GEMTC: **G**PU **E**nabled **M**any-**T**ask **C**omputing

Scott Krieder, Ioan Raicu
Department of Computer Science
Illinois Institute of Technology
Chicago, IL USA
skrieder@iit.edu, iraicu@cs.iit.edu

Abstract— **Current software and hardware limitations prevent Many-Task Computing (MTC) workloads from leveraging hardware accelerators (NVIDIA GPUs, Intel Xeon Phi) boasting Many-Core Computing architectures. Some broad application classes that fit the MTC paradigm are workflows, MapReduce, high-throughput computing, and a subset of high-performance computing. MTC emphasizes using many computing resources over short periods of time to accomplish many computational tasks (i.e. including both dependent and independent tasks), where the primary metrics are measured in seconds. MTC has already proven successful in Grid Computing and Supercomputing on MIMD architectures, but the SIMD architectures of today's accelerators pose many challenges in the efficient support of MTC workloads on accelerators. This work aims to address the programmability gap between MTC and accelerators, through an innovative middleware that enables MIMD programmability of SIMD architectures. This work will enable a broader class of applications to leverage the growing number of accelerated high-end computing systems.**

Index Terms— **GPGPU, MTC, CUDA, Swift, Accelerator, Workflows**

I. Introduction

This research involves pursuing the integration between data- flow driven parallel programming systems (e.g. Many-Task Computing - MTC)[1] and hardware accelerators (e.g. NVIDIA GPUs, AMD GPUs, and the Intel Xeon Phi). MTC aims to bridge the gap between two computing paradigms, high throughput computing (HTC) and high-performance computing (HPC). MTC emphasizes using many computing resources over short periods of time to accomplish many computational tasks (i.e. including both dependent and independent tasks), where the primary metrics are measured in seconds. Swift[2] is a particular implementation of the MTC paradigm, and is a parallel programming system that has been successfully used in many large-scale computing applications. The scientific community has adopted Swift as a great way to increase productivity in running complex applications via a dataflow driven programming model, which intrinsically allows implicit parallelism to be harnessed based on data access patterns and dependencies. Swift is a parallel programming system that fits the MTC model, and has been shown to run well on tens of thousands of nodes with task graphs in the range of hundreds of thousands of tasks.[2] This work aims to enable Swift to efficiently use accelerators (such as NVIDIA GPUs, AMD GPUs, and the Intel Xeon Phi) to further accelerate a wide range of applications, on a growing portion of high-end systems. GPUs are one of the most effective ways to provide acceleration on HPC resources. However, a programmability gap still exists between applications and accelerators. Researchers and developers are forced to work within the constraints of closed environments such as the CUDA GPU Programming Framework[3] (for NVIDIA GPUs). The goal of this work is to improve the performance of MTC workloads running on GPUs through the use of the GEMTC framework. The CUDA framework can only support 16 kernels running concurrently, one kernel per streaming multiprocessor (SM). One problem with this approach is that all kernels must start and end at the same time, causing extreme inefficiencies in heterogeneous workloads. By working at the warp level, (which sits between cores and SMs) I can trade local memory for concurrency, and I am able to run up to 200 concurrent kernels. Our middleware allows independent kernels to be launched and managed on many-core architectures that traditionally only supported SIMD. Our preliminary results in the costs associated with managing and launching concurrent kernels on NVIDIA Kepler GPUs show that our framework is able to achieve a higher level of efficiency for the MTC workloads I tested. I expect results to be applicable to the many HPC resources where GPUs are now common. Finally, I plan to explore applications from different domains such as medicine, economics, astronomy, bioinformatics, physics, and many more. I will continue to push the performance envelope by enabling many MTC applications and systems to leverage the growing number of accelerated high-end computing systems. I also expect this work to enable other classes of applications to leverage accelerators, such as MapReduce and ensemble MPI. I also hope to influence future accelerator architectures by highlighting the need for hardware support for MIMD workloads.

This work contains the following contributions:

1) I present GEMTC, a framework for enabling MTC workloads to run efficiently on NVIDIA GPUs.

2) The GEMTC Framework improves the programmability model of NVIDIA GPUs by more closely representing the MIMD model.

3) Swift/T integration provides increased programmability, efficient scaling, and future support for real applications.

This paper is organized as follows: Section 2 introduces related work and how the GEMTC framework differs and improves upon these works. In an effort to make this paper as self-contained as possible Section 3 provides the necessary background information for understanding the terms and technologies referenced in this paper. Section 4 introduces GEMTC, and the multiple components that make up the framework. Section 5 presents the evaluation method and results. Section 6 highlights future work directions, and Section 7 concludes the paper.

II. Related Work

The goal of this work is to provide support for MTC workloads on Accelerators. By providing this support I intend to improve the programmability of these devices and counter natural hardware drawbacks in the process. Currently, this work is unique due to its consideration of MTC workloads on accelerators. However, there are many related works regarding the improved programmability of Accelerators. This section categorizes related works in two ways 1) Frameworks and 2) Virtualization. First, frameworks are analyzed that aim to enable improved programmability of accelerators for workloads that closely match MTC. Next, the state of virtualization on accelerators is discussed as a means of providing support for improved accelerator performance.

A. GPGPU Frameworks

CrystalGPU[4] is a project for harnessing GPU power and the highest efficiencies available. CrystalGPU has similar goals to our project, improving programmability of GPUs, SMs are treated as workers and the level of control is not as low level as within the GeMTC framework.

Grophecy[5] attempts to improve the ease of use of GPU programming through code skeletonization. Grophecy can analyze CPU code and determine if it will achieve speedup if ported to GPU code saving development time.

StarPU is a task-programming library for hybrid architectures from Inria.

B. Virtualization

Pegasus[6] aims to improve GPU utilization through virtualization. The Pegasus project runs at the hypervisor level and promotes GPU sharing across virtual machines. The Pegasus project also includes its own custom DomA scheduler for GPU task scheduling.

Ravi presents a framework to enable GPU sharing amongst GPUs in the cloud.[7] By computing the affinity score they are able to determine which applications can benefit from consolidation.

III. Background Information

This section aims to provide the necessary background information to make this paper as self-contained as possible.

A. Many-Task Computing (MTC)

Many-task Computing (MTC)[1] is a programming paradigm that aims to bridge the gap between HPC and HTC. MTC focuses on running many tasks over a short period of time. Where tasks can be either dependent or independent, and are organized as Directed Acyclical Graphs (DAG)s. The primary metrics of these tasks are measures in seconds. There are several projects capable of supporting MTC workloads including Condor, Falkon[8], Swift[2, 9], Jets[10], Coasters[11], MapReduce, Hadoop, Boinc, and Cobalt.

B. Swift and the dataflow model

One solution that makes parallel programming implicit rather than explicit is the dataflow model. Conceived roughly 35 years ago, the model comes and goes in and out of "vogue" as researchers try again to make it useful. I believe Swift successfully encompasses "implicitly parallel functional dataflow." The Swift parallel scripting language has proven increasingly useful in scientific programming. This dataflow programming model has been characterized to be a perfect fit in the Many-Task Computing (MTC) paradigm. Swift is a parallel programming framework that has been widely adopted by many scientific applications.[1, 2, 8, 9, 12-14] The original implementation of Swift, Swift/K, had limitations in regards to scalability. Swift/T is a redesigned implementation of Swift and improves upon these limitations greatly while proving support for an arbitrary number of nodes. The GEMTC Framework interacts with Swift/T through a C API, which allows Swift script to call C function wrappers to CUDA code precompiled into the GEMTC framework before runtime.

C. Acceleration: GPUs and Coprocessors

General Purpose Computation on Graphics Processing Units (GPGPU)[15] is a great source of acceleration on high performance resources. Acceleration is achieved by offloading computations from the CPU host to a connected accelerator, which relieves the host of cycles. While acceleration can provide many benefits, in some cases the overheads associated with acceleration that may provide significant drawbacks, which are addressed in this work. Accelerators are now commonplace in many machines within the TOP500 and the Green500 due to both their performance and power advantages.

CUDA, OpenCL, and OpenACC are some of the most interesting ways to program GPUs for HPC. However, these languages are specific to accelerators and do not provide a high a enough level of abstraction.

While the Intel Xeon Phi coprocessor is still a relatively new product, the Texas Advanced Computing Center (TACC) recently launched Stampede, a resource built around the Xeon Phi coprocessor. The Xeon Phi coprocessor is an accelerator that supports several different programming models, including an offloading library which makes it appear similar to a GPU, but it also supports a native mode that allows for very interesting OpenMP programmability and avoids some common GPGPU drawbacks.

D. GPGPU Drawbacks

While GPUs are certainly an efficient source of acceleration, there are still drawbacks of accelerating with GPGPU technology. This work also aims to improve upon these shortcomings within the GEMTC framework.

1) Data Transfers

The time to transfer data is limited by the speed of the PCIe bus, these overheads become significant for data intensive tasks. The GEMTC framework attempts to coalesce data transfers and overlap data transfers with the execution of other tasks in attempts to mask data transfer times and latencies.

2) Architecture

GPUs are traditionally considered to be Single Input Multiple Data(SIMD) devices. The GEMTC Framework helps to facilitate a shift towards viewing GPUs as Multiple Input Multiple Data(MIMD) devices. There are many reasons why it would be beneficial to consider GPUs as MIMD. For example, viewing the GPU as MIMD allows access to the GPU as a small cluster. Through the use of the GEMTC framework, the GPU is represented as a MIMD collection of SIMD workers. Each worker runs on a single warp, the tightest level of control available on the GPU, which is an accumulation of 32 threads.

3) Programmability

GPU programming requires the use of specific languages and frameworks such as CUDA , OpenCL, or OpenACC. By making key GPU functionality available to parallel programming frameworks and workflow systems, such as Swift, I aim to improve the programmability and scalability of accelerators.

4) Dynamic Memory Management

The default CUDA programming framework was not designed for dynamic memory management. Our experiments show that the time required for a device allocation call grows in proportion to the number of device allocations already performed. Currently, CUDA applications generally allocate all memory requirements at the beginning of their lifetime to avoid this drop in performance. GEMTC incorporates a sub-allocator that allows the host to allocate device memory at a much faster rate than possible under the default CUDA framework.

IV. GEMTC DESIGN

This section discusses the approaches evaluated for MTC workloads on accelerators and introduces the GEMTC framework. This work evaluates 3 approaches 1) An initial naïve approach 2) A batch middleware 3) The GEMTC Framework.

A. NATIVE BLACK-BOX APPROACH

To determine how the CUDA GPU Programming Framework would handle MTC workloads natively. I interacted with the GPU in a black box fashion, and launched MTC workloads with a simple scripting middleware and the command line interface.

At this level, integration was done with Swift/K[2] for launching tasks. Swift scripts launched each task to the GPU as a separate application in parallel.

The problem with this approach is that the GPU will only execute device kernels from a single process at any given time. So for the purpose of MTC, the many small tasks will have very poor utilization. Tasks that only use a fraction of the device are unable to run in parallel. There is also 100msec of overhead in launching each individual GPU application. This resulted in extremely poor efficiency for MTC workloads, which emphasize many smaller tasks.

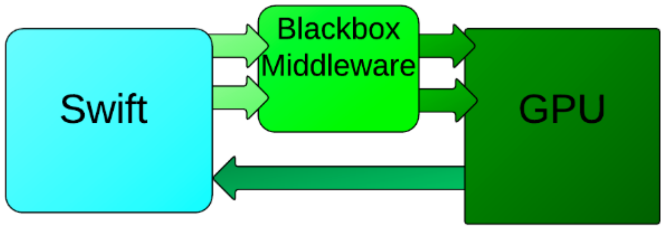

Figure 1 - The flow of a native black box middleware using CUDA.

With this native approach, the entire GPU is treated as a single worker. The above diagram demonstrates the drawbacks of launching single applications on the GPU.

Because the GPU does not allow parallel execution of multiple applications, I decided to create a single application which would batch together tasks to be executed on the GPU.

B. CPU BASED BATCH SCHEDULER

Due to the inefficiencies in the black box approach, I developed a CPU based scheduler that would consolidate all the tasks into batches to then run through a single application on the GPU. The basic unit of offloading work to the GPU is a kernel. This approach will launch a kernel for every task that is handed to it by a workflow system.

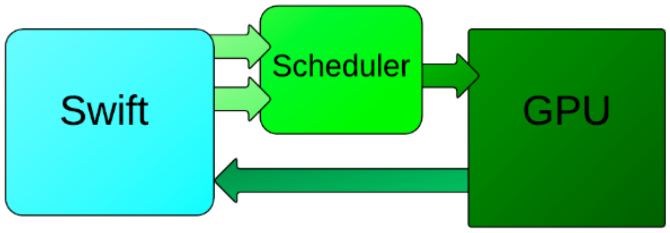

Figure 2 - The flow of a CPU based middleware that implements batching.

The CPU based middleware is capable of launching multiple kernels concurrently to the GPU. It is also able to overlap the memory transfer of tasks with the execution of others.

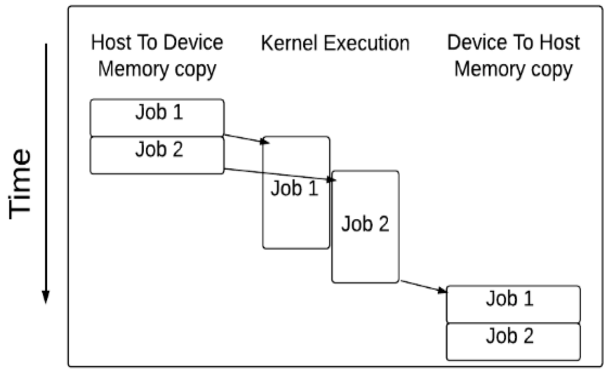

Figure 3 - Overlapping incoming data transfers with kernel execution.

One limitation of the batch approach is the limitation on the number of concurrent kernels that are capable of executing in parallel. Even the highest end devices are limited to 16 concurrent kernels. While NVIDIA has announced that number to be increased to 32 with Kepler, this still will not achieve the tightest level of control available on the device. In addition to the limitation on concurrency, this approach is limited by an age-old batching problem. The shortest running task cannot return a result until the longest running task has completed, which means that the results from any of the concurrent kernels will have to wait until the last task in that batch completes. Since a group or tasks must wait for the longest task to complete, a heterogeneous workflow will likely have poor utilization.

Due to these limitations on the concurrency of kernels, I decided that a much higher level granularity and utilization could be achieved if the scheduler was integrated more closely to the GPU. This motivated the development of GEMTC, which works at a per task basis allowing almost 200 independent workers to run heterogeneous workflows with high utilization.

C. GPU Based Scheduler

The GPU based scheduler runs MTC workloads with a much higher level of efficiency compared to the previous approaches.

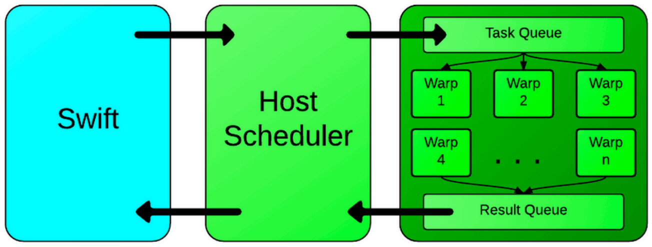

Figure 4 - Flow of a task in GEMTC.

In this approach a single kernel executes on the GPU, which I refer to as the SuperKernel. This is the GPU application that is seen by CUDA and the GPU. This kernel and the CPU component of the framework communicate through shared memory on the GPU. The host will copy task descriptions into a queue on the device, and then workers on the device will pick up that task and execute the work associated with that task. Finally, the task description now including the result is written into a result queue on the device, where the host thread will read it back to the CPU. I now introduce the major components included in the GEMTC framework.

1) *Task Descriptions*

Task descriptions are used to encode key information regarding the MTC workloads on the host and device. These are initially copied into an incoming work queue on the device, and tell the running SuperKernel to execute the task. After the work is completed then this particular *TaskDescription* is then placed on the results queue to inform the CPU that the task has completed. A task description is made up of the following 3 subcomponents and represented as a struct in the source code 1) A *TaskID*, a unique identifier for a particular task which distinguishes it from all other tasks. 2) The *TaskType*, is an integer value that refers to a particular pre-compiled micro-kernel. All micro-kernels have a numerical number mapped to them and this indicates which precompiled micro kernel the task should execute. 3) *numThreads*, is the number of threads/workers the task will require. Threads are assigned in groups of 32, due to a hard limit on what the architecture can allow. At the lowest level, these groups of 32 threads execute in a SIMD fashion. For example, if a task requests 16 threads then 32 will be assigned, but half will be idle. Similarly, requesting 65 threads will result in 96 threads being assigned, but 31 idle threads. 4) *params*, a void pointer to device memory for the parameters of a given task. This includes both the input to a particular micro-kernel and it's resulting output.

2) *Worker Granularity*

A major advantage to the GEMTC framework is that it allows the execution of tasks at the warp level which is the lowest possible level of control on the device. Under the default CUDA framework, applications are normally programmed at the block level, which is a logical level similar to running at the physical SM level.

The transition to executing at a warp level allows for a much higher granularity within tasks, without hindering the performance of tasks which might need large amounts of the device to execute. Furthermore, by viewing the device as a collection of separate warps executing separate operations, the device can be treated very similar to a MIMD device, except that the basic computational unit is a 32 core SIMD processor rather than a single core. This fundamental shift will enable systems like Swift/T to more easily and efficiently assign work to the device.

3) *Memory Transfers*

Due to the fact that the communication between the host component of the framework and the SuperKernel is through shared memory, it is critical to have fast memory transfer times for small amounts of data. It requires three memory copies to en-queue a single task (one to read the queue metadata, one to write the task description, and one to update the queue metadata). Similarly it takes three memory copies to de-queue a task from the resulting queue.

To evaluate the cost of memory copies on the GPU, I ran a series of benchmarks copying to and from the device, including both asynchronous and synchronous calls. Figure 5 shows that memory copies to the device take 2μsec and copies from the device take 10μsec.

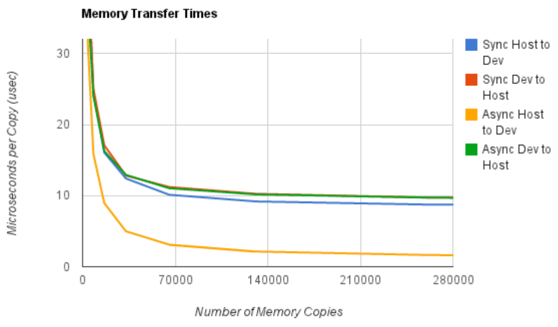

Figure 5 - Average time to copy data to and from the device as either an asynchronous or synchronous call.

These results are significantly faster than the required time for a cudaMalloc() and cudaFree() and is the motivation for having a sub-allocator within the framework.

D. Dynamic Memory Management

The GEMTC framework also requires efficient device memory allocation. Each task that is en-queued requires one device allocation and then the task itself may need to allocate memory for its own parameters and results.

The existing CUDA memory management system was not designed for dynamic memory management. This resulted in poor performance of workloads that required a large number of device allocations. To evaluate the existing memory management, measurements were taken for the average time to cudaMalloc() a small amount of memory repeatedly and to cudaMalloc() and free a small amount of memory, these results are shown in figure 6.

In the best case, a cudaMalloc() and cudaFree() takes more than 100µsec to execute. The GeMTC framework must allocate memory on a per task basis. The overhead of using this memory management is substantial compared to other overheads (e.g. memory copies). To reduce the cost of cudaMalloc() on device memory, the GEMTC framework has its own sub-allocator designed to efficiently handle many requests for dynamic memory allocation. GEMTC uses cudaMalloc() to allocate large contiguous pieces of device memory. The pointers to these free chunks and their sizes are stored in a linked list on the CPU.

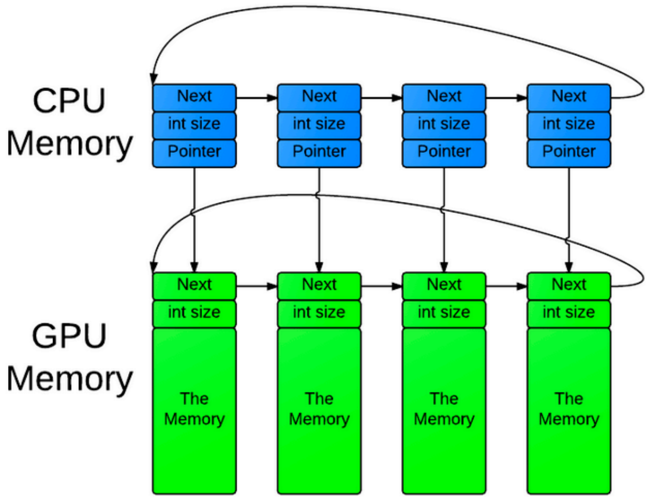

Figure 6 - A memory mapping of free memory available to the device.

Upon memory allocation requests, the sub-allocator will find a large enough chuck of free device memory in its list, or request more using cudaMalloc(). Next, it will write a header to the device memory indicating its size immediately before the free chuck. This operation takes roughly the same time as a memory copy to device.

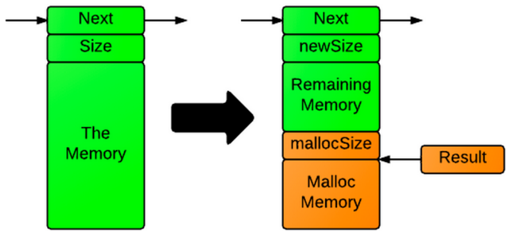

Figure 7 - The effects of gemtcMalloc() on free memory.

When device memory is freed, a header is read to identify the size of memory and adds it into the list of free memory. Freeing device memory takes roughly the same amount of time as a memory copy from the device.

To evaluate GEMTC's memory management, I measured the average time to malloc a small amount of memory repeatedly and to malloc and free some memory. Shown in figure 7. This sub-allocator scales very well and has an execution time on the same order of magnitude as a memory transfer to/from the device.

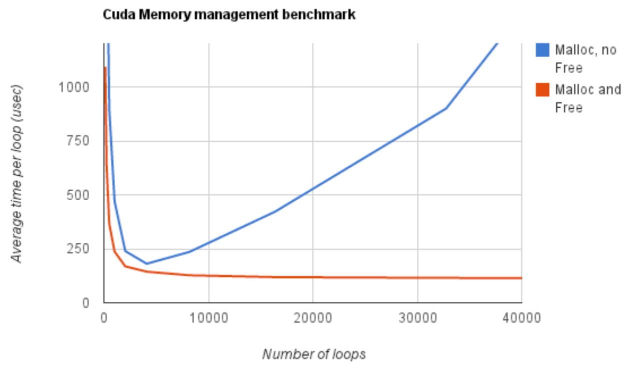

Figure 8 - CUDA memory management execution times.

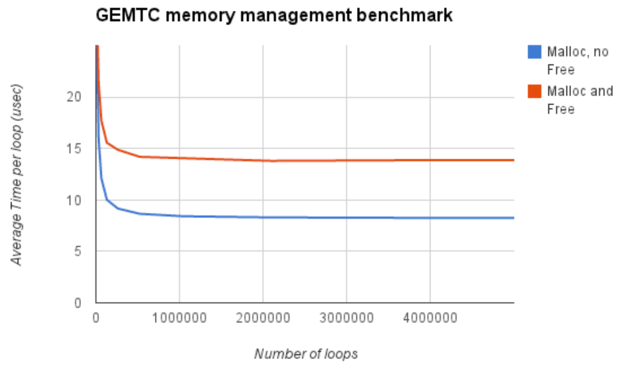

Figure 9 - Average GEMTC memory management execution times.

E. GEMTC Architecture

The SuperKernel is the single kernel that runs on the entire device and will execute all tasks. It is composed of many warps (32 thread SIMD groups) that execute independently. All computations are done by the warps that make up the

SuperKernel. Warps will busy wait on the incoming task queue when no tasks are available to execute.

All tasks descriptions are enqueued in a queue located in device memory. All of the SuperKernels free warps will wait on this queue for a task to be enqueued. When a task arrives, the requested number of warps (obtained from the *numThreads*) will dequeue and execute the task. Once it finishes, the warps will enqueue the task description into a finished work queue in device memory. A thread on the CPU, which manages the results, reads this queue.

Micro-kernels are the pre-compiled CUDA codes that tasks request to run. Micro-kernels are designed to run at the warp level on the GPU. The *TaskType* within a *TaskDescription* specifies a particular numerical value which maps to a certain micro-kernel that the worker needs to execute for that task.

V. Evaluation and Results

The framework is evaluated with multiple micro-kernels, where micro-kernels serve as MTC applications that are pre-compiled into the GEMTC framework. The micro-kernels that available within GEMTC are shown in the chart below. Note that while final results are not shown for all of the following micro-kernels in this paper, preliminary results have already tested all of the following micro-kernels.

#	Micro-Kernel	Parameters	Demonstrates
1	Sleep(t)	Time to sleep.	Mass parallelism
2	Matrix-Square(m)	Matrix	Compute and Data Transfer
3	FFT()	Multidimensional arrays	Compute and Data Transfer
4	Array Min/Max/Ave.	Array	Compute and Data Transfer
5	Matrix Inversion	Matrix	Compute and Data Transfer
6	Matrix Transpose	Matrix	Compute and Data Transfer
7	Matrix Multiply	Matrix	Compute and Data Transfer
8	Matrix Vector	Matrix	Compute and Data Transfer
9	Linear Algebra Solver (Gauss Jordan)	Data Matrices and Vectors	Advanced mathematical manipulation
10	Vector Add	(Vector1, Vector2)	Compute and Data Transfer
11	Vector Dot	Vector	Compute and Data Transfer
12	Stencil Algorithm	Data Stencil	Advanced mathematical manipulation
13	Black Scholes	Economic modeling data	Economics manipulation
14	PI calc. (Monte Carlo Method)	Integer values	Advanced mathematical manipulation

Figure 10 - Chart of available Micro-Kernels within GEMTC.

A. Environment

This paper evaluates the sleep micro-kernel currently within the GEMTC framework on the following environment:

- A single node workstation with a 6 core AMD 3.0 GHz CPU, 8 GB of DDR3 RAM, and a NVIDIA GTX-670 GPU.

B. Micro-Kernel Benchmarking

1) Sleep

This is a micro-kernel that executes for a given number of microseconds. It serves to provide benchmarking data on the efficiency that the framework can achieve with workloads of different length tasks. It sleeps with a busy wait on all of the threads executing within a warp worker. The wait consists of a number of additions based on the sleep time and the number of additions the specific GPU hardware can do per unit time. The latter value was found experimentally by testing GPU hardware.

The results from executing sleep tasks indicate that there is minimal overhead in launching a single task. With a single warp executing, high efficiency is achieved for tasks that execute >80μsec (See figure 9). When running the shortest tasks (10 μsec), the average time per task is 63μsec. This indicates that the framework has approximately this much overhead, which is roughly the same amount of time it takes to do all the memory copies involved in each tasks en-queuing and de-queuing. These results indicate that the framework is able to handle 15,000 tasks per second internally. After an externally available API is placed on top of the framework this number levels out at 7,000 tasks per second.

Similarly, with the entire GPU working to execute sleep tasks, the framework can achieve high efficiency for tasks of >5000μsec (See figure 10). This result is justified by the need for enough tasks to keep all warps busy (i.e. Task Duration >Number of warps*Overhead per task).

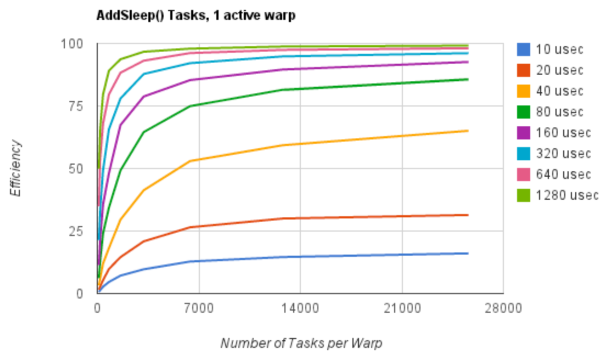

Figure 11 - Efficiency of sleep task workloads with 1 warp executing all tasks.

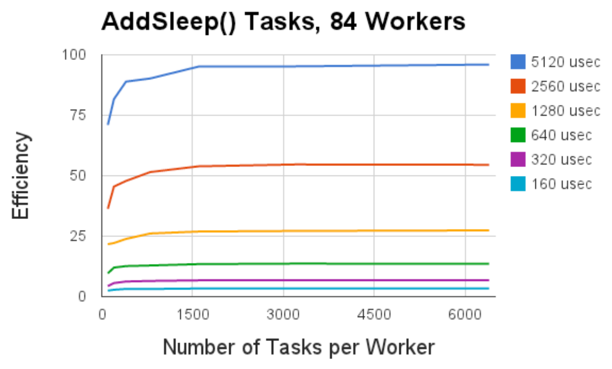

Figure 12 - Efficiency of sleep task workloads with many warps executing.

VI. FUTURE WORK

A. Alternative Accelerator Support

While NVIDIA GPUs make up a large presence of accelerators in HPC resources, there are many different types of accelerators. Other accelerators include AMD based GPUs and the Intel Xeon Phi coprocessor. Intel recently launched the Xeon Phi hardware as a response to the recent drive towards GPGPU computing in HPC. The Xeon Phi is a PCIe coprocessor accelerator, which physically appears similar to a GPU but is quite unique due to the programming model. Ongoing work is currently evaluating how MTC workloads are handled on the Xeon Phi through both the offloading libraries and the native programmability.

B. Real Application Support

This paper demonstrated the potential of the GEMTC framework through the use of micro-kernels. However, the importance of supporting real applications is a strong motivator. Ongoing work is currently evaluating the following two real science applications.

A chemistry glass material modeling application: Glen Hocky of the Reichman group at Columbia is currently evaluating a glass material modeling application.[16] This application has already been programmed to run in Swift. The computationally intensive portions have been identified and ongoing work is integrating GEMTC calls for improved performance.

The Open Protein Simulator (OOPS) builds on the Protein Library (PL). Currently, the OOPS application makes use of an array of python scripts running in parallel. Ongoing work is developing a swift version of the application and identifying computationally intensive portions of the code that could be integrated with GEMTC.

C. Evaluating MTC on Simulators

Evaluating MTC workloads on real systems is critical, but it is also important to evaluate simulation as another means of further understanding MTC workloads.[17, 18] Analyzing MTC workloads through GEMTC on GPU simulators such as GPGPU-SIM[19] allows the ability to test the framework under varying system configurations.

D. Many Node/GPU Evaluation

Through a collaboration with the University of Chicago I have gained access to a 20 node cluster "Breadboard" and a 28 node Cray Supercomputer "Raven." Ongoing work is evaluating the GEMTC framework on both of these resources. I am also evaluating the Amazon compute cloud with GPU nodes as a means of providing a readily accessible test bed of up to 100 nodes.

I currently lack a test bed larger than 100 nodes, to further evaluate our GEMTC framework. There is an ongoing project evaluating MTC and Swift/T on Blue Waters[20] a Cray based high performance resource with over 25k compute nodes.

VII. CONCLUSIONS

In conclusion I have presented GEMTC, a framework for enabling MTC workloads to run efficiently on NVIDIA GPUs. The GEMTC framework encompasses the entire GPU running as a single GPU application similar to a daemon. The GEMTC framework is responsible for receiving work from a host though the use of the C API, scheduling and running that work on many independent GPU workers. Finally, results are returned back through the C API to the host and the workflow system. The GEMTC API enables the framework to integrate closely with workflow systems such as Swift/T.

The GEMTC framework has simplified the programming model of the GPU by allowing GPUs to be treated as a collection of independent SIMD workers, enabling a MIMD view of the device.

The novel sub-allocator implemented within GEMTC allows for an efficient dynamic allocation of memory once an application is running. In addition GEMTC provides an alternative to cudaMalloc() that runs 8 times faster.

Finally, I have shown that GEMTC has a throughput of roughly 7k tasks per second. Integration with Swift/T improves programmability of accelerators, while demonstrating the ability to increase scalability to many nodes with many cores in clusters, clouds, grids, and HPC resources. I will continue to push the envelope in enabling applications from many different domains to run on such resources.

REFERENCES

[1] I. Raicu, Z. Zhang, M. Wilde, I. Foster, P. Beckman, K. Iskra, *et al.*, "Toward loosely coupled programming on petascale systems," presented at the Proceedings of the 2008 ACM/IEEE conference on Supercomputing, Austin, Texas, 2008.

[2] M. Wilde, M. Hategan, J. M. Wozniak, B. Clifford, D. S. Katz, and I. Foster, "Swift: A language for distributed parallel scripting," *Parallel Comput.,* vol. 37, pp. 633-652, 2011.

[3] (2012, October 3). *Parallel Programming and Computing Platform | CUDA | NVIDIA*. Available: http://www.nvidia.com/object/cuda_home_new.html

[4] A. Gharaibeh, S. Al-Kiswany, S. Gopalakrishnan, and M. Ripeanu, "A GPU accelerated storage system," presented at the Proceedings of the 19th ACM

International Symposium on High Performance Distributed Computing, Chicago, Illinois, 2010.

[5] J. Meng, V. A. Morozov, K. Kumaran, V. Vishwanath, and T. D. Uram, "GROPHECY: GPU performance projection from CPU code skeletons," presented at the Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis, Seattle, Washington, 2011.

[6] V. Gupta, K. Schwan, N. Tolia, V. Talwar, and P. Ranganathan, "Pegasus: coordinated scheduling for virtualized accelerator-based systems," presented at the Proceedings of the 2011 USENIX conference on USENIX annual technical conference, Portland, OR, 2011.

[7] V. T. Ravi, M. Becchi, G. Agrawal, and S. Chakradhar, "Supporting GPU sharing in cloud environments with a transparent runtime consolidation framework," presented at the Proceedings of the 20th international symposium on High performance distributed computing, San Jose, California, USA, 2011.

[8] I. Raicu, Y. Zhao, C. Dumitrescu, I. Foster, and M. Wilde, "Falkon: a Fast and Light-weight tasK executiON framework," presented at the Proceedings of the 2007 ACM/IEEE conference on Supercomputing, Reno, Nevada, 2007.

[9] Z. Yong, M. Hategan, B. Clifford, I. Foster, G. von Laszewski, V. Nefedova, *et al.*, "Swift: Fast, Reliable, Loosely Coupled Parallel Computation," in *Services, 2007 IEEE Congress on*, 2007, pp. 199-206.

[10] J. M. Wozniak and M. Wilde, "JETS: Language and System Support for Many-Parallel-Task Computing," in *Parallel Processing Workshops (ICPPW), 2011 40th International Conference on*, 2011, pp. 249-258.

[11] M. Hategan, J. Wozniak, and K. Maheshwari, "Coasters: Uniform Resource Provisioning and Access for Clouds and Grids," in *Utility and Cloud Computing (UCC), 2011 Fourth IEEE International Conference on*, 2011, pp. 114-121.

[12] I. Raicu, I. T. Foster, and Z. Yong, "Many-task computing for grids and supercomputers," in *Many-Task Computing on Grids and Supercomputers, 2008. MTAGS 2008. Workshop on*, 2008, pp. 1-11.

[13] M. Wilde, I. Foster, K. Iskra, P. Beckman, Z. Zhao, A. Espinosa, *et al.*, "Parallel Scripting for Applications at the Petascale and Beyond," *Computer,* vol. 42, pp. 50-60, 2009.

[14] Z. Yong, I. Raicu, and I. Foster, "Scientific Workflow Systems for 21st Century, New Bottle or New Wine?," in *Services - Part I, 2008. IEEE Congress on*, 2008, pp. 467-471.

[15] J. D. Owens, D. Luebke, N. Govindaraju, M. Harris, J. Krüger, A. E. Lefohn, *et al.*, "A Survey of General‐Purpose Computation on Graphics Hardware," *Computer Graphics Forum,* vol. 26, pp. 80-113, 2007.

[16] J. DeBartolo, G. Hocky, M. Wilde, J. Xu, K. F. Freed, and T. R. Sosnick, "Protein structure prediction enhanced with evolutionary diversity: SPEED," *Protein Science,* vol. 19, pp. 520-534, 2010.

[17] K. Wang and I. Raicu, "SimMatrix: SIMulator for MAny-Task computing execution fabRIc at eXascales," 2012.

[18] K. Wang, J. C. H. Munuera, I. Raicu, and H. Jin, "Centralized and Distributed Job Scheduling System Simulation at Exascale," 2011.

[19] A. Bakhoda, G. L. Yuan, W. W. L. Fung, H. Wong, and T. M. Aamodt, "Analyzing CUDA workloads using a detailed GPU simulator," in *Performance Analysis of Systems and Software, 2009. ISPASS 2009. IEEE International Symposium on*, 2009, pp. 163-174.

[20] D. S. Katz, T. G. Armstrong, Z. Zhang, M. Wilde, and J. M. Wozniak, "Many-Task Computing and Blue Waters," *Arxiv preprint arXiv:1202.3943,* 2012.