Design and Evaluation of the GeMTC Framework for GPU-enabled Many-Task Computing

Scott J. Krieder,[*] Justin M. Wozniak,[†] Timothy Armstrong,[§] Michael Wilde[†‡]
Daniel S. Katz,[‡] Benjamin Grimmer,[*] Ian T. Foster,[§†‡] Ioan Raicu[*†]
[*]Department of Computer Science, Illinois Institute of Technology
[†]Mathematics and Computer Science Division, Argonne National Laboratory
[§]Department of Computer Science, University of Chicago
[‡]Computation Institute, University of Chicago & Argonne National Laboratory

ABSTRACT

We present the design and first performance and usability evaluation of GeMTC, a novel execution model and runtime system that enables accelerators to be programmed with many concurrent and independent tasks of potentially short or variable duration. With GeMTC, a broad class of such "many-task" applications can leverage the increasing number of accelerated and hybrid high-end computing systems. GeMTC overcomes the obstacles to using GPUs in a many-task manner by scheduling and launching independent tasks on hardware designed for SIMD-style vector processing. We demonstrate the use of a high-level MTC programming model (the Swift parallel dataflow language) to run tasks on many accelerators and thus provide a high-productivity programming model for the growing number of supercomputers that are accelerator-enabled. While still in an experimental stage, GeMTC can already support tasks of fine (subsecond) granularity and execute concurrent heterogeneous tasks on 86,000 independent GPU warps spanning 2.7M GPU threads on the Blue Waters supercomputer.

Categories and Subject Descriptors

D.1.3 [**Programming Techniques**]: Concurrent Programming

Keywords

Many-task computing; GPGPU; CUDA; Accelerators; Hybrid execution; Workflow; Programming models; Execution models.

1. INTRODUCTION

This work explores methods for, and potential benefits of, applying the increasingly abundant and economical general-purpose graphics processing units (GPGPU) to a broader class of applications. It extends the utility of GPGPU from the class of heavily vectorizable applications to irregularly-structured many-task applications. Such applications are increasingly common, stemming from both problem-solving approaches (i.e., parameter sweeps, simulated annealing or branch-and-bound optimizations, uncertainty quantification) and application domains (climate modeling, rational materials design, molecular dynamics, bioinformatics).

In many-task computing (MTC) [1, 2], tasks may be of short (even subsecond) duration or highly variable (ranging from milliseconds to minutes). Their dependency and data passing characteristics may range from many similar tasks to complex, and possibly dynamically determined, dependency patterns. Tasks typically run to completion: they follow the simple input-process-output model of procedures, rather than retaining state as in web services or MPI processes.

Efficient MTC implementations are now commonplace on clusters, grids, and clouds. In recent years we have extended MTC to applications on homogeneous supercomputers, using tools such as Falkon [3], Swift [4], JETS [5], and Coasters [6]. Other programming models and tools that support MTC include MapReduce, volunteer computing [7], SLURM [8], and Cobalt [9], which allow supercomputer tasks to be subdivided into asynchronous subtasks [10]. All these approaches can benefit from the MTC-enabling accelerator work we describe here. The contributions of this work are as follows:

- Design and implementation of GeMTC, a framework enabling MTC workloads to run efficiently on NVIDIA GPUs.
- Improved dynamic GPU memory management, providing efficient scaling and a 10x improvement over native CUDA dynamic memory management.
- Integration of GeMTC with Swift, enabling a broad class of dataflow-based scientific applications, and improving programmability for both hybrid multicore hosts and extreme scale systems. Work is load balanced among large numbers of GPUs.
- Performance evaluation on synthetic benchmarks and a proxy code representing molecular dynamics simulation workloads.

This paper is organized as follows: Section 2 describes the challenges of many-task computing on GPGPUs. Section 3 describes the GeMTC framework and its underlying architecture. Section 4 describes Swift and its integration as a

HPDC'14, June 23–27, Vancouver, BC, Canada.

http://dx.doi.org/10.1145/2600212.2600228.

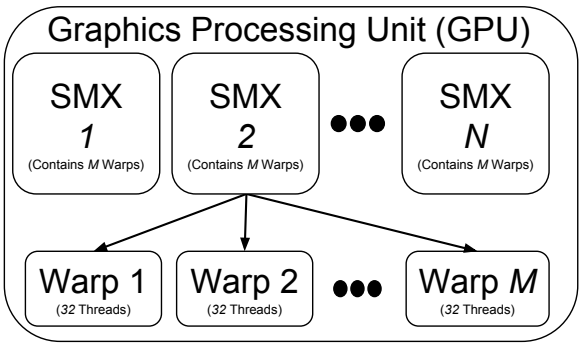

Figure 1: Diagram of GPU architecture hierarchy.

GeMTC programming model. Section 5 presents a performance evaluation, and Section 6 discusses related work. We summarize our contributions in Section 7 and briefly discuss related work.

2. CHALLENGES OF MANY-TASK COMPUTING ON GPGPUS

Our GeMTC work is motivated by the fact that with current mainstream programming models, a significant portion of GPU processing capabilities underutilized by MTC workloads. We advocate sending a larger number of smaller, concurrent, independent tasks to a GPU. The results presented here indicate that this approach enables higher utilization of GPU resources, greater concurrency, and hence higher many-task throughput.

2.1 NVIDIA GPUs and GPGPU Computing.

General-purpose computing on graphics processing units (GPGPU) allows a host CPU to offload a wide variety of computation, not just graphics, to a graphics processing unit (GPU). GPUs are designed for vector parallelism: they contain many lightweight cores designed to support parallel bulk processing of graphics data. GPGPU leverages this parallel architecture for nongraphic computations such as matrix multiplication. In the context of this paper, all references to GPU refer to this GPGPU approach. In addition to application speedup, other benefits to leveraging accelerators include power efficiency (improved Flops/watt) and cost savings (improved Flops/$).

As shown in Figure 1, a NVIDIA GPU (which dominates the GPGPU and HPC marketplace) is comprised of many Streaming Multiprocessors (SMXs). A SMX contains many warps, and each warp provides 32 concurrent threads of execution. All threads within a warp run in a Single Instruction Multiple Thread (SIMT) fashion. As we describe below, GeMTC schedules independent computations on the GPU at the warp level, a level of independent task concurrency not provided by any mainstream GPU programming model.

Our GeMTC work targets the latest generation of NVIDIA GPUs, specifically the Kepler K20X. This device has 14 SMXs with 192 cores per SMX, a maximum of 168 warps, and a total core count of 2,688. MTC workloads that send only single tasks, or small numbers of large tasks, to accelerator devices observe near-serialized performance, and leave a significant portion of device processor capability unused. Ousterhout et al., [11] make a compelling argument for the pervasive use of tiny tasks in compute clusters. We apply a similar argument to motivate the GeMTC model of running many small independent tasks on accelerators. Driven by this tiny-task motivation, GeMTC provides an architecture for "overdecomposition" [12] of accelerator-resident tasks, which can then be tightly packed into a GPU to maximize efficiency and minimize time to solution. While Swift load balances tasks and applies compiler optimizations in support of overdecomposition, the user must write applications with suitably fine-grained tasks.

2.2 Mainstream GPU Support for MTC

The dominant CUDA and OpenCL GPGPU programming models both provide extensions to traditional programming languages such as C with added API calls to interact with accelerators. CUDA is supported by NVIDIA and works on NVIDIA GPUs. OpenCL is based on an open standard that aims to provide improved portability across a variety of accelerators and other compute devices. OpenACC is a newer pragma-based technology that is gaining momentum. As in OpenMP, OpenACC programmers provide hints to the compiler where they believe a computation would benefit from being offloaded to an accelerator. OpenACC is an open standard and aims to provide the portability of OpenCL while requiring less detailed knowledge of accelerator architecture than is required in CUDA and OpenCL programming. In many cases OpenACC may require significantly less coding, but early measurements (e.g., by Wienke et al. [13]) suggest that OpenACC is not yet capable of delivering equivalent performance.

Concurrent Kernels [14] is a CUDA feature that enables the developer to launch parallel work on a GPU. However, the maximum number of concurrent kernels is limited to 32, far less than the number of 168 independent warps provided by the latest Kepler GPUs. HyperQ and Dynamic Parallelism [15], recent CUDA enhancements introduced by NVIDIA with the Kepler architecture, are a step toward MTC support. HyperQ allows more parallel work to be sent to the GPU, while Dynamic Parallelism allows threads to spawn more threads on the device. The current model of GeMTC and Swift relies on communication between the CPU and GPU to drive tasks to and from the Swift script. If a task sent to GeMTC from Swift was represented by compact code and could be decomposed even further (e.g., loop unrolling) it is possible that GeMTC could utilize Dynamic Parallelism to dynamically launch new tasks and process the parent task with even more improved performance, but we leave that as future work. Most other programming models, however, still treat the GPU as a solution to large vector-oriented SIMD computations and do not adequately support the potential for speedup of many-task applications.

A primary motivation for our work on GeMTC is that none of these mainstream accelerator programming models provides the flexible support for independent concurrent tasks required by many-task applications. In order to effectively utilize an accelerator, MTC applications with complex task dependencies need task results rapidly returned from device to host so that the application can process its dataflow-driven dependencies. To the best of our knowledge, no solution prior to GeMTC offers this capability.

Figure 2(A) illustrates why many-task computing workloads experience low efficiencies through Concurrent Ker-

nels, the best available standard CUDA concurrency model for independent tasks launched by the host. In this model, tasks must be submitted at the same time, and no additional tasks can be submitted until all tasks are complete. With unbalanced task durations, a significant number of GPU processor cores will be underutilized. In addition, to process workflows with complex dependencies, the developer must group tasks into batches and block on batch completion before executing dependent kernels, an inadequate approach for supporting heterogeneous concurrent tasks. Figure 2(B) demonstrates how GeMTC provides support for heterogeneous tasks by treating every warp worker as an independently operating SIMD compute device. Because the warps are operating independently they are able to pick up work immediately rather than block on other warps for completion. Figure 2(C) demonstrates how overdecomposition can be utilized by GeMTC to pack tiny tasks neatly into the GPU, maximizing device core utilization and reducing application time to solution.

3. GEMTC ARCHITECTURE

Given that our target test bed consisted of NVIDIA GPUs and that we wanted to examine the GPU at the finest granularity possible, we opted to implement our framework using CUDA. This decision allowed us to work at the finest granularity possible but limited our evaluation to NVIDIA based hardware. While GeMTC was originally developed on NVIDIA CUDA devices, its architecture is general, and has also been implemented on the Intel Xeon Phi [16]. The Phi, however, represents a different accelerator architecture, meriting separate study, and is not addressed in this paper.

Figure 3 shows a high-level diagram of GeMTC driven by tasks generated by the Swift parallel functional dataflow language (described in Section IV). GeMTC launches a daemon on the GPU that enables independent tasks to be multiplexed onto warp-level GPU workers. A work queue in GPU memory is populated from calls to a C-based API, and GPU workers pick up and execute these tasks. After a worker has completed a computation, the results are placed on an outgoing result queue and returned to the caller.

3.1 Kernel Structure and Task Descriptions

A key element of GeMTC is the daemon launched on the GPU, named the Super Kernel, which enables many hardware level workers (at the warp level) on the GPU. A work queue in GPU memory is populated from calls to a C API, and GPU workers pick up and execute these tasks. After a worker has completed a computation, the results are placed on an outgoing result queue and returned to the caller.

Within traditional GPU programming, a user defined function that runs on the GPU is called a *kernel*. An application may define many GPU kernels, and application logic may be written to execute some or all kernels in parallel. These *concurrent kernels* are a key technology in the GeMTC framework. Once the GeMTC framework is initialized, the *Super Kernel* daemon is started, the memory management system is set up, and calls can begin to Application Kernels (*App-Kernels*). The Super Kernel gathers hardware information from the GPU and dynamically starts the maximum number of workers available on that GPU. A worker consists of a single warp, and therefore the maximum number of workers is equal to the maximum number of warps.

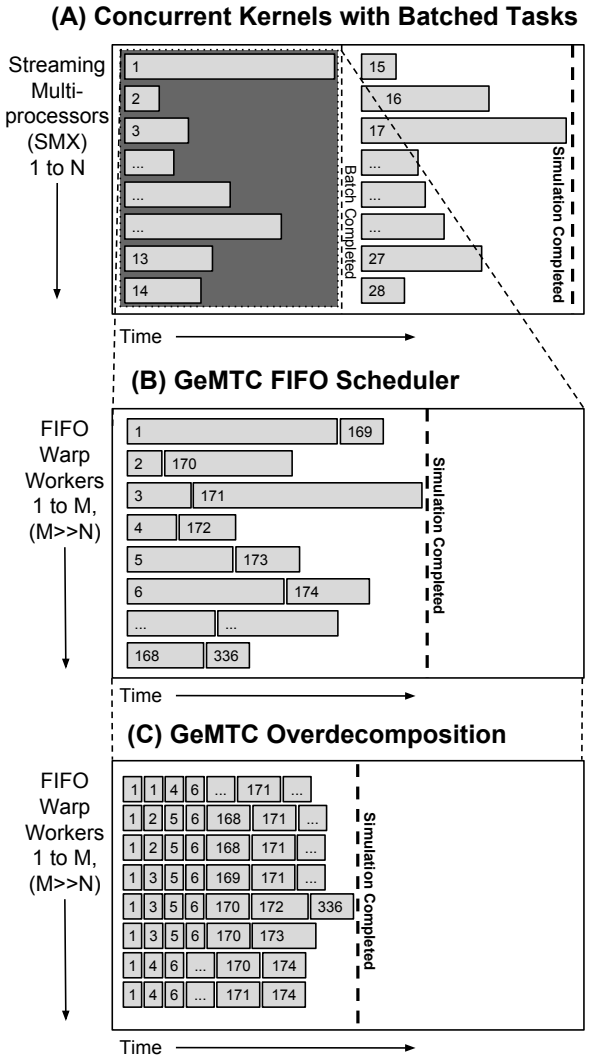

Figure 2: GeMTC FIFO scheduler processes tasks as soon as they are available, rather than blocking on batches for completion. The warps required to execute cases (B) and (C) are provided by all the streaming multiprocessor's within the shaded area of (A). While the hardware available remains the same, the number of parallel channels is increased for the amount of concurrent parallel work.

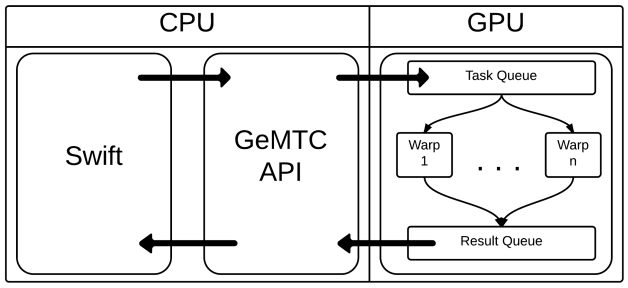

Figure 3: Flow of a task in GeMTC.

AppKernels are the computations that are executed by a GeMTC worker. The AppKernels are modular in design, and users can quickly contribute to the AppKernel Library by writing their own AppKernels based on pre-existing templates. A major appeal of the GeMTC framework is the decomposition of the GPU programming model. Instead of an application launching hundreds or thousands of threads, which could quickly become more challenging to manage, GeMTC AppKernels are optimized at the warp level, meaning the programmer and AppKernel logic are responsible for managing only 32 threads in a given application. Furthermore, run-time logic can be used to control concurrency of tasks to ensure that GPU cores are kept utilized without exhausting the GPU memory.

The *Task Description* is a C struct that contains relevant information for executing an AppKernel as a task on GeMTC. The Task Description is passed from a client via the GeMTC API (e.g., by Swift) to the GPU and queued with parameters on the device to the input queue or queued with task results on the outgoing result queue.

Figure 4 shows how a sample AppKernel could be written to compute a naive square matrix multiplication through GeMTC. Swift stubs have marshaled AppKernel parameters into a single boxed parameter. Therefore, after calibrating for warp size, the first step is to unbox the parameters. After executing an algorithm optimized for the warp size, the result is stored in a location identified from unboxing the input parameters. The result is then placed on an outgoing result queue, and the warp is ready to pick up new work.

3.2 GeMTC API

The GeMTC API is a C-based API which consists of eight major functions identified in Table 1. Figure 5 uses a simple molecular dynamics (MD) example to demonstrate how a user can leverage the GeMTC API to launch a simulation on the GPU. For the MD example, the user defines the initial universe of molecules as a parameter to the MD function. Once these parameters have been transferred into GPU memory the user pushes the task to the GPU along with all the information needed to create the task description on the device. The push operation contains, as parameters, the four pieces of data necessary to construct the task description; in this case, TaskType = MDLite, TASK_ID is set to a unique integer value (for tracking the task throughout its lifetime), numThreads = 32, and *params = a pointer to device memory where the task parameters are stored.

```
1  __device__ void MatrixMultiply(void *boxed_input)
2  {
3    //  calibrate for warp size
4    int warp_size = 32;
5    int thread = threadIdx.x % warp_size;
6    //  unbox host parameters
7    float* inputParams = (float*)boxed_input;
8    int matrixWidth = inputParams[0];
9    int matrixSize = matrixWidth * matrixWidth;
10   float *matrixA = inputParams+1;
11   float *matrixB = matrixA + matrixSize;
12   float *matrixOut = matrixA + 2 * matrixSize;
13   //  compute Matrix Multiplication
14   for (unsigned int i = thread; i < matrixWidth;
15        i=i+warp_size){
16     for (unsigned int j = 0; j < matrixWidth; j++) {
17       float sum = 0;
18       for (unsigned int k = 0; k < matrixWidth; k++) {
19 float a = matrixA[i * matrixWidth + k];
20 float b = matrixB[k * matrixWidth + j];
21 sum += a * b;
22       }
23       //  result location from input parameters
24       matrixOut[i * matrixWidth + j ] = sum;
25     }
26   }
27 }
```

Figure 4: GeMTC Mat-Mul AppKernel

Table 1: GeMTC API

API Call	Functionality Provided
gemtc(Setup/Cleanup)	(Start/Stop) GeMTC
gemtc(Push/Poll)	(Submit/Return) Tasks
gemtcMemcpyHostToDevice	Memory Copy
gemtcMemcpyDeviceToHost	Memory Copy
gemtcGPU(Malloc/Free)	(Allocate/Free) Memory

At this point the user can begin polling for a result. The precompiled MD AppKernel already knows how to pack and unpack the function parameters from memory; and once the function completes, the result is packed into memory and placed on the result queue. When the gemtcPoll function returns a result, the user can then unpack the memory and move to the next operation. The gemtcPoll function does not block on a specific task, and it automatically pops any completed task(s) off the result queue. This strategy is explained in further detail in the Task Bundling subsection. In addition, the example shown in Figure 5 is specific to users leveraging the C API. It is expected that end users will utilize high-level Swift scripts to launch their tasks on GeMTC. The calls described above are implicitly handled by the GeMTC and Swift integration, as explained in further detail in Section 4.

3.3 Queues, Tasks, and Memory Management

GeMTC manages two queues on the device. The *Incoming Work Queue* is populated by calls to the GeMTC API and contains tasks that are ready to execute. The tasks in this queue contain a *TaskDescription* and the necessary parameters to execute the task. Both in-memory queues are configured as circular linked-lists with pointers indicating the front and rear of the queue. When a worker picks up a task, it will dequeue from the front, and any new work is placed at the rear. Figure 6 demonstrates how workers interact with the queues.

```
1  # include "gemtc.cu"
2  main(){
3    // Start GeMTC
4    gemtcSetup(QUEUE_SIZE);
5    // Allocate device memory
6    device_params = gemtcGPUMalloc(MALLOC_SIZE);
7    // Populate device memory
8    gemtcMemcpyHostToDevice(device_params,
9      host_params, MALLOC_SIZE);
10   // Push a task to the GPU
11   gemtcPush(MD_Lite, NUM_THREADS,
12       TaskID, device_params);
13   // Poll for completed results
14   gemtcPoll(TaskID, pointer);
15   // Copy back results
16   gemtcMemcpyDeviceToHost(host_params,
17       pointer, MALLOC_SIZE);
18   // Free GPU memory
19   gemtcGPUFree(pointer);
20   // Shutdown GeMTC
21   gemtcCleanup();
22 }
```

Figure 5: Code sample of GeMTC API.

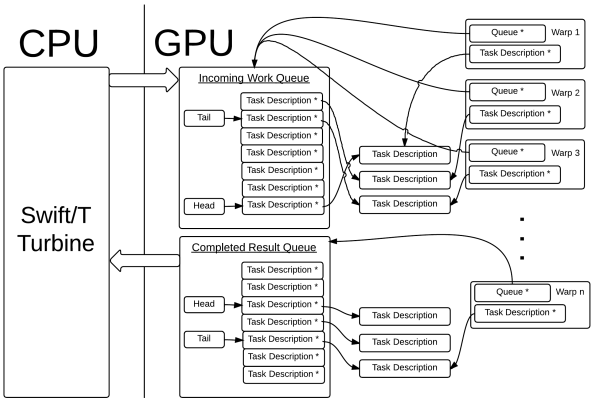

Figure 6: GPU Workers interacting with queues.

The GeMTC framework requires efficient device memory allocation on a per task basis. Each task enqueued requires at least two device allocations: the first for the task itself and the second for parameters and results. The existing CUDA memory management system was not designed for a large number of independent memory allocations. With traditional CUDA programming models the current best practice is to allocate all memory needed by an application at launch time and then manually manage and reuse this memory as needed.

To reduce the large overhead of individual memory allocations for MTC workloads, GeMTC includes a sub-allocator designed to efficiently handle many requests for dynamic allocation. The sub-allocator uses the existing CUDA malloc to allocate large contiguous pieces of device memory, allocating more as needed. Then pointers to these free chunks and their sizes are stored in a circular linked list on the CPU (see Figure 7). This list is ordered by increasing device address to allow for easy memory coalescing of adjacent memory chunks.

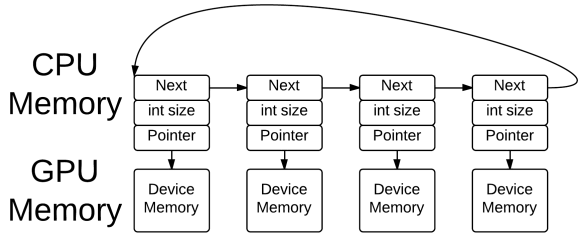

Figure 7: Memory mapping of free memory available to the device.

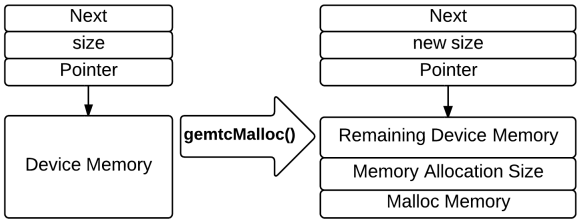

Figure 8: Result of gemtcMalloc on free memory.

When a GeMTC memory allocation request is sent from the host to the GPU, the sub-allocator will traverse the list and select the first chunk of free device memory meeting the allocation requirements. Figure 8 demonstrates how the header is then updated to reflect the remaining free device memory available in that chunk. This operation runs in the same order of time as a single memory copy to the device.

Upon freeing device memory, the header is read to identify the size of the chunk. Then it is added to the list of free memory in the correct location. If there is any free consecutive memory, the chunk is coalesced to provide a single larger contiguous chunk of memory. The operation to free device memory takes roughly the same amount of time as reading the header (i.e., a device memory copy).

Both malloc() and free() within GeMTC's memory management run in O(n), where n is the length of the free memory list. In addition, the size of the list is proportional to the amount of memory fragmentation since each element is recorded as a separate chunk of memory. Because malloc and free both need to write and read to the GPU memory, these operations may scale poorly under workloads with high fragmentation. However, the MTC workloads we examine show no signs of high fragmentation. The original cudaMalloc ran in ∼100 microseconds, and our gemtcMalloc runs in ∼10 microseconds.

To optimize the GeMTC framework for fine-grained tasks, we have implemented a task-bundling system to reduce the amount of communication between the host and GPU. The main bottleneck for obtaining high task throughput through GeMTC is the latency associated with writing a task to the GPU DRAM memory. This bundling system as shown in Figure 9 creates a buffer of tasks that need to be written to the GPU, and flushes it periodically or when it is full. This

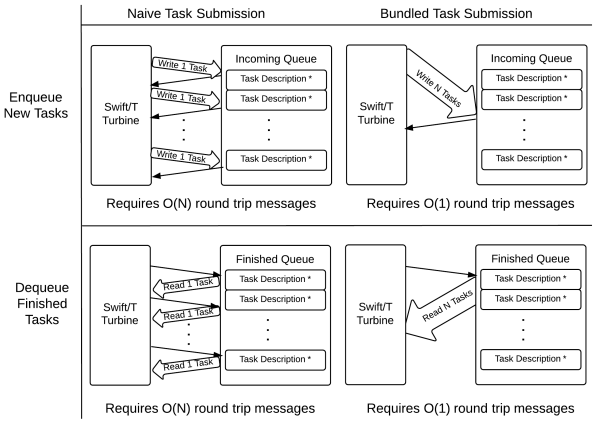

Figure 9: GeMTC implicitly bundles tasks to efficiently utilize PCI bandwidth and latency.

whole buffer can be written to the device in a single copy. Similarly, results of finished tasks can be read from device memory in bulk. The performance improvements of this optimization are substantial. With no bundling (buffer size of one), GeMTC is able to run approximately 6,000 tasks per second on a single GPU. With bundling of 100 task groups, we achieve 22,000 tasks per second, more than a 3x increase in throughput. This optimization allows GeMTC to support many more fine-grained tasks.

4. SWIFT: DATAFLOW EXECUTION AND PROGRAMMING MODEL FOR MTC

Swift [4] is an *implicitly parallel functional dataflow* programming language that is proving increasingly useful to express the higher-level logic of scientific and engineering applications. In this work, we enable Swift to serve as a high-level, high-productivity programming language for hybrid CPU/GPU applications, paving the way for a seamless programming environment for extreme-scale systems composed of hybrid, accelerated nodes.

Many important application classes and programming techniques that are driving the requirements for such extreme-scale systems include branch and bound, stochastic programming, materials by design, and uncertainty quantification. All these classes can be productively expressed as many-task dataflow programs. The dataflow programming model of the Swift parallel scripting language can elegantly express, through implicit parallelism, the massive concurrency demanded by these applications while retaining the productivity benefits of a high-level language. Swift programs can be written with little or no experience in parallel programming, making it a productive language for scientists and engineers to leverage parallel systems.

Swift was originally developed as a scripting language for executing scripts composed from the execution of ordinary application programs on distributed systems such as clusters, grids, and clouds [17]. In this mode, a Swift interpreter, written in Java, executes on a single (possibly multicore) host, and sends work to distributed systems using a variety of "providers" that interface with remote systems.

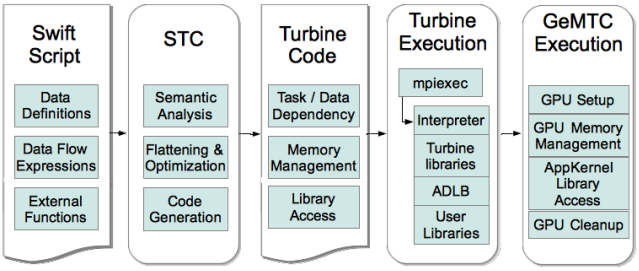

Figure 10: Swift/T stack including GeMTC.

When using its own resource provisioner [6] Swift is capable of sending approximate 500 tasks per second to a set of resources. When using Falkon [3], Swift achieved over 1,000 tasks per second.

To overcome the limits of this centralized single-node program evaluation model, a new Swift implementation named Swift/T [18] was implemented to achieve extreme scalability. Its innovations are a dataflow engine that provides highly parallel, distributed-memory evaluation; a data store that enables distributed access to memory-resident objects; and a load balancer for scalable low-latency task distribution to systems with millions of cores. While the original Swift language could only specify the execution of application programs as its leaf tasks, and only return files from these tasks, Swift/T enables finer-grained programming of in-memory functions that pass in-memory data objects. Performance (and granularity) has been improved by four orders of magnitude beyond the centralized Swift system, to $>$ 15M tasks/sec (achieved on 128K integer cores of Blue Waters). This enables Swift to express a far broader set of applications, and makes it a productive coordination language for hybrid CPU+accelerator nodes and systems.

GeMTC Integration with Swift

The integration with Swift provides many mutual benefits for both Swift and GeMTC. The four boxes on the left side of Figure 10 show the original Swift/T stack before adding GeMTC support. Users write high-level Swift scripts that are compiled by STC [19] into code which is then executed by Turbine [20]. The final box on the right illustrates how GeMTC fits into the Swift/T stack. Once code has been generated and ready for execution, the Turbine runtime uses the GeMTC API to establish memory for tasks, move tasks into the GPU, and return results from the GPU back to the high level Swift script. Currently each GPU is dedicated to one Swift application. Swift provides logging information for all application runs to allow programmers to pinpoint errors and evaluate safety/reliability issues.

All the GeMTC API calls are managed dynamically within the Turbine worker node of the Swift runtime environment. Thus, the user's Swift application can simply call any function mapped to an AppKernel from the high level Swift program. Figure 11 shows an example of how the user would write a Swift application to utilize a set of GPUs through GeMTC.

Data transfers overlap with ongoing GPU computations implicitly and automatically. And because the GeMTC API calls are handled at the Turbine worker level, the Swift programmer is freed from the burden of writing complex mem-

```
1  import gemtc;
2  main
3  {
4    float input_array[];
5    float result_array[];
6
7    input_array = populate_array(SIZE);
8    result_array = gemtc_mdlite(
9        MD_CONFIGURATIONS, input_array);
10 }
```

Figure 11: Swift script launching GeMTC.

ory management code for the GPU. Overlapping data transfer with compute is a common way to achieve increased accelerator performance, and the GeMTC + Swift stack provides this functionality automatically.

Integrating GeMTC with Swift allows GeMTC to launch on very large sets of cluster nodes with very low overhead. Swift automatically performs load balancing across all worker CPUs and across nodes, while GeMTC is optimized from the CPU-GPU level down to the GPU warps.

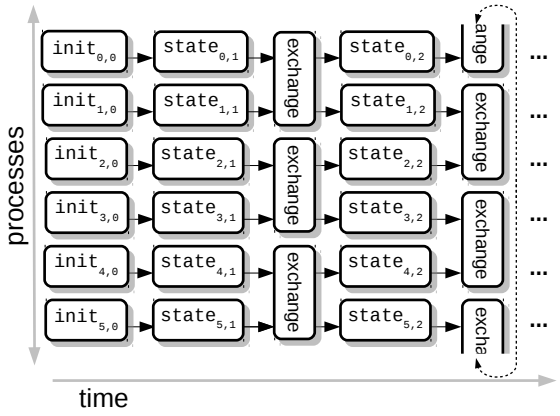

Figure 12: Diagram demonstrating execution model for molecular dynamics with replica exchange. Short simulation segments are run in an ensemble with asynchronous data exchanges [24].

5. PERFORMANCE EVALUATION

This section evaluates the GeMTC framework with a set of AppKernels from the GeMTC AppKernel Library. AppKernels are CUDA device functions that have been tuned to deliver high performance under MTC workloads. AppKernels have been precompiled into the GeMTC framework.

We work with a lightweight molecular dynamics simulation called MDLite. We first evaluate MDLite at the level of a single GPU warp and then over all warps in the GPU. We conclude with an analysis of MDLite over multiple XK7 nodes and examine a set of simple adder benchmarks to highlight throughput and efficiency.

The target test bed for this work is the Blue Waters HPC resource at NCSA. Blue Waters contains ∼20K Cray XE6 CPU based nodes and ∼4K Cray XK7 GPU nodes. While future work aims to address heterogeneous scheduling, this work focuses on the XK7 nodes. Each node contains a single AMD 6276 Interlagos Processor with 8 Bulldozer cores and 16 integer scheduling units. In addition, each node contains 32 GB of memory. Each XK7 node is equipped with a Kepler K20X GPU with 6 GB of memory, 2688 CUDA cores, a peak GPU performance of 1.31 TF (double precision), and a memory bandwidth of 200 GB/s [21].

5.1 Molecular Dynamics

MDLite is a simple molecular dynamics simulation, based on an educational code [22]. The user specifies the number of particles in a "universe" along with their starting positions, the number of dimensions, and a starting mass. MDLite runs a simulation that determines how the potential and kinetic energy in the system changes as the particles change position. MDLite then simulates a particle system with coupled, time-step-discretized differential equations. A MDLite task consists of loading the simulation into GPU memory, running the simulation, and returning the results to Swift. The MDLite workflow may contain dependencies as demonstrated in Figure 12, where simulations exchange data before continuing execution. MDLite is an excellent proxy application for Protlib-2 and InsEnds [23] and demonstrates data movement and fine-grained task execution on accelerators.

GPU computing traditionally involves managing thousands of threads of execution. First, we scale down to show that it is indeed possible to gain increased performance at a low level using a varied of number of active threads within a single warp. Figure 13 demonstrates that GeMTC is capable of scaling within a single warp. By varying the number of active threads included in a warp computation, we prove that for the right application it could indeed benefit from the 32 threads in a GPU warp. In this figure, 1 thread is a lower bound on performance, and 32 threads are the maximum number of threads available within a single warp. Figure 14 highlights the speedup obtained by varying the level of concurrency for MDLite within a single warp of execution. While the walltime of MDLite successfully decreases as more threads are added, the speedup obtained is significantly less than ideal after 8 threads are active within a single warp. While improved performance could certainly be achieved with added development cycles of fine tuning, we argue for the overdecomposition of large tasks into smaller, easier-to-manage MTC tasks. By identifying the task granularity that observes the best combined performance of walltime and thread utilization, we can run many tasks of that granularity on the GPU.

Scaling an application down to the level of concurrency available within a single warp can provide the highest level of thread utilization for some applications. Next, we evaluate how efficient many of these independent warp workers are when working in parallel across a single GPU.

Figure 15 evaluates a varied number of MDLite simulations running over a K20X GPU. In this experiment a single warp provides a baseline and lower bound on performance. This baseline is achieved by running a single task on a single warp and varying the input vector size from several hundred elements to several thousand. The plot lines in Figure 15 represent active GPU workers in executing the MD application, conveying GeMTC scalability from 1 to the maximum number of GPU workers.

As more workers are added, there is increasingly higher demand on the in-memory queues; specifically, the two locks that keep each queue synchronized become bottlenecks for

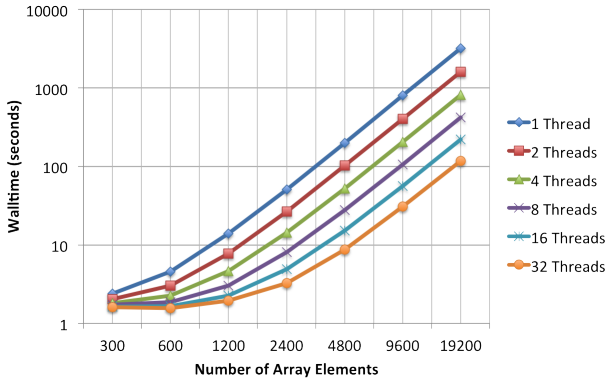

Figure 13: GeMTC scales MD within a single warp and achieves decreased walltime as the level of concurrency within the computation is increased.

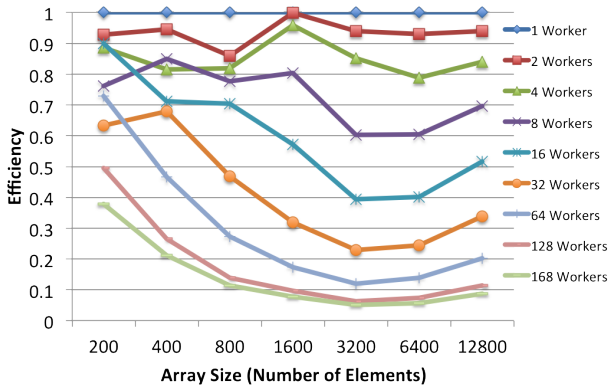

Figure 15: GeMTC utilization on the K20X running MD codes with varied worker counts from 1 to 168.

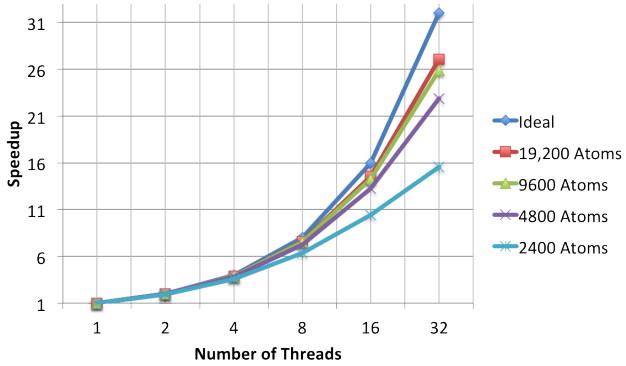

Figure 14: Speedup achieved with varied concurrency (1-32 threads) within a single warp, launching MDLite tasks from 2,500 atoms to 19,200 atoms.

Figure 16: GeMTC and MDLite scaling over 1344 workers on Blue Waters.

the system. We anticipate that implementing a lock-free queueing system on the device will alleviate a significant portion of the latency and provide improved performance; this is left as future work.

We have shown GeMTC is capable of scaling down to manage threads both within a single warp and across multiple warps within a single GPU. Next we highlight how the integration with Swift enables GeMTC to scale across multiple XK7 nodes within Blue Waters. Figure 16 is a multinode scaling experiment where the number of simulations is set equal to the number of workers. At each data point there are two times as many workers as the previous, so we run twice as much work. In an ideal system without any overheads we would expect a flat line demonstrating the ability to conduct the same amount of work at each step. Even after 8 nodes we achieve 96% utilization. Future work aims to evaluate our system at even larger scales on Blue Waters.

5.2 Throughput and Efficiency

Next, we evaluate GeMTC with a simple adder benchmark. The benchmark launches a series of additions on the GPU, for which we have calculated expected runtimes. Afterwards, we can easily measure the efficiency and overhead of our system: efficiency = (expected runtime/observed runtime). Expected runtime is calculated during a short calibration phase where kernel runtime is measured with and without our framework.

First, a CPU version of the simple adder is executed through Swift/T on XE6 nodes. The results shown in Figure 17 confirm that Swift/T is highly capable of driving fine-grained work over many nodes on Blue Waters. In particular, tasks with runtimes exceeding 10 ms observe high efficiencies well over 10K processes (where 1 process is running per core) and fits our target duration and scale for the GPU evaluation.

Secondly, we evaluate Swift on a single GPU equipped node to calculate a per node throughput rate for GPU tasks. Figure 18 highlights throughput rates for Swift and GeMTC on a Kepler K20X. In this benchmark the maximum number of available GeMTC workers is enabled (168). Figure 19 demonstrates that Swift + GeMTC is capable of driving a high throughput of fine grain GPU work over many nodes on Blue Waters. As shown in Figure 19 we obtain ~70% of ideal throughput with 10,000-way concurrency. Next, we evaluate GeMTC efficiencies on a single XK7 node with workloads comprising varied task granularities. Figure 20 highlights the single-node efficiency of GeMTC running with

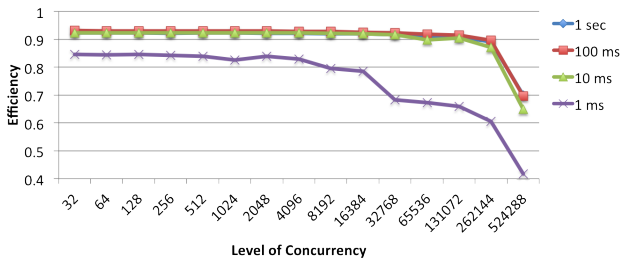

Figure 17: Fine-grained Swift CPU workloads on Blue Waters, demonstrating the ability to drive fine-grained workloads with high efficiency.

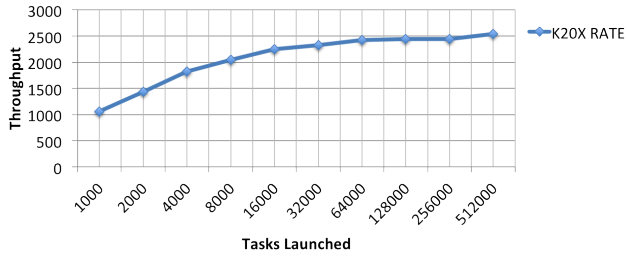

Figure 18: Swift driving GeMTC tasks on a Cray XK7(K20X equipped) node of Blue Waters.

168 active workers per GPU. At a single node we continue to observe high efficiencies for tasks with runtimes exceeding ~150ms. In both Figure 19 and Figure 20 the tasks measuring throughput are simple summations with small data parameters (and thus include data movement overhead).

Figure 21 demonstrates an upper bound of GeMTC by launching efficiency workloads on multiple GPU nodes with only a single active GeMTC worker per GPU. We next enable 168 GeMTC warp workers per GPU (the maximum) and evaluate the efficiency of workflows with varied task granularities up to 86k individually operating GPU workers of Blue Waters. After adding 167 additional workers per GPU we do require longer lasting tasks to achieve high efficiency. We attribute this drop in performance to greater worker contention on the device queues and the fact that Swift must now drive 168 times the amount of work per

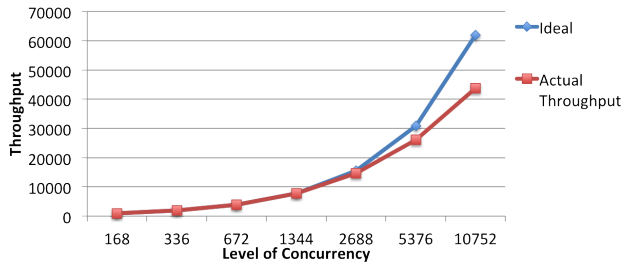

Figure 19: GeMTC + Swift Throughput over 10,000 GPU workers.

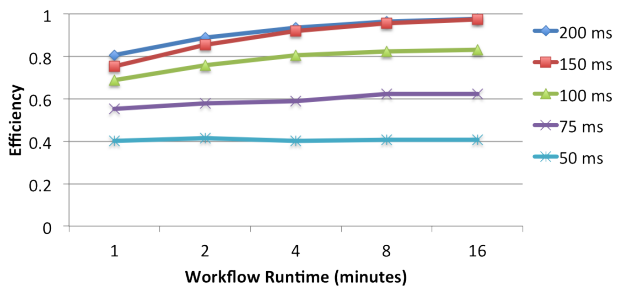

Figure 20: Single-node efficiency on Blue Waters for parallel work with varied task granularities running 168 GPU workers.

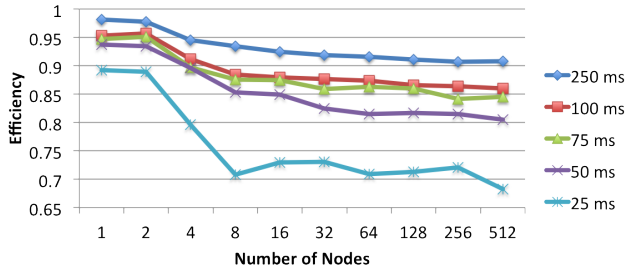

Figure 21: GeMTC + Swift efficiency for varying task granularities up to 512 nodes on Blue Waters with a single GeMTC worker active per node.

node. In Figure 22 we observe that tasks exceeding ~1 second achieve high efficiency up to scales of 40K workers. Although we have not yet identified the cause for this drop in performance, we expect that the performance degradation at extreme levels of concurrency comes from the loading of shared libraries from the remote parallel filesystem. In future work we will continue to improve systemwide performance by reducing the reliance on dynamic loadable shared libraries and through larger scale evaluation on all 4K XK7 nodes of Blue Waters.

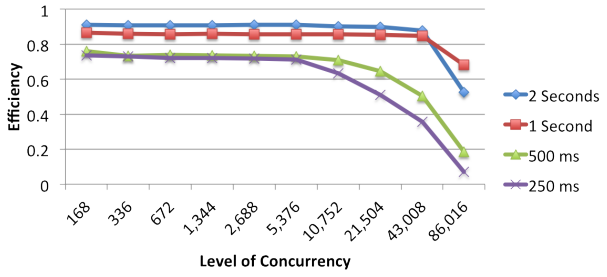

Figure 22: Efficiency for workloads with varied task granularities up to 86K independent warps of Blue Waters. 168 active workers/GPU.

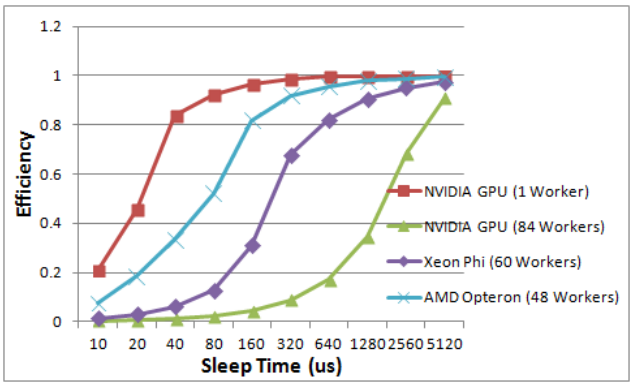

Figure 23: Microbenchmark measuring efficiency for tasks with varied granularities on a variety of hardware: a 1344-core NVIDIA GPU, a 60-core Intel Xeon Phi, and a 48-core AMD Opteron SMP.

5.3 Preliminary MTC Xeon Phi Results

We have also gathered preliminary results for supporting MTC workloads on the Intel Xeon Phi Coprocessor. As shown in Figure 23 we can achieve the same level of efficiency with shorter running tasks (50% shorter) on a Xeon Phi compared with a GTX-680 NVIDIA GPU. We highlight the fact that with GeMTC on its own we observe upwards of 90% efficiency with tasks lasting 5 ms. The AMD Opteron has an even higher level of improved performance. This means that a fully general purpose framework would be capable of launching tasks an order of magnitude faster. We will continue to improve performance to ensure all components of the system can keep up with these task dispatch rates.

6. RELATED WORK

Warp Level Execution. Hong et al. [25] developed and evaluated methods that obtained 9X speedups of breadth-first search in graphs over prior GPU implementations by enabling each warp to run independent threads and even multiple "virtual" threads. While it stopped short of the generalized mechanism for managing independent per-warp tasks presented here, it characterizes the memory access regimes in which many-task programming of accelerators will provide effective speedup, and describes methods for integrating SISD and SIMD code segments within warp-resident tasks that can guide the implementation of GeMTC AppKernels. Chen et al. observed how unbalanced task-based workloads resulted in low utilization of CUDA hardware [26] and multi-GPU systems [27]. While still not a general-purpose solution like GeMTC, our work could benefit from the lock-free queueing system Chen described. While GeMTC is currently optimized for a single GPU per node we believe supporting multi-GPUs on a single node would only require launching additional Swift workers configured to target additional accelerators.

Dataflow on Accelerators. PTask [28], by Rossbach et al. aims to treat GPUs as compute devices from the view of the Operating System. PTask supports the dataflow programming model and provides several scheduling options for tasks including priority, first-available, and data-aware. Unlike Swift, PTask does not support dynamic DAGs. Dandelion is built on top of PTask and cross compiles .NET code into CUDA code that is then executed on the PTask runtime [29].

Accelerator Virtualization. Virtualization is another technique to decompose hardware. Ravi et al. present a framework to enable GPU sharing amongst GPUs in the cloud [30]. Computing the affinity score allows the authors to determine which applications can benefit from consolidation. Becchi et al. extend the framework to provide additional abstractions for GPU sharing while allowing isolation of concurrent applications [31]. Gupta et al. describe Pegasus [32], which aims to improve GPU utilization through virtualization. The Pegasus project runs at the hypervisor level and promotes GPU sharing across virtual machines, while including a custom DomA scheduler for GPU task scheduling.

Runtime Systems. StarPU [33] is a task-based runtime designed to improve the programmability of accelerators while maintaining efficiencies. StarPU does allow for data dependencies and hints regarding task priorities, but GeMTC provides a finer granularity by leveraging warps as workers. Zhang et al. present GPU-TLS [34], a loop speculative parallelization framework for decomposing large loops into smaller pieces which are then executed in parallel by GPU kernels. Chatterjee et al. describe a runtime [35] that allows different tasks to execute on the same SM in a similar fashion to GeMTC. However, their runtime schedules workers based on the Concurrent Collections (CnC) model and leverages work stealing among SM managed queues to execute tasks. At startup their runtime treats every SM as a single worker, but allows tasks to launch more tasks. Our work could benefit from the idea of additional worker queues to reduce contention. However, our model leverages Swift to manage the dependencies of run-to-completion tasks that are assumed to have already been decomposed to the finest granularity possible. Not only does the Turbine engine within Swift support data dependencies, but the execution of tasks is based on the dataflow model. Added benefits of leveraging Swift include access to the built-in load balancer ADLB, and the ability to easily span multiple nodes. COSMIC [36] is a middleware for multiprocessing on the Intel Xeon Phi. The GeMTC implementation on the Xeon-Phi will benefit greatly from avoiding memory and thread oversubscription, as highlighted in this work.

Alternative Accelerator Programming Methods. Intelligent compilers are another way to avoid low level accelerator development and still gain high performance. If the compiler is able to generate device code and parallel instructions, the developer may opt to write sequential code and benefit from accelerator speedup. OpenMPC [37] is a project that takes OpenMP code and converts it to CUDA, thus enabling many scientific applications already written in OpenMP to take advantage of the accelerator. Other work aims to target accelerators directly from OpenMP [38]. Grophecy [39] attempts to improve the process of migrating codes to the GPU through the analysis of code skeletonization. Grophecy can analyze CPU codes and determine whether they may benefit from being moved to the GPU, saving valuable development cycles. Singe [40] is a Domain Specific Language compiler for combustion chemistry applications. GeMTC could benefit from a Grophecy or Singe-like module for creating warp-optimized AppKernels and vice versa. MPI-ACC [41] aims to provide integrated MPI support for accelerators to allow the programmer to easily execute code on a CPU or GPU.

7. CONCLUSIONS

We 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 through the use of the C API, and scheduling and running that work on many independent GPU workers. Results are returned through the C API to the host and then to Swift. The GeMTC API enables the framework to closely integrate with parallel scripting systems such as Swift/T. The GeMTC framework simplifies 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 API call to cudaMalloc, achieving a speedup of roughly 10x. GeMTC has a throughput of roughly 23K tasks per second on a single node. 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.

Our studies suggest that not every workload will achieve increased performance with a fine-grained many-task model on a GPU. Applications that can generate thousands of SIMD threads may prefer to use traditional CUDA programming techniques. GeMTC is currently optimized for executing within environments containing a single GPU per node, such as Blue Waters; but future work aims to address heterogeneous accelerator environments. Under the current configurations, users are required to write their own AppKernels. While a user may use many preexisting AppKernel templates, a system to automatically tune CUDA/C functions to AppKernels would streamline the development process. We leave this for future work.

Future work also includes performance evaluation of diverse application kernels; analysis of the ability of such kernels to effectively utilize concurrent warps; enabling of virtual warps [25] which can both subdivide and span physical warps; support for other accelerators such as the Xeon Phi; and continued performance refinement.

8. ACKNOWLEDGEMENTS

The Blue Waters sustained-petascale computing project is supported by the National Science Foundation (OCI 0725070) and the State of Illinois. Blue Waters is a joint effort of the University of Illinois at Urbana-Champaign and its National Center for Supercomputing Applications. We gratefully acknowledge support from NSF ACI-1148443 (Swift) and DOE DE-SC0005380 (ExM). Work by Katz was supported by NSF while working at the Foundation. Any opinion, finding, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the National Science Foundation. The information, data, or work presented herein was funded in part by the Starr Foundation through its generous donation to Illinois Institute of Technology. The content is solely the responsibility of the authors and does not necessarily represent the official views of the Starr Foundation. The authors also recognize Dustin Shahidehpour and Jeffrey Johnson for their contributions to GeMTC and MDLite.

9. REFERENCES

[1] I. Raicu, Z. Zhang, M. Wilde, I. Foster, P. Beckman, K. Iskra, and B. Clifford, "Toward loosely coupled programming on petascale systems," in *Proc. of 2008 ACM/IEEE Conf. on Supercomputing*, ser. SC '08. Piscataway, NJ: IEEE Press, 2008, pp. 22:1–22:12.

[2] I. Raicu, *Many-task computing: bridging the gap between high-throughput computing and high-performance computing.* ProQuest, 2009.

[3] I. Raicu, Y. Zhao, C. Dumitrescu, I. Foster, and M. Wilde, "Falkon: a Fast and Light-weight tasK executiON framework," in *Proc. of the 2007 ACM/IEEE Conf. on Supercomputing (SC'07).* New York, NY, USA: ACM, 2007, pp. 43:1–43:12.

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

[5] J. M. Wozniak, M. Wilde, and D. S. Katz, "JETS: Language and system support for many-parallel-task workflows," *J. Grid Computing*, vol. 11, no. 3, pp. 341–360, 2013.

[6] M. Hategan, J. Wozniak, and K. Maheshwari, "Coasters: uniform resource provisioning and access for scientific computing on clouds and grids," in *Proc. Utility and Cloud Computing*, 2011, pp. 114–121.

[7] D. P. Anderson, "Boinc: A system for public-resource computing and storage," in *Proc of 5th IEEE/ACM Intl. Workshop on Grid Computing.* IEEE, 2004.

[8] A. B. Yoo, M. A. Jette, and M. Grondona, "Slurm: Simple linux utility for resource management," in *Job Scheduling Strategies for Parallel Processing.* Springer, 2003, pp. 44–60.

[9] N. Desai, "Cobalt: an open source platform for hpc system software research," in *Edinburgh BG/L System Software Workshop*, 2005.

[10] IBM, "Sub-block jobs," in *IBM System Blue Gene Solution: Blue Gene/Q System Administration*, 2013, pp. 80–81, Sec. 6.3.

[11] K. Ousterhout, A. Panda, J. Rosen, S. Venkataraman, R. Xin, S. Ratnasamy, S. Shenker, and I. Stoica, "The case for tiny tasks in compute clusters," in *Proc. of the 14th USENIX Conf. on Hot Topics in Operating Systems.* USENIX Association, 2013, pp. 14–14.

[12] L. V. Kale and G. Zheng, "Charm++ and ampi: Adaptive runtime strategies via migratable objects," *Advanced Computational Infrastructures for Parallel and Distributed Applications*, pp. 265–282, 2009.

[13] S. Wienke, P. Springer, C. Terboven, and D. an Mey, "OpenACC - first experiences with real-world applications," in *Euro-Par 2012 Parallel Processing.* Springer, 2012, pp. 859–870.

[14] NVIDIA Inc., "CUDA C Programming Guide PG-02829-001 v5.5, Section 3.2.5, Asynchronous Concurrent Execution," 2013.

[15] NVIDIA Inc. , "CUDA C Programming Guide PG-02829-001 v5.5, Appendix C, Dynamic Parallelism Execution," 2013.

[16] J. Johnson, S. J. Krieder, B. Grimmer, J. M. Wozniak, M. Wilde, and I. Raicu, "Understanding the costs of many-task computing workloads on intel xeon phi

coprocessors," in *2nd Greater Chicago Area System Research Workshop (GCASR)*, 2013.

[17] Y. Zhao, M. Hategan, B. Clifford, I. Foster, G. von Laszewski, V. Nefedova, I. Raicu, T. Stef-Praun, and M. Wilde, "Swift: Fast, reliable, loosely coupled parallel computation," in *Services, 2007 IEEE Congress on*, 2007, pp. 199–206.

[18] J. M. Wozniak, T. G. Armstrong, M. Wilde, D. S. Katz, E. Lusk, and I. T. Foster, "Swift/T: Scalable data flow programming for many-task applications," in *Proc. CCGrid*, 2013.

[19] T. G. Armstrong, J. M. Wozniak, M. Wilde, and I. T. Foster, "Compiler optimization for data-driven task parallelism on distributed memory systems," ANL/MCS-P5030-1013.

[20] J. M. Wozniak, T. G. Armstrong, K. Maheshwari, E. L. Lusk, D. S. Katz, M. Wilde, and I. T. Foster, "Turbine: A distributed-memory dataflow engine for high performance many-task applications," vol. 28, no. 3, pp. 337–366, 2013, fundamenta Informaticae 128(3).

[21] NCSA, "Blue Waters User Portal," 2014, https://bluewaters.ncsa.illinois.edu/hardware-summary.

[22] J. Burkardt, "MD - molecular dynamics," 2013, http://people.sc.fsu.edu/~jburkardt/cpp_src/md/md.html.

[23] A. N. Adhikari, J. Peng, M. Wilde, J. Xu, K. F. Freed, and T. R. Sosnick, "Modeling large regions in proteins: Applications to loops, termini, and folding," *Protein Science*, vol. 21, no. 1, pp. 107–121, 2012.

[24] S. S. Hampton, P. Brenner, A. Wenger, S. Chatterjee, and J. A. Izaguirre, "Biomolecular sampling: Algorithms,test molecules, and metrics," in *New Algorithms for Macromolecular Simulation*, ser. Lecture Notes in Computational Science and Engineering, B. Leimkuhler, C. Chipot, R. Elber, A. Laaksonen, A. Mark, T. Schlick, C. SchÃijtte, and R. Skeel, Eds. Springer-Verlag, New York, 2006, vol. 49, pp. 103–121.

[25] S. Hong, S. K. Kim, T. Oguntebi, and K. Olukotun, "Accelerating CUDA graph algorithms at maximum warp," in *Proc. of the 16th ACM Symp. on Principles and practice of parallel programming*, ser. PPoPP '11. New York, NY, USA: ACM, 2011, pp. 267–276.

[26] L. Chen, O. Villa, S. Krishnamoorthy, and G. R. Gao, "Dynamic load balancing on single-and multi-gpu systems," in *IEEE Intl. Symp. on Parallel & Distributed Processing (IPDPS)*. IEEE, 2010.

[27] L. Chen, O. Villa, and G. R. Gao, "Exploring fine-grained task-based execution on multi-gpu systems," in *2011 IEEE Intl. Conf. on Cluster Computing (CLUSTER)*. IEEE, 2011, pp. 386–394.

[28] C. J. Rossbach, J. Currey, M. Silberstein, B. Ray, and E. Witchel, "Ptask: operating system abstractions to manage GPUs as compute devices," in *Proc. of the Twenty-Third ACM Symp. on Operating Systems Principles*. ACM, 2011, pp. 233–248.

[29] C. J. Rossbach, Y. Yu, J. Currey, J.-P. Martin, and D. Fetterly, "Dandelion: a compiler and runtime for heterogeneous systems," in *Proc. of the Twenty-Fourth ACM Symp. on Operating Systems Principles*. ACM, 2013, pp. 49–68.

[30] V. T. Ravi, M. Becchi, G. Agrawal, and S. Chakradhar, "Supporting GPU sharing in cloud environments with a transparent runtime consolidation framework," in *Proc. of the 20th Intl. Symp. on High performance distributed computing*, ser. HPDC '11. New York, NY, USA: ACM, 2011, pp. 217–228.

[31] M. Becchi, K. Sajjapongse, I. Graves, A. Procter, V. Ravi, and S. Chakradhar, "A virtual memory based runtime to support multi-tenancy in clusters with GPUs," in *Proc. of the 21st Intl. Symp. on High-Performance Parallel and Distributed Computing*. ACM, 2012, pp. 97–108.

[32] V. Gupta, K. Schwan, N. Tolia, V. Talwar, and P. Ranganathan, "Pegasus: coordinated scheduling for virtualized accelerator-based systems," in *Proc. of the 2011 USENIX Annual Technical Conf.*, ser. USENIXATC'11. Berkeley, CA, USA: USENIX Association, 2011, pp. 3–3.

[33] C. Augonnet, S. Thibault, R. Namyst, and P.-A. Wacrenier, "StarPU: a unified platform for task scheduling on heterogeneous multicore architectures," *Concurrency and Computation: Practice and Experience*, vol. 23, no. 2, pp. 187–198, 2011.

[34] C. Zhang, G. Han, and C.-L. Wang, "GPU-TLS: An efficient runtime for speculative loop parallelization on gpus," in *13th IEEE/ACM Intl. Symp. on Cluster, Cloud and Grid Computing (CCGrid)*. IEEE, 2013.

[35] S. Chatterjee, M. Grossman, A. Sbîrlea, and V. Sarkar, "Dynamic task parallelism with a gpu work-stealing runtime system," in *Languages and Compilers for Parallel Computing*. Springer, 2013, pp. 203–217.

[36] S. Cadambi, G. Coviello, C.-H. Li, R. Phull, K. Rao, M. Sankaradass, and S. Chakradhar, "COSMIC: middleware for high performance and reliable multiprocessing on xeon phi coprocessors," in *Proc. of the 22nd Intl. Symp. on High-performance parallel and distributed computing*. ACM, 2013, pp. 215–226.

[37] S. Lee and R. Eigenmann, "OpenMPC: Extended OpenMP for efficient programming and tuning on GPUs," *Intl. J. of Computational Science and Eng.*, 2012.

[38] T. R. Scogland, B. Rountree, W.-c. Feng, and B. R. de Supinski, "Heterogeneous task scheduling for accelerated OpenMP," in *IEEE 26th Intl. Parallel & Distributed Processing Symp. (IPDPS)*. IEEE, 2012.

[39] J. Meng, V. A. Morozov, K. Kumaran, V. Vishwanath, and T. D. Uram, "GROPHECY: GPU performance projection from CPU code skeletons," in *Proc. of 2011 Intl. Conf. for High Performance Computing, Networking, Storage and Analysis*, ser. SC '11. New York, NY, USA: ACM, 2011, pp. 14:1–14:11.

[40] M. Bauer, S. Treichler, and A. Aiken, "Singe: Leveraging Warp Specialization for High Performance on GPUs," in *Proc. of the 19th ACM SIGPLAN Symp. on Principles and Practice of Parallel Programming*, ser. PPoPP '14. New York, NY, USA: ACM, 2014.

[41] A. M. Aji, L. S. Panwar, F. Ji, M. Chabbi, K. Murthy, P. Balaji, K. R. Bisset, J. Dinan, W.-c. Feng, J. Mellor-Crummey *et al.*, "On the efficacy of GPU-integrated MPI for scientific applications," in *Proc. of the 22nd Intl. Symp. on High-Performance Parallel and Distributed Computing*. ACM, 2013.