Efficient address translation structures for multi-GPUs supporting UVM

Aman Choudhary

Mid-term MTech Project Report

Abstract

Modern GPUs are equipped with unified virtual memory (UVM) that provides a coherent view of the virtual memory address space between CPUs and GPUs in the system. UVM along with demand paging eliminates the need for manual page migrations, which greatly reduces the programmer's burden. It also allows execution of applications whose working sets cannot fit in GPU memory. However, UVM comes with its own drawbacks such as long latency page faults (also called far-faults), frequent TLB misses and associated stalling of warps etc. Interestingly, majority of the address translation structures like TLBs, page tables etc. have identical designs in both UVM and CPU's virtual memory. Our aim, in this work is to study the efficiency of these mechanisms and the associated hardware components that constitute the virtual memory infrastructure. Specifically, we would like to explore newer and efficient design points for these address translation mechanisms, which are more customized to suit the characteristics of GPU applications.

1 INTRODUCTION

With steady increase in GPU compute density and advancement in high-level programming tools, we witness a wider adoption of GPUs by general purpose applications in high-performance computing. In such devices, each computing node typically consists of a CPU with one or more GPU accelerators. Despite industrial investment in both on-die GPUs and next generation interconnect technology, the highest performing parallel accelerators shipping today continue to be discrete GPUs.[6] Connected via PCIe, these GPUs utilize their own privately managed physical memory.

Traditionally, these separate memories have forced GPU programmers to follow the "copy then execute" programming model. This has remained the major roadblock in the general adaptation of GPUs as the responsibility of maintaining complex data structures, and explicit data migration falls on the application developers. To simplify this process, GPU vendors have developed 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. This virtual memory infrastructure, known as unified virtual memory greatly simplifies GPU programming by providing a virtual address space abstraction similar to CPUs and eliminating manual memory management.

However, the introduction of unified virtual memory results in significant performance overheads during (1) address translation and (2) page faults. [1]

1.1 Address Translation

UVM which is basically a mechanism to virtualize memory, relies on page tables to store virtual-to-physical address translations. Conventionally, systems store one translation for every 4KB page. To translate a virtual address on demand, a series of serialized memory accesses are required to traverse the page table. This traversal also known as page table walk is a high latency operation, and can become a bottleneck during accessing memory.

Translation lookaside buffers (TLBs) can reduce the latency of address translation by caching recently-used address translation information. Unfortunately, as application working sets and DRAM capacity have increased in recent years, state-of-the-art GPU TLB designs suffer due to inter-application interference and stagnant TLB sizes.[1] Consequently, contemporary GPUs suffer from poor TLB reach, i.e., the TLB covers only a small fraction of the physical memory working set of an application. This often leads to high TLB

miss rates.

A GPU relies on high degrees of thread-level parallelism (TLP) to hide memory latency. Poor TLB reach can undermine TLP, as a single miss in the TLB invokes the expensive serialized page table walk that often stalls multiple threads. With shortage of threads to mask the memory latency, GPU applications suffer memory stalls and hence performance degradation.

1.2 Demand Paging

With demand paging, an application can request data that is not currently resident in GPU memory. This in turn, triggers a page fault, which requires a long-latency data transfer over the system I/O bus. A single page fault can cause multiple threads to stall at once, as threads often access data in the same page due to data locality. As a result, page faults can significantly reduce the amount of TLP that the GPU can exploit, which significantly hurts performance.

In order to reduce page faults, pages are proactively prefetched into GPU memory speculatively. [5][6] This is done by the GPU runtime on CPU, which is also responsible for resolving the page faults. A successful prefetch avoids a far fault, and saves the associated overhead.

1.3 Proposed Work

Most of the design parameters in UVM have been replicated as is from CPU domain like the 4-level radix-tree page table structure of the x86 architecture, base page size of 4KB, the two-level TLB architecture and so on. However, the memory requirements and nature of GPU applications differ in many ways from their CPU counterparts. In light of this, our goal is to revisit the traditional virtual memory infrastructure and study whether there is scope for improvements. We hope to leverage the insights gained from studies of how memory is managed on GPU and make enhancements accordingly, to UVM such that it is tailored to serve GPU applications. Specifically, we want to explore newer design points for address translation structures including, page tables and TLBs, in order to make them more efficient. To this extent, we propose to perform a series of experiments briefly outlined below:

- GPGPU applications typically allocate memory en masse (i.e., they allocate a large number of pages at a time). [1] The en masse allocation takes place when an application kernel is about to be launched, and the allocation requests are often for a large contiguous region of virtual memory. Hence, we speculate that GPU execution is based around certain hot spots in the virtual address space. Accordingly, we would like to study the nature of memory allocations/ deallocations done by the GPU driver. Also, if it turns out that there is scope for enhancement, we intend to explore a distributed page table design (a set of smaller page tables with lesser depth, one for each contiguous virtual memory region).

- There exists a fundamental trade-off between many conflicting goals like higher TLB reach, lesser memory bloat, lower page fault handling overhead, minimal load-to-use latency etc. based upon the size of page used by the architecture (explained in more detail in Section 3.1). Larger pages result in high TLB reach, but fare poorly for other factors, and reverse is true for small page sizes. Hence, we would like to evaluate whether 4KB is the best choice for base page in GPU scene, or there exists better alternatives.

- Prior work [3] have found out that profiling of TLB-sensitive workloads reveals that there exists significant sharing of pages between across different SMs of GPU. This behavior can range from applications where pages are being shared by only a few GPU cores, to applications where pages are shared by all the cores on the GPU. We would like to utilize this insight to try and come up with a more efficient organization for the TLB hierarchy.

2 BACKGROUND

This section presents key features from start-of-the-art UVM systems for GPU.

2.1 Address Translation

The key to virtualizing memory is to map each virtual address to a physical address in GPU memory. Address translations are maintained at page-granularity that are stored in a multi-level page table. Whenever, a SIMD core in GPU issues an access to a virtual memory address, a page table walk is performed where a page table walker traverses through each level of the page table, until the walker locates the page table entry for the requested mapping in the last level of the table. The physical address is extracted from the page

table entry, which can be used for further lookup in the memory hierarchy.

2.2 Translation Lookaside Buffer

In order to avoid the long latency incurred during page walk, GPUs have caches for storing recently used address mappings, known as TLBs (Translation Lookaside Buffers). With the introduction of TLB into address translation mechanism, any request that hits in the TLB does not require a page walk. However, similar to CPUs, TLB misses trigger a page walk. A multilevel page table requires many memory accesses to translate a single address, hence it is a long latency operation. Therefore, a TLB miss hurts performance as it can stall multiple threads, all waiting for the page walk to finish for the requested page.

In order to reduce TLB misses, GPUs employ multiple levels of TLB, typically private per-SM L1 TLBs and a shared L2 TLB. A shared L2 TLB entry is extended with address space identifiers, to distinguish between different address spaces of applications running on the GPU.[4] As an alternative to shared L2 TLB, another design scheme utilizes a shared page walk cache after the L1 TLB. [4] Both variants incorporate private per-core L1 TLBs, and all cores share the page table walker. GPUs access an order of magnitude more pages than CPUs, requiring a commensurate number of translations. In light of this, the page table walker is highly threaded. [4]

2.3 Demand Paging

In the context of discrete GPUs, UVM extends support for demand paging where all of the memory used by a GPU application does not need to be transferred to the GPU memory at the beginning of application execution. Instead, when an application executes, any request to a GPU page that is not yet present in the GPU memory, a page fault (also known as far-fault) is issued. At this point, the requested data is transferred from the host memory over the PCIe bus to the GPU memory. This page fault handling is expensive because (1) it requires long latency communications between the CPU and the GPU over the PCIe bus, and (2) the GPU runtime on CPU performs a very expensive fault handling service routine. To amortize the overhead, the GPU runtime processes a group of page faults together, which is also referred to as batch processing.[2]

2.4 Replayable Far Faults

Upon first access to a page, which GPU expects to be available in the local memory, a TLB miss occurs in the core's private TLB structure. This translation miss is then relayed to the GMMU (GPU Memory Unit) which performs a local page table lookup, without any indication yet that the page may be valid but not yet present in GPU memory. Upon detecting that a translation request has transitioned into a far-fault, the GMMU inserts an entry into the far-fault MSHR list (a bookkeeping structure to track potentially multiple outstanding page migration requests to the CPU).[6] Additionally, the GMMU also notifies the faulting TLB that this particular fault may need to be re-issued to the GMMU for translation at a later time. Subsequently, the GMMU inserts the page translation request in the page fault buffer and interrupts the CPU to initiate fault servicing. Once the page is migrated to the GPU, the corresponding entry in the far-fault MSHRs is used to alert the appropriate TLBs to replay their translation request for this page. This translation is then handled locally a second time, successfully translated, and returned to the TLB.

2.5 Large Pages

Traditionally GPUs have performed memory mapping at the granularity of 4KB pages. However, TLB has a fixed number of entries, and using 4KB pages provides poor TLB reach. Therefore, proposals have been made for contemporary GPUs to support large pages (of size 2MB), which significantly increases the TLB reach and thus brings down the rate of TLB misses.[1] To simplify translation hardware in a GPU that uses multiple page sizes (i.e., both base pages and large pages), each TLB level contains two separate sets of entries, where one set of entries stores only base page translations, while the other set of entries stores only large page translations.[1]

2.6 Prefetching

Another technique used to reduce page faults and thus improve overall performance of the system is to proactively prefetch selected pages from CPU to GPU memory[5][6]. Successful prefetching of pages will result in the first touch to a page incurring only TLB miss that can be resolved locally by the GPU memory unit, rather than being converted into a far fault. Thus, a successful prefetch removes the far-fault overhead from the application's critical path, improv-

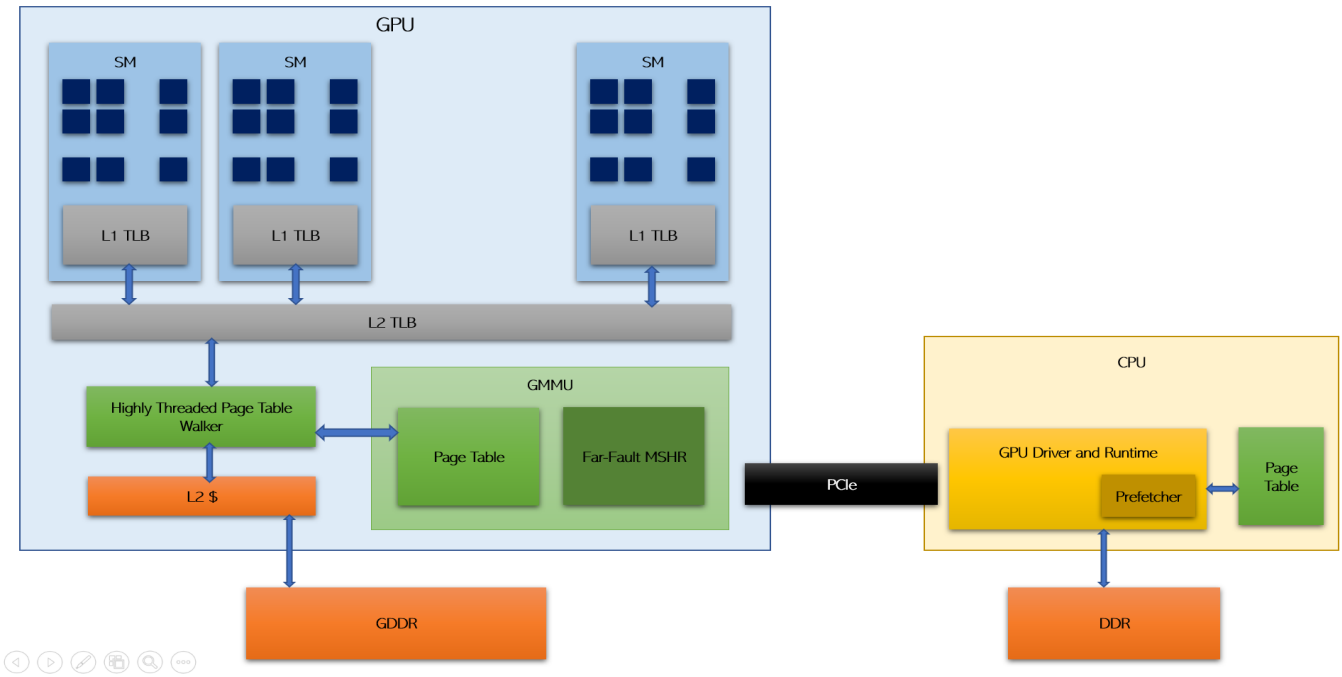

Figure 1: *A diagram illustrating different components in the UVM architecture*

ing average memory latency. Prefetching is done by the GPU runtime on CPU which also resolves the far fault requests. A variety of prefetching policies exists such as random, sequential, locality-based and oracular prefetching[6]. In case when the GPU memory is oversubscribed, physical page frames need to be freed up to accommodate the set of incoming pages from CPU. To this extent, page eviction policies like random and LRU eviction strategy are utilized[6].

2.7 Life cycle of a Memory Access

This section describe the steps involved in a memory access. Refer to Figure 1 which illustrates different components in the UVM architecture.

1. A thread running on a GPU core (or SM) makes a memory access.

2. The first step is to lookup the core's private L1 TLB. If the address mapping is found, then we have the required physical address, which can further be used to access memory.

3. On a miss in the L1 TLB, the GPU initializes a page table walk, which probes the shared page walk cache. For architectures that include a Page Walk Cache, any page walk requests that miss in the page walk cache go to the shared L2 cache and (if needed) main memory.[4]

4. For shared L2 TLB systems, on a miss in the L1 TLB, the GPU checks whether the translation is available in the L2 TLB. If the translation misses in the shared L2 TLB, the GPU initiates a page table walk.

5. The translation miss is forwarded to GMMU, which initiates a local page table walk. The page table walker reads the Page Table Base Register (PTBR) from the core that caused the TLB miss, which contains a pointer to the root of the page table. The walker then accesses each level of the page table, retrieving the page table data from either the shared L2 cache or the GPU main memory.

6. If the GMMU discovers that the page fault cannot be resolved locally, it issues a far fault, which is noted in far-fault MSHR list. The miss request is also inserted into a page fault buffer.

7. The warp that has generated the fault is descheduled until the time its memory request returns. As a result, warps that are not waiting on their own fault can continue to execute in an SM.

8. Periodically, the GMMU issues an interrupt to host to initiate exception handling. The handling starts by draining all of the page fault entries, and creating a batch to be processed together.

9. The batch of requests is dispatched to the GPU runtime on CPU. To handle a multitude of page faults efficiently, the GPU runtime preprocesses the page faults before performing page table walks. This preprocessing includes sorting the page faults and the analysis of page addresses to insert page prefetching requests. [2]

10. Finally, after completion of fault resolution, requested pages (along with prefetched pages) are transferred to the GPU memory over the PCIe bus. If the GPU memory happens to be oversubscribed, some pages need to be evicted in parallel to make room for the incoming pages.

11. Once the page is migrated to the GPU, the appropriate TLBs are notified by the GMMU to replay their translation request for this page. This translation will then be tried locally a second time, successfully translated, and returned to the TLB.

3 PROPOSED WORK

3.1 Optimal Page Size

The size of base pages used in GPUs is 4KB, same as that of CPU. However, there exists a significant tradeoff between several competing factors (all of which effect GPU performance) depending on the page size used. To quantify these tradeoff, a motivational study was conducted. In this study, memory address reference trace was collected for 14 CUDA applications. Using this trace, the set of unique pages accessed (for varying page granularity) was identified for each application. Specifically, the number of unique pages accessed and total memory consumed was evaluated for different page sizes, for each application. The collected data is tabulated on the next page. The various conflicting factors are elaborated below.

- **Number of address translations:** As we can see from Figure 2 and Figure 4, there is drastic decline (roughly 99%) in the number of unique pages accessed, and consequently, the count of address translations required as the page size increases from 4KB to 2MB. Therefore, increasing page size can help to reduce TLB miss rates by improving TLB coverage.

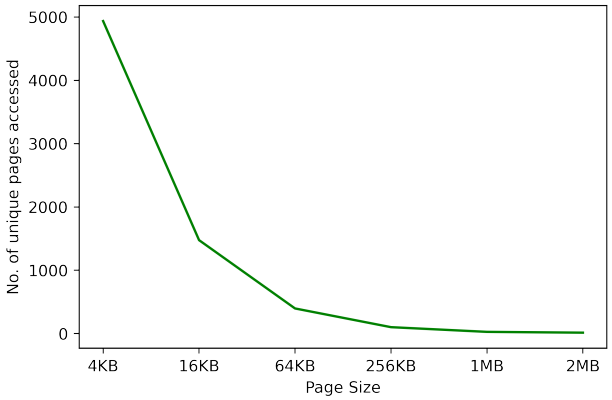

Figure 4: *No. of unique pages accessed vs Page size (Average over all applications)*

empty

- **Total Memory required:** Another trend to note from Figure 3 and Figure 5, is that as the size of pages increases, although the count of unique pages accessed drop, the total memory footprint of the application increases. The bloat is roughly 46% for 2MB page size. This can cause significant memory wastage in the form of internal fragmentation.

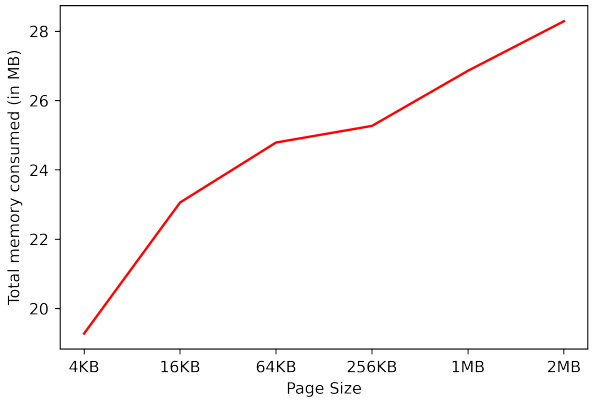

Figure 5: *Total Memory consumed (in MBs) vs Page size (Average over all applications)*

empty

- **Load-to-use Latency:** Prior work [1] observes that the load-to-use latency (i.e., the time between when a thread issues a load request and when the data is returned to the thread)

	No. of unique pages accessed					
Cuda Program	4KB	16KB	64KB	256KB	1MB	2MB
quasirandomGenerator_nvrtc	1937	501	168	42	12	6
shfl_scan	2121	608	163	43	12	7
BlackScholes_nvrtc	1087	366	112	30	11	6
conjugateGradientUM	1258	438	115	30	9	5
reduction	12196	3672	1021	259	66	34
UnifiedMemoryStreams	1742	494	126	33	10	6
conjugateGradient	1661	463	121	33	9	5
MC_EstimatePiInlineQ	1531	397	103	28	9	6
simpleMultiGPU	25145	7799	2001	502	127	64
BlackScholes	1401	447	136	37	10	5
immaTensorCoreGemm	6578	2023	550	139	36	19
cudaTensorCoreGemm	9720	2748	721	181	46	24
threadFenceReduction	1027	259	67	19	7	5
quasirandomGenerator	1707	443	149	39	12	6
MEAN	4936.5	1475.57	396.64	101.07	26.86	14.14

Figure 2: *No. of unique pages accessed for varying page granularity*

	Total Memory footprint (on GPU)					
Cuda Program	4KB	16KB	64KB	256KB	1MB	2MB
quasirandomGenerator_nvrtc	7.56641	7.82812	10.5	10.5	12	12
shfl_scan	8.28516	9.5	10.1875	10.75	12	14
BlackScholes_nvrtc	4.24609	5.71875	7	7.5	11	12
conjugateGradientUM	4.91406	6.84375	7.1875	7.5	9	10
reduction	47.6406	57.375	63.8125	64.75	66	68
UnifiedMemoryStreams	6.80469	7.71875	7.875	8.25	10	12
conjugateGradient	6.48828	7.23438	7.5625	8.25	9	10
MC_EstimatePiInlineQ	5.98047	6.20312	6.4375	7	9	12
simpleMultiGPU	98.2227	121.859	125.062	125.5	127	128
BlackScholes	5.47266	6.98438	8.5	9.25	10	10
immaTensorCoreGemm	25.6953	31.6094	34.375	34.75	36	38
cudaTensorCoreGemm	37.9688	42.9375	45.0625	45.25	46	48
threadFenceReduction	4.01172	4.04688	4.1875	4.75	7	10
quasirandomGenerator	6.66797	6.92188	9.3125	9.75	12	12
MEAN	19.28	23.06	24.79	25.27	26.86	28.29

Figure 3: *Total memory consumed (in MBs) for varying page granularity*

on a far fault increases significantly, when using large sized pages. This is due to high transfer latency for large pages over the I/O bus. The impact of far-faults is particularly harmful for workloads with high locality, as all warps touching the 2MB large page frame must stall, which limits the GPU's ability to overlap the system I/O bus transfer by executing other warps.

- **Page-fault handling overhead:** With increase in page size, the count of page faults also decreases. This is beneficial, as lesser number of faults need to be resolved by the GMMU on GPU, and GPU runtime on CPU, which goes on to minimize the overall page-fault handling time.

Based on the above observations, we intend to do a more comprehensive study to find the optimal base page size for GPU.

3.2 Page Table Structure

GPU uses the same 4-level radix-tree structure for page tables, like the x86 architecture. Prior work suggests that memory allocations in GPU applications occur in bulk before the kernel launch happens[1]. That is large contiguous chunks of virtual memory regions (analogous to VMA regions in CPU) are allocated in the application's address space. Consequently, we speculate that program execution is concentrated around these regions. Accordingly, we would like to conduct detailed experiments to study the true nature

of how memory allocations/ deallocations happen for GPU applications. Based on this study, we would like to explore a distributed design for the existing page table. That is instead of having a single 4-level page table, we would like implement and study whether a set of distributed page tables(with lesser number of levels) one for each contiguous regions yields better performance for GPU applications.

3.3 Efficient TLB Organization

The work done by Baruah et al. [3] brings forward an important revelation about page sharing among GPU threads. The parallel nature of some GPU applications causes threads running on different cores to access the same page or even the same cache block. Sharing in applications is common due to threads running on different GPU cores accessing the same common data structure, such as matrices and arrays. In summary they make three crucial observations about sharing behaviour:

1. The amount of page sharing can vary from application to application. To be specific, the number of sharer L1-TLBs per page and the fraction of total pages shared among these sharers is unique for every application.
2. Second, the shared pages are accessed in a temporal proximity to each other. That is, the same set of TLBs make access to the shared pages throughout the execution of the application.
3. Based on the observation that GPU cores access a similar set of pages, some of the L1-TLB misses on a core can potentially be serviced by another core's L1-TLB.

The paper talks about probing and prefetching mechanisms to deal with this sharing behaviour. We on the other hand have a different approach in mind, that is to explore a shared L1 TLB organization to leverage this observation. In summary, we want to study whether there exists a better approach to incorporate the above observation in the existing TLB structure.

4 PROGRESS MADE

- Completed literature survey of relevant research material and identified problem statement along with the tentative set of experiments/studies to be performed.
- Conducted an initial motivational study to analyze the tradeoff between number of address translations required and memory bloat with several different page sizes.
- Finished setting up of MGPUSim simulator environment for further experiments.

References

[1] R. Ausavarungnirun et al. *Mosaic: A GPU Memory Manager with Application Transparent Support for Multiple Page Sizes*. In Proceedings of The 50th Annual IEEE/ACM International Symposium on Microarchitecture, 2017 (MICRO-50).

[2] Hyojong Kim, Jaewoong Sim, Prasun Gera, Ramyad Hadidi, and Hyesoon Kim. 2020. *Batch-Aware Unified Memory Management in GPUs for Irregular Workloads*. In Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS 2020).

[3] Trinayan Baruah, Yifan Sun, Saiful A. Mojumder, José L. Abellán, Yash Ukidave, Ajay Joshi, Norman Rubin, John Kim, David Kael. *Valkyrie: Leveraging Inter-TLB Locality to Enhance GPU Performance*. In Proceedings of the International Conference on Parallel Architectures and Compilation Techniques (PACT 2020).

[4] R. Ausavarungnirun et al. *MASK: Redesigning the GPU Memory Hierarchy to Support Multi-Application Concurrency*. In Proceedings of Architectural Support for Programming Languages and Operating Systems, 2018 (ASPLOS'18).

[5] Debashis Ganguly, Ziyu Zhang, Jun Yang, and Rami Melhem. *Interplay between Hardware Prefetcher and Page Eviction Policy in CPU-GPU Unified Virtual Memory*. In The 46th Annual International Symposium on Computer Architecture (ISCA 2019).

[6] Tianhao Zheng, David Nellans, Arslan Zulfiqar, Mark Stephenson, and Stephen W Keckler. 2016. *Towards high performance paged memory for GPUs*. In 2016 IEEE International Symposium on High Performance Computer Architecture (HPCA).