CPU/GPU Heterogeneous compute platforms is prevalent now and a programming model specified for this heterogeneous computing is more important for the performance as well as programmability. Shared unified address space between the heterogeneous units is a programming model to eliminate the need to explicitly manage the movement of data between CPU and GPU.

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 runtimes support is not included in the GPGPU-Sim, a very commonly used framework models the features of a modern graphics processor that are relevant to non-graphics applications. To reflect this substantial advance, UVMSmart was proposed, which extended GPGPU-Sim 3.x to incorporate the modelling of on-demand pageing and data migration. This report discusses the integration of UVMSmart and GPGPU-Sim 4.0 and the modifications to improve simulation performance and accuracy.

A Introduction

Graphics processing units (GPUs) have become more general purpose and are increasingly used for a wider range of applications. As an accelerator device, however, a conventional discrete GPU only allows access to its own device memory, so programmers need to design their applications carefully to 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). 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 each 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, paging in and out of GPU memory requires costly communications between 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. Unfortunately, this page fault latency, which is in the order of microseconds, cannot be easily hidden even with ample thread-level parallelism (TLP) in GPUs.

Recently, Debashis proposed a simulation framework, called UVM Smart, to provide both functional and timing simulation support for UVM. To mitigate the costly page-faults handling, his work explores various hardware prefetches in the context of FPU's unified memory management. His result shows prefetching larger chunks of memory improves PCI-e utilization and reduces transfer latency. Further, prefetched pages reduce the number of page-faults and in turn the overhead to resolve them.

B Background

B.1 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.

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, re- turning 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.

UVM Smart 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. 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, it 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. 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.

B.2 GPU Page-Fault Handling

Previous section explained 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 1. 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 refered to as “replayable faults”.

Figure 1 shows a simplified architecture of a GPU that supports ‘replayable’ page faults. ① 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. ② 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’ 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 list. 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.

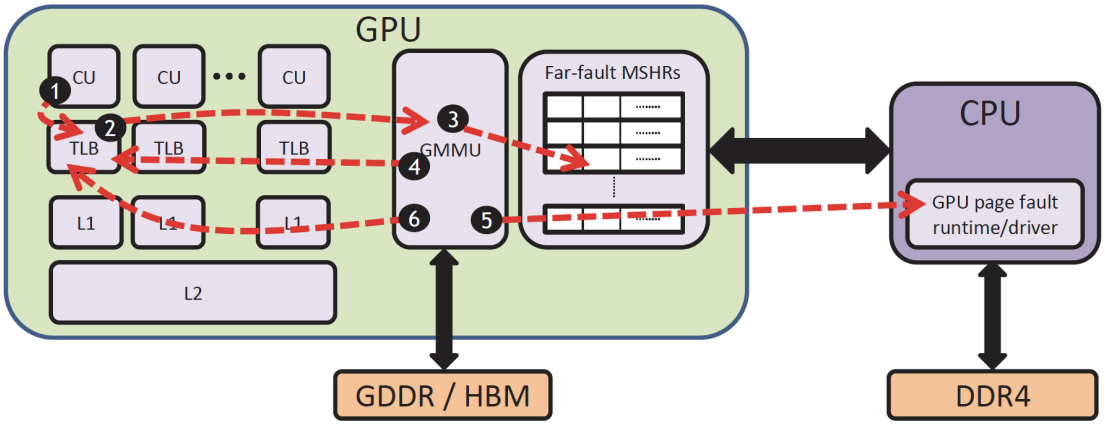

Figure 1: **Architectural view of GPU MMU and TLBs implementing compute unit (CU) transparent far page faults. sus**

B.3 Hardware Prefetchers

With UVM, kernel execution stalls at every far-fault for page allocation and data migration from host to device. 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 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. Hardware prefetchers take away the burden from the programmer by automatically deciding what and when to prefetch. Following their lead, we have incorporated the following hardware prefetchers in UVMSmart.

B.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.

B.3.2 Sequential-local Prefetcher

Zheng et al 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. We 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 in-alid 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 PCI-e 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.

B.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 size of the allocation breaks the principle of a full binary tree. To address this, the remainder allocation is rounded up to the next $2^i * 64$KB and another full binary tree is created. For example, if the programmer specifies 4MB and 192KB size for a cudaMallocManaged allocation, at the time of allocation, GMMU rounds this size up to 4MB and 256KB. Then two full binary trees for 2MB large pages and one full tree for 256KB are created and maintained by the GMMU transparent to the programmer's knowledge. This behavior can also be verified by running

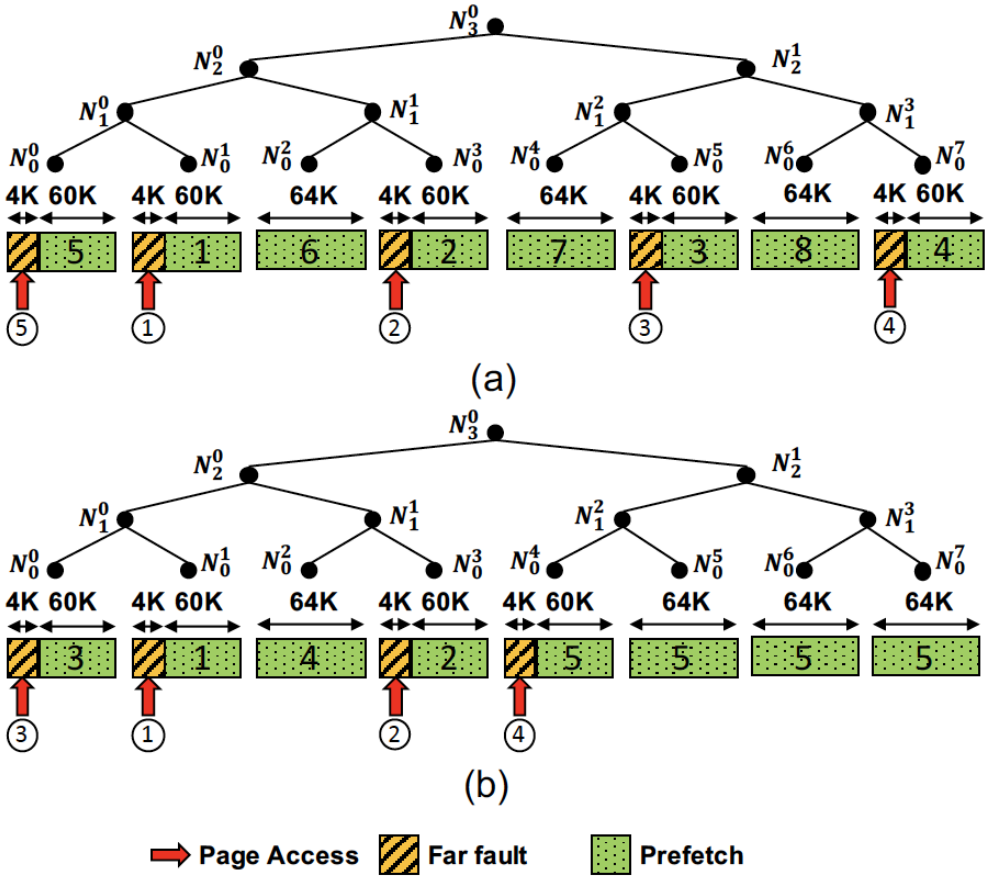

Figure 2: **Demonstration ofTBNp on 512 KB memory chunk for two different page access patterns.**

the micro-benchmarks we have published.

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 grand- parent up to the root node of the tree. Here and henceforth, by valid size we mean 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

Prefetching contiguous pages within 2MB boundary tries to ensure allocation of larger contiguous memory and can also help bypass traversing the nested page tables. This helps reduce the time to access memory. For this same reason, researchers introduced the concept of memory defragmentation to swap and coalesce fragmented memory chunks to ensure contiguous physical memory worth of 2MB large page. However, migrating 4KB pages on-demand and then defragmenting the memory space in the runtime has a substantial overhead. Whereas, TBNp is an adaptive scheme where the prefetch size can vary from 64KB to 1MB based on the access pattern and opportunity of prefetching. Thus, it can get close to 2MB large page locality without causing any additional

performance overhead.

TBNp can be demonstrated with the help of two examples in Figure 2. 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. We further 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. In the example, 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

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 50basic 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.

C UVM Smart Integration

C.1 Merge Code

Table 4 enumerates the CUDA API calls regarding to UVM currently supported by UVM Smart. These calls are enough to enable the execution of shared virtual memory space programming model. UVM Smart mainly adds on the modeling of far-fault handling latency and PCI-e transfer latency. Based on the Table 1, a function to express PCI-e bandwidth as a function of transfer size is deducted. In the simulator, PCI-e transfer latency is calculated based on this expression. an additional 100 core cycles for page table walk. The simulator makes simplified assumptions to model TLB and page table. The TLB look up is performed in a single core cycle based on the assumption of fully-associative TLB. A multi-threaded model for page table walk is used and an additional fixed 100 core cycles for page table walk is considered.

Table 1: **CUDA API Calls Supported by UVMSmart.**

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

Table 2: **CUDA API Calls Supported by UVMSmart.**

CUDACall
`cudaMallocManaged`
`cudaDeviceSynchronize`
`cudaMemprefetchAsync`

The first step in merging UVMSmart into GPGPU-Sim is to understand the difference between them. Since UVMSmart extended GPGPU-Sim v3.2, the major change is a new class, called `gmmu_t`, that handles the gpu memory management added to UVMSmart. This class stores necessary information about memory requests from all shader cores that missed in TLB. Then it figures out whether there is page-fault by looking up the page table. If page-fault, it would coalesce faults to the same page and handle these page faults one by one. If hardware prefetched is enabled, it would bring extra pages to GPU memory in the light of prefetching algorithms(Section B.4). And the update from GPGPU-Sim v3.2 to v4.0 has some minor changes, like making simulation cycle count a class variable instead of a global variable. Such minor changes would cause simulation crashes if not noticed and changed properly.

Table 3 lists the number of benchmarks from various benchmark suites (Rodinia, Parboil, Lonestar, Parboil, HPC Challenge) that modified to use UVM.

Table 3: **UVM Smart benchmarks**

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

The configuration for the GPGPU Simulator supporting Unified Virtual Memory is shown in Listing 1.

Listing 1: **Sample GPGPU-Sim UVM Smart Configuration**

```
[GPU]
clock: 1200MHz
gpu_cores: 80
gpu_l2_parts: 32
gpu_l2_capacity: 192KiB
gpu_cpu_latency: 23840ps
gpu_cpu_bandwidth: 16GB/s

[GPUMemory]
page_size: 4KB
page_fault_handling_latency: 45 $\mu s$
page_table_walk_latency: 100
```

```
[GPU-CPU Interconnect]
PCIe 3.0 16x, 8 GTPS per channel per direction We
```

C.2 Optimize Simulation Performance

In Section B.1 and B.2, we mentioned 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 C.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.

Table 4: **CUDA API Calls Supported by UVMSmart.**

CUDACall
`cudaMallocManaged`
`cudaDeviceSynchronize`
`cudaMemprefetchAsync`

C.3 Improve TLB performance

An important correctness issue of TLB shootdown seems promising to be studied. The GPU MMU design handles TLB flushes similarly to the CPU MMU. Particularly when the register which stored 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, 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, all TLBs need to be flushed because the MMU does not know which TLB has stale translation infomation.

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. 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.

With UVM Smart simulation framework, it would not be hard to gain the insight of TLB performance. Therefore, we run an experiment in terms of GPU TLB shootdown granularity. We

want to justify the positive impact of advanced TLBs. The result shows that the TLB shootdown penalty is not significantly varied among different shootdown granularity.

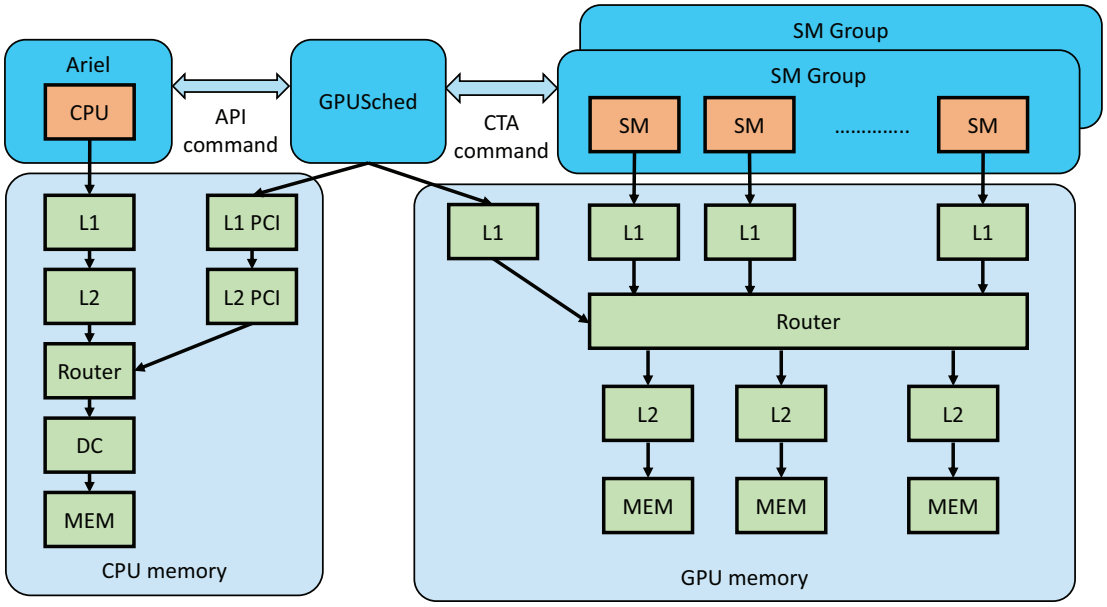

Figure 3: **Timing and memory model for SMs component**

D Evaluation

D.1 Correlation

A validation sweep was run using six benchmarks. These applications were run using UVM Smart model that approximates a Nvidia V100. The simulation parameters are shown in Table 5. The overall kernel runtime was compared with the results of running the six applications through nvprof on Nvidia Tesla V100. Figure 4 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.

D.2 Kokkos

As we stated in C.3, an experiment is designed to justify the positive impact of advanced TLBs. We compared two TLB shootdown granularities - per TLB entry v.s. 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 TLB coherence. Although TLB coherence requires additional hardware support, it should have similar behavior with per TLB entry shootdown in terms of TLB hit rate.

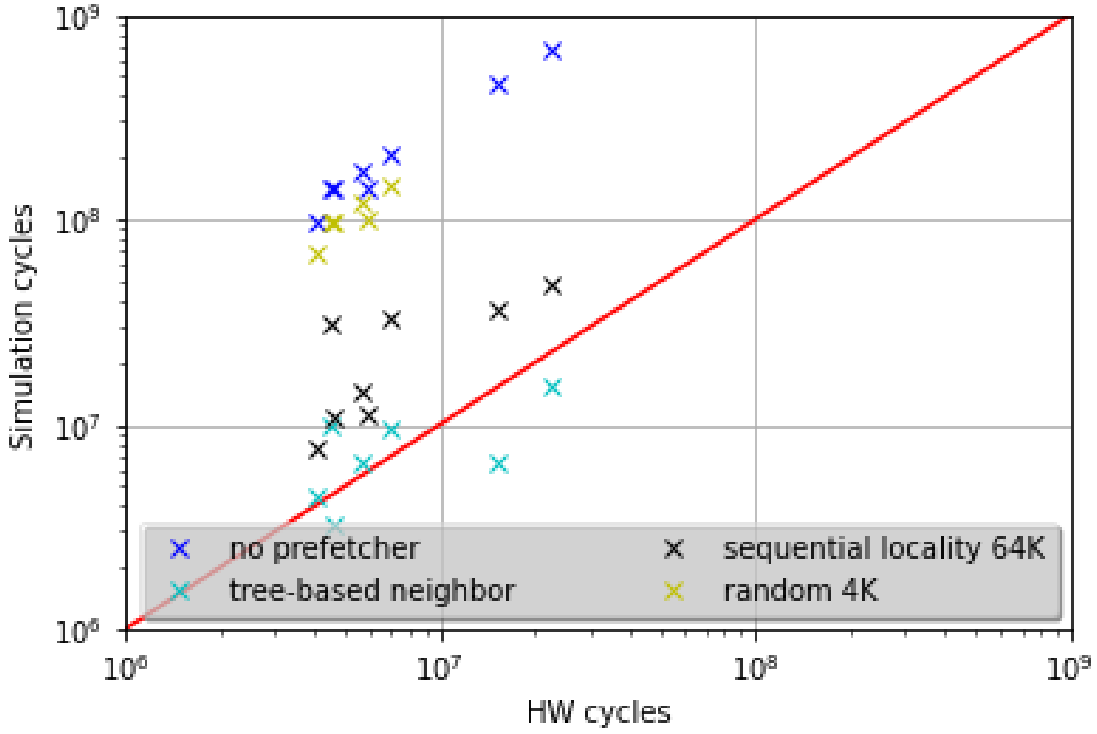

Figure 4: **Correlations between simulator and hardware.**

Table 5: **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

As expected, smaller shootdown granularity gives higher hit rate because it keeps as much as valid translation information at any given time. The average hit rate difference is 11However, TLB shootdown granularity has little effect on simulation cycles. This result demonstrates that if there is no significant latency improvement, the GPU barely benefit from TLB coherence model.

D.3 Lulesh

A parameter sweep was performed using LULESH, described in Section **??**. The device clock was varied from 500MHz to 1312MHz to 1800MHz. The memory clock was varied from 877MHz to 1200MHz to 1600MHz. Figure 7 shows the results, where lower runtime time is better.

As expected, changing the frequency of the backing store has little effect on LULESH for this

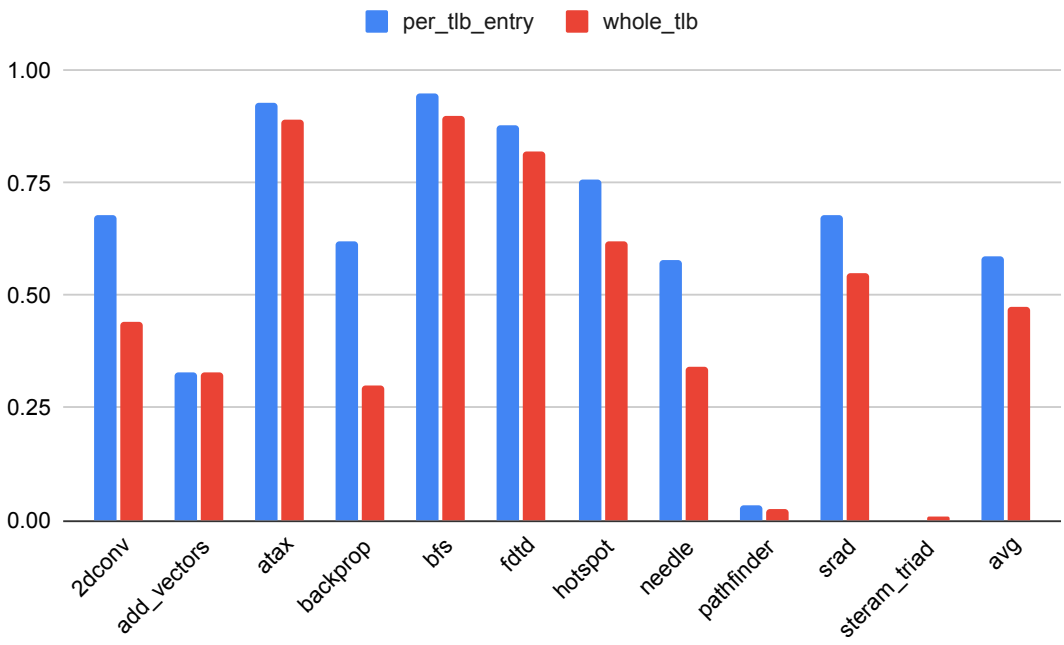

Figure 5: **TLB hit rate for per TLB entry shootdown and whole TLB shootdown.**

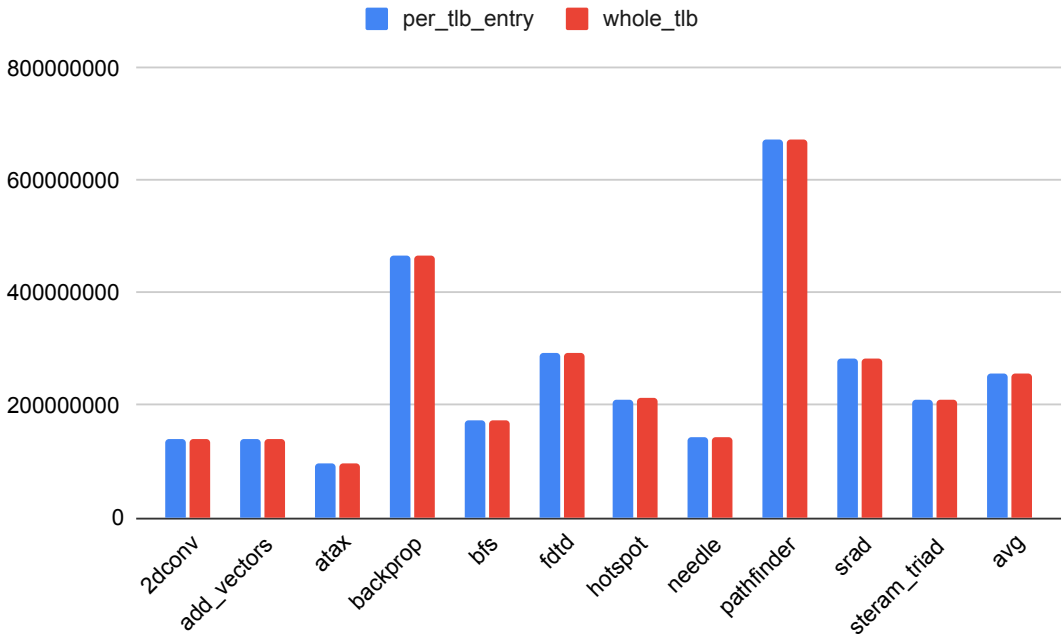

Figure 6: **Simulation cycles for per TLB entry shootdown and whole TLB shootdown.**

problem size because it is not memory bandwidth bound. The most improvement is seen at the low

device clock frequency, but at this frequency the speedup is still small at 1.04x. However, increasing the frequency of the SMs does improve the performance noticeably. Going from 500MHz to 1312MHz shows a 2.5x speedup; going from 1312MHz to 1800MHz shows a further 1.3x speedup.

Although this was a small study, one can imagine being able to run a more complete parameter sweep over any of the Balar parameters.

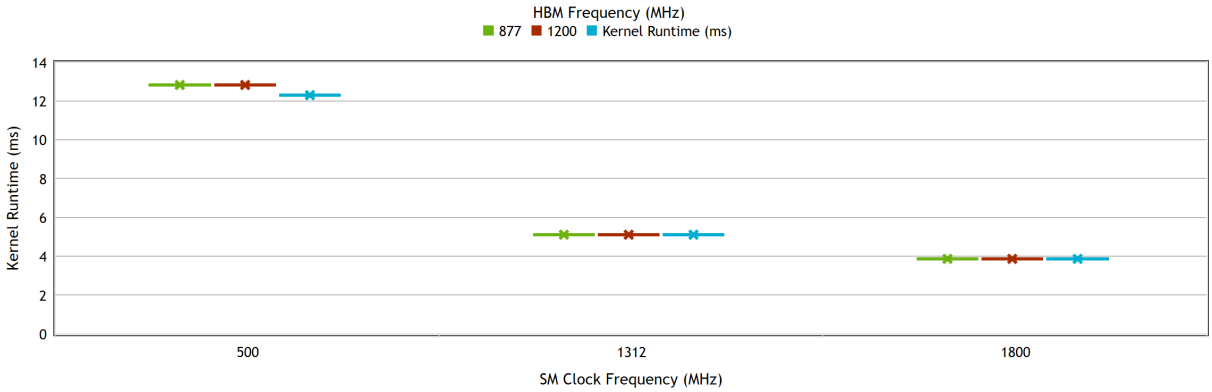

Figure 7: **GPU Parameter Sweep Using LULESH (Baseline was 1312MHz/877MHz)**

D.4 Parallel Simulation Performance

To show the parallel simulation performance with SST-GPU, we split V100 Volta GPU into 1, 2, and 4 SM groups and put each SM group in one thread. To show the advantage of parallel simulation, we use Vector Addition application and accumulate input vector elements 2000 times to its output vector elements. Thus, each thread cause more time at computing but not memory accesses. We launch 84 CTAs and each with 256 threads to make sure at least 1 CTA per SM.

Figure 8 shows the total time cost to run Vector Addition with the different numbers of SM groups. We assign CPU and its memory to one thread and each of the SM group and its memory hierarchy to one thread. The GPU scheduler is attached to the first SM group thread. We achieve on average 5% speedup with 2 SM groups. Note that a significant amount of time are due to initialization, configuration, and memory copy.

E Conclusion

This report described the final integration of the SST-GPU project. Functional validation against the Kokkos Kernels unit tests shows that the GPU component can successfully run more than 48.9% of the tests. Correlation with the Waterman V100 testbed is excellent, showing XX% error in the runtime for the applications considered. The final phase of the project has involved parallelizing the scheduler and groups of SMs, dubbed *SM Groups*, using multi-threaded. Initial performance results demonstrate good scalability using default scheduling policies with additional opportunities to improve parallel scheduling performance.

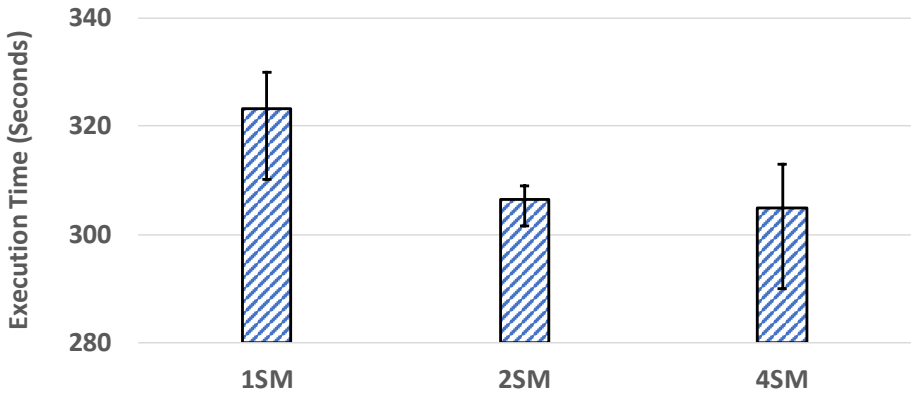

Figure 8: **SST-GPU total executon time with the different numbers of SM groups for V100 Volta GPU. We run 5 times for each configuration and mark the average as well as the maximum and minimum execution time in this figure.**

Acknowledgment

We would like to thank Gwen Voskuilen for her help with memHierarchy and recommendations on debugging problems with the NIC and interconnect. We would also like to thank Arun Rodrigues and Scott Hemmert for their support and help in defining the scope of the project.

References