Efficient Static and Dynamic Memory Management Techniques for Multi-GPU Systems

Max Grossman
Dept. of Computer Science - MS 132
Rice University, P.O. Box 1892
Houston, TX 77251, USA
jmg3@rice.edu

Mauricio Araya-Polo
Repsol USA (now at Shell Intl. E&P Inc.)
2455 Technology Forest Blvd
The Woodlands, TX 77381

ABSTRACT

There are four trends in modern high-performance computing (HPC) that have led to an increased need for efficient memory management techniques for heterogeneous systems (such as one fitted with GPUs). First, the average size of datasets for HPC applications is rapidly increasing. Read-only input matrices that used to be on the order of megabytes or low-order gigabytes are growing into the double-digit gigabyte range and beyond. Second, HPC applications are continually required to be more and more accurate. This trend leads to larger working set sizes in memory as the resolution of stored and computed data becomes more fine. Third, no matter how close accelerators are to the CPU, memory address spaces are still incoherent and automated memory management systems are not yet reaching the performance of hand-crafted solutions for HPC applications. Fourth, while the physical memory size of accelerators is growing it fails to grow at the same rate as the working set sizes of applications.

Taking these four trends together leads to the conclusion that future supercomputers will rely heavily on efficient memory management for accelerators to be able to handle future working set sizes, but that new techniques in this field are required.

In this paper we describe, evaluate, and discuss memory management techniques for two common classes of scientific computing applications. The first class is the simpler of the two and assumes that the locations of all memory accesses are known prior to a GPU kernel launch. The second class is characterized by an access pattern that is not predictable before performing the actual computation.

We focus on supporting data sets which do not fit in the physical memory of current GPUs and which are used in applications exhibiting both of these access patterns. Our approach considers GPU global memory as a cache for a large data set stored in system memory. We evaluate the techniques described in this paper on a production (industrial-strength) geophysics application as part of a larger GPU implementation. Our results demonstrate that these techniques flexibly support out-of-core datasets while minimizing overhead, future-proofing the target application against future generations of GPUs and dataset size increases. Our results demonstrate that using these out-of-core memory management techniques results in 80-100% GPU memory utilization while adding 7-13% of overhead. These overheads are offset by the performance improvement from using GPUs and using the memory management techniques described in this paper improves the flexibility of the overall application.

1. INTRODUCTION

GPUs offer a *highly parallel* and *energy efficient* alternative to conventional, commodity processors for high-performance computing platforms. The last decade of research and development into GPU numerical computing has proven that features such as many compute units, high memory bandwidth, specialized memory hierarchy, and low power consumption of GPUs make them a fruitful porting target for scientific applications [5].

However, the majority of that research has been focused on GPU-friendly kernels and datasets where one of the following is true:

1. The working set size of the kernel naturally fits into GPU memory without any extra programmer effort.
2. The working set size and access patterns of the kernel make a straightforward double-buffering approach to memory management feasible.

In this work, we present novel contributions to GPU memory management techniques developed as part of a GPU implementation of a production geophysics application. We evaluate our techniques on real-world geophysical datasets, where neither of the above two characteristics are present. Our contributions include:

1. An approach to managing out-of-core datasets on GPUs for kernels where the memory regions accessed can be computed efficiently prior to a kernel launch. Statically computing memory region accesses simplifies the problem of out-of-core dataset management by allowing the host application to assert that all necessary data is on the GPU before launching a kernel.
2. An approach to managing out-of-core datasets on GPUs for kernels where the memory regions accessed cannot be efficiently computed prior to a kernel launch.

This characteristic complicates the data dependencies of each kernel launch, but it is inherent to many classes of algorithm.

The paper is organized as follows: Section 2 will summarize related work in this area and describe how our work differs. Section 3 will briefly describe the legacy computational kernels we target in this work and relevant aspects of the encompassing application architecture. Section 4 will describe the memory management techniques used during a GPU port of these legacy computational kernels. Section 5 will evaluate the overheads added by these techniques and demonstrate that these overheads do not offset the performance gains made by using GPUs. Section 6 will discuss how the techniques presented in Section 4 are generally applicable and the factors that motivated their design. Section 7 will conclude with a summary of our contributions and results.

2. RELATED WORK

[3] offers the first example of a fully automatic system for managing host and GPU communication. In [3], a compile-time pass is used to transform application code so that stack, heap, and global allocations made by the program can be tracked by a runtime library. Then, the runtime inspects items passed to the kernel and identifies reachable entities that must be transferred to the GPU. However, because this work performs this analysis at the granularity of "allocation units", it does not automatically support out-of-core datasets.

The most common and effective approach to out-of-core datasets is to use tiling of the input space and process each tile separately on the GPU. [7] and [4] are both examples of this technique. However, this approach assumes a number of characteristics about the problem. First, that the input space is the part of the problem that is out-of-core and that tiling across it is a valid transformation. While this is true for many problems (e.g. MapReduce computation as in [7] and stencils as in [4]) it is not true for the application described later in Section 3. Second, this approach assumes that the read-set for a unit of parallel execution is straightforward and efficient to compute. Again, Section 3 will describe an important scientific application where this is not the case.

The work in [2] used an on-demand approach to data communication for out-of-core datasets in a heterogeneous platform that is similar to our second out-of-core memory management approach described in Section 4.0.3. However, the work in [2] does not consider deduplication of shared data on the GPU, multiple host threads sharing the GPU, and differs entirely from our first out-of-core approach, described in Section 4.0.2.

NVIDIA's Unified Memory [1] eliminates the need for GPU programmers to explicitly manage transfers between host and device, using memory faults on each to automatically perform communication on demand. While Unified Memory significantly reduces programmer burden and is a general tool for automatic memory management, it 1) relies on proprietary technology, 2) in our preliminary evaluations introduced too much overhead, and 3) does not support kernel working sets larger than device memory.

In general, past research in GPU memory management has either 1) avoided supporting out-of-core datasets entirely, or 2) focused on applications where the tiling of large datasets is straightforward, making the conversion of an out-of-core dataset to an in-memory dataset straightforward. In the next section, we will briefly describe a relevant geophysical application (heavily used in Oil and Gas exploration) for which these constraints are null to motivate the memory management techniques presented later in Section 4.

3. TARGET APPLICATION AND PROBLEM DESCRIPTION

Kirchhoff Migration (KM)[8] is a widely used subsurface reconstruction technique in hydrocarbon exploration [6]. KM is a two-step algorithm. First, a large three-dimensional matrix of traveltimes to points in the subsurface is calculated using wavefront (ray-based approximation) propagation. This stage is called *traveltime computation* (TT). Then, using seismic traces recorded at different source locations on the surface the actual subsurface structure can be reconstructed based on the traveltimes calculated. This stage is referred as *migration* (MIG).

In this work, we present memory management techniques used during the port of a legacy, production implementation of KM to GPUs. While the legacy implementation is distributed and multi-threaded, we only focus on the relevant characteristics of the main individual kernels. This section focuses on the memory access characteristics of each KM stage (TT and MIG) to motivate the techniques presented in Section 4.

3.1 TT Memory Accesses

The pseudocode for a single TT task is shown in 1. Primarily it consists of a time loop enclosing a pipeline of data-parallel kernels executed across a wavefront of points being propagated. Within this pipeline, several of those kernels read from a single large (GBytes) three-dimensional matrix. This single matrix is generally too large to fit into GPU memory. To complicate matters, the accesses performed by each kernel on this matrix are data dependent, therefore unpredictable. Calculating the memory locations referenced requires running the full computational kernel.

Algorithm 1: Pseudo-code for the main computational kernel of a map microjob in the travel time table computation step of KM

```
for each time step do
    for each point in wavefront do
        kernel1();
    end
    for each point in wavefront do
        kernel2();
    end
    ...
end
```

3.2 MIG Memory Accesses

The MIG kernel operates on "traces", which represent the recorded subsurface response at receivers on the surface from the perturbations introduced. MIG uses several hierarchies of data partitioning to reduce disk I/O. At the highest partitioning level, MIG starts by reading segments of traces from disk. Each trace segment includes many individual traces. Trace segments are sized to fit efficiently in the memory hierarchy of the computing node.

Algorithm 2: Migration step pseudo-code listing

```
while traces remain in microjob do
    segment = read_trace_segment_from_disk()
    while traces remain in segment do
        chunk = ∅
        curr_tables = segment.head().table_deps()
        while segment.head().table_deps() ==
        curr_tables do
            chunk.append(segment.pop())
        end
        for each trace in chunk do
            for each x in trace do
                for each y in trace do
                    for each z in trace do
                        ...
                        read from curr_tables
                        ...
                    end
                end
            end
        end
    end
end
```

Within each trace segment, MIG packages traces together based on the "traveltime tables" those traces access. Each traveltime table is a three-dimensional matrix generally on the order of tens of MBytes but with hundreds or thousands of them per microjob. Traces that relate to the same traveltime tables are processed together to avoid reading the same traveltime table from disk multiple times. As shown in Algorithm 2, each trace chunk is processed using a triply-nested, many-iteration loop that iterates over a physical three-dimensional space (of usually billions of cells). The innermost loop for the z-axis is not parallelizable, but the other two are.

In MIG, the largest consumer of memory is by far the traveltime tables. While each traveltime table is only on the order of tens of MBytes, there are generally hundreds to thousands referred to in a single MIG process. This far exceeds GPU memory capacity. However, the traveltime tables referenced from a single trace chunk are statically determinable. This simplifies the problem of caching the necessary data in GPU memory.

3.3 Summary

This section briefly summarized the memory access characteristics of the two stages of a legacy KM application. As part of the porting process from a multi-threaded CPU implementation to a GPU implementation, it was important to not reduce the flexibility of the application in terms of the datasets that could be handled. The limited size of GPU physical memory made the memory management techniques described in the following section necessary.

4. METHODOLOGY

This section will discuss the different dynamic memory management techniques used in TT and MIG to enable processing of datasets whose working sets do not fit in GPU memory. The approaches for TT and MIG share some common elements, which will be discussed first. Later, the differences in the techniques will be explained.

4.0.1 High Level Memory Management Design

In both TT and MIG, device memory is partitioned into a statically and a dynamically managed memory region. The static region is allocated during microjob initialization and stores small, global objects. After static allocation completes, the remainder of free GPU memory is considered the dynamically managed region and is used as a cache, temporarily storing data from host memory that is needed by running kernels.

While TT and MIG share this high-level strategy, the ways in which they manage the dynamically managed memory region differ based on the requirements of their particular kernels.

4.0.2 TT Memory Management

In TT, the dynamically managed memory region is exclusively used to cache submatrices of a large three-dimensional input matrix, called the velocity model. Given a large velocity model of the dimensions Nx × Ny × Nz, the steps for initializing the dynamically managed memory region are as follows:

1. The velocity model is partitioned in to smaller three-dimensional blocks that can be easily stored in device memory. The dimensions of these blocks is decided statically, and will be referred to as Bx × By × Bz.

2. The velocity matrix (stored as a one dimensional, row major array) is converted to an array of blocks, where each block is also stored as a flattened, row major, single-dimensional array.

3. All free device memory is allocated as a single, one-dimensional array which is logically split in to C cache slots, where C is the amount of free memory divided by the velocity model block size.

4. Host side data structures for managing the dynamically managed memory region are created. This includes a least-recently-used queue of velocity model blocks to help with cache eviction and an array mapping from block IDs to the cache slot on the device each block is currently stored at.

With this initialization, there exists a host-side representation of the velocity model that has been partitioned into blocks which are cache-able on the device (meaning they are small enough that many can be fit in to device memory). Host-side data structures have also been created for managing the GPU-side cache. There is now a large set of on-GPU cache slots to which we can transfer these blocks, making them accessible to GPU kernels.

One open topic remains: how does TT decide which velocity model blocks to transfer to cache slots on the device? An answer to this question is complicated by two factors. First, it is difficult to predict which velocity model blocks will be referenced by a kernel invocation without fully executing its computation. Therefore, kernels must be able to dynamically request that blocks be brought on to the GPU. Second, because there is only one velocity model cache on each GPU but a single GPU may be shared by multiple threads in TT, the transfer of velocity model blocks must be thread-safe. If a kernel is launched on the GPU assuming that some blocks have been placed in the GPU-side cache, that data must remain there until the kernel completes. Those blocks cannot be prematurely evicted by another thread.

The velocity model block cache management policy relies on three device-side data structures:

1. `superblock_requests`: A device bit vector whose length is the number of velocity model blocks created. Setting bit i in `superblock_requests` from a kernel indicates that block i is needed by a GPU kernel to complete. A bit vector is used to minimize the number of bytes that need to be transferred over the PCIe bus.

2. `block_id_to_slot`: A device array indicating if each velocity model block is stored on the device. If block i is stored in cache slot j on the device, element i of `block_id_to_slot` contains the value j. Otherwise, an invalid placeholder is stored.

3. `points_completed`: A device array indicating if the processing of each data point completed successfully. The processing of a point may fail if not all required velocity model blocks are present on the device.

The steps by which a velocity model block ends up in the GPU cache are detailed below:

1. A kernel is launched on the GPU that references the velocity model. At each point in kernel execution where the velocity model is referenced, the kernel marks any referenced blocks in the `superblock_requests` bit vector.

2. After all required blocks are marked, the kernel then uses `block_id_to_slot` to determine if each block being referenced is present on the GPU. If any required blocks are not present the processing of the current point in the wavefront is aborted and this point is marked incomplete in `points_completed`. If all blocks are present, the necessary values are fetched from the GPU cache and processing continues.

3. Once the processing of a point completes successfully, that point is marked complete in `points_completed` before the kernel exits.

4. While the kernel executes, the host is blocked on it. When it completes, the host transfers `superblock_requests` back and examines it for blocks which were required by the kernel but which were not present on the GPU. For each block requested, the host determines the best block to evict from the GPU. The victim block is selected from a pool of blocks comprised of the least recently referenced block for each thread using this GPU. Iterating over these candidate blocks and selecting the one which was the least recently used on average across all threads on this GPU limits the disruption that evicting this block will cause. The selected block is evicted from the cache by removing it from host-side data structures and updating `block_id_to_slot`. The requested block is then copied to the GPU.

5. Once all requested blocks have been transferred to the GPU, the host re-launches the same kernel. Future executions of the same kernel check to see if each point was completed in an earlier attempt, preventing re-execution of points. This process of execute, refresh, re-execute is continued until all points are marked completed. Then, host execution continues.

We store `superblock_requests` in CUDA shared memory and only flush it to GPU memory as each CUDA thread block exits. CUDA shared memory is on-chip and offers much lower access latencies than system/global memory. Using shared memory also decreased conflicting atomic operations when updating `superblock_requests`.

To ensure that the concurrent GPU cache is thread-safe, a read-write lock is used on the host. Any thread trying to alter the contents of the cache must hold the write lock. Any time a kernel is launched that references the GPU cache, the calling thread holds the read lock to prevent other threads from concurrently updating the contents of device memory, ensuring that the device-side mapping from block ID to cache slot is consistent with the actual contents of the GPU cache.

4.0.3 MIG Memory Management

In MIG, multiple types of data share the dynamically managed region of device memory. The main two types of data are:

- Small objects (hundreds of bytes) storing metadata on each trace chunk being processed on the GPU.

- Large (kilobytes or megabytes) read-only traveltime tables that are referenced by traces.

MIG must decide how to partition the dynamic region between them. Executing a trace chunk on the GPU always requires one metadata object, and may require transferring dozens of traveltime tables depending on which traveltime tables are already stored on the GPU. Establishing an accurate ratio of memory used to store each is important to improving GPU utilization for MIG.

To determine an appropriate partitioning of the dynamically managed region between metadata objects and traveltime tables, MIG performs a dry run of the main computational loop to determine 1) how many metadata objects will be transferred in a microjob, and 2) how many bytes of traveltime tables will be transferred to the GPU for those chunks. Using this ratio, MIG statically reserves space in the dynamic memory region for each type of data in the same ratio.

Trace chunk metadata objects on the GPU are simply cached in an appropriately typed array. Caching traveltime tables efficiently on the GPU is complicated by the fact that each table can be a different size. We use the memory reserved for traveltime tables as a single flat array and dynamically allocate variably-sized ranges from it to store TT tables in. The host tracks what device memory isn't currently reserved for a TT table, and stores a mapping from TT table IDs to the device memory addresses they are stored at. Unreserved device memory is tracked on the host using a linked list of free memory segments sorted by base address.

In addition to managing this pool of device memory for TT tables, MIG also pre-allocates a pool of page-locked host memory that is used to perform asynchronous copies from the host to the device. This pool of page-locked memory is managed in the same way as the device-side cache: as a flat array from which dynamically sized ranges are allocated. However, the size of the page-locked pool is smaller than the device-side cache as it only needs to contain tables whose copies that are in-progress. For our experiments we set the page-locked pool to a constant size of 2GB. On memory-constrained systems, this would be reduced.

Managing the TT table cache in MIG is simplified by the fact that the host is able to determine which traveltime tables are referenced by a trace chunk before processing it. Therefore, the execute-refresh-reexecute pattern used for TT is unnecessary. However, there are three preconditions that must be satisfied before a trace chunk can be executed on the device:

1. A trace chunk metadata slot on the device must be available to transfer this trace chunk's metadata in to. Free metadata slots are maintained in a FIFO queue on the host. When the GPU finishes processing a trace chunk, it marks that slot as free by setting a host flag which has been mapped into the device's address space. The host checks for freed metadata slots before processing each trace chunk and adds them to the free queue.

2. Sufficient space in the device table cache must be available to store all traveltime tables needed by this trace chunk. The technique for checking this is shown in Algorithm 3. First, a list of tables required by this trace chunk is constructed. Any tables already cached on the GPU and any duplicates are removed from this list. The remaining tables are sorted from largest to smallest to ensure that the allocations which are hardest to satisfy are attempted first. Then, the algorithm iterates over the uncached tables and attempts to allocate space in the device table cache for each. If all allocations succeed, the host-side reference count for each allocation is set to one and any re-used tables have their reference counts incremented. As each trace chunk completes on the GPU, it will decrement this reference count. When the reference count for a cached table reaches zero its memory is released.

3. Sufficient space must also be available in the pool of page-locked host memory to store the traveltime tables while they are asynchronously transferred to the GPU.

If all allocations succeed, the current trace chunk is grouped with other trace chunks in the current batch. All trace chunks in a single batch will be executed together in a single kernel launch. Unlike the technique used for static scheduling, each of these batches is executed on a single GPU, rather than across multiple GPUs. As a result, Algorithm 3 must only find traveltime table cache slots on the same GPU, which must also be the same GPU as the other trace chunks in the current batch. If sufficient cache slots can only be found on a different GPU, the current batch is executed immediately and a new batch is created with the current trace chunk as its first member. If insufficient cache slots are available across all GPUs, the current batch is still launched immediately but the current trace chunk executes on the host using OpenMP. Failing to find sufficient resources for a trace chunk indicates that all GPUs are subscribe and that we will be unable to process another trace chunk on the GPU for some time. Immediately launching the current trace batch prevents the contained trace chunks from being blocked, and means that the resources they are holding can be freed sooner.

Algorithm 3: Algorithm used to decide if a trace chunk fits on the GPU memory.

```
if Trace Chunk Slot Available then
    Slots = $\emptyset$
    for Table in TablesRequired do
        if Table Already On Device then
            Slots[Table] = Cache[Table]
            TablesRequired.remove(Table)
        end
    end
    RemoveDuplicateTables(TableRequired)
    SortBySize(TableRequired)
    for Table in TableRequired do
        Slots[Table] = AllocateSpace(Table->size)
        if Slots[Table] == NULL then
            ReleaseAllocated(Slots)
            Break
        end
    end
    if All Allocations Successful then
        for Slot in Slots do
            Slot->RefCount += 1
        end
    end
end
```

4.1 Summary

This section covered two distinct approaches to supporting out-of-core datasets on GPUs, motivated by the unique requirements of TT and MIG as described in Section 3. In Section 5 we evaluate the these techniques. In Section 6 we discuss where these techniques are most relevant and how these techniques can be generalized to other applications.

5. RESULTS

This section starts by introducing the baseline performance improvement that resulted from the porting of a legacy distributed and multi-threaded KM implementation to GPUs. We then focus more closely on the overheads associated with each of the out-of-core memory management techniques described in Section 4.

5.1 Experimental Setup

5.1.1 Experimental Platform Description

The tests described below were performed on 1, 2, 4, 8, or 16 nodes. The number of nodes used is limited by resource availability on the production cluster. Table 1 presents the

Table 1: Compute node configuration. Each computing node sports 3 accelerators, and 2 host processors. GDDR5 is a variant of DDR3. Each K10 card sports two GK104 processors

Item	Host Processor	Accelerator
Machine Type	x86_64	GPGPU
Model	Intel Xeon E5-2670	NVIDIA K10
Cores	16	1536x2
Frequency	2.60 GHz	745 MHz
Memory Total	64 GB DDR3	(4x2) GB GDDR5

hardware resources available in each node. Compute nodes are connected by 10 Gbit/s Ethernet and share access to a Panasas parallel filesystem. For our TT experiments we use cache blocks of size 32 *times* 32 *times* 32 cells.

5.1.2 *Input Datasets Description*

The input dataset to the TT stage of the KM pipeline is primarily composed of a single 11.95GB three-dimensional matrix, the velocity model. The output of running TT on this dataset is a single 1.12GB traveltime matrix. The peak memory utilization of a single TT microjob is 18.52 GB.

The input to MIG consists of 24,000,000 traces. The traveltime tables read by each trace range from a few hundred kilobytes to 20 megabytes in size. All datasets hold real data used in hydrocarbon exploration, and some of output datasets were later used in decision making workflows. The peak memory utilization of a single MIG microjob is 73.23 GB.

5.2 Overall Application Performance

Tables 2 and 3 list the overall performance of the legacy and GPU implementations of KM running TT and MIG on a varying number of nodes. MIG achieves higher speedup than TT due to more regular computation that is better suited for GPU execution.

Neither the legacy or GPU implementations show perfect scalability for TT or MIG. The distributed system design consists of a single master node coordinating many worker nodes. The master node acts as a single bottleneck for the whole system, reducing scalability. However, no loss of scalability is observed for the GPU implementation.

The only outlier in these results is the GPU implementation running MIG on 16 nodes, where we observe a speedup of only 5.02x and scalability of only 1.26x relative to an 8-node GPU run. For this test, a single task sits on the critical path of the application, limiting overall speedup.

Table 2: TT Overall performance measurements. Speedup is computed with respect to the legacy implementation.

Nodes	Legacy	GPU	Speedup
1	522,683,687 ms	197,304,873 ms	2.65x
2	262,785,314 ms	97,675,037 ms	2.69x
4	136,127,437 ms	55,244,779 ms	2.46x
8	75,380,638 ms	29,965,100 ms	2.52x
16	42,170,935 ms	17,668,498 ms	2.39x

5.3 TT Performance Analysis

Table 3: MIG Overall performance measurements. Speedup is computed with respect to the legacy implementation.

Nodes	Legacy	GPU	Speedup
1	590,894,312 ms	67,738,228 ms	8.72x
2	294,213,144 ms	32,060,760 ms	9.18x
4	152,519,695 ms	16,381,245 ms	9.31x
8	80,803,623 ms	8,755,180 ms	9.23x
16	34,937,989 ms	6,960,823 ms	5.02x

In the following sections, we will focus on in-depth performance analysis of the dynamic memory management techniques used in TT. Later sections will turn to MIG and perform a similar analysis.

5.3.1 *TT GPU Memory Utilization*

In this section, we look at GPU memory utilization in TT. For the results presented here, a single node run was performed while GPU utilization was sampled every three seconds with *nvidia-smi*.

Figure 1 contains a plot of GPU memory utilization for each GPU in the node, but the individual lines overlap completely. The TT implementation makes all GPU memory allocations at the very start of the microjob and does not free them until the end, keeping memory utilization high throughout.

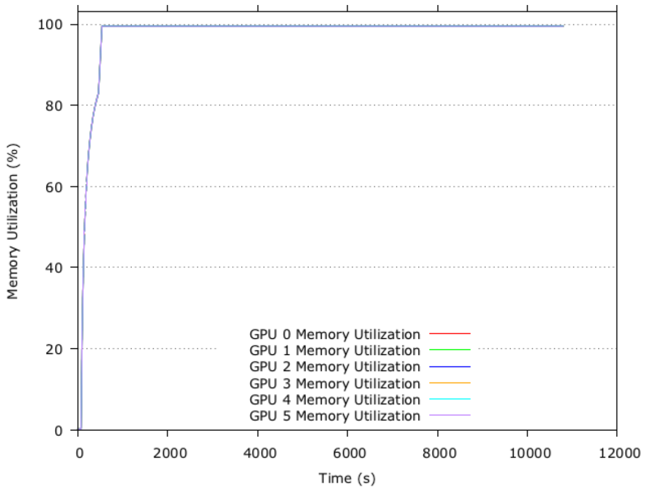

Figure 1: TT GPU Memory Utilization

5.3.2 *Effectiveness of Device-Side Caching Techniques in TT*

To understand how effective the memory management techniques used in TT are, it is important to study two metrics: utilization of the caches and overhead added. We calculate utilization and overhead by parsing diagnostic logs generated.

Figure 2 shows the utilization of cache slots on all six GPUs in a single TT node. We climb to 100% utilization of 5,401 cache slots on each GPU and remain there for ∼85% of the microjob.

We also measure that ∼7.58% of CPU time is spent managing the TT table cache, including CPU cycles spent in management functions and CPU time spent transferring data to the GPU cache over the PCIe bus. This is the result of a large number of cache updates, with an average of 46.57 cache updates and 46.46 evictions per cache refresh and 302,966 cache refreshes in total. We calculate that ∼0.91 ms is spent per cache update. The overhead introduced is small enough that the application still achieves a performance benefit over the legacy implementation.

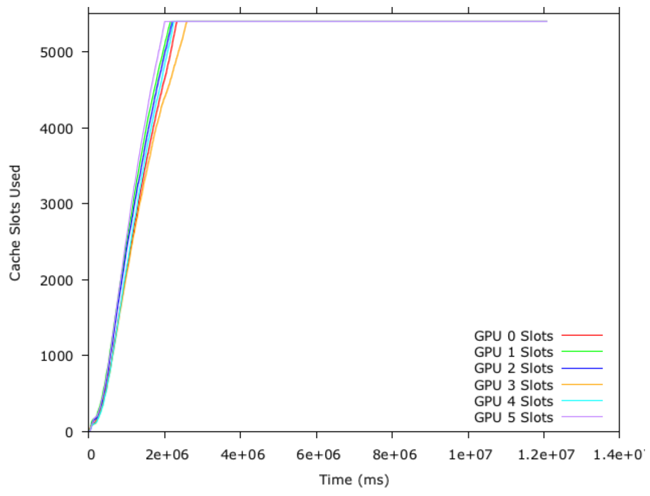

Figure 2: TT Device-Side Cache Utilization

5.4 MIG Performance Analysis

The preceding sections analyzed the performance of TT's memory management techniques. In the following sections, we analyze the performance of MIG's memory management using similar metrics.

5.4.1 *MIG Memory Utilization*

The same methodology was used to study MIG memory utilization as was used for TT in Section 5.3.1.

Figure 3 shows the utilization of GPU memory in a single node run of MIG. MIG GPU memory utilization fluctuates over time much more than in TT. MIG dynamically allocates and deallocates GPU memory more frequently than TT. For the majority of MIG execution GPU memory utilization oscillates between 90% and 100%, indicating that when a workload is applied and the application is not blocked performing 1) I/O, 2) pre-processing, or 3) post-processing, GPU resources are well-utilized.

5.4.2 *Effectiveness of Device-Side Caching Techniques in MIG*

In this section, we study the utilization and overhead of the MIG GPU-side caches.

Figure 4 shows the utilization of table and buffer slots for MIG running on a single GPU. We are able to keep metadata buffer slots well-utilized with an average utilization of 94.39% across the microjob. Table cache usage is erratic, but at its maximum reaches 80.82% utilization. Given that the table cache is partitioned dynamically, even utilization levels of less than 100% may fail allocation requests if a sufficiently large piece of contiguous GPU memory is not available.

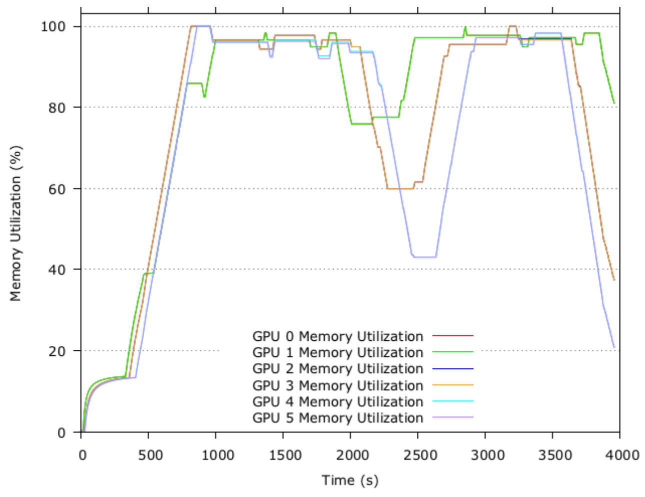

Figure 3: MIG GPU Resource Utilization

Further, log analysis shows that this microjob's ability to execute traces on the GPU is entirely bound by the travel-time table cache. Execution only reverts to the CPU when table memory allocation failed, and never due to metadata buffer slot exhaustion.

We find that MIG spends 13.61% of execution time managing the cache, a much higher percentage than was observed in TT. This is a result of MIG's more complex cache, supporting dynamically sized GPU cache slots. Future work on the MIG cache management should focus on more intelligent ways of creating dynamically sized table slots, with the goal of achieving higher than ∼80% utilization of GPU cache memory and reducing overheads.

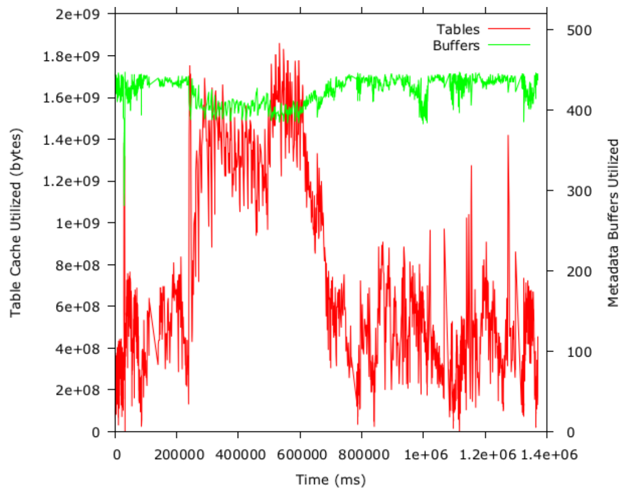

Figure 4: MIG Device-Side Cache Utilization

5.5 Summary

The preceding sections summarized the performance, overhead, and memory utilization of a GPU implementation of KM. We considered overall speedup, demonstrating that the GPU port still outperforms an equivalent and performant legacy KM implementation. We then looked at the memory utilization achieved by each technique to understand if either led to under-utilized hardware. We also evaluated the overhead added by supporting out-of-core datasets. This evaluation shows that while out-of-core dataset support does add significant overhead to our GPU implementation, it is not so significant that it offsets the performance benefit of GPU execution.

6. DISCUSSION

In this paper, we presented two different device memory management techniques that use device memory as a cache for large, out-of-core data structures in host memory.

The MIG caching technique was designed for a kernel where the data access patterns of a single kernel instance are known prior to launching that kernel. However, MIG's caching mechanism must also support caching variably sized matrices. The dynamic device memory management system described in Section 4.0.3 utilizes 80.80% of the dynamically partitioned cache. Dynamically managing GPU memory adds overhead as it must search for a best-fit memory segment for each matrix being transferred to the GPU. 13.61% of total execution time is spent managing this cache (which includes time spent transferring data to the GPUs). Future work will investigate a system that statically partitions device memory into variably sized cache slots, which would remove the overhead of the dynamic memory management system while allowing for a better fit between slot and matrix sizes.

TT used an on-demand caching technique that optimistically executed kernels without the ability to predict if all data dependencies were satisfied on the GPU. This approach is flexible, generally applicable, and adaptable to future code changes. In our experiments, 100% cache utilization was achieved. However, this approach adds overhead from 1) the bookkeeping necessary to detect unsatisfied data dependencies, 2) from blocking execution/transfers, and 3) from retried kernel executions as a result of unsatisfied data dependencies. We measured 7.58% of total execution time in TT being consumed by cache management operations, including transferring data to the GPU. However, this on-demand approach was primarily selected as a robust technique that would remain useful through changes to the algorithms used, as it is generally useful for processing of any out-of-core read-only matrix. Future work will include investigation into more intelligent caching algorithms that reduce the number of data blocks transferred across the PCIe bus, as well as techniques for reducing the amount of blocking transfers and blocking kernel executions necessary in our design. For example, taking a double-buffering approach would allow for computation-communication overlap but would greatly complicate the logic necessary to maintain cache coherency on the GPU.

Looking forward, physically or virtually shared, coherent host-accelerator address spaces are becoming more common (e.g., AMD APUs, CUDA Unified Memory). Porting KM-H to one of these platforms would greatly simplify the MIG and TT caching techniques used in this work. Depending on the platform used, these techniques may even become unnecessary. Future work will compare the performance of these automatic approaches to our manual, workload-tuned implementation.

7. CONCLUSION

In this paper we describe techniques for supporting out-of-core datasets on GPU accelerators, focusing on a single geophysical application containing many kernels that exhibit different memory access characteristics. We demonstrate two techniques for supporting out-of-core datasets based on the different constraints of two different stages of the KM application and evaluate the overhead added by these techniques as well as the memory utilization they achieve. We also discussed how these techniques could be generalized to other applications.

Our results demonstrate that using these out-of-core memory management techniques results in 80-100% GPU memory utilization while adding 7-13% of overhead. These overheads are offset by the performance improvement from using GPUs and using the memory management techniques described in this paper improves the flexibility of the overall application.

As dataset sizes, working set sizes, and application precision requirements all continue to outstrip GPU memory sizes the important of out-of-core memory management techniques on accelerators grows. It is generally accepted that reaching exascale computing will require some form of heterogeneous computing in which accelerators form the computational engine of future computing clusters. If those future computing clusters are to be able to support future application requirements, techniques like the ones presented in this paper will be necessary to ensure that their applicability is not limited to in-memory, regular problems.

Keywords

Distributed, Heterogeneous, GPU, resource management

8. REFERENCES

[1] CUDA Unified Memory. http://devblogs.nvidia.com/parallelforall/unified-memory-in-cuda-6/.

[2] B. Budge, T. Bernardin, J. A. Stuart, S. Sengupta, K. I. Joy, and J. D. Owens. Out-of-core data management for path tracing on hybrid resources. In *Computer Graphics Forum*, volume 28, pages 385–396. Wiley Online Library, 2009.

[3] T. B. Jablin, P. Prabhu, J. A. Jablin, N. P. Johnson, S. R. Beard, and D. I. August. Automatic cpu-gpu communication management and optimization. In *ACM SIGPLAN Notices*, volume 46, pages 142–151, 2011.

[4] G. Jin, T. Endo, and S. Matsuoka. A parallel optimization method for stencil computation on the domain that is bigger than memory capacity of gpus. In *Cluster Computing (CLUSTER), 2013 IEEE International Conference on*, pages 1–8. IEEE, 2013.

[5] V. Kindratenko. *Numerical Computations with GPUs*. Springer, 2014.

[6] J. Panetta et al. Accelerating kirchhoff migration by cpu and gpu cooperation. In *Computer Architecture and HPC. SBAC-PAD '09. 21st International Symposium on*, pages 26–32, Oct 2009.

[7] K. Shirahata, H. Sato, and S. Matsuoka. Out-of-core gpu memory management for mapreduce-based large-scale graph processing. In *Cluster Computing (CLUSTER), 2014 IEEE International Conference on*, pages 221–229. IEEE, 2014.

[8] Wiggins, J. W. Kirchhoff integral extrapolation and migration of nonplanar data. In *Geophysics*, 1984.