SANDIA REPORT
SAND2022-1528
Unclassified Unlimited Release
Printed February 10, 2022

Sandia National Laboratories

Unified Memory: GPGPU-Sim/UVM Smart Integration

C. Hughes
Center for Computing Research
Sandia National Laboratories
Albuquerque, NM 87185
{chughes}@sandia.gov

Y. Liu and T. Rogers
AALP Research Group
Purdue University
West Lafayette, IN 47907
{liu2550, timrogers}@purdue.edu

Prepared by
Sandia National Laboratories
Albuquerque, New Mexico 87185
Livermore, California 94550

Issued by Sandia National Laboratories, operated for the United States Department of Energy by National Technology & Engineering Solutions of Sandia, LLC.

Available to DOE and DOE contractors from

U.S. Department of Energy
Office of Scientific and Technical Information
P.O. Box 62
Oak Ridge, TN 37831

Telephone: (865) 576-8401
Facsimile: (865) 576-5728
E-Mail: reports@osti.gov
Online ordering: http://www.osti.gov/scitech

Available to the public from

U.S. Department of Commerce
National Technical Information Service
5301 Shawnee Road
Alexandria, VA 22312

Telephone: (800) 553-6847
Facsimile: (703) 605-6900
E-Mail: orders@ntis.gov
Online order: https://classic.ntis.gov/help/order-methods

Unified Memory: GPGPU-Sim/UVM Smart Integration

Y. Liu[1], T. Rogers[1], and C. Hughes[2]

[1]AALP Research Group, Purdue University, West Lafayette, IN 47907

[2]Center for Computing Research, National Laboratories, Albuquerque, NM 87185

SAND2022-1528

ABSTRACT

CPU/GPU heterogeneous compute platforms are an ubiquitous element in computing and a programming model specified for this heterogeneous computing model is important for both performance and programmability. A programming model that exposes the shared, unified, address space between the heterogeneous units is a necessary step in this direction as it removes the burden of explicit data movement from the programmer while maintaining performance. GPU vendors, such as 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. However, this runtime support is not included in GPGPU-Sim [1], a commonly used framework that models the features of a modern graphics processor that are relevant to non-graphics applications. UVM Smart [2] was developed, which extended GPGPU-Sim 3.x to incorporate the modeling of on-demand pageing and data migration through the runtime. This report discusses the integration of UVM Smart and GPGPU-Sim 4.0 and the modifications to improve simulation performance and accuracy.

ACKNOWLEDGMENT

The authors acknowledge financial support from the DOE Advanced Simulation and Computing program. Sandia National Laboratories is a multi-mission laboratory managed and operated by National Technology and Engineering Solutions of Sandia, LLC, a wholly owned subsidiary of Honeywell International, Inc., for the U.S. Department of Energy's National Nuclear Security Administration under contract DE-NA0003525. This paper describes technical results and analysis. Any subjective views or opinions that might be expressed in the paper do not necessarily represent the views of the U.S. Department of Energy or the United States Government.

This page intentionally left blank.

CONTENTS

LIST OF FIGURES

LIST OF TABLES

NOMENCLATURE

Table 0-1.

Abbreviation	Definition
CU	Compute Unit
GPU	Graphics Processing Unit
MMU	Memory Management Unit
TLP	Thread-Level Parallelism
UVM	Unified Virtual Memory

1. INTRODUCTION

Graphics processing units (GPUs) have become more general purpose and are increasingly used for a wide range of applications. As an accelerator device, however, a conventional discrete GPU only allows access to its own device memory, which can force developers to make tradeoffs in problem size and performance to ensure that kernels fit in the device memory. This makes it very challenging and costly to run large-scale applications with hundreds of GBs of memory footprint, such as Graph Computing workloads, because it requires careful data and algorithm partitioning in addition to purchasing more GPUs just for memory capacity. To address this issue, recent GPUs support Unified Virtual Memory (UVM) [3]. UVM provides a coherent view of a single virtual address space between CPUs and GPUs with automatic data migration via demand paging. This allows GPUs to access a page that resides in the CPU memory as if it were in the GPU memory, thereby allowing GPU applications to run without worrying about the device memory capacity limit. As such, UVM frees programmers from tuning an application for an individual GPU and allows the application to run on a variety of GPUs with different physical memory sizes without any source code changes. This is good for programmability and portability.

While the feature sounds promising, in reality the benefit comes with a non-negligible performance cost. Virtual memory support requires address translation for every memory request, and its performance impact is more substantial than in CPUs because GPUs can issue a significantly larger number of memory requests in a short period of time. In addition, transferring GPU pages requires large communication overhead between the CPU and GPU over an interconnect such as PCIe and an interrupt handler invocation. Prior work reports that page fault handling latency ranges from 20µs to 50µs [4]. Unfortunately, this page-fault latency cannot be easily hidden even with thread-level parallelism (TLP) in GPUs.

Recently, Debashis explored various hardware prefetchers in the context of FPU's unified memory management [2]. His results show prefetching larger chunks of memory improves PCIe utilization and reduces transfer latency. Further, prefetched pages reduce the number of page-faults and the overhead to resolve them. To explore this design space, he developed a simulation framework, GPGPU-Sim UVM Smart [2], which provides both functional and timing simulation support for UVM.

This page intentionally left blank.

2. BACKGROUND

This chapter explores on-demand memory paging and its impact on page faults.

2.1. On-Demand GPU Memory

On-demand paged GPU memory can improve performance over up-front bulk memory transfer by overlapping concurrent GPU execution with memory transfers. However, fine-grain migration of memory pages to the GPU might cause significant overheads to be incurred on each transfer rather than amortized across many pages in an efficient bulk transfer.

CPUs are able to hide the long-latency of page-faults by context switching. However, 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. 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 (10s of μs) 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.

UVM Smart [2] explores two techniques that are able to hide on-demand GPU page-fault latencies rather than trying to reduce them. First, page-fault latency can potentially be hidden by not only decoupling GPU CUs from each other under page-faults, but by allowing each CU to continue executing in the presence of a page-fault. GPUs are efficient 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. 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, UVM Smart explores the option of augmenting the GPU memory system, which already supports long latency memory operations, to gracefully handle occasional ultra-long latency memory operations. Second, 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.

2.2. GPU Page-Fault Handling

The previous section explained that allowing GPU compute units to execute independently and stalling execution only on their own page-faults, was insufficient to hide the effects of long latency page-fault handling. Due to the fact that 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. The architectural support for this augmentation was proposed in [4], as shown in Figure 2-1. Since 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, both the CU TLB and GMMU structures need to be extended with new capabilities to track and replay page translation requests once they have been handled by the software runtime, a capability refered to as "replayable faults".

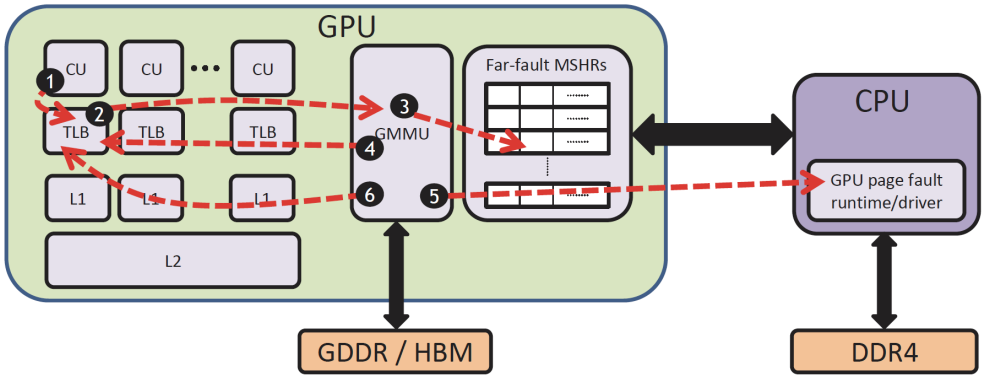


Figure 2-1. Architectural View of GPU MMU and TLBs Implementing CU Transparent Far Page-faults

Figure 2-1 shows a simplified architecture of a GPU that supports 'replayable' page-faults. ① Upon first access to a page that is not present in GPU memory, a TLB miss will occur in the CU's local TLB structure. ② This translation miss will be forwarded to the GMMU which performs a local page table lookup. Once 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, the GPU's GMMU employs a book-keeping structure called 'far-fault MSHRs' to track potentially multiple outstanding page migration requests to the CPU. ③ Upon discovery that a translation request has transitioned into a far-fault, the GMMU inserts an entry into the far-fault MSHR table. ④ 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. ⑤ 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. ⑥ 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.

2.3. Hardware Prefetchers

As described in previous sections, while the CU is able to continue executing upon a page-fault, it is still difficult to completely hide the page-fault latency. Thus the total kernel execution time increases dramatically as it includes far-fault handling latency and memory copy time. cudaMemPrefetchAsync, is an asynchronous construct in CUDA 8.0, that allows programmers to specify an address range to migrate in parallel to the kernel execution. Prefetching later referenced pages helps reduce the number of page-faults and also ensures overlap between data migration and kernel execution. However, the responsibility of what to prefetch and when to prefetch still belongs to the programmer. Zheng *et al.* [4] are the first to propose programmer-agnostic hardware prefetchers to overlap kernel execution and data migration. They introduced (i) random, (ii) sequential, and (iii) locality-aware hardware prefetchers. Debashis *et al.* [2] explore and verify a tree-based hardware prefetcher, called (iv) tree-based neighborhood prefetcher, that is implemented by NVIDIA. Hardware prefetchers take away the burden from the programmer by automatically deciding what and when to prefetch. These hardware prefetchers are incorporated in UVM Smart.

2.3.1. Random Prefetcher

A random prefetcher prefetches a random 4KB page along with the 4KB page for which the far-fault occurred in the current cycle. The prefetch candidate is selected randomly from the 2MB large page boundary to which the faulty page belongs. This not only helps CUDA workloads with random access pattern, but also selecting from 2MB large page boundary instead of the whole virtual address space helps in cases of locality of memory accesses.

2.3.2. Sequential-local Prefetcher

Zheng *et al.* [4] describe their sequential prefetcher as the process of bringing a sequence of 4KB pages from the lowest to the highest order of virtual address irrespective of page access pattern or far-faults. Their locality aware prefetcher migrates consecutive 128 4KB pages (or total 512KB memory chunk) starting from the faulty-page. Debashis *et al.* [2] propose a different variation called sequential-local hardware prefetcher. Each cudaMallocManaged allocation is logically split into multiple 64KB basic blocks. GMMU upon discovering the pages corresponding to the coalesced memory requests are invalid in the GPU page table, first calculates the base addresses of the 64KB logical chunks to which these faulty 4KB pages belong. Thus, GMMU identifies these 64KB basic blocks as prefetch candidates. Further, it divides these candidate basic blocks into prefetch groups and page-fault groups based on the position of the faulty page in the current basic block and then schedules them for sequential transfers by the PCIe interconnect. Prefetching 64KB basic blocks ensures contiguous 16 4KB pages local to the current faulty pages. The position of a faulty page can be anywhere within the corresponding 64KB basic block. Further, multiple faulty

pages are taken in consideration while choosing a basic block for prefetching and can be grouped within the same 64KB boundary.

2.3.3. Tree-based Neighborhood Prefetcher

The semantics of TBNp demands that every cudaMallocManaged allocation is first logically divided into 2MB large pages. Then, these 2MB large pages are further divided into logical 64KB basic blocks to create a full binary tree per large page boundary. By the definition of a full binary tree, every node has exactly 2 children nodes. The root node of each binary tree corresponds to the virtual address of a 2MB large page and the leaf-level nodes correspond to the virtual addresses of the 64KB basic blocks. If the user-specified size of an allocation is not a perfect multiple of 2MB, then the remainder allocation is rounded up to the next $2^i * 64$KB and another full binary tree is created.

The maximum memory capacity of a node in the full binary tree can be calculated as $2^h * 64$KB, where h is the height of a node and h = 0 at the leaf level. On every far-fault, the GMMU first identifies the 64KB basic block corresponding to the faulty page being requested. With the understanding that upon migrating, 16 pages in the basic block will be validated in the GPU page table, GMMU then recalculates the to-be valid size of its parent and grandparent up to the root node of the tree. Here and henceforth, valid size is the size of all valid pages corresponding to the leaf-nodes belonging to a given node. At any point, if GMMU discovers the to-be valid size of a node is strictly greater than 50% of the maximum memory capacity at this level, it tries to balance the valid sizes between the two children of that node. This balancing process is recursively pushed down to the children which have not reached the maximum valid size quota. This balancing act identifies basic blocks for prefetching. This process continues till no more basic blocks at leaf level can be identified as prefetch candidates and the to-be valid size of any non-leaf node including root is not more than 50% of maximum size capacity at its level.

In Figure 2-2, Tree-based Neighborhood Prefetcher is demonstrated by two examples. Both of these examples explain the semantics on 512KB memory chunk for simplicity. These examples use N_h^I to denote a node in the full binary tree, where h is the height of the node and i is the numeric position of the node in that particular level. These examples assume initially all pages in this 512KB allocation are invalid with valid bit not set in the GPU's page table and thus every first access to a page causes a far-fault. In the first example, for the first four far-faults, GMMU identifies the corresponding basic blocks N_0^1 , N_0^3 , N_0^5 , and N_0^7 for migration. As the first byte of every basic block is accessed, the basic blocks are split into 4KB page-fault groups and 60KB prefetch groups. All memory transfers are serialized in time. After these first four accesses, each of nodes N_0^1 , N_0^3 , N_0^5 , and N_0^7 has 64KB valid pages. Then, GMMU traverses the full tree to update the valid page size for all the parent nodes and thus each node at $h = 1$ (N_0^1 , N_1^1 , N_1^2 , and N_1^3) has 64KB valid pages. When the fifth access occurs, GMMU discovers that N_1^0 and N_2^0 will have 128KB and 192KB valid pages respectively. For N_2^0, the to-be valid size is greater than 50% of the maximum valid size of 256KB. Hence, the right child N_1^1 is identified for prefetching. This decision is then pushed down to the children. This process identifies the basic block N_0^2 as a prefetch candidate. Further, GMMU discovers that after prefetching N_0^2, N_3^0 will have 320KB of valid pages which is more than 50% of the maximum valid size of 512KB. Then, node N_3^0 pushes

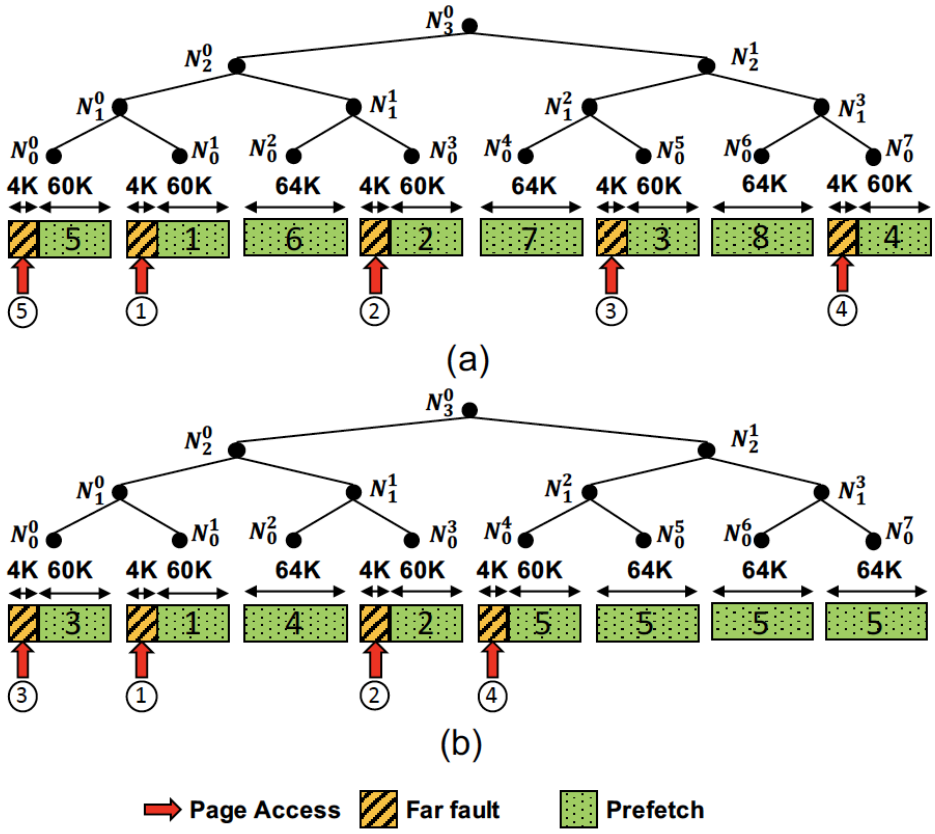


Figure 2-2. Demonstration of TBNp on 512 KB memory chunk for two different page access patterns.

prefetch request to the node N_2^1 which in turn pushes it to its children. This process identifies basic blocks N_0^4 and N_0^6 for further prefetching.

In the second example, the first two far-faults cause migration of basic blocks N_0^1 and N_0^3. GMMU traverses the tree to update the valid size of nodes N_1^0 and N_1^1 as 64KB each. At the third far-fault, as basic block N_0^0 is migrated, the estimated valid sizes for nodes N_1^0, and N_2^0 are updated as 128KB and 192KB respectively. As the valid size of N_2^0 is more than 50% of the maximum valid size of 256KB, N_0^2 is identified for prefetching. After this point, the N_2^0 is fully balanced and both N_2^0 and N_3^0 have exactly 256KB of valid pages. On fourth access, GMMU discovers that the valid size of N_3^0 will be 320KB which is more than 50% of the maximum memory size it can hold. This imbalance causes prefetching of nodes N_0^5 , N_0^6, and N_0^7. Note at this point as GMMU finds four consecutive basic blocks, it groups them together to take advantage of higher bandwidth. Then, based on the page-fault, it splits this 256KB into two transfers: 4KB and 252KB. An interesting point to observe here is that for a full binary tree of 2MB size, TBNp can prefetch at most 1020KB at once in a scenario similar to the second example.

This page intentionally left blank.

3. UVM SMART INTEGRATION

This chapter discusses the integration of UVM Smart with GPGPU-Sim.

3.1. Merge Code

Table 3-1 enumerates the relevant CUDA API calls currently supported by UVM Smart. These calls are enough to enable the execution of the shared virtual memory space programming model. UVM Smart adds the ability to model far-fault handling latency and PCIe transfer latency. Based on Table 3-2, a function to express PCIe bandwidth as a function of transfer size can be deduced. In the simulator, PCIe transfer latency is calculated based on this expression – an additional 100 core cycles for page table walk. The simulator makes simplified assumptions to model the TLB and page table. TLB look ups are performed in a single core cycle, based on the assumption of fully-associative TLB. A multi-threaded model for a page table walk is used and an additional fixed 100 core cycles for the page table walk is add.

Table 3-1. CUDA API Calls Supported by UVM Smart

CUDACall
`cudaMallocManaged`
`cudaDeviceSynchronize`
`cudaMemprefetchAsync`

Table 3-2. CUDA API Calls Supported by UVM Smart

Transfer Size (KB)	PCIe Bandwidth (GB/s)
4	3.2219
16	6.4437
64	8.4771
256	10.508
1024	11.223

The first step in merging UVM Smart into GPGPU-Sim is to understand the difference between the two simulators. Since UVM Smart extended GPGPU-Sim v3.2, the major change is a new class, called `gmmu_t`, that handles the GPU memory management added to UVM Smart. This class stores necessary information about memory requests from all shader cores that missed in the TLB. If a page-fault occurs, it coalesces faults to the same page and handles these page-faults one by one. If hardware prefetch is enabled, it brings extra pages to GPU memory based on the chosen prefetching algorithm (Section 2.3). The update from GPGPU-Sim v3.2 to v4.0 has some minor

changes, such as making simulation cycle count a class variable instead of a global variable. While minor, these changes can cause simulation crashes if not found and manged properly.

3.2. Optimize Simulation Performance

In Section 2, it was discussed that 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 GMMU can successfully return a page translation. Under such a scenario, each individual CU could be blocked for thousands of cycles while its page-fault is handled, but other non-faulting compute units can continue make progress, enabling some overlap between GPU kernel execution and on-demand memory migration. Alternatively, to enable the CU to continue executing in the presence of a page-fault, the CU TLB and GMMU need to be augmented. Even though UVM Smart choose the latter that enables compute unit execution under page-faults, in the worst case, page-fault latency cannot be hidden if all warps are waiting for their page-fault handling requests, especially common at the beginning of kernel execution.

The page-fault latency includes the page-fault handling latency and page migration time. As described in Section 3.1, the page-fault handling latency is fixed and the page migration time is calculated once the memory transfer size is known, thus the simulator knows in which cycle the pages is ready in GPU memory before the page-fault handling request is sent. This simulator assumption is a opportunity to skip those cycles when all warps are stalled due to page-fault handling.

3.3. Improving TLB performance

The GPU MMU design handles TLB flushes similar to the CPU MMU. When the register that stores the pointer to the page table is written, the GPU MMU is notified via inter-processor communication and all of the GPU TLBs are flushed. This is a rare event that usually happens between two different kernels. A more common case is when a page-fault occurs and a new page is brought to GPU memory. When this occurs, all TLBs need to be flushed because the MMU does not know which TLB has stale translation information.

Mechanisms to reduce the cost of TLB shootdowns on CPUs, and emerging heterogeneous memory systems, have attracted significant attention over the last decade. This is due to the rising cost of TLB shootdowns, especially as core counts continue to scale and heterogeneous memory makes its way into mainstream systems. Previous work by Agarwal et al. [5] have studied on mechanisms to reduce the occurrence of TLB shootdowns on a CPU-GPU system. Reducing the cost for translation coherence on virtualized systems has also been studied.

4. EVALUATION

With the UVM Smart simulation framework, we can gain insight into TLB performance. A number of benchmarks have been modified to use UVM, shown in Table 4-1. With these benchmarks, we correlate the timing reported by the simulator with a real GPU and examine the impact of TLB shootdown granularity.

Table 4-1. UVM Smart Benchmarks

Benchmark	Input
bfs	4096
hotspot	30 6 40
pathfinder	1000 20 5
backprop	65536
srad	1024 127 .5 4

4.1. Correlation

A validation sweep was run using six benchmarks. These applications were run using a UVM Smart model that approximates a NVIDIA V100. The simulation parameters are shown in Table 4-2. The overall kernel runtime was compared with the results of running the six applications through nvprof, the NVIDIA profiling tool, on NVIDIA Tesla V100. Figure 4-1 shows the total number cycles that each application took on the simulation model and on the native V100. Note that this is only cycles where a kernel was running and does not include host execution time. The performance gap mainly comes from prefetching algorithms. The blue cross points represent the result of no prefetcher applied, the yellow represents random prefetched, the black represents the sequential locality prefetcher, and the cyan represents the tree-based neighbor prefetcher. It is very clear that the tree-based neighbor prefetcher has the best correlation, which seems very close to the tree-based hardware prefetcher implemented by NVIDIA CUDA driver.

4.2. TLB Performance

Once the models were correlated with a real device, an experiment was designed to justify the positive impact of advanced TLBs. We compared two TLB shootdown granularities: per-TLB entry and whole TLB when the GPU page table is updated. The default implementation is to invalidate the whole TLB of every CU. Alternatively, only one TLB entry will be modified with

Table 4-2. CPU/V100 Model Parameters

Clock	1312MHz
SMs	84
L2 Slices	32
L2 Capactiy	192KiB per slice
HBM Capacity	16384MiB
HBM Stacks	4
Crossbar Frequency	1200MHz
Crossbar Input Ports	2
Crossbar Output Ports	1

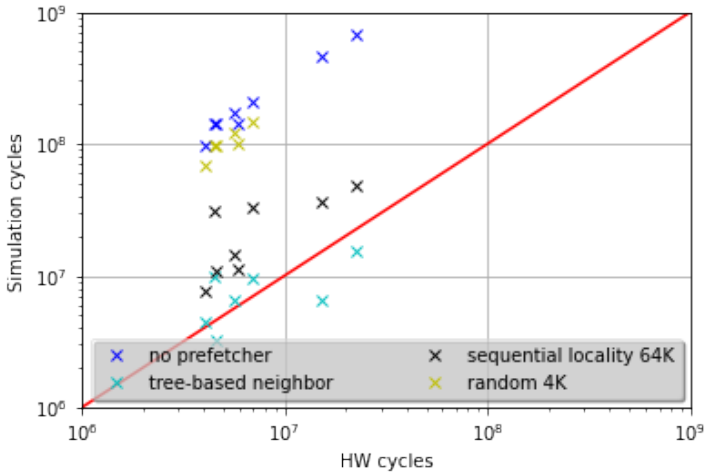
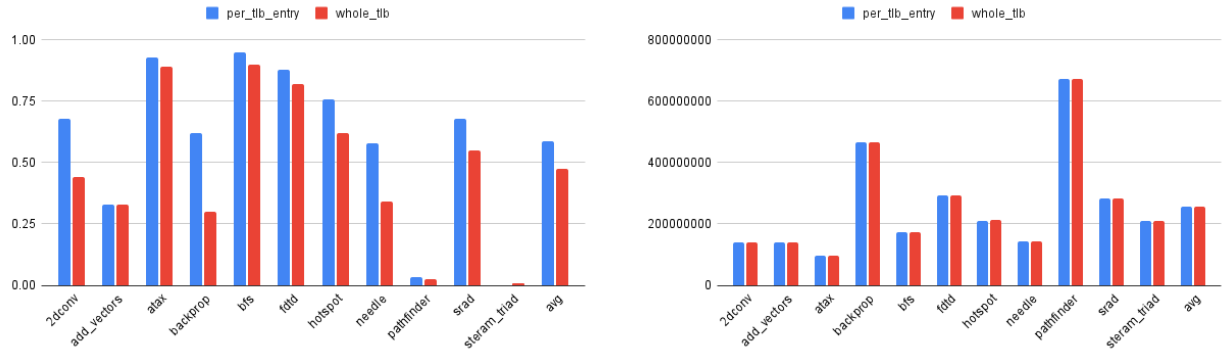


Figure 4-1. Correlations between simulator and hardware.

TLB coherence. Although TLB coherence requires additional hardware support, it should have similar behavior with per-TLB entry shootdown in terms of the TLB hit rate.

From Figure 4-2a, the per-TLB shootdown implementation has a higher hit rate than the whole-TLB shootdown due to it's small granularity. However, this does not translate to performance. As can be seen in Figure 4-2b, there is very little difference in terms of cycle counts between the two implementations. This result demonstrates that if there is no significant latency improvement, the GPU barely benefits from a TLB coherence model.



(a) TLB Hit Rate **(b) Simulation Cycles**

Figure 4-2. TLB Entry Shootdown and TLB Shootdown

This page intentionally left blank.

5. CONCLUSIONS AND FUTURE WORK

This report describes the integration of Unified Virtual Memory (UVM) with GPGPU-Sim and potential research opportunities to improve UVM performance. The integrated GPGPU-Sim with UVM-enabled is able to reduce 92.7% cycles and correspondingly reduce 20% simulation time on average compared to original UVM Smart. Correlation with the NVIDIA V100 is excellent when the hardware prefetcher is enabled, showing 37% error in the runtime for the applications considered. The final phase of the project has involved investigating how TLB performance is affected by different TLB shootdown granularities. Initial performance results demonstrate poor latency improvement but good hit rate improvement.

REFERENCES

[1] T. M. Aamodt, W. W. L. Fung, I. Singh, A. El-Shafiey, J. Kwa, T. Hetherington, A. Gubran, A. Boktor, T. Rogers, A. Bakhoda, and H. Jooybar, "GPGPU-Sim 3.x Manual." http://gpgpu-sim.org/manual/index.php/Main, June 2016.

[2] D. Ganguly, Z. Zhang, J. Yang, and R. Melhem, "Adaptive page migration for irregular data-intensive applications under gpu memory oversubscription," in *2020 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pp. 451–461, 2020.

[3] "NVIDIA Volta V100 White Paper," tech. rep., Nvidia, 2017.

[4] T. Zheng, D. Nellans, A. Zulfiqar, M. Stephenson, and S. W. Keckler, "Towards high performance paged memory for gpus," in *2016 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pp. 345–357, 2016.

[5] N. Agarwal, D. Nellans, M. O'Connor, S. W. Keckler, and T. F. Wenisch, "Unlocking bandwidth for gpus in cc-numa systems," in *2015 IEEE 21st International Symposium on High Performance Computer Architecture (HPCA)*, pp. 354–365, 2015.

DISTRIBUTION

Email—External [redacted]

Name	Company Email Address	Company Name
Tim Rogers	timrogers@purdue.edu	Purdue University

Email—Internal [redacted]

Name	Org.	Sandia Email Address
Simon D. Hammond	01422	sdhammo@sandia.gov
Clay Hughes	01422	chughes@sandia.gov
Technical Library	1911	sanddocs@sandia.gov

Sandia National Laboratories

Sandia National Laboratories is a multimission laboratory managed and operated by National Technology & Engineering Solutions of Sandia LLC, a wholly owned subsidiary of Honeywell International Inc., for the U.S. Department of Energy's National Nuclear Security Administration under contract DE-NA0003525.