Distributed, Heterogeneous Scheduling Techniques Motivated by Production Geophysical Applications

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

When developing applications on distributed, heterogeneous platforms in high-performance computing (HPC) we identify two common options software engineers choose between when designing the parallelism and resource management of their application.

On the one hand, an application can spawn one process per core and statically assign accelerators evenly across those processes. Using separate processes enables isolation from soft errors in other processes, but also makes coordination and resource management between different processes more difficult. This may reduce the efficiency or system utilization of the application.

On the other hand, creating a single process per socket with one thread per core improves coordination, decreases overhead, and makes it possible to dynamically schedule work across multiple accelerators and cores. However, it reduces isolation and may lead to underutilized resources if the granularity of tasks is too fine-grained to use all available hardware.

That is not to say these are the only two choices. Indeed, a full spectrum of options lies between these two extremes. In this paper, we explore the application of various scheduling techniques that lie along this spectrum on distributed, heterogeneous systems. We introduce a novel intra-process load-balancing technique for accelerators named *donation*, which follows a push model. We also describe a greedy host-device task scheduler. We focus our investigation on a two-stage production hydrocarbon exploration application with different scheduling requirements in each stage. Our exploration covers different granularity levels and characteristics of the target application, identifies the benefit of techniques described here, and measures the overall performance improvement. This work results in a 2.69x speedup for the first stage of the target application and a 9.31x speedup for the second stage, relative to an equivalent homogeneous, production implementation.

1. INTRODUCTION

General-purpose computing on GPUs has moved from being a specialized field to one that is both applicable to a wide range of applications[10][6][4][1] and accessible to programmers with a range of backgrounds. In particular, it has gained a strong foothold in the high-performance computing (HPC) community. In HPC, heterogeneous systems have been deployed to successfully accelerate computationally heavy workloads at widely varied granularities, from shared memory machines up to thousands of heterogeneous nodes in a single distributed system.

This increasing prevalence is a result of the performance and energy improvements possible with heterogeneous systems[10], in which specialized architectures trade some generality for other improvements. Adding multiple architectures to a single system increases programming complexity, increases peak power consumption, and increases cost-of-purchase/maintenance. To offset these costs, the performance benefits of porting a legacy application to heterogeneous systems must be significant. Fortunately, expert implementation and optimization of hybrid software can lead to a magnitude of performance improvement, relative to legacy homogeneous implementations. However, as the architectural complexity of shared-memory systems increases, it becomes more challenging to keep all computational resources well utilized.

This paper describes novel scheduling techniques used in the development of an architecturally heterogeneous implementation of Kirchhoff Migration (KM) [12], an important workload in the Oil & Gas exploration. We focus on achieving high utilization of computational resources to gain performance improvement relative to a legacy, distributed, multi-threaded, CPU-only implementation. For the remainder of this paper the legacy KM implementation will be referred to as KM-L. The architecturally heterogeneous implementation developed as part of this work will be referred to as KM-H. The main contributions of this work include:

- Dynamic scheduling of KM computation in a distributed system across multiple architectures in each node.
- Combining task and data parallelism in a single application at multiple levels.
- Techniques in automatic resource detection and management, used to maximize hardware utilization.

Section 2 will summarize related work in this area and describe how our work differs. Section 3 will briefly describe the legacy Kirchhoff Migration implementation that

this work builds on. Section 4 will describe the modifications to the legacy system made as part of this work. Section 5 will present the resulting performance improvements achieved and analyze any remaining bottlenecks. Section 6 will conclude with a summary of our contributions and results.

2. RELATED WORK

One of the primary challenges addressed in this paper is efficient scheduling of unbalanced and complex workloads across a heterogeneous system. While the techniques presented in this paper are used specifically to accelerate Kirchhoff Migration, they are generally applicable. Prior work has investigated other methods of dynamically scheduling work on heterogeneous systems.

In [8] and [2] historical performance data is used to predict future task performance on different architectures. These performance predictions are then used to make scheduling decisions. While this approach is generally applicable to a wide range of computations, it also adds computational overhead at runtime and assumes the programmer has no knowledge of application or hardware characteristics. As this work focuses exclusively on accelerating Kirchhoff Migration, we can take advantage of expert knowledge in the application and architecture domains to guide efficient scheduling decisions for each task across all architectures without the overhead of the approaches used in [8, 2].

Hamano et al [11] consider both the performance of tasks on an accelerator as well as that accelerator's energy consumption when performing scheduling decisions on a heterogeneous system. In this work, we do not optimize our solution for energy efficiency and exclusively focus on performance. However, future work could include using techniques like the ones described in [11] to improve the energy efficiency of our Kirchhoff Migration implementation.

The work in [7] proposes a scheduling algorithm for heterogeneous and distributed systems using message priorities to schedule work on different processors by greedily scheduling high-priority messages immediately on the CPU and buffering low-priority messages for batched execution on the GPU. By building this scheduling algorithm on Charm++, this work supports a unified heterogeneous and distributed programming model. However, as Section 3 will explain, the priority or workload of a task in our application is unknown prior to execution, therefore requiring a more dynamic approach.

3. BACKGROUND

Kirchhoff Migration (KM)[12] is a commonly used subsurface reconstruction technique in geophysical exploration. KM is a two-step algorithm. First, a large three-dimensional matrix of traveltimes to points in the subsurface is calculated using wavefront propagation (through a raytracing approximation). Then, using seismic traces recorded at different source locations on the surface, the actual subsurface structure can be reconstructed based on the traveltimes calculated.

KM is a well studied and optimized application algorithmically [5, 3]. In this work we take a legacy, production implementation of KM and improve performance by adding support for heterogeneous systems. In this section, we briefly describe the architecture of the legacy implementation to provide context for the work described in Section 4.

The original homogeneous application (KM-L) is a multi-node and multi-threaded application with a classic master-worker distributed architecture. Intra-node parallelization is performed using OpenMP.

KM-L is composed of two sub-applications which form a two-stage pipeline. The first implements traveltime calculation, and is referred to as TT. The second takes TT's output as input and migrates seismic traces to construct the final output image, and is referred to as MIG. In this paper, we discuss and accelerate each sub-application individually. Though these applications are packaged as a single executable and share utilities, their computational characteristics are distinct.

At the highest level, a TT or MIG job encapsulates all of the work to be performed to process a given dataset. Each job is decomposed into many microjobs, each of which is scheduled in a single node of a multi-node distributed platform. The work in each microjob is then parallelized across cores using OpenMP.

KM-L runs on two types of nodes: master nodes and worker nodes. Each job uses a single master node and one or more worker nodes which are responsible for the execution of the majority of a job's computation.

In KM-L, each worker node contains two processes: a management process and a compute process. The management process in each worker node is responsible for fetching microjobs from the master node and launching a compute process in that worker node. That compute process then executes the retrieved work. Using two processes makes the system robust against transient faults in the compute process.

Each TT microjob propagates multiple wavefronts through three-dimensional space. Each wavefront is handled by a separate thread. As a rule of thumb there are approximately twice as many wavefronts in each microjob as there are CPU cores in a node. Each wavefront is propagated starting where a perturbation (or "shot") was introduced to the subsurface for thousands of time steps. On each time step, the wavefront (represented by a list of points) is passed through a multi-kernel pipeline. Each kernel in the pipeline iterates over all points in the wavefront. Many of these loops-over-points have no loop-carried dependencies and can be parallelized.

The control flow of a MIG microjob is shown in Algorithm 1. MIG microjobs operate on "traces", which represent the recorded echoes (reflections) at receivers on the surface from perturbations introduced into the ground. Each MIG microjob is given a collection of traces to process. Rather than processing traces one at a time, MIG chunks traces together at multiple granularities to minimize disk I/O.

As shown in Algorithm 1, at the finest granularity of trace chunking a triply-nested, many-iteration loop iterates over a physical three-dimensional space. The innermost loop across the z-axis is not parallelizable, but the outer two are.

4. METHODOLOGY

This section describes the approach taken to maximize hardware utilization and performance of the Kirchhoff Migration application on a heterogeneous platform using dynamic scheduling. The legacy architecture was described

Algorithm 1: Pseudo-code for the migration computation step of Kirchhoff Migration

```
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
                        ...
                    end
                end
            end
        end
    end
end
```

in Section 3. This section focuses on the changes made to the legacy architecture to efficiently execute across all CPU cores and GPUs in a platform.

4.1 TT Device Donation Algorithm

In TT, we continue to rely on the OpenMP runtime to distribute the many TT tasks in a single TT microjob across all of the CPU cores in a single node. We add an inter-thread device assignment algorithm which assigns GPUs to CPU threads. Past work has statically assigned GPUs to CPU threads and moved tasks that are to be run on GPUs to the appropriate CPU thread [9]. However, in TT the workload and internal parallelism of a task can not be known prior to its launch. Therefore, we must both dynamically assign tasks to CPU threads as a means of load balancing across the host, and dynamically associate GPUs with different CPU threads as the workload of their currently executing task changes.

Our approach uses a push-based model that has any thread which owns a GPU but has a light computational workload *donate* its GPU to a thread with a greater computational load. For the remainder of this paper we refer to this algorithm as the device donation algorithm. This approach relies on:

- A compact way of representing the workload on a thread.
- Shared data structures for arbitrating GPU ownership.
- The ability to dynamically transition per-thread state from the CPU to the GPU and back.

The inter-thread device donation algorithm works as follows. Each thread in a microjob which is not currently executing on a GPU is responsible for periodically broadcasting information on its current computational workload to all threads. Threads that currently own a GPU are responsible for periodically checking that their computational workload is greater than that of any thread which does not own a GPU. If a GPU thread finds that a non-GPU thread has surpassed it in computational workload, it donates the device it currently owns to that thread and switches to CPU execution. Non-GPU threads must also periodically check for a donated device, and switch to GPU execution if one is found. Non-GPU threads also check for unclaimed devices, e.g. at the start of microjob execution when no GPUs are owned. This design is non-blocking, avoiding wasted cycles waiting on a resource to become available.

In TT, the workload of a thread can be represented by a single integer: the length of the wavefront being propagated for the current shot. Using this length, each thread can efficiently communicate its computational workload relative to the others.

Because inter-thread communication is required for device donation, inter-thread synchronization must be added. This adds overhead that would not be present if static scheduling were used. By 1) only reading shared state periodically, and 2) setting tolerances to avoid ping-ponging of GPUs between threads with similar workloads, the overhead of dynamic device assignment is minimized. In practice, we found these techniques were sufficient. Future work could investigate using the current pattern of a thread's workload (i.e. increasing or decreasing parallelism) to predict whether acquiring a GPU would be beneficial in the long run.

As part of this work, oversubscribing GPUs with more than one thread was beneficial to overall performance. As this is a real world application with significant time spent in I/O and sequential code regions on the host, over-subscription allows one thread to be using a GPU while other threads assigned to the same GPU perform other operations. We do not implement any custom logic to prevent multiple threads from contending for the GPU at the same time. While a single GPU may be used by multiple threads, each thread can only use one GPU. For the performance evaluation in Section 5 we empirically chose to share each GPU between two threads.

4.2 MIG Microjob Scheduling

In contrast to the scheduling techniques used for TT, the scheduling techniques used for MIG modify both the inter-node microjob scheduling framework and intra-node GPU management.

For MIG, the management process in each worker node was modified to support the execution of multiple microjobs concurrently in a single node. With this change, multiple compute processes running different microjobs can be created in each node by a single worker node management process. Each microjob in a node is now assigned a subset of the GPUs in that node. The number of GPUs assigned to each microjob is configurable in a job parameter file, and defaults to one. For our experiments, we assign two GPUs to each microjob. The number of concurrent microjobs, M, in each node is defined as $M = ceil(G/P)$, where G is the number of GPUs in a node and P is the number of GPUs assigned to each microjob.

Supporting multiple concurrent microjobs per node means that smaller microjobs which may not have sufficient parallelism to occupy the CPU and GPU resources of a whole node now cause less hardware under-utilization. For workloads where the microjobs are already large enough to utilize a full node, we can limit the number of concurrent microjobs using a job parameter to prevent out-of-memory failures. Running multiple microjobs in a single node does increase the fragility of the system to hardware errors: a single error can now affect multiple microjobs.

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

4.3 MIG GPU Management

In addition to the changes to the inter-node microjob scheduler, MIG also changes how work is scheduled across CPU cores and GPUs within a node.

Recall from the discussion of the MIG pseudocode in Algorithm 1 that traces in MIG are grouped at multiple granularities. At the finest granularity, traces are grouped into trace chunks based on the traveltime tables they access.

In this work, we dynamically batch trace chunks together to be processed in a single GPU kernel launch. These larger trace batches increase the amount of parallelism exposed to the GPU. Multiple trace batches can be in-flight on the GPU at once.

The number of trace chunks that can be grouped into a single trace batch is artificially limited to ensure that multiple trace batches are in flight at once, encouraging computation-communication overlap.

The MIG intra-microjob scheduling policy greedily populates the GPU with as many trace batches as device resources will allow. If device resources are exhausted, the host processes a single trace chunk with the legacy OpenMP implementation on a subset of the CPU cores in a node before again trying to create trace batches for the GPU. This strategy keeps GPUs busy by maintaining a deep queue of trace batches to be processed. The host is kept busy either preparing new trace batches for GPU execution or processing trace chunks while waiting for GPU kernels to complete.

5. EXPERIMENTAL RESULTS

This section introduces the performance improvements that resulted from the implementation of the techniques described in Section 4. The measurements below were taken under production conditions, i.e. using real three-dimensional input data sets acquired in exploration fields and processed in a cluster shared with exploration geophysicists, which does not affect the results since multiple repetitions were carried out to discard spurious effects. We analyze the application performance from all relevant perspectives and reason about observed bottlenecks and improvements.

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 hardware resources available in each node. Compute nodes are connected by 10 Gbit/s Ethernet and share access to a Panasas parallel filesystem.

5.1.2 *Input Datasets Description*

Two real-world input datasets are used in the evaluation of this work, one for TT and one for MIG. Dataset A is used to test TT and consists of 20 microjobs, each of which contains 12, 24, or 36 shots depending on the geographic coordinates it processes. Within each shot, wavefronts are limited to a maximum of 200,000 points.

Dataset B is used to test MIG and consists of 96 microjobs, each of which processes 250,000 seismic traces.

5.2 Overall Application Performance

Tables 2 and 3 list the overall performance of KM-L and KM-H running TT and MIG on a varying number of nodes. While KM-H outperforms KM-L for both, MIG sees higher speedup than TT due to more regular computation that is better suited for GPU execution.

The outlier in these results is KM-H running MIG on 16 nodes, where we observe a speedup of only 5.02x relative to KM-L and scalability of only 1.26x relative to 8-node KM-H. The reason behind this is that a single microjob sits on the critical path of the application, limiting overall speedup.

Table 2: TT Overall performance measurements. Speedup is computed with respect to KM-L.

Nodes	KM-L	KM-H	Speedup
1	522,683.687 s	197,304.873 s	2.65x
2	262,785.314 s	97,675.037 s	2.69x
4	136,127.437 s	55,244.779 s	2.46x
8	75,380.638 s	29,965.100 s	2.52x
16	42,170.935 s	17,668.498 s	2.39x

Table 3: MIG Overall performance measurements. Speedup is computed with respect to KM-L.

Nodes	KM-L	KM-H	Speedup
1	590,894.312 s	67,738.228 s	8.72x
2	294,213.144 s	32,060.760 s	9.18x
4	152,519.695 s	16,381.245 s	9.31x
8	80,803.623 s	8,755.180 s	9.23x
16	34,937.989 s	6,960.823 s	5.02x

5.3 TT Performance Analysis

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

5.3.1 *TT Shot Performance*

Shots in TT are the largest unit of work that is executed single-threaded on the CPU. Observing shot performance focuses on computation only, enabling a more direct comparison of the performance of CPU and GPU kernels.

Figure 2 plots the per-shot speedup for each shot in dataset A as a function of the percentage of execution time a shot spends using a GPU. The positive relationship shown in Figure 2 demonstrates that threads which had the benefit of a GPU for a higher percentage of their time steps also achieved a higher speedup. If networking and disk overheads are disregarded, the hybrid implementation of TT achieves better than 3.5x speedup per shot on average. Note that for all

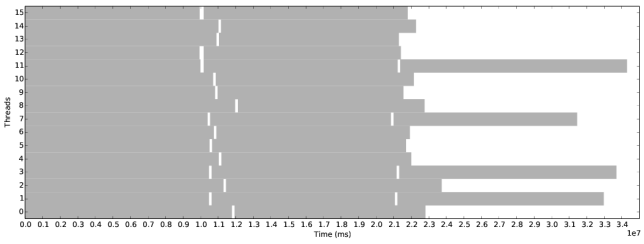
shots some amount of computation is still performed on the CPU because 1) a shot may not own a GPU for the entirety of its execution, and 2) the processing of a shot includes work that cannot be parallelized for GPU execution.

5.3.2 *Benefits of Dynamic Device Assignment*

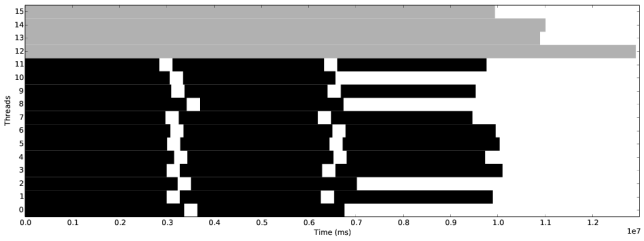
For TT's device donation algorithm, we expect a speedup relative to 1) CPU-only execution, and 2) static assignment of GPUs to CPU threads.

Figure 1 shows sample task schedules in a single microjob running on 1) KM-L, 2) static GPU assignment in KM-H, and 2) the device donation algorithm in KM-H.

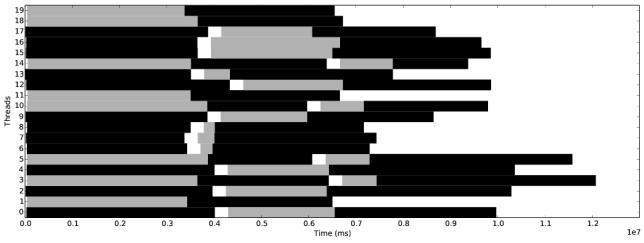
Assigning GPUs to threads statically results in a 2.65x speedup relative to KM-L. This speedup is purely a result of executing computation on the GPU instead of the CPU. Dynamically assigning GPUs to threads through the device donation mechanism improves speedup to 2.84x relative to KM-L. This speedup is primarily a result of the ability to dynamically switch straggler shots to GPU execution, which would not be possible with static GPU assignments.



(a) Scheduling of shots within one TT microjob in KM-L.



(b) Static scheduling of shots on threads and GPUs within one TT microjob in KM-H.



(c) Dynamic scheduling of shots on threads and GPUs within one TT microjob in KM-H.

Figure 1: Gray bars mark CPU shot time, black bars mark GPU shot time, and white space marks idle time between processing shots or after no shots remain to be processed. Note the change in time scale along the x axes (from 3.4ms for a) to 1.3 for b) and c))

5.4 MIG Performance Analysis

The preceding sections analyzed the performance of TT.

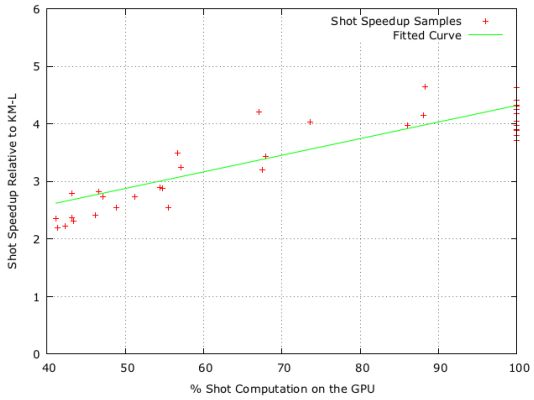


Figure 2: Per-shot speedup as a function of the percentage of shot execution time spent on the GPU

In the following sections, we analyze the performance of MIG using similar techniques.

5.4.1 *MIG Trace Performance*

Similar to shots in TT, traces in MIG are a small but relevant unit of work that we can use to measure the relative performance of KM-L and KM-H without including I/O. Figure 3 shows the rate at which the legacy and hybrid implementations are able to process traces, as well as the gains made by the hybrid implementation. The bimodal distribution of trace rates is caused by different sizes of the main input matrix for each microjob. The earlier microjobs all have an input size of ∼1GB while later microjobs only consume ∼1.5MB. Both implementations' performance is impacted by the input size, but KM-H achieves higher speedup when the input matrix is larger. The reasons behind this are:

1. Large input sizes allow better utilization of the GPU resources both in terms of concurrency and regular memory access.

2. Using the GPUs allows the preprocessing of each trace chunk on the CPU to overlap with work being done on the GPU. This preprocessing consumes more time on larger inputs. KM-L MIG results improve when there is more overlap of preprocessing and GPU computation.

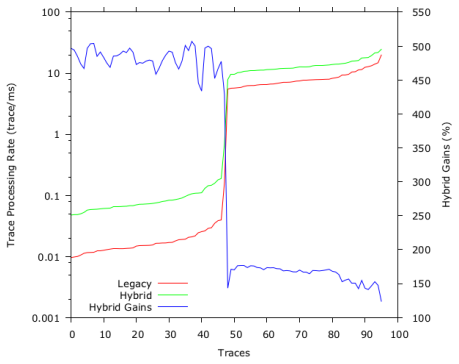


Figure 3: Trace-level performance improvement

Note that the maximum improvement by KM-H relative to KM-L in Figure 3 is ∼5x, but the overall speedup of MIG reported in Table 3 is ∼9x. This discrepancy is the result of executing multiple concurrent microjobs in a single node in KM-H, which increases the parallelism possible at the microjob level given a constant number of nodes.

5.4.2 *Improvement from Hybrid Device Switching*

Log analysis shows that 75.90% of all traces in MIG are processed on GPUs. In this section, we characterize the relative performance of 1) running 100% of the traces on the CPU, 2) running 100% of the traces on the GPU, and 3) dynamically switching between the CPU and GPU. Table 4 lists the processing rates of each of these execution configurations. Hybrid execution with dynamic device switching demonstrates the highest trace processing rate.

Because dynamically switching trace chunk processing between the CPU and GPU adds overhead, selecting a static ratio for the number of trace chunks to be scheduled on each processor could improve performance. However, this would require re-tweaking that ratio for every new dataset and every change to the MIG kernel which would rapidly become unfeasible in a production setting.

Table 4: System-wide trace processing rate for CPU-only, GPU-only, and dynamic hybrid switching

Platform	Trace Processing Rate (traces/ms)
CPU-only	1.442E-3
GPU-only	9.070E-3
Hybrid	10.36E-3

6. CONCLUSION

In this paper we describe the techniques used in the port of a large, production geophysics application to a heterogeneous system. In particular, we emphasize an integrated and hybrid approach in our implementation by considering all system computational resources as part of our porting and optimization plan.

We introduce device-sharing techniques and cooperative execution techniques that focus on effective and efficient system resource utilization. As a result of the application of these techniques to a legacy, distributed, homogeneous Kirchhoff Migration implementation an overall speedup of ∼2.5x was achieved for the TT sub-application, and ∼9x for MIG.

Future work will investigate integrating these techniques into a more general runtime system. The device donation technqiues developed for TT require hints from either the programmer or compiler on the workload of an executing task, as well as a mechanism for transparently moving tasks between processors. Device donation is appropriate for long-running tasks whose internal parallelism changes dynamically. The device switching technique used for MIG requires the ability to aggregate multiple tasks together for GPU execution as well as an estimation of task completion time on the GPU so that an appropriate amount of work can be overlapped with it on the host. Device switching is more useful for fine-grain tasks that must be batched together to achieve high GPU utilization.

With the focus on dynamic, runtime, adaptive techniques this implementation is robust to changes to its code, datasets, and platforms. Furthermore, this work demonstrated significant performance improvement over a large, production-ready, legacy implementation without a loss in generality. Using a hybrid, dynamic approach results in higher system computational throughput than existing static or homogeneous techniques.

Keywords

Distributed, Heterogeneous, GPU, resource management

7. REFERENCES

[1] Catanzaro, Bryan, Narayanan Sundaram, and Kurt Keutzer. Fast support vector machine training and classification on graphics processors. In *Proceedings of the 25th international conference on Machine learning*, 2008.

[2] Chi-Keung Luk, et al. Qilin: Exploiting Parallelism on Heterogeneous Multiprocessors with Adaptive Mapping. In *IEEE/ACM International Symposium on Microarchitecture*, 2009.

[3] Docherty, Paul. A brief comparison of some Kirchhoff integral formulas for migration and inversion. In *Geophysics 56.8 pgs. 1164-1169*, 1991.

[4] Fan, Zhe, et al. GPU cluster for high performance computing. In *Proceedings of the 2004 ACM/IEEE conference on Supercomputing*, 2004.

[5] Gray, Samuel H. Efficient traveltime calculations for Kirchhoff migration. In *Geophysics 51.8 pgs. 1685-1688*, 1986.

[6] KrÃijger, Jens, and RÃijdiger Westermann. Linear algebra operators for GPU implementation of numerical algorithms. In *ACM Transactions on Graphics (TOG)*, 2003.

[7] J. Lifflander, G. Evans, A. Arya, and L. Kale. Dynamic Scheduling for Work Agglomeration on Heterogeneous Clusters. In *Parallel and Distributed Processing Symposium Workshops PhD Forum (IPDPSW), 2012 IEEE 26th International*, pages 2404–2413, 2012.

[8] Max Grossman, et al. HadoopCL2: Motivating the Design of a Distributed, Heterogeneous Programming System With Machine-Learning Applications. In *IEEE Transactions on Parallel and Distributed Systems*, 2015.

[9] A. Sbîrlea, Y. Zou, Z. Budimlíc, J. Cong, and V. Sarkar. Mapping a data-flow programming model onto heterogeneous platforms. In *ACM SIGPLAN Notices*, volume 47, pages 61–70. ACM, 2012.

[10] Tian, Xiang, and Khaled Benkrid. High-performance quasi-monte carlo financial simulation: FPGA vs. GPP vs. GPU. In *ACM Transactions on Reconfigurable Technology and Systems (TRETS)*, 2010.

[11] Tomoaki Hamano, et al. Power-Aware Dynamic Task Scheduling for Heterogeneous Accelerated Clusters. In *IEEE International Symposium on Parallel and Distributed Processing*, 2009.

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