiple outstanding page migration requests to the CPU. ❸ Upon discovery that a translation request has transitioned into a far-fault, the GMMU inserts an entry into the far-fault MSHR list. ❹ Additionally,the GMMU also sends a new 'Nack-Replayable' message to CU's requesting TLB. This Nack response tells the CU's TLB that this particular fault may need to be re-issued to the GMMU for translation at a later time. ❺ Once this Nack-Replayable message has been sent, the GMMU initiates the SW handling routine for page fault servicing by putting its page translation request in memory and interrupting the CPU to initiate fault servicing. ❻ Once the page is migrated to the GPU, the corresponding entry in the far-fault MSHRs is used to notify the appropriate TLBs to replay their translation request for this page. This translation will then be handled locally a second time, successfully translated, and returned to the TLB as though the original TLB translation request had taken tens of microseconds to complete.

This far-fault handling routine is fully transparent to the GPU CU, the warp that has generated the fault is descheduled until the time its memory request returns. As a re-

sult, within the CU all warps that are not waiting on their own fault can continue to execute. Because all TLB misses can potentially turn into far-faults, one might think that the GMMU should implement as many far-fault MSHRs as there are CU TLB entries. However, these additional entries do not add to the GPU's TLB reach like normal TLB entries making them logically more expensive to implement.

Instead, the GMMU can implement a smaller number of far-fault MSHRs and rely on the Nack-Replayable mechanism to force replay of TLB translation requests for which it has not initiated the SW page faulting process nor is tracking. For example, if the GMMU supports just two concurrent far-faults per CU, a third in-flight translation request will turn into a far-fault. The GMMU can then drop this request and issue the Nack-Replayable to the TLB. This marks this request as needing replay.

Because the TLB knows the supported number of far-fault MSHRs for each CU, the TLB simply maintains a counter of how many far-fault Nack-Replayable messages it has pending. If it receives a Nack that pushes this count above the GMMU threshold, it now knows that this translation request is not being handled and it will have to replay this translation once a GMMU pending far-fault has been resolved. The Nack-Replay mechanism effectively implements a far-fault request throttling mechanism under which the GPU CU can continue execution for TLB hits and page translation requests that can be resolved locally by the GMMU hardware page walk unit. Under this architecture, there is now a critical design trade-off to determine the number of GMMU far-fault MSHRs that must be supported to enable CU's to continue execution under the presence of replayable far-faults without backing up into the CU TLB and blocking non-faulting warps from performing the necessary address translation.

3.1 Methodology

To evaluate the performance impact of our replayable far-faults within a paged GPU memory, we model a hypothetical GPU with the ability of the GMMU to track and trigger far-fault page migrations from the CPU to the GPU as shown in Figure 4. Under the on-demand paged GPU memory model, all pages initially reside in CPU memory at the beginning of kernel execution. Upon first access to a page by the GPU, the TLB will miss causing the GMMU to issue a far-fault to the CPU to migrate the page to GPU memory. In this case, the warp that has missed in the TLB will be stalled, but all other warps within the CU can continue to issue and be serviced by both the TLB and cache and memory hierarchies pending availability of those microarchitectural resources. When the GMMU interrupts the CPU to signal a page-fault we also assume that the SW fault handler will drain the GMMU pending fault queue if there are multiple faults available in a polled fashion. This is a common technique used in device drivers to hide interrupt latency and overhead for devices that signal the CPU at high rates.

We have sized our GPU TLB structures aggressively to try and understand the performance impact of far-misses on GPU paged memory when compared to TLB implementations that provides near ideal performance for applications that execute resident within GPU memory. Our local multi-threaded GMMU page walk and per CU TLB implementation is based largely on those proposed by Pichai et al. [38] and Power et al. [39]. These groups have shown that a highly threaded GMMU page walker and 128 entry, 4-port per-CU TLB supporting hit-under-miss is able to achieve performance within 2% of an idealized TLB design for most workloads. Our workload footprints vary between 2.6MB and 144MB with an average footprint size of 43MB. While these footprints are smaller than contemporary GPU memories they are non-cache resident, thus depending heavily on the GPU memory system to achieve good performance. Additionally, as shown in Figure 2 memory transfer time dominates the GPU execution time for half of these workloads.

Simulator	GPGPU-Sim 3.x
GPU Arch	NVIDIA GTX-480 Fermi-like
GPU Cores	15 CUs @ 1.4GHz
L1 Caches	16kB/CU
L2 Caches	Memory Side 128kB/DRAM Channel
L2 MSHRs	128 Entries/L2 Slice
Per CU TLBs	128-entry, 4-port, per CU TLB supporting hit under miss
GMMU	Local page walk supported by 8KB 20-cycle latency PW-cache
Memory system	
GPU GDDR5	12-channels, 384GB/sec aggregate
DRAM Timings	RCD=RP=12,RC=40,CL=WR=12
CPU–GPU Interconnect	PCIe 3.0 X16, 16GB/sec 20μs far-fault service time

Table 1: Simulation parameters for GPU supporting on-demand paged GPU memory.

We extend GPGPU-Sim [5] with our GMMU memory model as shown in Table 1. It should be noted that we extend the baseline GTX-480 model to better match the bandwidth and compute capabilities of modern GPUs (e.g. increasing the number of miss status handling registers and increasing clock frequency on both CUs and memory). Although the exact steps involved during the page migration process between CPU and GPU memories are undisclosed, we estimate software managed page faults to take somewhere between 20μs and 50μs. We model an optimistic 20μs far-fault handling latency in the rest of our evaluations assuming that software page fault implementations will continue to be optimized over time thus trending to the lower end of this range.

3.2 Experimental Results

Figure 5 shows the performance comparison of on-demand paged GPU memory when varying the number of per CU concurrent GMMU far-faults. Although some applications such as `nw` and `sgemm` show little sensitivity to supporting multiple outstanding far-faults per-CU, the majority of applications see significant performance gains if the GMMU supports multiple pending far-faults per-CU indicating that allowing CUs to continue executing under a pending page fault is an important microarchitectural feature for paged-memory GPUs to support. While we do not esti-

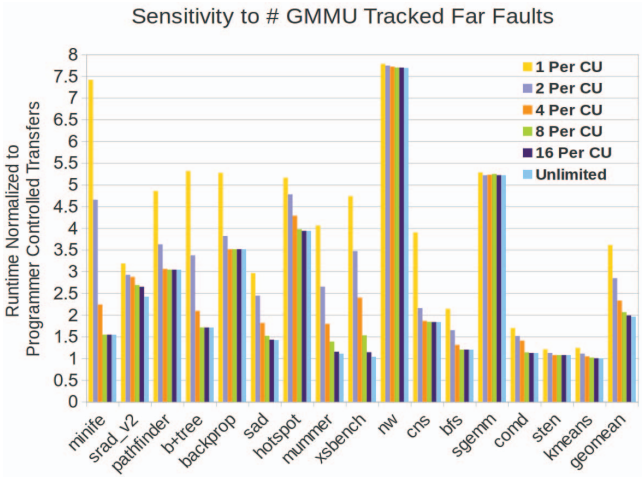

Figure 5: Workload runtime (lower is better) as a function of number of supported GMMU far-faults (per CU).

mate the area impact of our GMMU far-fault MSHRs, this structure essentially duplicates the per CU L1 MSHR entries without providing any additional TLB reach.

Because the GMMU MSHR structure size is the union of all per CU entries that are awaiting far-faults, supporting a large number of entries in the far-fault MSHR structure can be expensive. At application startup, when most TLB misses are likely to be translated into far-faults, supporting a large number of GMMU far-fault MSHRs is ideal, but once most GPU accessed pages are on the GPU, these MSHRs are likely to go underutilized. Therefore, we would like to minimize the total number of entries required in this structure as much as possible. Fortunately, our evaluations show that tracking only *four* in-flight far-faults per-CU at the GMMU is sufficient to sustain good performance and there is little reason to go beyond 16 in flight faults per CU. While it is conceptually possible to allow the GMMU to simply use the state information available at the per-CU TLBs without requiring separate far-fault MSHRs; allowing the centralized GMMU to have direct access into all TLB entries in all CUs is not feasible from an implementation point of view.

Virtual vs Physically Tagged L1 Caches: In this work we have assumed that the per CU L1 caches are physically tagged, which requires the TLB lookup to be completed before the L1 cache tag lookup can be made. Because GPUs typically do not need to support L1 cache coherence, they may choose to be implemented with virtual tags rather than physical. This removes the TLB latency from the critical path of a cache access if permissions checking is also not required. In our evaluation however, use of physical or virtual L1 caches should not change our conclusions primarily because the performance impact of paged GPU memory is dominated by the service time for first access to a page. The virtual or physical implementation of the L1 cache tags does not change the overall compulsory miss rate, on which GPU paged memory faults occur.

Virtual–virtual L1 caches can reduce the overall TLB pressure by reducing the TLB translation rate, but the primary impact of this will be in the porting and number of entries required in the TLB to not limit performance. Because we have assumed an aggressive design for our per CU L1 TLBs, their performance is already near ideal, moving to virtual-virtual L1 caches should result in little application performance change in our simulated GPU. Despite the performance improvement seen when supporting multiple outstanding faults the performance of our proposed paged memory implementation is still 2–7× worse than programmer directed transfers for half of our applications indicating that there are further inefficiencies in the paged memory design that need to be explored. We present further optimizations in Section 4 aimed at closing this performance gap.

4. PREFETCHING GPU PAGED MEMORY

In Section 3 we explored the performance improvements of enabling the GPU memory system to hide pending software-handled page faults from the GPU CUs. By augmenting the TLB and GMMU to allow for multiple pending far-faults, we are able to reduce the performance overhead of paged GPU memory by nearly 45% compared to supporting just a single outstanding far-fault. This improvement comes strictly from enabling additional overlapping of data migration and computation and not from improving the page-fault latency itself. However, even with this improvement our paged memory implementation still has a ∼100% performance overhead compared to programmer directed data transfers. To understand where the remaining overhead is rooted, we examined the PCIe bus traffic generated by our replayable far-fault architecture using multiple pending on-demand faults.

Figure 6 shows the PCIe bandwidth utilization of our workloads as a function of normalized application run time. For each application if the line is below 1.0 it means that it is underutilizing the PCIe bus. We see that only a few applications are able to generate enough on-demand requests to fully saturate the available PCIe bus at any time during execution and the average bandwidth utilization being just 25% across all application's run times. Programmer controlled memory transfers, on the other hand, are able to transfer their datasets to the GPU at nearly full PCIe bandwidth when utilizing efficient up-front memory transfers. This on-demand underutilization provides an opportunity to speculatively prefetch pages across PCIe with little opportunity cost. Successful prefetching of pages will result in the first touch to a page incurring only TLB miss that can be resolved locally by the GMMU, rather than being converted into a far-fault. Thus, a successful prefetch removes the far-fault overhead from the application's critical path, improving average memory latency and freeing up TLB and GMMU far-fault MSHR resources to handle other on-demand faults.

4.1 Demand Prioritized Prefetching

The concept of prefetching has been been explored in many microarchitectural and software-based systems (e.g. [3, 15–17, 27, 33, 41]) but there are several unique properties of prefetching paged GPU memory that need to be met. First, because we are examining workloads that are expected to fit in GPU memory (a requirement we later relax in Section 5), there is little downside to aggressive prefetching. As

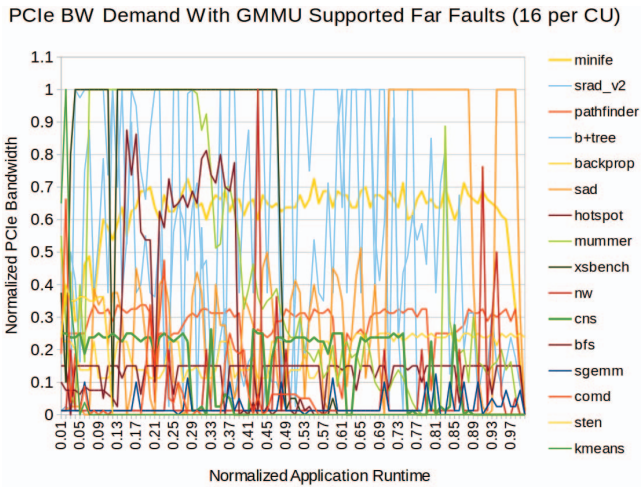

Figure 6: PCIe bus transfer rate showing applications underutilizing transfer BW despite multiple pending GMMU far-faults.

long as the GPU memory has free physical pages, no page must be evicted to allow a prefetched page to be inserted. Second, because the GPU is nearly constantly generating on-demand fetches, we do not want to flood the PCIe interface with prefetch requests or risk throttling on-demand page fetches (that are guaranteed to be useful) with fetches of possibly useless pages. Large DMA transfers over PCIe (the primary mechanism for transferring pages) are efficient because they amortize the cost of the actual DMA setup across many pages. Similarly, batched processing of page fault handling by the GPU SW runtime amortizes the cost of page table updates and CPU–GPU communication across multiple faults.

Because of the use of DMA batched transfers and a finite number of DMA transfers that can be pending, our software prefetcher cannot reactively insert single page prefetches onto the PCIe bus when it detects that the bus is underutilized. Instead it calculates the number of in-flight pages on the PCIe bus required to maximize bandwidth and groups "on-demand" and "prefetch" pages into *transfer sets* that are then issued for DMA. Based on PCIe bandwidth (16GB/s), we can calculate the total number of pages that must be in flight at all times to keep the bandwidth capacity of the PCIe link saturated. At 16GB/s it takes ∼250 nanoseconds to transfer a single 4KB page across PCIe. With an unloaded page fault service request taking ∼20 microseconds, PCIe can thus transfer 80 4KB pages within that interval. Thus, at steady-state, if the software prefetcher can issue an 80 page transfer set every 20μs, regardless of the pages in the group being on-demand or prefetch based, our prefetcher should be able to fully saturate the PCIe bus.[2]

Therefore, our prefetching approach segments execution time into 20μs intervals, and within each interval it constructs an 80-page transfer set that it batch-submits at the end of the interval. Because we do not want to impede on-demand fetches, we add prefetch requests to a transfer set only as a last resort. More specifically, our approach inserts the first unique 80 pages that far-fault to the SW runtime within an interval to the transfer set, S. At the end of the interval, if the transfer set is not full (i.e., $|S| < 80$), the approach inserts speculative prefetch requests to fill it. Because our approach waits until the end of each interval to perform prefetch selection, it is worth noting that prefetch selection is performed in software within the GPU runtime on the CPU, thus it is on the critical path for fault handling latency. We will later show that prefetch selection algorithms that cost even 30μs, have minimal negative effect on GPU performance.

[2]If multiple DMA channels are supported, the arithmetic is similar. For example, with four channels, we can issue 20 page transfer sets every 5μs to maintain full bandwidth.

4.2 Prefetch Page Selection

With a policy for creating and issuing transfer sets that prioritize on-demand fetches, we now consider what pages should be prioritized as part of prefetch page selection. To understand the importance of prefetch selection (versus maximizing PCIe bandwidth via balanced prefetching) we implemented several different prefetching policies. *Random* prefetching makes page selections by choosing candidates randomly from the allocated virtual address (VA) range and inserting them into the transfer set. *Sequential* prefetch, chooses pages from lowest to highest virtual page number order (without regard for the memory access order by the GPU), much like the way programmer controlled memory transfers often perform bulk memory copies. Our *locality* prefetcher tries to leverage spatial locality between recently touched pages. It identifies the last far-faulting page, and then inserts the subsequent $80 - |S|$ pages required to complete the transfer set from the next VA sequential 128 pages beyond this faulting page. The intuition is that because GPUs often stream through memory pages the next N pages will likely see immediate use by the GPU. In all cases, if prefetch selection identifies pages that have already been transfered to the GPU, they are not inserted in the transfer set and the next prefetch candidate is used. If the locality prefetcher runs out of local pages to prefetch it will also choose candidate pages in VA-sequential order.

Finally, to understand the limit of prefetch effectiveness we implement *oracular profiled prefetching*, in which the transfer set is supplemented with prefetch candidates that are inserted in the same order that the application actually touches the pages. We implemented this by using a two execution-pass approach where we track page touch order in the first pass and then feed that reference stream into the prefetcher during the second execution pass. While this is not strictly oracular for non-deterministic applications, our analysis shows it results in very few on-demand fetches caused by candidate misprediction for applications in our benchmark suite.

4.3 Results

Figure 7 shows the performance of these five prefetching policies combined with replayable far-faults, utilizing 16 outstanding faults per CU, and a 20μs unloaded far-fault handling latency. Not surprisingly, oracular prefetching is the best-performing prefetch policy. The magnitude of improvement from this prefetching is significant however, improving performance over 10× for some applications. The locality based prefetcher is the next best policy

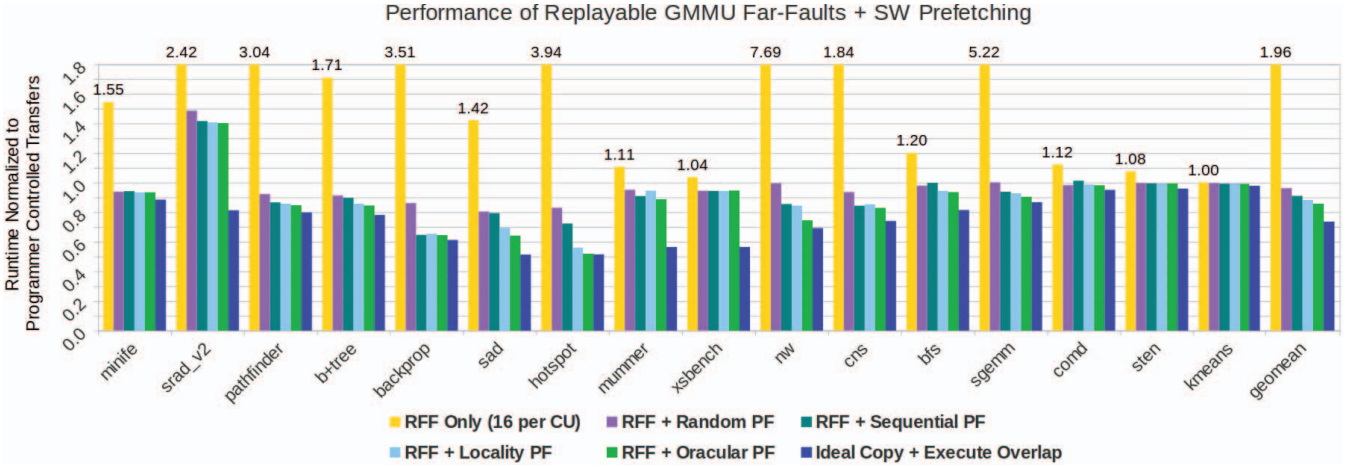

Figure 7: Workload runtime (lower is better) when combining replayable far-faults with demand prioritized prefetching.

and comes within 3% of the absolute performance of the oracular prefetching engine and manages to reduce the geometric mean application runtime below that of legacy memory copy by 12%.

We find that sequential prefetching also works well, primarily because for existing workloads the program allocation order of pages tends to occur in a similar order of first access by the application. Our results show that there is a 11% difference in geometric mean performance between the best and worst prefetcher examined. While the bulk of the benefit of prefetching is clearly from improving PCIe bandwidth utilization, the differentiation in prefetching algorithm becomes important primarily when trying to improve performance beyond legacy memory transfers.

For all prefetchers (including random), we were surprised that the geometric mean performance *surpasses* that of legacy programmer directed memory transfers. As discussed in Section 2, applications that are dominated by either memory transfer time or execution time, the potential savings of overlapping memory transfers with execution is limited. This can be seen by comparing the oracular prefetching performance to the lower limit of execution time possible (labeled "Ideal Copy + Execute Overlap"). Applications such as `hotspot` and `sad` have relatively balanced memory transfer and execution phases, prefetching effectively overlaps these phases, yielding over 20%+ speedups compared to programmer directed memory transfers using locality based prefetching.

For other applications such as `minife` or `sgemm`, prefetching can still provide a large improvement over replayable far-faults alone, but there is insignificant headroom for absolute performance improvement. Finally, applications that are kernel execution bound, such as `comd` or `sten`, prefetching can only improve performance to match programmer directed transfers which are already near the performance limit of efficient transfer and execution overlap.

Replayable Faults Versus Prefetching: One additional design point evaluated was the effect of aggressive prefetching without architectural support for replayable far-faults. We found that aggressive prefetching alone (blocking the SM during fault handling) can bridge much of the performance gap for paged GPU memory, resulting in just a 5% slowdown versus programmer controlled transfers. However, combined with replayable far-faults our final design results in a 12% improvement, indicating both are necessary to improve upon programmer directed transfers.

Effect of Page Size: In this work we have assumed the GPU will utilize 4KB OS pages. There is a growing trend toward using large (2MB+) pages for high performance computing workloads. Current GPU software stacks are unable to leverage transparent large pages and superpages are typically split into the native small page size before being migrated to the GPU via their software runtime. If this software limitation was removed, 2MB page transfers would provide implicit locality prefetching (which our results show is good). However, over PCIe a 2MB transfer takes nearly 128μs of transfer time. This causes a minimum load-to-use interval of 128μs for on-demand requests, significantly worse than the 20μs observed when using 4KB pages.

Sensitivity to Far-Fault Handling Latency: While we have used 20μs as our unloaded far-fault handling latency, the prefetch selection process can add >5μs of software overhead to any existing fault handling implementation. Additionally, 20μs may prove to be too aggressive an estimate for unloaded fault handling latency as GPU runtimes evolve. Because we have designed replayable far-faults and prefetching to *hide* page fault latency rather than reduce it, understanding the performance sensitivity to fault latency is important. Figure 8 shows the performance of our implemented GMMU with far-faults and locality prefetching under unloaded far-fault latencies ranging from 20–40μs. We see that our fault latency hiding techniques are nearly as effective for fault latencies as high as 40μs. Performance decreases just 4% when moving from a 20μs fault latency to a 40μs latency.

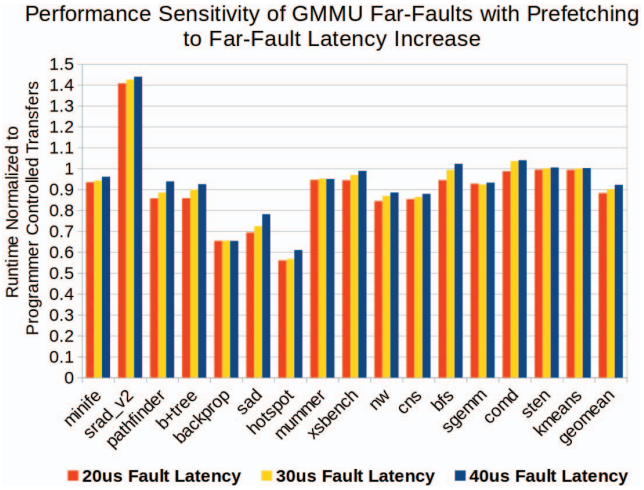

Figure 8: Sensitivity to increasing the end-to-end latency of constructing and servicing a transfer set.

Concluding Discussion: Paged GPU memory has been positioned as a programming convenience feature by the GPU industry. With initial implementations showing significant performance degradation compared to programmer controlled memory management [25], this convenience appears to be the primary upside of early paged memory GPU implementations. We have shown that by combining replayable far-faults with demand prioritized prefetching, it will be possible for paged memory GPUs to not just match the performance of programmer controlled memory transfers, but in many cases exceed it. By overlapping GPU execution with on-demand memory transfers, parallel GPU accelerators may see more applicability to problems that were previously not candidates for offload (due to the memory transfer time hindering the overall GPU throughput). While programming convenience, and now performance, are compelling reasons to used paged memory for GPUs, paged memory also enables another feature that has not been available for GPU programmers until recently: memory oversubscription. We explore the impact of memory oversubscription on GPU workloads in the remainder of the paper.

5. GPU MEMORY OVERSUBSCRIPTION

In Sections 3 and 4 we showed that for workloads with memory footprints that fit within GPU memory, paged GPU memory implementations can be competitive with and sometimes exceed the performance of programmer controlled memory transfers. Achieving performance parity with programmer controlled transfers increases the likelihood that GPU programmers will adopt automatic memory management, but this programmatic convenience is not the only new feature that paged GPU memory provides. The ability to page memory in and out of the GPU on-demand opens the possibility that GPUs can now compute across datasets that are larger than their physical memory, without any programmer heroics. Memory oversubscription allows application developers to make conscious decisions about scale-up or scale-out use of GPUs for their environment depending on application performance characteristics.

While we have shown that supporting replayable far-faults within the GMMU and aggressive software prefetching is good for performance when an application's data set fits within GPU memory, it is not clear that these policies are good, or even sufficient, when GPU memory is oversubscribed. For example, aggressive prefetch of pages is beneficial when no pages have to be evicted from GPU memory, but this could be counterproductive in an oversubscribed GPU memory situation, displacing other heavily referenced pages. When GPU memory has been filled to capacity but on-demand fetches are required for execution to proceed; the GPU must now displace a page that was within GPU memory, creating the need for an *eviction selection* policy. Oversubscribed paged memory implementations must balance a complex interaction of insertion, eviction, and prefetching policies, as well as being heavily dependent on application locality profiles and re-use distances. To understand the interaction of these components for a GPU paged memory system, we decompose these interactions into three individual constituents before re-combining them into our proposed oversubscribed page-memory implementation.

The three axis that we evaluate independently are *eviction policy*, *prefetching*, and *oversubscription rate*. To evaluate these factors, we first must precondition the GPU execution and memory to be in a known starting state. We do this by picking a 50% oversubscription point, meaning only 50% of the application footprint fits in GPU memory, the other pages must reside in CPU memory. To achieve this, we adjust the simulated GPU main memory size down until it can only contain the appropriate fraction of application pages. We then execute the application using the oracular prefetcher described in Section 4 until GPU memory is at capacity. Only at this point do we vary the eviction and prefetch policies, described next. When comparing application run times, we include the warmup execution time in addition to the execution time after GPU memory has become oversubscribed. In this section we provide results for only 11 (out of 16 shown previously) workloads which have application footprints exceeding 32MB.

5.1 Page Eviction Policy

To evaluate the effect of eviction policy on oversubscription, after the GPU memory reaches capacity we turn off page prefetching and migrate pages to the GPU on-demand only. We then model two replacement policies for choosing the eviction candidate: *random* and *LRU*. While true LRU implementations are typically too expensive to implement, it provides a commonly recognized upper bound for the performance achievable via eviction policy changes. While we had initially believed we would need to implement a variety of eviction policies, Figure 9 shows that for all applications except for `xsbench`, when using on-demand only migration, a random eviction policy is surprisingly competitive with LRU.

Analysis shows that despite there being a hot-set of pages in GPU memory, at a 50% oversubscription rate, the hot set typically remains much smaller than the capacity of the GPU memory. Because of this, random selection is likely to pick a cold page for GPU eviction. Due to the streaming nature of many GPU workloads, there is a high incidence of pages that

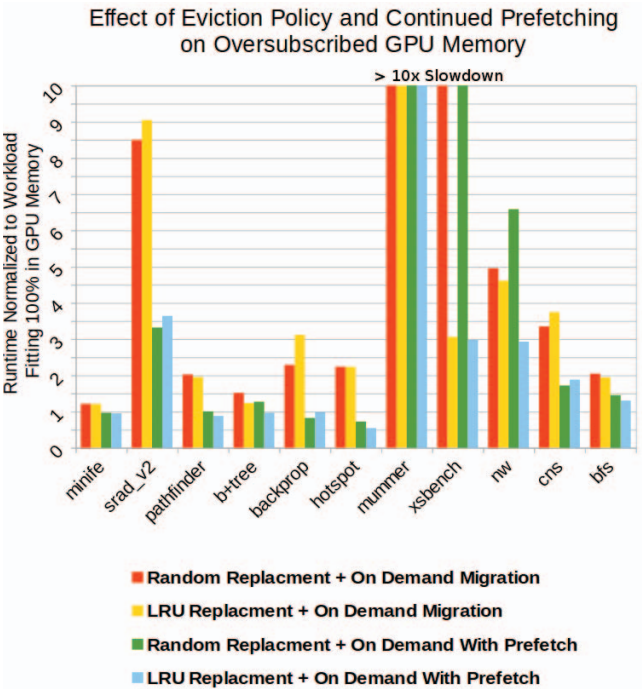

Figure 9: Workload runtime (lower is better) when only 50% of application footprint fits in GPU memory while varying LRU vs random replacement policy and on-demand only vs continuous prefetch.

are migrated to the GPU, accessed several times in a short time period and then becoming dead once the GPU moves on to a different block of memory. `xsbench` is the notable exception to this observation, with LRU policy performing nearly 7× better than random replacement. While random replacement works well for many applications, LRU does perform better on average. True LRU is often approximated by maintaining two, or even one bit per tracked item with periodic resets. It is not clear that even this small amount of per page storage is feasible for paged GPU memory which will need to track millions of pages. Though tracking sets and other techniques have been developed to reduce the storage overheads of LRU, we question whether supporting LRU eviction (rather than random) is worth the implementation effort; especially given that despite perfect LRU, the application slowdown due to memory oversubscription may be too high for application developers to even consider using this feature. This slowdown may also be worse if the GPU caches must be fully flushed, rather than selectively flushed (lines corresponding to the evicted page only) as modeled in this work.

5.2 Prefetching While Oversubscribed

When evaluating eviction policies we disabled speculative prefetching of pages once the GPU memory was oversubscribed because prefetching can cause unnecessary cache thrashing. However the effectiveness of the random eviction policy, compared to LRU, indicates that many of the pages residing in GPU memory are actually dead. If there are a significant number of dead pages within GPU memory, then continuing to prefetch additional pages even while GPU memory is oversubscribed, can potentially improve performance. While these additional prefetches cause page evictions, which consume bandwidth and energy, if the prefetched pages are referenced before they are themselves evicted then the page fault latency can be hidden from the GPU, just as they were in Section 4.

To evaluate the value of continued prefetching, we ran 50% oversubscription simulations comparable to those described in the prior subsection, but once we reach the GPU "memory full" condition, we allow continued locality based prefetching. The prefetching algorithm takes the current on-demand page and prefetches pages that are not on the GPU yet but are within the next 128 virtual contiguous pages in the application address space, as in the locality prefetch algorithm described in Section 4. Rather than attempting to maximize PCIe bandwidth however, oversubscribed-prefetching needs to transfer pages that are more likely to be referenced than pages that are being evicted, or prefetching will waste bandwidth and possibly hurt performance.

The results of performing prefetching during oversubscription are shown in Figure 9 when utilizing both random and LRU eviction policies. We see that when combining prefetching with random replacement, on average, performance improves from 3.9× to 2.5× slower than if the applications fit entirely in GPU memory, but for applications `xsbench` and `nw` prefetching actually hurts performance substantially when used with random replacement. Prefetching combined with LRU replacement does not always improve performance, but on average is the best performing combination of eviction and prefetch policy. Because of the strong variability of application sensitivity to eviction policy and continued prefetching, it is hard to make broad generalizations about the efficacy of page replacement decisions for paged memory GPUs; while LRU in combination with prefetching performs best, improving performance of outlier applications may not justify the implementation overheads required for tracking eviction candidates or implementing more complex prefetching routines.

5.3 Memory Oversubscription Sensitivity

When using oversubscribed GPU memory, we have shown that there can be significant differences in application performance when varying the page eviction and prefetching policies. While interesting, the importance of these differences may be non-consequential if the performance achievable is simply not compelling enough for application developers to consider oversubscribing GPU memory. To shed light on this issue, we examined the effect of oversubscription ratio (the amount of application footprint that fits within GPU memory) on application performance. Figure 10 shows the workload execution time, as the application footprint is varied from 95% fitting in GPU memory down to just 25% fitting in GPU memory. We consider the 95% case to understand if there will be noticeable performance impact on application developers who wish to use the convenience of GPU paged memory, but are not tracking their application memory usage carefully.

We see that for applications such as `backprop`, `minife`, or `hotspot` it may be possible to expand the application

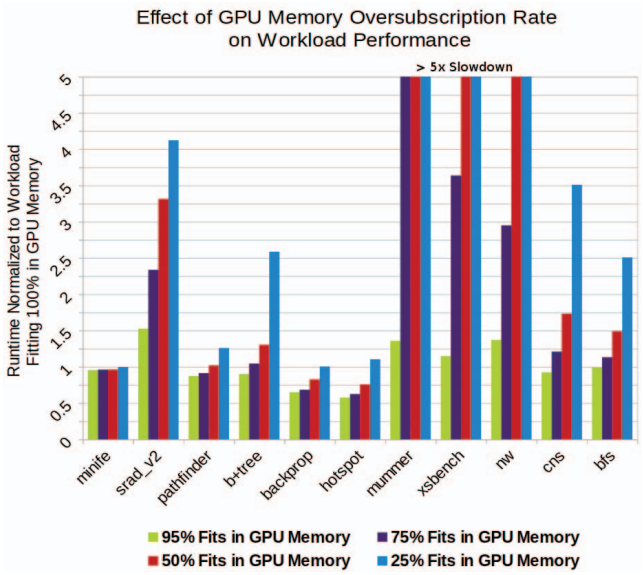

Figure 10: Workload runtime (lower is better) sensitivity to GPU memory oversubscription rate where N% indicates fraction of GPU workload that fits within GPU memory.

footprint 2× or more beyond the GPU memory capacity (shown as 50%) with little performance degradation. These applications respond well to oversubscription because their memory footprints show strong locality and streaming properties that on-demand page migration can exploit. For other applications however, utilizing active paging between CPU and GPU memory will result in significant slowdowns, likely to be deemed unacceptable when evaluating the speedups achievable on a GPU parallel accelerator. With oversubscription slowdowns varying from 0–60×, system designers will likely have to evaluate using oversubscription on a case-by-case basis, first determining whether the application slowdown is acceptable before buying additional (or more expensive) GPUs or HPC nodes to be able to compute across ever larger growing datasets.

6. RELATED WORK

A significant amount of research has investigated page placement and migration strategies for CC-NUMA systems [9, 10, 21, 26, 46, 48]. Much of the effort has been spent trying to understand the implications of memory paging on systems for which memory latency is of primary importance. GPUs are not critically sensitive to memory latency however, and the absolute latency differences explored in CC-NUMA systems is typically an order of magnitude smaller than the latencies in a CPU–GPU PCIe attached system. More recent work on page placement and migration [4,8,13,14,23,44,50] considers interconnect utilization and data sharing characteristics, but the primary focus is again on reducing memory latency rather than increasing bandwidth or hiding long-latency memory operations.

Several recent papers have explored hybrid DRAM + Non-volatile memory subsystems for GPUs where the software runtime or hardware attempts to move data between these memory tiers to optimize GPU performance [47, 49]. While the memory latencies of the non-volatile memories are higher than that of system DRAM, they are not orders of magnitude slower like a memory attached across PCIe. Additionally these studies do not consider the initial overheads of moving data from the host CPU into the GPU memory tiers, and instead measure the GPU performance once the data is present in the GPU's memory. Similarly, several groups have explored using mixed DRAM technologies or DRAM + Non-volatile memories to improve CPU power consumption or capacity but these studies did not consider GPUs or the effect of a (relatively) low performance link between the memory tiers [7,24,31,36,40]. Kim et al. propose using GPU memory as a hardware managed cache of CPU memory but do not address the coherency issues that this solution introduces into a multi-CPU–GPU system [22].

The notion of GPUs supporting virtual memory, a prerequisite for paged memory implementations, is relatively new, and brought on by the need for virtualizing GPUs within the datacenter. Though details about commercial GPU VM implementations are not public several groups have provided proposals about how to implement TLBs that work well with the unique memory access patterns of GPUs [38, 39]. However both focus on GPU TLB implementation and do not support the notion of paged GPU memory. Our paged-memory GPU model effectively builds on top of the GMMU models proposed in these works. Paged memory and oversubscription has an impact on TLB reach which has been explored for CPUs as large-dataset processing has put renewed pressure on the VM subsystem [6, 37, 43].

Additionally, NVIDIA has explored the notion of paged migration from CPU to GPU memory, but their target architecture is not a traditional PCIe-attached GPU. They examine a future GPU where page migration is optional and the GPU may also perform direct cache-line-sized accesses to CPU attached memory [1, 2]. This problem is functionally distinct from paged GPU memory, the approach that NVIDIA and AMD support on current devices, where all pages start in CPU memory and pages referenced by the GPU must first be transfered over PCIe to GPU memory. Finally, Lustig et al. propose a system for data dependency tracking along with program-level APIs to help ensure good execution and data transfer overlap [28]. While effective, this approach places more, rather than less burden on the programmer to manage data transfers between CPU and GPU.

7. CONCLUSIONS

In this work, we have examined the impact that GPU paged memory implementations may have on application performance. Pageable memory for GPUs has been positioned by industry as a feature for improving programmer convenience. Our results show that without architectural and software support, the performance overhead of using pageable memory may give programmers pause, despite the programming convenience. While GPUs are designed to cover longer memory latencies than CPUs, via massive multi-threading, pageable memory introduces a memory latency cost that is nearly 20× what the GPU is designed to hide. Our proposal for replayable GPU far-faults combined with software based prefetching allows us to not just reduce

the performance impact of paged memory for GPUs but exceed the performance of programmer controlled transfers by 12% on average. We also show that, on average, the performance of our paged memory implementation comes within 15% of a perfect overlapping of memory transfer and execution. This leaves little headroom for application designers to further improve performance using programmer pipelined memory transfers, in addition to avoiding yet another layer of implementation overhead.

While many GPU programs are sized to fit within GPU memory, for optimal performance, paged GPU memory enables computation over datasets larger than the available GPU capacity. In an oversubscribed state, the GPUs eviction policy is equally important as the decision to use purely on-demand or prefetched page migrations. Our results show that despite its simplicity, random eviction performs surprisingly well for many applications, causing us to question if the storage overhead and implementation complexity of a pseudo LRU eviction policy is worth the effort to implement in future GPUs. For applications where LRU eviction is significantly better than a random policy, the absolute slowdown of using GPU paged memory over PCIe may not be attractive, even with with improved eviction policies. Our analysis of oversubscribed GPU memory has only scratched the surface in terms of performance optimization and evaluation, but our relatively simple GMMU replayable far-fault microarchitectural model combined with SW prefetching should provide a reasonable baseline upon which more complex policies can be evaluated. Our initial results indicate there is a good probability that GPU paged memory implementations can reach the level of performance where programmers may decide to simply allow the GPU runtime to manage data transfer for them, simultaneously improving both programmer productivity *and* GPU performance.

8. REFERENCES

[1] N. Agarwal, D. Nellans, M. O'Connor, S. W. Keckler, and T. F. Wenisch, "Unlocking Bandwidth for GPUs in CC-NUMA Systems," in *International Symposium on High-Performance Computer Architecture (HPCA)*, February 2015, pp. 354–365.

[2] N. Agarwal, D. Nellans, M. Stephenson, M. O'Connor, and S. W. Keckler, "Page Placement Strategies for GPUs within Heterogeneous Memory Systems," in *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, March 2015, pp. 607–618.

[3] M. Annavaram, J. Patel, and E. Davidson, "Data Prefetching by Dependence Graph Precomputation," in *International Symposium on Computer Architecture (ISCA)*, July 2001, pp. 52–61.

[4] M. Awasthi, D. Nellans, K. Sudan, R. Balasubramonian, and A. Davis, "Handling the Problems and Opportunities Posed by Multiple On-Chip Memory Controllers," in *International Conference on Parallel Architectures and Compilation Techniques (PACT)*, September 2010, pp. 319–330.

[5] A. Bakhoda, G. L. Yuan, W. W. L. Fung, H. Wong, and T. M. Aamodt, "Analyzing CUDA Workloads Using a Detailed GPU Simulator," in *International Symposium on Performance Analysis of Systems and Software (ISPASS)*, April 2009, pp. 163–174.

[6] A. Basu, J. Gandhi, J. Chang, M. D. Hill, and M. M. Swift, "Efficient Virtual Memory for Big Memory Servers," in *International Symposium on Computer Architecture (ISCA)*, June 2013, pp. 237–248.

[7] R. A. Bheda, J. A. Poovey, J. G. Beu, and T. M. Conte, "Energy Efficient Phase Change Memory Based Main Memory for Future High Performance Systems," in *International Green Computing Conference (IGCC)*, July 2011, pp. 1–8.

[8] S. Blagodurov, S. Zhuravlev, M. Dashti, and A. Fedorova, "A Case for NUMA-aware Contention Management on Multicore Systems," in *USENIX Annual Technical Conference (USENIXATC)*, June 2011, pp. 1–15.

[9] W. Bolosky, R. Fitzgerald, and M. Scott, "Simple but Effective Techniques for NUMA Memory Management," in *Symposium on Operating Systems Principles (SOSP)*, December 1989, pp. 19–31.

[10] T. Brecht, "On the Importance of Parallel Application Placement in NUMA Multiprocessors," in *Symposium on Experiences with Distributed and Multiprocessor Systems (SEDMS)*, September 1993, pp. 1–18.

[11] C. Chan, D. Unat, M. Lijewski, W. Zhang, J. Bell, and J. Shalf, "Software Design Space Exploration for Exascale Combustion Co-design," in *International Supercomputing Conference (ISC)*, June 2013, pp. 196–212.

[12] S. Che, M. Boyer, J. Meng, D. Tarjan, J. W. Sheaffer, S.-H. Lee, and K. Skadron, "Rodinia: A Benchmark Suite for Heterogeneous Computing," in *International Symposium on Workload Characterization (IISWC)*, October 2009, pp. 44–54.

[13] J. Corbet, "AutoNUMA: the other approach to NUMA scheduling," http://lwn.net/Articles/488709/, 2012, [Online; accessed 29-May-2014].

[14] M. Dashti, A. Fedorova, J. Funston, F. Gaud, R. Lachaize, B. Lepers, V. Quema, and M. Roth, "Traffic Management: A Holistic Approach to Memory Placement on NUMA Systems," in *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, March 2013, pp. 381–394.

[15] E. Ebrahimi, O. Mutlu, C. Lee, and Y. Patt, "Coordinated Control of Multiple Prefetchers in Multi-Core Systems," in *International Symposium on Microarchitecture (MICRO)*, December 2009, pp. 316–326.

[16] E. Ebrahimi, O. Mutlu, and Y. Patt, "Techniques for Bandwidth-Efficient Prefetching of Linked Data Structures in Hybrid Prefetching Systems," in *International Symposium on High-Performance Computer Architecture (HPCA)*, February 2009, pp. 7–17.

[17] E. Gornish, E. Granston, and A. Veidenbaum, "Compiler-Directed Data Prefetching in Multiprocessors with Memory Hierarchies," in *International Conference on Supercomputing (ICS)*, June 1990, pp. 354–368.

[18] C. Gregg and K. Hazelwood, "Where is the Data? Why You Cannot Debate GPU vs. CPU Performance Without the Answer," in *International Symposium on Performance Analysis of Systems and Software (ISPASS)*, April 2011.

[19] M. Heroux, D. Doerfler, J. Crozier, H. Edwards, A. Williams, M. Rajan, E. Keiter, H. Thornquist, and R. Numrich, "Improving Performance via Mini-applications," Sandia National Laboratories SAND2009-5574, Tech. Rep., 2009.

[20] HSA Foundation, "HSA Platform System Architecture Specification - Provisional 1.0," http://www.slideshare.net/hsafoundation/hsa-platform-system-architecture-specification-provisional-verl-10-ratifed, 2014, [Online; accessed 09-Sept-2015].

[21] R. Iyer, H. Wang, and L. Bhuyan, "Design and Analysis of Static Memory Management Policies for CC-NUMA Multiprocessors," *Journal of Systems Architecture*, vol. 48, no. 1, pp. 59–80, September 2002.

[22] Y. Kim, J. Lee, D. Kim, and J. Kim, "ScaleGPU: GPU Architecture for Memory-Unaware GPU Programming," *IEEE Computer Architecture Letters*, vol. 13, no. 2, pp. 101–104, July 2014.

[23] R. Knauerhase, P. Brett, B. Hohlt, T. Li, and S. Hahn, "Using OS Observations to Improve Performance in Multicore Systems," *IEEE Micro*, vol. 28, no. 3, pp. 54–66, May 2008.

[24] E. Kultursay, M. Kandemir, A. Sivasubramaniam, and O. Mutlu, "Evaluating STT-RAM as an Energy-efficient Main Memory Alternative," in *International Symposium on Performance Analysis of Systems and Software (ISPASS)*, April 2013, pp. 256–267.

[25] R. Landaverde, T. Zhang, A. Coskun, and M. Herbordt, "An Investigation of Unified Memory Access Performance in CUDA," in *High Performance Extreme Computing Conference (HPEC)*, September 2014, pp. 1–6.

[26] R. LaRowe, Jr., C. Ellis, and M. Holliday, "Evaluation of NUMA Memory Management Through Modeling and Measurements," *IEEE Transactions on Parallel Distribibuted Systems*, vol. 3, no. 6, pp. 686–701, November 1992.

[27] W. Lin, S. Reinhardt, and D. Burger, "Designing a Modern Memory Hierarchy with Hardware Prefetching," *Proceedings of IEEE Transactions on Computers*, vol. 50, no. 11, pp. 1202–1218, November 2001.

[28] D. Lustig and M. Martonosi, "Reducing GPU Offload Latency via Fine-grained CPU-GPU Synchronization," in *International Symposium on High-Performance Computer Architecture (HPCA)*, 2013, pp. 354–365.

[29] J. Menon, M. De Kruijf, and K. Sankaralingam, "iGPU: Exception Support and Speculative Execution on GPUs," in *International Symposium on Computer Architecture (ISCA)*, 2012, pp. 72–83.

[30] D. J. Miller, P. M. Watts, and A. W. Moore, "Motivating Future Interconnects: A Differential Measurement Analysis of PCI Latency," in *Symposium on Architectures for Networking and Communications Systems (ANCS)*, October 2009, pp. 94–103.

[31] J. Mogul, E. Argollo, M. Shah, and P. Faraboschi, "Operating System Support for NVM+DRAM Hybrid Main Memory," in *Workshop on Hot Topics in Operating Systems (HotOS)*, May 2009, pp. 14–18.

[32] J. Mohd-Yusof and N. Sakharnykh, "Optimizing CoMD: A Molecular Dynamics Proxy Application Study," in *GPU Technology Conference (GTC)*, March 2014.

[33] T. Mowry and A. Gupta, "Tolerating Latency through Software-Controlled Prefetching in Shared-Memory Multiprocessors," *Journal of Parallel and Distributed Computing*, vol. 12, no. 2, June 1991.

[34] NVIDIA Corporation, "Unified Memory in CUDA 6," http://devblogs.nvidia.com/parallelforall/unified-memory-in-cuda-6/, 2013, [Online; accessed 09-Sept-2015].

[35] ——, "NVIDIA Launches World's First High-Speed GPU Interconnect, Helping Pave the Way to Exascale Computing," http://nvidianews.nvidia.com/News/NVIDIA-Launches-World-s-First-High-Speed-GPU-Interconnect-Helping-Pave-the-Way-to-Exascale-Computin-ad6.aspx, 2014, [Online; accessed 28-May-2014].

[36] S. Phadke and S. Narayanasamy, "MLP-Aware Heterogeneous Memory System," in *Design, Automation & Test in Europe (DATE)*, March 2011, pp. 1–6.

[37] B. Pham, A. Bhattacharjee, Y. Eckert, and G. Loh, "Increasing TLB Reach by Exploiting Clustering in Page Translations," in *International Symposium on High-Performance Computer Architecture (HPCA)*, February 2014, pp. 558–567.

[38] B. Pichai, L. Hsu, and A. Bhattacharjee, "Architectural Support for Address Translation on GPUs: Designing Memory Management Units for CPU/GPUs with Unified Address Spaces," in *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, March 2014, pp. 743–758.

[39] J. Power, M. Hill, and D. Wood, "Supporting x86-64 Address Translation for 100s of GPU Lanes," in *International Symposium on High-Performance Computer Architecture (HPCA)*, 2014, pp. 568–578.

[40] L. Ramos, E. Gorbatov, and R. Bianchini, "Page Placement in Hybrid Memory Systems," in *International Conference on Supercomputing (ICS)*, June 2011, pp. 85–99.

[41] S. Srinath, O. Mutlu, H. Kim, and Y. Patt, "Feedback Directed Prefetching: Improving the Performance and Bandwidth-Efficiency of Hardware Prefetchers," in *International Symposium on High-Performance Computer Architecture (HPCA)*, February 2007, pp. 63–74.

[42] J. A. Stratton, C. Rodrigues, I.-J. Sung, N. Obeid, L.-W. Chang, N. Anssari, G. D. Liu, and W.-m. W. Hwu, "Parboil: A Revised Benchmark Suite for Scientific and Commercial Throughput Computing," IMPACT Technical Report, IMPACT-12-01, University of Illinois, at Urbana-Champaign, Tech. Rep., March 2012.

[43] M. Swanson, L. Stoller, and J. Carter, "Increasing TLB Reach using Superpages Backed by Shadow Memory," in *International Symposium on Computer Architecture (ISCA)*, June 1998, pp. 204–213.

[44] D. Tam, R. Azimi, and M. Stumm, "Thread Clustering: Sharing-aware Scheduling on SMP-CMP-SMT Multiprocessors," in *European Conference on Computer Systems (EuroSys)*, March 2007, pp. 47–58.

[45] J. Tramm, A. Siegel, T. Islam, and M. Schulz, "XSBench - The Development and Verification of a Performance Abstraction for Monte Carlo Reactor Analysis," *The Role of Reactor Physics toward a Sustainable Future (PHYSOR)*, September 2014.

[46] B. Verghese, S. Devine, A. Gupta, and M. Rosenblum, "Operating System Support for Improving Data Locality on CC-NUMA Compute Servers," in *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, September 1996, pp. 279–289.

[47] B. Wang, B. Wu, D. Li, X. Shen, W. Yu, Y. Jiao, and J. Vetter, "Exploring Hybrid Memory for GPU Energy Efficiency Through Software-hardware Co-design," in *International Conference on Parallel Architectures and Compilation Techniques (PACT)*, September 2013, pp. 93–103.

[48] K. Wilson and B. Aglietti, "Dynamic Page Placement to Improve Locality in CC-NUMA Multiprocessors for TPC-C," in *International Conference on High Performance Networking and Computing (Supercomputing)*, November 2001, pp. 33–35.

[49] J. Zhao, G. Sun, G. Loh, and Y. Xie, "Optimizing GPU Energy Efficiency with 3D Die-stacking Graphics Memory and Reconfigurable Memory Interface," *ACM Transactions on Architecture and Code Optimization*, vol. 10, no. 4, pp. 24:1–24:25, December 2013.

[50] S. Zhuravlev, S. Blagodurov, and A. Fedorova, "Addressing Shared Resource Contention in Multicore Processors via Scheduling," in *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, March 2010, pp. 129–142.