APPROVAL SHEET

Title of Thesis: The Hybrid Task Graph Scheduler

Name of Candidate: Timothy James Blattner
Ph.D. in Computer Science,
2016

Thesis and Abstract Approved: _____

Dr. Milton Halem
Research Professor
Department of Computer Science and
Electrical Engineering

Date Approved: _____

ABSTRACT

Scalability of applications is a key requirement to gaining performance in hybrid and cluster computing. Implementing code to utilize multiple accelerators and CPUs is difficult, particularly when dealing with dependencies, memory management, data locality, and processor occupancy. The Hybrid Task Graph Scheduler (HTGS) is designed to increase programmer productivity to develop applications for single nodes with multiple CPUs and accelerators. Current task graph schedulers provide APIs, directives, and compilers to schedule work on nodes; however, many fail to expose the locality of data and often use a single address space to represent memory resulting in inefficient data transfer patterns for accelerators. HTGS merges dataflow and traditional task graph schedulers into a novel model to assist developers in exposing the parallelism and data locality of their algorithm. With the HTGS model, an algorithm is represented at a high level of abstraction and modularizes the computationally intensive components as a series of concurrent tasks. Using this approach, the model explicitly defines memory for each address space and provides interfaces to express the locality of data between tasks. The result achieves the full performance of the node comparable to the best of breed implementations of algorithms such as matrix multiplication and LU decomposition. The performance gains are demonstrated with a modest effort using the HTGS C++ API, which improves programmer productivity with obtaining that performance.

The Hybrid Task Graph Scheduler

by
Timothy Blattner

Thesis submitted to the Faculty of the Graduate School
of the University of Maryland in partial fulfillment
of the requirements for the degree of
Ph.D. in Computer Science
2016

I would like to dedicate this thesis to my brother John Blattner. His strong will and determination through his life and fight against cancer continues to be an inspiration to me. Although John is not with us anymore, he still lives on in my heart as I strive to better the world and to hopefully one day help the fight against cancer.

Additionally, I would like to dedicate this thesis to my new born son Benjamin and my wife Kimberly. Throughout my writing of this thesis, my wife was just a few months from delivering our son. During this time, both of them were my motivators to help me keep my focus and write. Shortly after I finished writing, our son was born and he is a joyful baby boy that continues to motivate me.

ACKNOWLEDGMENTS

I would like to thank Milton Halem, Yelena Yesha, John Dorband, Shujia Zhou, and Walid Keyrouz for their help and guidance through this long and arduous journey. I would also like to thank Mary Brady, the National Institute of Standards and Technology, and the Center for Hybrid Multicore Productivity Research for providing me funding throughout my work. Thank you to my family, particularly my parents for their encouragement. Most importantly, to my wife Kimberly and our new baby Benjamin, their encouragement and patience have helped me push through the many struggles and long hours while working on this research.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

Chapter 1

INTRODUCTION

1.1 Background

Hybrid clusters now play a prominent role in high performance computing; they make up four of the top ten fastest supercomputers as of Jun 2016 (TOP500 2016). These petascale clusters consist of nodes that contain one or more CPUs with one or more co-processors (Intel Xeon Phi (Intel 2015)/NVIDIA Tesla (NVIDIA 2015)). The next generation of hybrid architectures will contain fat cores coupled with many thin cores/accelerators on a single chip, as seen on Intel's Knights Landing (Sodani *et al.* 2016) and NVIDIA's DGX-1 (NVIDIA 2016b), (Foley 2014). Each single fat node can be thought of as high performance computing in the small. Programming for these nodes for performance requires special consideration for data locality and parallelism to minimize data motion while maximizing the arithmetic density of computations applied to the data (Ang *et al.* 2014).

High-level programming models are needed to aid developers with obtaining performance on these hierarchical computational systems. The model must have a clear representation for data locality and coarse-grained parallelism to be able to properly utilize both the CPUs and the co-processors. Previously, a hybrid node contained one or more CPU processors, which submitted compute intensive work to a single co-processor. Algorithms tuned for these systems require rework to scale on systems that contain multiple CPUs scheduling work for multiple co-processors. Distributing work among multiple GPUs is challenging to implement. Ideally, an algorithm's implementation will need to overlap computation with data transfers to ensure the transmission to/from all of the GPUs do not overwhelm the overall runtime. Additionally, newer interconnects such as NVLink,

now require simultaneous multi-GPU bi-directional data transfers to saturate the wider bus (NVIDIA-NVLink 2016).

The Hybrid Task Graph Scheduler (HTGS) model is one such high-level programming model that aids developers in utilizing these desktop super computers. There are four attributes that the HTGS model provides. First, the model uses coarse-grained parallelism by distributing work concurrently across computational tasks. Second, dependencies are satisfied using dataflow representations of algorithms and micro-schedulers managed by bookkeepers, which hold the global state of the computation. Third, scalability is handled by execution pipelines that prescribe data based on decomposition strategies, which can be used to scale for multi-GPU systems. Finally, the model uses an explicit memory representations to express data locality, which tunes a task graph to support memory release based on access patterns. Using these attributes, the productivity of a programmer is increased as there is a simple procedure to follow that enables high utilization for fat nodes. The primary addition that the HTGS model provides is the merging of dataflow semantics and task graph scheduling.

Dataflow is often used in digital signal processing software environments to model the interaction among heterogeneous devices. These interactions are laid out using a dataflow graph that consists of vertices that define computational functions and edges that connect vertices based on data dependencies. HTGS uses the dataflow graph semantics to assist in modularizing an algorithm's computationally intensive and logical operations to formulate concurrent execution assuming data rates can pipeline within the graph.

Task graph schedulers define a mechanism for scaling computation among multi-core CPUs. Using an API, such as OpenMP tasks, a task queue is created by inserting tasks with various descriptors that describe data dependencies between tasks. The task queue is ordered based on these data dependencies and a CPU thread pops the task from the queue and begins processing that task. The interaction between the tasks within the queue forms a task graph; however, the task graph representation is lost in the implementation. There are post-processing tools available to extract the task graph and visualize the path of the execution. The notion of a shared queue and thread pool are adapted into HTGS and are distributed among a task graph, which is formulated from the dataflow representation. The runtime system that was developed for HTGS distributes threads to tasks rather than a pool of threads waiting to be assigned a task. This allows for a thread to bind to a specific task and all contexts that are created by that task will be bound to that thread. This enables

for accelerators to be bound to specific tasks without the need to context switch the GPU context between threads. Threads are awakened when data is available for its task and begin processing the task with that data. If a task is computationally intensive then multiple instances of that task are created with each new instance bound to a separate thread. Each task shares an input and output thread safe queue among the multiple threads to safely distribute data among the threads.

Combining dataflow and task graphs in HTGS provides two main attributes that impact programmer productivity. First, the analysis phase of representing an algorithm in HTGS is not lost in the implementation. This means that all of the decisions that were made during implementation can be easily mapped back to the analysis phase as the physical task graph is created and scheduled within the implementation. Second, the threading model enables efficient scalability for multi-core CPUs and accelerators as threads are bound to physical tasks, which can then instantiate contexts for accelerators.

Traditional approaches to parallelism requires significant programmer effort to fully utilize complex hybrid systems; identifying parallelism, handling synchronization, and managing data distribution and locality (Chamberlain, Callahan, & Zima 2007). The effort spent analyzing the parallelism of an algorithm often does not map well into the implementation for hybrid systems. This is often leads to mishandling of data motion and memory capacities, which ends up with poor CPU and GPU utilization.

The issues that programmers face is demonstrated in our previous work (Blattner 2013). We observed that directly porting compute-intensive functions from a sequential implementation to the GPU did not yield sufficient performance improvements. This was due to the low utilization of the GPU, a result of the default synchronous approach to CPU-GPU memory copies. This led us to develop an implementation based on hybrid pipeline workflow systems (Blattner 2013), which are designed to keep GPUs and CPUs busy while overlapping data movement with computations. This approach stays within memory limits and operates across multiple GPUs, but requires a significant amount of programmer effort to implement. The HTGS model is a generalization of the hybrid pipeline workflow system, which combines elements from dataflow semantics and task graph scheduling.

1.2 Hybrid Pipeline Workflows

The hybrid pipeline workflow system is a technique to schedule tasks on both CPU and GPU resources simultaneously, while effectively managing dependencies and memory between tasks. The system also overlaps memory transfers with computation, keeps multiple GPUs occupied, and utilizes all of the available compute resources. A hybrid workflow system is implemented by setting up computational tasks, bound to pools of threads, and connecting them with FIFO queues. Dependency management is done by setting up bookkeepers between computational tasks, which manage the global state of the algorithm. Incorporating this methodology into an implementation was a challenge; however, there were significant performance gains.

The performance of using hybrid pipeline workflows is demonstrated in image stitching (Blattner *et al.* 2014). Image stitching is used to address the scale mismatch between the dimensions of the microscope's field of view and the plate under study. To image a plate, a motorized stage acquires a grid of overlapping images. The positions of these images are computed by stitching neighboring tiles together. The positions are used to construct the image mosaic. The algorithm consists of three compute stages: (S1) the fast Fourier transform (FFT) of an image, (S2) the phase correlation image alignment method (PCIAM) (Kuglin & Hines 1975b) that acts on two neighboring images' FFTs, and (S3) the cross correlation factors (CCFs) between two neighboring images focused around a maximum intensity point identified from the PCIAM, as shown in Figure 1.1. These operations are done for each pair of neighboring images within a grid of images.

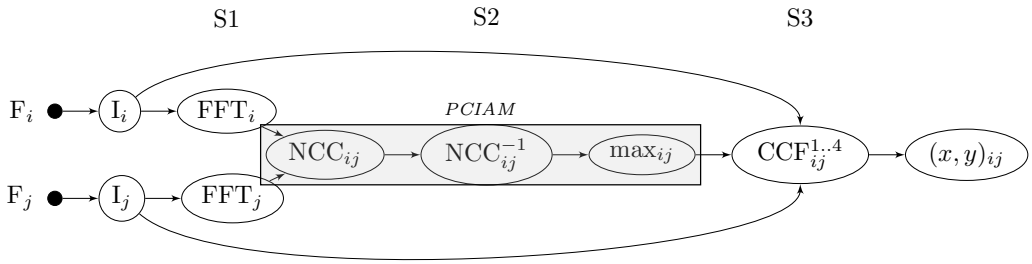

Figure 1.1: Image stitching data flow

Our implementation of image stitching started with a sequential CPU implementation,

which was ported to the GPU (*Simple-GPU*). In *Simple-GPU*, NVIDIA's CUDA (NVIDIA CUDA 2011) is used to process images and data is copied to the GPU as needed. The results showed a 14% speedup compared to the sequential CPU implementation. Data motion between co-processors and CPUs dominates the performance. Using the existing compute kernels from *Simple-GPU* and scheduling their invocations in a hybrid pipeline workflow (*Pipelined-GPU*) that properly manages memory and overlaps computations with data motion improves the *Simple-GPU* implementation by $\mathbf{24}$x. Adding a second GPU to the pipeline improves the performance by an additional 1.86x. The implementation of the *Pipelined-GPU* requires a significant amount of programming effort to prevent race conditions, satisfy dependencies, and maintain memory limitations.

Figures 1.2 and 1.3 show 1 and 0.2 second profiles, respectively, of the *Simple-GPU* implementation. The figures were generated using NVProf, which gathers profiling data during execution (NVIDIA 2016a). The *Simple-GPU* has numerous gaps in the GPU computation, which indicate that the GPU is idly waiting for data copies to be sent to the GPU. These gaps result in low GPU utilization.

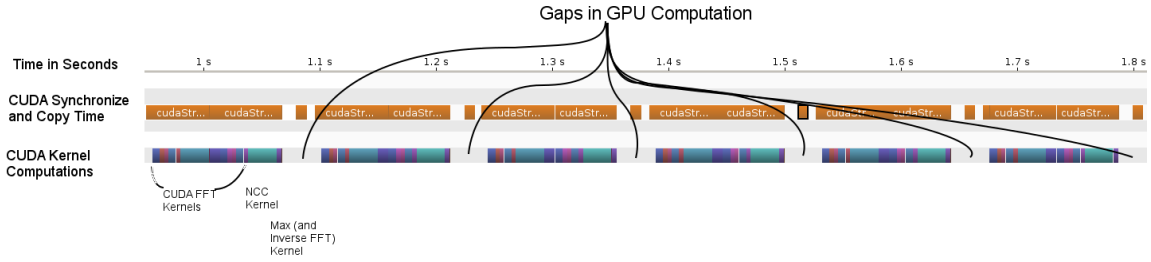

Figure 1.2: One second profile of image stitching implementation using one thread with GPU (*Simple-GPU*) profile. Demonstrates multiple gaps between computing relative displacements between neighbors.

Next, the hybrid pipeline workflow implementation is profiled. In Figures 1.4 and 1.5 we show the same 1 and 0.2 second profiles. In the hybrid pipeline workflow profiles, we see a sharp contrast in the scheduling behavior compared to the *Simple-GPU* profiles. Using hybrid pipeline workflows allowed the execution to continue as soon as data is available. Using this method, the next iteration of data can be fetched, enabling the GPU to keep busy, overlapping computation with data motion. Additionally, through the use of streams, multiple compute kernels were able to overlap enabling better instruction-level

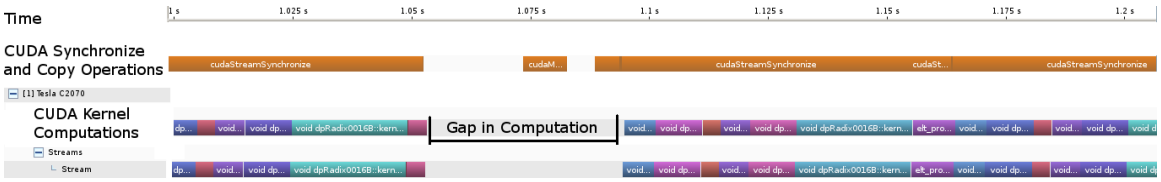

Figure 1.3: 0.2 second profile of image stitching implementation using one thread with GPU (*Simple-GPU*) profile. Zooms in on one of the gaps from Figure 1.2

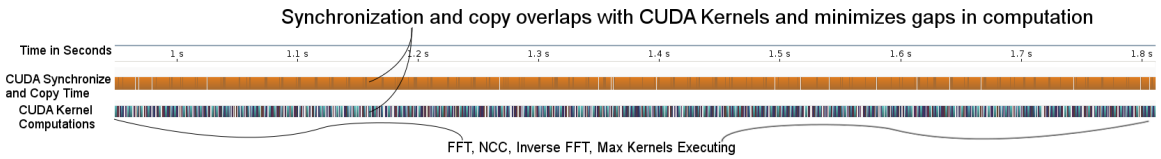

Figure 1.4: One second profile of image stitching implementation using hybrid pipeline workflows with GPU (*Pipelined-GPU*) profile. Shows overlapping of data transfer with compute.

Figure 1.5: 0.2 second profile of image stitching implementation using hybrid pipeline workflows with GPU (*Pipelined-GPU*) profile. Zooms in on a section of the profile from Figure 1.4

parallelism on the GPU. Enabling this component assumes that the GPU contains enough resources (registers/shared memory) to schedule work from two separate kernels on the same GPU.

Using the hybrid pipeline workflow system abstracts the GPUs into pipelines. By

increasing the number of pipelines from the one to two enables multi-GPU computation. Each pipeline is bound to a separate GPU and executes concurrently. Using the execution pipeline, data is distributed to each GPU and dependencies from halo regions are copied between GPUs using direct peer to peer copies. Figure 1.6 shows the same 1 second profile as before, except using two NVIDIA Tesla C2070s. This profile shows the concurrent multi-GPU execution, keeping multiple GPUs busy.

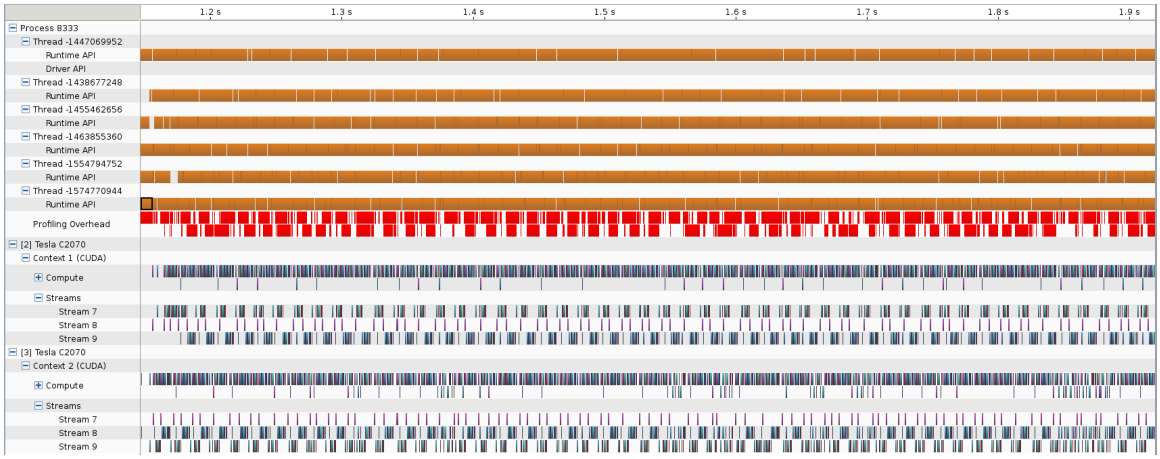

Figure 1.6: 1 second profile of image stitching implementation using hybrid pipeline workflows with execution pipelines across 2 GPUs.

Using this implementation as a baseline we incorporate the benefits of the hybrid pipeline workflow into the design of the hybrid task graph scheduler model.

1.3 Motivation

Implementing and scheduling applications on hybrid systems with multiple GPUs is a challenging task, particularly when trying to minimize data movement and maximize the amount of computations done on that data, all the while, staying within memory limits. Many task scheduling models schedule work on hybrid systems by using work stealing through a shared prioritized work queue. This approach typically requires a global address space and results in inefficient data transfer patterns. The HTGS model exposes data locality to the programmer to provide more fine-grained control over when and where data gets allocated and copied between CPUs and GPUs at a higher level of abstraction.

Also, HTGS explicitly provides an interface for scheduling work across multiple GPUs by encapsulating a task graph into an execution pipeline. In an execution pipeline, a task graph is duplicated and data is distributed evenly between multiple GPUs based on decomposition rules. This approach effectively pipelines tasks to overlap computations with memory transfers, while operating on multiple GPUs.

The benefits of this approach is shown with hybrid pipeline workflows; however, creating hybrid pipeline workflows is complex and time consuming. The HTGS model is designed to aid programmers with implementing hybrid pipeline workflows. This is achieved by combining dataflow and task graph schedulers. Dataflow is a representation that is used to expose the parallel attributes of an algorithm, which connects computation nodes with data dependency edges. Each node can be executed concurrently assuming dependencies are satisfied. Task schedulers are incorporated into each of these computational nodes, such that each node will have one or more threads processing data that is consumed. Typically task schedulers use a single work queue that distributes tasks to threads. HTGS alters this design and uses dataflow semantics to send data to threads, which are bound to tasks. Combining dataflow and task schedulers into one unified model is the essence in what the HTGS model provides for programmer productivity.

Using the HTGS model, we implemented the HTGS C++ API, which provides functions to create task graphs and includes a runtime system to schedule on hybrid collections of compute resources (i.e., CPUs and GPUs). The task graphs that HTGS helps build handle dependencies, manages memory in multiple native address spaces (CPU/GPU), scales to multi-GPU systems through execution pipelines, and overlaps data motion with computations. Every task created through HTGS exposes the computational resources and automatically binds tasks to physical hardware. This approach is used to help developers utilize single fat nodes with multiple CPUs and accelerators.

We will demonstrate the HTGS model on three algorithms: (1) Image processing through an implementation of image stitching using Kuglin and Hines phase correlation image alignment method (Kuglin & Hines 1975b), (2) Matrix multiplication, and (3) Linear algebra with LU factorization. The HTGS model and API aims to improve programmer productivity, with a modest effort, to obtain high performance, particularly for scheduling on hybrid multi-GPU systems.

1.4 Contents

In the next Chapter, we provide our thesis statement, problem description, and contributions. In Chapter 3 we present various related works in the field of task-based schedulers and other methods for parallelization. Chapter 4 describes the proposed HTGS model. Chapter 5 describes the HTGS C++ implementation. In Chapters 6, 7 and 8, we present three case studies that demonstrates the impacts of using the HTGS model and implementation compared with highly tuned libraries. Finally the conclusions are in Chapter 9.

Chapter 2

THESIS STATEMENT

We propose the Hybrid Task Graph Scheduler (HTGS), which improves programmer productivity by implementing and optimizing parallel algorithms to fully utilize single fat nodes consisting of many-core CPUs and multiple accelerators, while effectively managing dependencies, overlapping data motion with computation, and expressing the locality of data.

HTGS defines a model that builds upon two paradigms; dataflow semantics and task graph schedulers to build hybrid pipeline workflows. Using this approach, HTGS exposes the parallel nature of the algorithm and modularizes the execution, which improves programmer productivity for implementing and optimizing parallel algorithms. HTGS expresses algorithms as a series of highly optimized computational tasks that execute concurrently assuming data dependencies are satisfied. HTGS aims to fully utilize machines with multiple CPUs and multiple accelerators. This is achieved through a novel recursive execution pipeline task that scales an HTGS task graph to multiple accelerators. HTGS is a formalization of the extensive work to build hybrid pipeline workflow systems that we presented in (Blattner 2013).

We will demonstrate the HTGS model using our HTGS C++ API implementation. Using the API, we will apply HTGS to three distinct algorithms: (i) an image processing application that stitches thousands of micro-biological images together to produce large format images, (ii) matrix multiplication, and (iii) LU decomposition used for solving large systems of linear equations. Each of these algorithms are approached with a modest effort over a reasonable amount of time and follow the HTGS model and framework for exposing the parallel nature of the algorithms. By using the proposed model and API we will show performance gains comparable or better than the standard implementations that are used in

high performance computing.

CPUs and co-processors, each contain separate address spaces, which have high latency and low bandwidth for shipping data. These data transfer costs cannot be ignored. Traditional task scheduler approaches to CPU and co-processor interoperability often fail to expose scheduling behavior to improve data locality. In order to utilize high performance computers, it is necessary to pay careful consideration to the location of data and ways to express that locality to ensure the data resides next to the compute hardware for as long as needed. These concerns are multiplied by the need to micromanage differing memory capacities, such as between CPUs and co-processors, which forces algorithms to operate using out-of-core techniques. With the HTGS model, algorithms are represented in a way that exposes these data access behaviors to allow restructuring of scheduling at a high level, with the aim of improving locality for computational hardware.

2.1 Problem Definition

HTGS uses dataflow and task scheduler semantics to create a model to effectively implement algorithms for a single fat node. Dataflow is used to express an algorithm as a series of computational tasks that execute concurrently assuming data dependencies are satisfied. Task schedulers are used as the execution model. HTGS expands and combines these definitions by incorporating the task schedulers into the dataflow representation and adds additional tasks, defined within the HTGS framework, to ensure data dependencies are satisfied. In HTGS, the task graph is built from a dataflow representation where each task is connected by dependency edges. HTGS then expresses a work queue within each task that stores data. Each work queue has a pool of threads, which are bound to a specific task that consume data from the work queue. This formulation creates a pipeline where tasks concurrently consume and produce data. By binding threads to tasks allows for accelerator contexts to be bound to the thread, preventing unneeded GPU context switching.

Hybrid workflows can be considered a method for scaling software by making use of task graph pipelining. This method is effective for overlapping data motion with computational tasks, while keeping fat nodes busy.

Models for exploiting high performance computations also require careful consideration of data locality, dependencies, and parallelism. Keeping these attributes in mind aid in maximizing the concurrency of scheduling independent compute kernels.

We will expand on these concepts to show that using the HTGS model and API can simplify the design and implementation of algorithms to handle scaling, data locality, dependencies, and parallelism for compute nodes with multiple CPUs and accelerators.

2.2 Contributions

Abstract Model for Single Fat Nodes

HTGS is a high-level programming model that combines dataflow semantics and task scheduling to assist programmers to represent hybrid pipeline workflows. The model distributes work among optimized computational tasks, manages dependencies through micro-schedulers, handles scalability with execution pipelines that distribute data based on decomposition strategies, and explicitly handles data locality with memory managers. Through these attributes, HTGS executes efficiently on single fat nodes making full use of the highly parallel architectures within.

This model is further supported through the implementation of matrix multiplication and LU decomposition using the HTGS C++ API and a modest effort. The results show that, HTGS is able to perform as good or better than the best performing implementations available.

Abstract Model for Data Locality

Keeping data close to the compute hardware is the most important attribute when scheduling compute intensive algorithms. If data needs to be shipped to/from hardware multiple times, then the cost of shipping the data will often outweigh the benefits gained for using that hardware. Many algorithms employ techniques to schedule data in a way that maximizes the locality of data; however, due to differing capacities between traditional CPUs and accelerator memories, it is a difficult task to implement. It becomes especially challenging when dealing with multiple accelerators each with their own address spaces.

The HTGS framework defines HTGS memory managers to assist with data locality and manages each address space independently, whether it is for traditional or accelerator memories. Using this approach, the data access patterns are exposed allowing for better decisions to be specified within HTGS for scheduling the data to improve the data locality. These decisions are further tuned through HTGS data release rules applied to any data allocated by memory managers. These rules provide

a high-level abstract model for how the memory is used as it flows through the task graph. In addition, HTGS can export the compute profile, visualizing the task graph, exposing where the bottlenecks exist. Using this information, tuning and optimizations can be applied to allow for better behavior for data locality. This type of analysis is demonstrated for LU decomposition in Chapter 8.

Scaling with Execution Pipelines

Fat nodes contain multiple co-processors/accelerators. Scheduling work to utilize these accelerators without a high-level abstract model is very challenging. Each accelerator has its own memory address space and data must be sent to/from each device. One of the main contributions that the HTGS framework defines is its ability to represent a task graph and scale that task graph across multiple accelerators. First, an HTGS task graph is partitioned into an accelerator graph and a main graph. The accelerator graph is encapsulated into a recursive HTGS execution pipeline task. Within the execution pipeline task, the accelerator graph is duplicated, one per accelerator on a system. The execution pipeline task distributes data using custom decomposition rules among the accelerators, which concurrently execute on the data. The execution pipeline task is then inserted back into the main graph.

Each graph copy is a mirror image of all the components of the original graph such as bookkeepers, memory managers, and computational tasks. Each memory manager binds to the physical device allowing allocation and data copying to process on its designated accelerator. Using this design philosophy, HTGS enables operation on a variety of machines, which is enabled by using allocation routines specific to the device the task is operating with.

Improving Parallel Programming Efficiency

Writing code that scales to multi-core CPUs and accelerators often requires a significant amount of programming effort. Issues arise such as dealing with race conditions, deadlock, or load imbalance. Often the parallel algorithm analysis does not map well into the implementation, and the effort spent breaking down an algorithm is lost. The HTGS model is an attempt at bridging the gap by directly mapping the analysis phase into implementation for parallel algorithms.

In HTGS, there are five phases that are used to aid parallel programmers and improve their efficiency with implementing parallel algorithms. First, identifying

the computational bottlenecks and the data patterns within the parallel algorithm. Then, using this analysis, design a dataflow representation where nodes represent computational steps and edges are data dependencies or parameters. From the dataflow representation, a task graph is designed that contains bookkeepers that manage dependencies, and memory managers to handle data locality. The task graph representation is then implemented using the HTGS C++ API. Finally, using profiling tools within the HTGS API, analyze the behavior of the graph and identify the tasks that are overwhelming the computation. This leads to redesign of the graph to best evenly distribute the work and improve locality. Approaching an algorithm in this way minimizes the amount of analysis that is lost during implementation. Additionally, the implementation can map back to the analysis allowing efficient understanding into the design decisions made.

The decisions made for one particular algorithm can be applied to other algorithms that share similar data access patterns. This allows for efficient algorithm implementation as the data access decisions can then be interpreted and applied into implementations of other algorithms.

Pipelining

The HTGS framework consists of a series of tasks connected based on data dependencies. Each of these tasks execute concurrently and are bound to a pool of threads. If a task is computationally intensive, then that task is assigned multiple threads to aid in processing data. This thread pool acts to accelerate the task to improve the task's consumption rate. With this method of coarse-grained parallelism, the task graph represents multiple producers and consumers that modularizes an algorithm and enables concurrent execution of its tasks. By assigning some tasks for computationally intensive operations and others to data motion, the execution of an algorithm in HTGS overlaps components such as disk I/O or PCI express I/O with compute. Through this pipelining, HTGS utilizes the multi-core CPUs, multiple accelerators, and distributes the workload using customizable thread configurations that are assigned based on the consumption and production rates of the computationally intensive tasks.

Memory Management

Pipelining is an excellent approach for parallelizing an algorithm, especially if the datasets fit into memory. When the data for an algorithm does not fit into memory, then

a pure pipelining approach will cause issues with running out of memory. The HTGS framework adds memory management into the task graph explicitly to help throttle the execution of the task graph. In addition, the memory allocated contains state that can be used to represent the access patterns to enable better locality. Using this methodology, any two tasks within a task graph can be connected with a memory edge, which is managed by a memory manager. The memory manager is a separate task whose purpose is to update the state of the memory received, which then determines when the memory can be released/recycled. The memory manager initializes a fixed sized pool of memory that is used during execution. Using this memory pool, the memory manager produces memory for the task that is requesting memory. If the memory pool is empty, then the task requesting memory will have to wait for another task in the graph to release the memory. This method throttles the graph based on these memory pools and prevents the system to stay within memory limits across multiple address spaces.

The memory managers within HTGS are bound to specific address spaces. If the memory resides on an accelerator, then the memory manager will allocate and distribute memory from that accelerator's address space. This design allows the task graph to represent each address space independently and provides fine-grained control over the locality of the data and when to copy between the various address spaces.

Memory Management within Execution Pipelines

Execution pipelines are used to create copies of task graphs to scale among multiple accelerators. Each memory manager within these task graphs are copied. The memory manager copies are bound to separate accelerators and allocate from their appropriate accelerator memories. The execution pipeline task distributes data among these accelerators using decomposition rules that split data into separate regions, each accelerator will operate on their designated region. If there is overlap among the regions, such as a halo, then memory may need to be shared among multiple address spaces. The memory manager is aware of the address space it is allocating. If one memory manager receives memory from the wrong address space, then that memory is forwarded to the memory manager that is processing that address space. This service is handled automatically from within the HTGS model implementation.

Overlapping Data Motion with Compute

Moving data to/from compute hardware is difficult to manage. This is particularly true when dealing with single fat nodes that contain multiple accelerators each with their own address spaces. HTGS uses its pipelining methodology to simplify separating the data motion operations with the computational tasks. Through this separation, data motion and computational tasks operate concurrently, which overlaps the cost of shipping the data with the computational operations. These components are co-dependent such that if the pipeline cannot be filled then the benefits of overlapping will be minimal.

High Performance Implementation

The HTGS model and framework is designed for high performance super computers. The HTGS C++ API is an implementation of the model and framework. The API is high performance and operates efficiently using lightweight monitor-based locks, zero-copy pointer-based data structures, and efficient use of high-level C++11 threading. In Chapters 6, 7, and 8 we demonstrate similar or better performance of the HTGS API compared with the standard implementations that are used in high performance computing. Chapter 5 will go into more details with regard to the C++ API implementation.

Mapping Analysis to Implementation

The main methodology for parallelizing an algorithm involves identifying the independent operations and the locality of data. Understanding these components allows for operations to be done in parallel all-the-while keeping the data local to the compute hardware. This is particularly important for working on accelerator systems. The analysis phase often does not map very well into the actual implementation. In addition, it is challenging to map the implementation back to the analysis. The HTGS model efficiently maps the analysis phase into the implementation and back. This is achieved through the intermediary steps of dataflow and task graph representations. The dataflow describes the concurrent operations and basic dependencies, whereas the task graph describes complex dependencies that is micro-managed through bookkeepers, and data access patterns from scheduling and memory managers. Using the HTGS methodology allows for more productive development of parallel algorithms that can be iterated on efficiently to improve parallelism and data locality.

Customizable and Open

The HTGS C++ API is implemented using an object-oriented approach. The API is split into two components; (1) Core API and (2) User API. The core API implements the low-level features of the HTGS model, such as thread safe queues and memory managers. Whereas the user API contains the high-level routines that programmers use to implement their task graphs. The majority of routines within HTGS are abstract and are implemented using object-oriented design. For example, execution pipelines, memory managers, CUDA tasks, and bookkeepers all implement and use the task interface. The task interface is inserted into task graphs, which are connected through sub-routines defined within the task graph object. All of these functions are exposed for programmers to add new abilities or improve upon existing components. Additionally, the runtime system is decoupled from the task graph representation allowing further customization with alternate threading models.

The implementation of the HTGS model is customizable and is open source. The user API can be used to represent most algorithms, but if advanced users need more functionality, then the core API can also be altered. Additional routines that bridge the gap between the core API and the user API are also provided, such as the custom edge interface, which can be used to describe a new type of edge connecting two tasks, such as the memory manager edge.

Availability

The HTGS C++ API is freely available for download now at `https://pages.nist.gov/HTGS/` or on github at `https://github.com/usnistgov/htgs`. The HTGS tutorials are also freely available at `https://github.com/usnistgov/HTGS-Tutorials`. The full documentation is available on-line at `https://pages.nist.gov/HTGS/doxygen/index.html`. Both the HTGS tutorials and HTGS API are open source. The tutorials include all implementations of matrix multiplication and LU decomposition shown in Chapters 7 and 8.

Light Weight

The HTGS C++ API is implemented using a series a header files. To use the API, developers include the header files within their code. The code base is implemented in 4000 lines of code across 37 files. The API runs efficiently with minimal overhead. Large task graph are constructed in < 100 ms and data is scheduled between tasks in < 1 us.

Profiling and Debugging HTGS Task Graphs

Profiling and debugging of task graphs is difficult without tools. HTGS provides functionality to profile and visualize HTGS task graphs. Using these tools, each task within the graph can output useful information about how it performed such as compute time, wait time, lock time, and the maximum size that its input queue achieved. Using this information each task can be tuned to improve utilization. In addition, these attributes can be visualized by exporting the graph to a Graphviz dot file (Gansner & North 2000). The dot file can then be parsed into an image where nodes represent tasks that are color coded based on profiling data. This enables the graph to be visually analyzed to identify issues. In addition, the visualization is an excellent debugging tool to visually observe the graph during construction and can be generated during execution to identify locations of deadlock. The HTGS C++ API implements these tools and are available now.

HTGS Community

Creating a system that scales and gains performance with modest a effort is useful, but a community of developers need to use this system. A developer public forum has been created; `https://groups.google.com/forum/#!forum/htgs`, to encourage collaboration and communication. The end goal is to create a repository of task graphs, tasks, data representations, and libraries to assist development of new complex algorithms with minimal effort within HTGS. This forum is newly created, but expresses the intent of moving forward with HTGS to expand and garner support from universities and industry.

Chapter 3

RELATED WORK

Implementing parallel algorithms in distributed environments poses challenges with data locality, fault tolerance, load balancing, heterogeneity, and coding complexity. There are many programming interfaces that provide tools for programmers to function around these issues. We classify the interfaces into four categories: (1) Map reduce frameworks, (2) Dataflow graphs, (3) Task graphs, and (4) Concurrent Collections. Various implementations of Map reduce frameworks are used for processing embarrassingly parallel algorithms. Dataflow graphs represent algorithms as graphs where nodes are logical or computational function and edges are data flow. In a dataflow graph, the graph interacts with a scheduler that issues work for nodes based on some ordering and dependencies. Task graphs have similar qualities of dataflow graphs, except the scheduler is inherently defined within the graph representation for an algorithm by using a shared task queue that is reordered based on data dependencies. Threads pop and execute on tasks from this queue. Concurrent collections use a multiple producer, consumer model, which is formulated by defining operations and ordering to form a CnC graph. These are similar to task graphs, except there is an explicit representation of the dependencies and data.

Each of these categories have different mechanics for handling the challenges of distributed environments. HTGS focuses on the issues of data locality, load balancing, heterogeneity, and coding complexity on a single fat node with an emphasis on multi-GPU systems. Distributed computing is left for future work. Fault tolerance can be implemented through check-pointing, but there is no explicit design to automate handling resiliency. In this section we look at different programming interfaces within each of the three categories.

3.1 Map Reduce Frameworks

The first presentation of Map reduce (Dean & Ghemawat 2008) provides an interface for representing two functions, a map and a reduce function that operate on data sets distributed across multiple processors. This enables programmers to avoid writing explicit parallel code. This approach was first proposed in 2002 by Google researchers and has been integrated and adapted to form new programming interfaces that aim to simplify and expand the map reduce paradigm. Map reduce provides a method to direct computation to the data without worrying about data movement between nodes in a cluster, all the while processing disk I/O in parallel. Below is a list of various implementations and expansions of the map reduce framework.

3.1.1 Hadoop

Hadoop is a java-based implementation of map reduce from Yahoo (Shvachko *et al.* 2010). The framework uses the hadoop distributed file system (HDFS). In HDFS, data is evenly distributed and duplicated among nodes in a hadoop cluster. Map and reduce functions are sent to the nodes that own the data and are processed in a distributed fashion. This system requires that an algorithm fits into the map reduce framework. Hadoop scales well for very large clusters and is fault tolerant through redundant storage and reassignment of work on node failure.

While many algorithms may fit into hadoop, there are a subset of algorithms that are not well suited for map reduce, such as gaussian elimination, iterative methods, n-body problems, etc. Through the use of task graphs, HTGS enables programmers to represent many of the algorithms that cannot be easily mapped to hadoop. However, a mapping task that links directly into HDFS could be integrated into HTGS, thus making HTGS into a map reduce framework.

3.1.2 Spark

Spark is an expansion of HDFS and hadoop (Zaharia *et al.* 2010). The main contribution is the introduction to an abstraction called resilient distributed datasets (RDDs). RDDs are read-only objects and enables faster processing of iterative jobs that required shared data among hadoop nodes. The RDDs provide a mechanism to quickly

broadcast and accumulate data such as lookup tables very quickly. Adding these to hadoop improved performance by an order of magnitude for problems that benefited from this in-memory data structure.

RDDs are an excellent interface for sharing state information while maintaining resiliency. HTGS currently handles state through bookkeepers, although that state is not effectively distributed and is not resilient. Approaches for handling resiliency and sharing state information efficiently are being analyzed, with RDDs as a possible solution for distributed state across bookkeepers within HTGS.

3.1.3 Map Reduce in MPI

Map reduce in MPI is a C implementation that uses the message passing interface (MPI) to handle commmunication between nodes in a cluster (Plimpton & Devine 2011). This study presents the map reduce framework using MPI as the basis for data communication. The results show an improvement in performance compared to hadoop; however, it does not provide fault tolerance or data redundancy. There are many variations of map reduce that have been implemented that use different methods of communication. Map reduce in MPI is one such example.

3.2 Dataflow Graphs

Dataflow is a computational model that is often used in digital signal processing (DSP) software environments. HTGS incorporates dataflow semantics to assist in identifying independent computationally intensive components prior to constructing task graphs. Each node within dataflow represent logical or computational functions and edges represent the flow of data. (Dennis 1974) (Lee & Parks 1995) The fundamental difference between HTGS task graphs and dataflow is with scheduling. For example, one dataflow model represents a node as an actor whose functionality is fired based on input tokens and firing rules. Each actor in a dataflow graph is invoked by the scheduler upon successful satisfaction of actor invocation rules, which could be actor interface (i.e., edge, port) requirements, or other specific per application requirements. There is a distinct separation of the dataflow scheduler and dataflow graphs. The distinct separation between the dataflow graph scheduler and the underlying dataflow graph is a key aspect of dataflow based modeling of applications. Such separation allows the dataflow graph scheduler

to integrate other runtime measurements (i.e., memory, bandwidth, latency) into the scheduling mechanism of the dataflow graph conveniently. (Pino, Bhattacharyya, & Lee 1995) HTGS has similar constructs, but defines the scheduling inherently within the each task graph. Each task is enabled as soon as data is available on an edge. To handle dependencies HTGS provides a bookkeeper task, which acts as a micro-scheduler. The bookkeeper is added into the graph and processes dependency rules, which may produce data for other tasks. In essence, HTGS contains an inherent scheduler based on the structure of the task graph and multiple bookkeepers operating within the task graph using dependency rules. The bookkeepers are used to maintain the state for a computation.

Another major difference between dataflow and HTGS is their methodologies for threading. Many dataflow models allocates actors or sets of actors to a particular thread based on dependencies and parallelization properties. This parallelization is done either within the scheduler or prior to execution. This step is a form of graph decomposition with the aim to minimize dependencies between sub-graphs to maximize pipelining the computation and logic for an algorithm. (Sane 2011) On the contrary, HTGS binds threads to tasks. Each task is bound to one or more threads, which is defined when a task is created. The threads act as a work pool to process data for a task in parallel. For GPU tasks, one thread is used to schedule work for the GPU that the thread/GPU task is bound, which prevents unnecessary GPU context switching.

3.2.1 Heterogeneous Dataflow using Anthill

Anthill is a framework to aid in developing parallel applications. (Teodoro *et al.* 2012) Algorithms are represented as dataflow models and implemented using filter-streams. In the filter-stream model, actors/tasks are represented as filters, and data flow as streams. Anthill spawns the instances of each filter across multiple nodes in a cluster and automatically handles run-time communication among the instances. Each of these filters can be executed independently providing excellent parallelism. The duplication of filters across multiple nodes contains some similarities to HTGS execution pipelines. The main difference is an execution pipeline holds a sub-graph, and the entire sub-graph is duplicated, one per GPU. Whereas anthill duplicates a single filter/task and distributes the distributed tasks among nodes. One aspect that may be adapted to HTGS is the use of scheduling based on the run-time of each task. In some instances if there are two tasks that are executed on

two different architectures, it may be more beneficial to manage how data is sent to each architecture. If an HTGS graph contains these options, it would be interesting to analyze the impact of sending data based on run-time performance of each task. Anthill provides this level of scheduling by profiling each filter prior to execution and determining an effective scheduling of the dataflow graph based on the execution times of each heterogenous filter. Currently HTGS relies on the programmer to distinguish which task is more effective for a given architecture.

3.2.2 Qilin

Qilin provides an approach that takes a dataflow representation, shown in directed acyclic graphs (DAGs) and automatically maps the computations to processing elements using run-time adaption. (Luk, Hong, & Kim 2009) To process applications at run-time, Qilin dynamically compiles and generates Intel Threading Building Blocks (TBB) and CUDA source code on the fly. The primary steps in a Qilin dynamic compilation are: (1) building DAGs using the Qilin API, (2) determine mapping from computations to processing elements, (3) perform optimization on DAGs, and (4) code generation. This approach has a number of difference to both HTGS and traditional dataflow models. First, using the Qilin API, existing API routines for common computationally intensive functions must be defined within Qilin. These routines provide Qilin with the information needed to generate the code for GPUs or CPUs. Second, the use of code generators aids in simplifying programming complexities, but can also cause additional overhead as the code generator is executed at run-time.

3.2.3 Kaapi

Kaapi is a runtime scheduler that implements a work-stealing algorithm for dataflow programs on a cluster of multi-processors. (Gautier, Besseron, & Pigeon 2007) To manage data dependencies between tasks a global address space is used. The extension ”XKaapi” (Gautier *et al.* 2013) enables execution on multi-CPU and multi-GPU architectures, while maintaining similar functionality with Kaapi. Within XKaapi, scheduling is accomplished with locality-aware work stealing and asynchronous task execution. Sequential code is annotated to create tasks that are scheduled using the XKaapi runtime system. To execute across multiple architectures, task versioning is used such that different versions

of the same task are implemented for each architectures. Locality-aware scheduling is accomplished by adding meta-data to work that indicates the location of the data. Kaapi is not a traditional dataflow representation and shows many similarities to task graph implementations. HTGS has a similar design philosophy as XKaapi, but has uses a different approach to the problem of locality-aware programming. Both APIs emphasize locality-aware scheduling and asynchronous task execution. The benefit of this approach is to effectively hide the PCI express data transfer bottleneck and to better utilize the GPU. The main difference between the two is the representation of multi-GPU scheduling and GPU task scheduling. In HTGS a task graph is bound to a particular GPU. For multi-GPUs that task graph is encapsulated into an execution pipeline, which duplicates the entire graph and binds each sub-graph to a different GPU. With XKaapi, each GPU is scheduled for individual tasks based on the location of data. This approach increases the complexity of the scheduler as the scheduler must concern itself about the location of data at each task step. This may cause additional memory copies if the data resides in the wrong location or having to pass the data to the correct task. In HTGS a sub-graph that receives data will not be copied to another sub-graph unless there is a dependency that needs to be satisfied (such as ghost regions). The programmer is responsible for distributing data to each sub-graph to minimize the size of a ghost region and minimize the need to copy between regions.

3.2.4 TensorFlow

TensorFlow is an interface for expressing machine-learning algorithms using a dataflow-like model to execute on a variety of heterogeneous systems. (Abadi *et al.* 2015) The computation within TensorFlow is represented as a directed graph consisting of nodes. The TensorFlow graph contains similar attributes as HTGS in that each node represents computation, whereas in TensorFlow some nodes are modified to maintain persistent state for things like branching and looping. However, the HTGS representation is mapped directly into the implementation, whereas TensorFlow loses that analysis step. Additionally, the behavior for scheduling and threading for parallel algorithms is very different, this is mostly due to TensorFlow being designed specifically for machine learning projects. Each node in TensorFlow has zero or more inputs and zero or more outputs. Data that flows between nodes are called *tensors*. Ordering is enforced using control dependence to ensure a source node must finish executing before the destination node begins. The

primary API within TensorFlow contains high-level routines that can be prescribed to work on various components within the graph. These high-level routines contain various implementations that are compiled for heterogeneous architectures such as computers or GPUs. A cluster scheduling system is used to manage the jobs and submit work for the various jobs. Each job is responsible for managing memory (allocation and freeing) for each device the job is executed on. Each tensor supports multi-dimensional arrays that support specific data types, such as IEEE float, double, or complex types.

3.3 Task Graphs

Task graphs and dataflow graphs have fundamental similarities. Both have the general idea of representing algorithms as a series of connected computational and logical tasks/actors. Data flows between tasks, and operate on the data. The main difference between dataflow and task graphs is the management of scheduling and how tasks are scheduled for threads. In a task graph, the data flow is inherently designed within the graph itself. The task graph is constructed using a prioritized queue that is restructured based on dependencies between tasks that are inserted into the queue. Threads pop tasks from these queues and processes the task that is received with data that is bound to the task instance. HTGS borrows ideas from task graphs, but adds a number of specialty tasks that enable effective design for HPC algorithms. Namely, a memory interface for managing data reuse, GPU tasks that bind to physical GPU hardware, bookkeeper tasks for managing dependencies, and execution pipelines for scaling sub-graphs to multi-GPU systems. In addition, HTGS alters the queue representation by binding threads to a specific task that consumes and produces data using shared queues that are connected to form the task graph. In traditional task graph schedulers, the graph representation is mostly lost in the implementation, whereas HTGS inherently uses the graph representation in its implementation to schedule and process data.

3.3.1 OpenMP Tasks

OpenMP v4.0 and beyond have employed new directives that can prescribe work as tasks with dependencies to formulate tasks. (OpenMP Architecture Review Board 2013) OpenMP tasks are computational functions that are employed into a work queue. An available thread picks up the work from within the work queue if dependencies have

been satisfied. Each task contains additional directives that are used to describe task dependencies to create restrictions among multiple tasks. Synchronization constructs are also provided such as barriers, grouping, and atomics. Using these constructs, OpenMP formulates a task graph, although the task graph representation is not inherent to the implementation, and requires external tools to extract and visiualize the graph. The main benefit of this model is the ability to quickly annotate existing code to create a task-based representation. However, this representation has a detrimental effect in that analyzing the behavior and design decisions becomes challenging. There have been a number of studies that extend OpenMP tasks to improve profiling and debugging of this model. (Lorenz *et al.* 2012) and (Qian, Ding, & Sun 2013) HTGS maintains the graph representation throughout the analysis and implementation phases. This enables more productivity with building and improving upon the task graphs that are implemented using the HTGS model.

3.3.2 QUARK

QUARK is a runtime environment to aid in dynamically scheduling applications, which prescribe precedence-constrained compute kernels for shared memory systems. (YarKhan, Dongarra, & Kurzak 2007) QUARK uses a dataflow model to represent schedules of data based on dependencies between computational tasks in a task graph. The data dependencies are determined using runtime analysis that detect data hazards within kernels. Similar to OpenMP Tasks, QUARK uses a queue to manage task scheduling, where a task is submitted into an execution queue, which is ordered based on data dependencies. Threads then pull from the queue to process each task. QUARK is used in the high performance Parallel Linear Algebra Software for Multicore Architectures (PLASMA). (Kurzak *et al.* 2010), (Buttari *et al.* 2009) Recently, QUARK and PLASMA have been ported to use OpenMP tasks. (YarKhan *et al.* 2016)

Another library that uses constructs similar to that of QUARK is MAGMA. (Tomov, Dongarra, & Baboulin 2010), (Tomov *et al.* 2010), and (Dongarra *et al.* 2014) MAGMA is a linear algebra library that uses task-based scheduling similar to QUARK, but for hybrid architectures. The scheduler used within MAGMA is coupled with the MAGMA LAPACK implementation, so further expansion of this scheduler is not available for algorithms beyond linear algebra. QUARK represents a similar system for implementing parallel algorithms through its use of dataflow representations. But as described for OpenMP Tasks,

this analysis is lost in the actual implementation.

3.3.3 Dryad

Dryad is a system to parallelize sequential programs into distributed parallel programs (Isard *et al.* 2007). A job in Dryad is a directed acyclic graph where each vertex is a program and edges represent data channels. Using a runtime, the computation graph is mapped onto physical resources. Jobs are defined using Dryad's graph library. Executing a graph is done using Dryad's runtime system. Using this system, Dryad effectively pipelines execution where multiple Dryad vertices contain purely sequential code, although it is possible to support using a shared thread pool using their event-based programming. HTGS represents a task graph to represent an algorithm containing multiple vertices and edges to form a single parallel high performance program. Dryad demonstrates an alternate way of represent distributed computing of multiple sequential programs.

3.3.4 StarPU

StarPU is a task graph scheduler that is effective at processing workloads on hybrid CPU/GPU architectures. (Augonnet *et al.* 2011) In StarPU the task graph is composed of a series computational nodes and data edges. Each node implements one or more variations of the computation, one for each architecture. At runtime, the scheduler distributes work for each compute function, and if an architecture is available for computation, then that resource is used to process the data. Data locality is done using a global address space memory manager to represent both CPU and GPU address spaces as a single address space. In this system, data resides in the global address space and automatically applies data transfers on the data for each architecture. StarPU also implements work stealing such that if a computation is finished and is looking for work, it can steal work from another architecture to speed up processing that data. For example, if the GPU is an order of magnitude faster at computing than the CPU, then it might be better to steal work from the CPU to process on the GPU. HTGS provides a different approach than StarPU. In HTGS each node is bound to a particular architecture and is designed for that architecture. Although it would be possible to provide similar functionality of StarPU in HTGS, such as implementing multiple variations of the same function for different architectures, it is not inherent to the design of HTGS. Also, the use of a global address space for CPU and GPU

memories simplifies data transfers, but will often result in inefficient data transfer patterns.

3.3.5 Pegasus

Pegasus is a framework for mapping scientific workflows onto distributed systems. (Deelman *et al.* 2005) Using a distributed scheduler, Pegasus manages applications and schedules them on grid-based clusters. The task graph in this case consists of nodes that represent programs, and edges are dependencies between programs in the form of input/output files. These tasks are clustered based on computational resource requirements for the job. Pegasus aims to maximize the usage of grid-based clusters for many-task jobs. This approach shows an alternative method for scheduling algorithms by managing program scheduling in grid-based clusters. HTGS focuses on the implementation and parallelization of an algorithm and improving concurrency for a single program.

3.3.6 Intel Threading Building Blocks

Intel Threading Building Blocks (TBB) provides a C++ template library that handles concurrency through task graphs. (Reinders 2007) (Kukanov & Voss 2007) The main difference in TBB and HTGS is the method in which it provides threading. In TBB, threads are sent to tasks, so the total number of threads operating on the graph is equal to the number of logical cores on a computer, which are multiplexed when data reaches tasks. This system is excellent for CPU computation as when a node in a graph receives a message, a task is spawned to execute on the incoming message, which is operated on by some available thread. However, when moving to multi-GPU computation, GPU tasks are typically bound to specific GPU contexts (one context per GPU). Therefore, the thread that executes the task becomes responsible for that GPU context. If the thread executing the task changes the next time data enters that GPU task, then the GPU context may become invalid. This requires that every time a thread binds to a task in Intel TBB, that task must first bind the thread to the GPU context, which adds additional overhead to schedule the task. HTGS provides an alternative method, which binds threads to tasks, so the same thread is always executing on the same task. This guarantees that GPU tasks will reuse the same thread during the entire execution of a task graph.

3.4 Concurrent Collections (CnC)

Concurrent collections (CnC) is a programming model that supports a combination of task and data parallelism. (Budimlic *et al.* 2010) The model builds on past work on TStreams, which executes functions in parallel using multiple producers and consumers. (Kathleen Knobe 2005) In CnC programmers define operations and ordering to formulate a CnC graph. A CnC graph is constructed as a series of step, data, and control collections. These collections define two operations; get and put. Instances of the collections are dynamically generated by the CnC runtime and is executed concurrently. The data and control collections handle dependencies and data management. CnC graphs are similar to task graphs, except their is an explicit representation of the dependencies and data. HTGS and CnC are similar in how they represent a multiple producer, consumer application to pipeline algorithms, except data dependencies and scaling are handled differently in HTGS. In the next sections, we present different implementations of the concurrent collections programming model.

3.4.1 Multi-core Concurrent Collections

Multi-core CnC presents two implementations of the CnC model for multi-core architectures. (Budimlic *et al.* 2009) The first builds on top of Intel Threading Building Blocks (TBB) as a runtime system and implementations for the step, data, and control collections. In their implementation, they use C/C++ classes to represent concurrent objects and graphs. Steps are defined by user-written C++ functions wrapped in a function object and data uses Intel TBB concurrent hash maps. When a tag prescribes a step, an instance of the step is created and mapped to a TBB task. The TBB scheduler takes the step and executes as a TBB task. This implementation is distributed by Intel as *Intel CnC*. (Schlimbach 2014) The second implementation is the Habanero-Java project, which is based on the X10 language. This implementation largely uses Scala and Java to hook into the CnC model. (Cavé *et al.* 2011)

3.4.2 CnC-CUDA

The CnC model is an implementation of CnC for CUDA GPUs. (Grossman *et al.* 2011) This implementation expands on the Habanero-Java project to include CUDA steps

into CnC. This execution model requires a Java-to-native code interface, which is provided by JCuda. (JCuda 2015) CnC CUDA manages data locality by automatically generating data and control flow between CPU and GPU steps. This is done through automatic data transfers over the PCI express. The main challenge with CnC CUDA is how they represent data locality. CUDA compute kernels are implemented by the user and data allocation and copying of data is automatically handled through CnC CUDA's translator, which generates stub code for these steps. Extraneous device memory copies are automatically removed by analyzing the CnC graph for contiguous GPU nodes that operate on that data. HTGS relies on the programmer to specify data motion, and manages data reuse through memory managers. The memory manager allocates data during initialization and resuses the memory throughout the execution through memory rules.

Chapter 4

THE HYBRID TASK GRAPH SCHEDULER MODEL

The Hybrid Task Graph Scheduler (HTGS) model consists of constructing task graphs to represent an algorithm and execute the core computational kernels concurrently using coarse-grained parallelism. From the model, a framework is defined such that each task graph is made up of a series of vertices and edges, where vertices are computational or logical tasks, and edges are data dependencies or input parameters for the tasks. Every task is bound to one or more concurrent threads, see Figure 4.1.

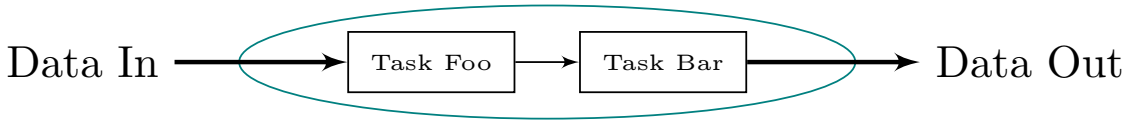

Figure 4.1: HTGS task graph.

Transforming an algorithm into an HTGS task graph involves three steps. First, represent the algorithm as a parallel algorithm and identify the core computational components of that algorithm. Next, define a dataflow graph that interprets the parallel algorithm as a series of computational tasks connected based on data dependencies. The dataflow graph is then transformed into an HTGS task graph. Converting the algorithm in this way exposes the concurrency across multiple computational tasks, allowing the algorithm to operate as a pipelined workflow system. The task graph has specific ordering, in which each task consumes data from a work queue and produces data for the next task's queue. These shared queues are thread safe, allowing a pool of threads to access the queue concurrently. In addition, a task can also produce data for a previous task within the graph, which will create a cycle and requires that the task producing the data define a termination

condition.

Each task has one input type and one output type; however, multiple tasks can have edges producing data for a single task. A pool of threads can be associated with that task. If more than one thread is specified for a task, then that task is copied, with each copy binding to a separate thread. The pool of threads accelerates the consumption and production rates of that task.

Tasks within HTGS have four primary phases: Initialize, Execute, Terminate, and Shutdown. These phases are used to define the various states that the task operates and provides hooks for the task to interact with custom behavior.

Initialize

In the initialize phase, a thread has attached to the task. This allows for any co-processors to initialize and bind device contexts to the task to prepare the task for scheduling work for that co-processor. The thread that is attached will never detach from the task until the task has terminated.

Execute

The execution phase defines the core functionality of a task and consumes data from the task's input queue and produces data for its output queue. Every task has one input type and one output type. The type that is defined represents and abstract object that can encapsulate multiple pieces of data that may be required for the current task's functionality or any future tasks later in a task graph. Each of the queues within a task graph are managed by a connector, which binds two tasks together and manages the number of active connections producing data.

Terminate?

The *terminate?* phase is checked anytime a task is awakened prior to getting data from the task's input queue. If the task is terminated, then the task enters the shutdown phase.

Shutdown

The shutdown phase is called when a task is terminating to free up any resources allocated within the lifetime of the task.

The novelty in the HTGS model is its methodology for handling threading, scheduling, and scalability. This methodology is defined using the HTGS framework. Each task in

the framework is bound to one or more threads to assist in processing the input queue for the task. This threading design concurrently process all tasks within a task graph. Using this threading model, if there is a task that is allocating memory, then task graphs will process data without waiting and eventually will run out of system memory. The *Memory Manager* is used to define a separate edge between two tasks to throttle a graph and stay within memory limits. Scheduling is defined through the task graph specification. *Bookkeepers*, micro-schedulers, manage the state of the computation to aid in properly scheduling complex algorithms that contain data dependencies. Scalability is represented using *Execution Pipelines*, which are tasks that assist in scaling a task graph. The scheduling, memory management, and scalability within the HTGS framework are all discussed in more detail in the next sections.

4.1 Scheduling – Bookkeepers

HTGS defines the bookkeeper task, to support a single input type and multiple output edges, potentially each with their own types. Each output edge has a rule associated with it, which defines when data is produced for that edge. The bookkeeper rules are intended to not be compute intensive and should only handle state updates to avoid stalls during task graph execution.

When data enters the bookkeeper, a control passes that data to each rule that is attributed with the bookkeeper. The rules update the state with that data and determine how/when new data is produced for the task connected by the rule. Figure 4.2 shows a diagram of the bookkeeper task and its multiple output edges.

Each bookkeeper is thought of as a micro-scheduler within the task graph. For example, in a matrix multiplication task graph, two tasks load matrices to be multiplied, the bookkeeper would be used to gather these matrices and produce data to do the matrix multiplication. The bookkeeper task complements the basic scheduling behavior of HTGS task graphs by defining the state of the computation between two tasks, where one or more tasks are producing state updates and another task is waiting for a dependency to be satisfied prior to execution. For example, in image stitching, two neighboring images must have their Fourier transforms computed prior to computing the phase correlation image alignment method. The bookkeeper in this case would be responsible for gathering image Fourier transforms and producing work when two neighboring images' FFTs are ready.

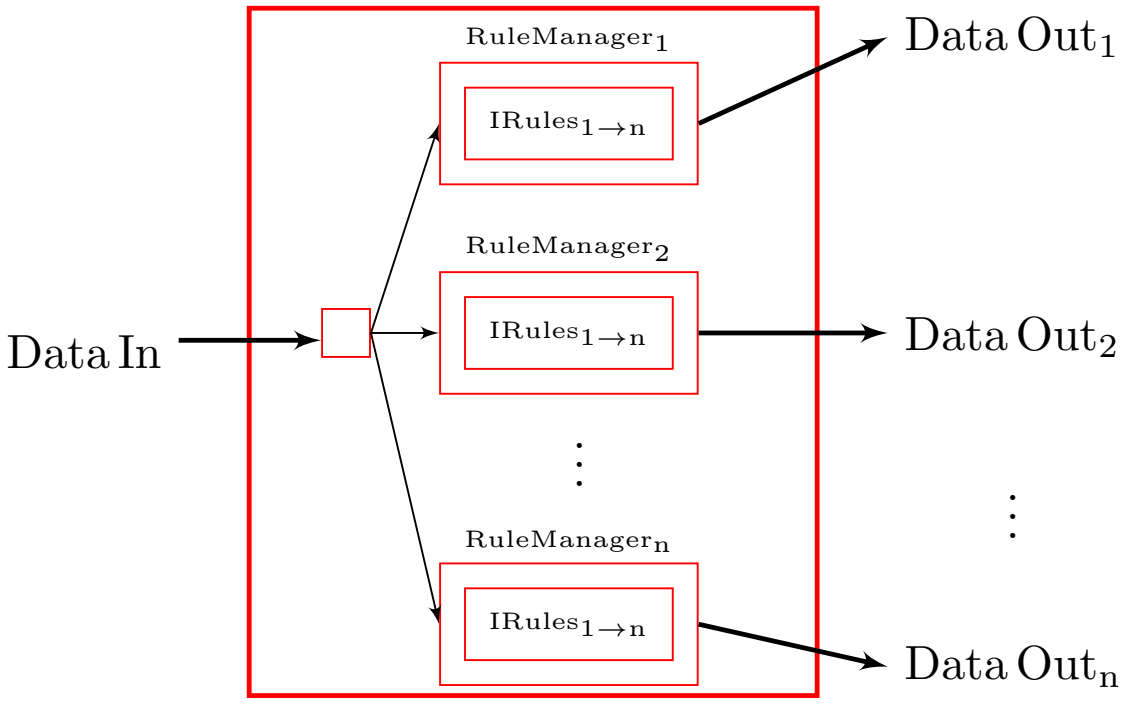

Figure 4.2: HTGS bookkeeper task with IRule interfaces for scheduling management.

Using the bookkeeper expands the capabilities of HTGS task graphs to represent complex dependencies seen in many algorithms.

4.2 Memory Management

Each task is bound to one or more threads within the task graph, which concurrently execute and consume data. In some cases, one task is allocating memory to process the data concurrently, whereas another task within the graph releases that memory. If the allocating task is less computationally intensive than the releasing task, then the graph will consume memory faster than it is being freed, until no memory is available. The HTGS model specifies a special memory edge to help throttle a task graph, which ensures a specified number of memory elements will flow within the task graph.

The memory edge represents a data channel between two tasks, which is managed by a memory manager. This data channel is separate from traditional edges within the HTGS task graphs and only processes memory data. One task gets memory from the edge and the other releases. The memory manager, like the bookkeeper, is a task. Because of this, the memory manager can be easily customized to support different types of memory allocation. For example, during the initialization phase, the memory manager can specify a co-processor that the memory manager uses to allocate. This allows each memory manager to bind data specific to a particular address space and distribute memory from that device to a task.

During the initialization phase, the memory manager allocates a pool of memory for the edge. The execution phase receives memory to be processed and produces memory for the task requesting memory. If the memory pool is empty, then the edge is empty and forces the task getting memory to wait. The memory data that is passed along this edge contains a memory release rule. The memory manager uses the release rule to progress the state of the memory and determine when memory is ready to be released/recycled. These memory rules provide a mechanism for expressing data locality. A graphical depiction of the memory manager is shown in Figure 4.3.

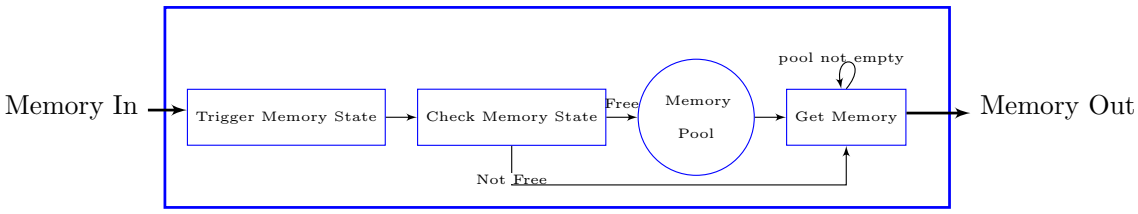

Figure 4.3: HTGS memory manager task.

The reuse of memory is a key optimization for processing data on accelerators to prevent unneeded data transfers. The access patterns of an algorithm must be carefully analyzed to find the best traversal to optimize data locality. Some of these decisions need to be incorporated into rules defined within bookkeepers. This analysis applies to the design of the task graph and the specification of the memory release rules.

4.3 Scaling – Execution Pipelines

There are three fundamental types of parallelism that programmers rely on to gain performance: (1) Data parallelism, (2) Task parallelism, and (3) Instruction level parallelism.

Data Parallelism

CPUs operate on the same function across multiple data; for example single instruction multiple data (SIMD).

Task Parallelism

Distributing computations to compute workers where each worker processes a specific set of instructions.

Instruction Level Parallelism

Processes operations within a program concurrently, effectively overlapping instructions.

HTGS expresses data parallelism through the use of CPU thread worker pools. A task in HTGS can have one or more CPU threads attached to it. Each thread receives different data from its work queue and processes the same computational function on that data. This type of parallelism is ideal for CPU tasks, whereas a GPU task only needs one CPU thread to issue work for a GPU.

Task parallelism is defined in HTGS through task graphs. A task graph decomposes an algorithm into a series of computational steps and can be executed concurrently when dependencies are satisfied. One major benefit to this method of parallelism is the ability to easily overlap I/O with computation. For example, with task-level parallelism the PCI express can be overlapped with computation, thus reducing the impact of the data transfer costs assuming there is enough computation complexity for each data element shipped.

HTGS does not explicitly expose instruction level parallelism. Instruction level parallelism is left up to the programmer to define within their computational tasks. In many of the examples provided, HTGS uses well-defined libraries for basic linear algebra operations and fast Fourier transforms.

The data and task parallelism featured in HTGS provides an environment that streamlines computation for multi-CPU systems. HTGS uses this to express an algorithm as a series of independent tasks that produce/consume data through edge connections defined within the task graph, exposing coarse-grained parallelism.

To process data on accelerators, a task simply needs to bind to the co-processor during the initialization phase and execute using the context for the accelerator, see Figure 4.4.

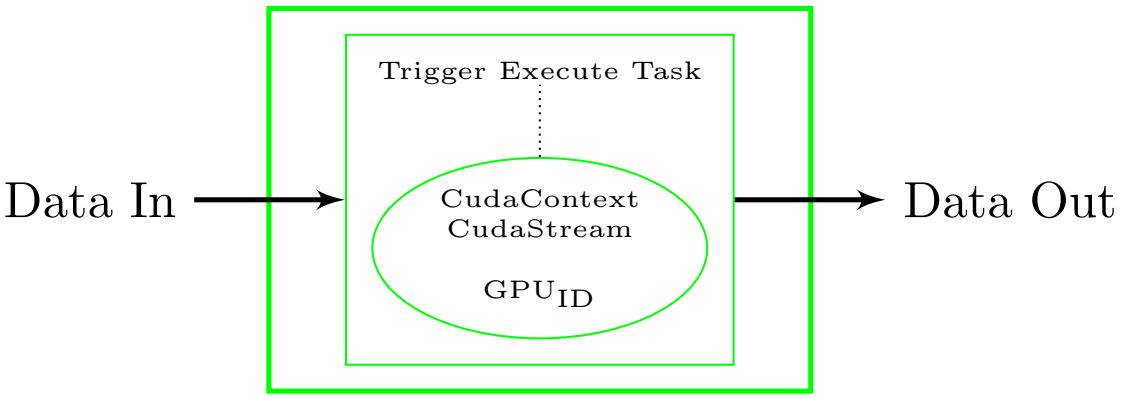

Figure 4.4: HTGS NVIDIA CUDA task.

This design is sufficient for systems with one accelerator. HTGS uses the execution pipeline task to scale on machines with multiple accelerators, such as fat nodes. The execution pipeline is a task that encapsulates a task graph and creates copies of that task graph. Data is distributed among the copies based on decomposition strategies provided to the execution pipeline, which are managed by a bookkeeper within the task. See Figure 4.5.

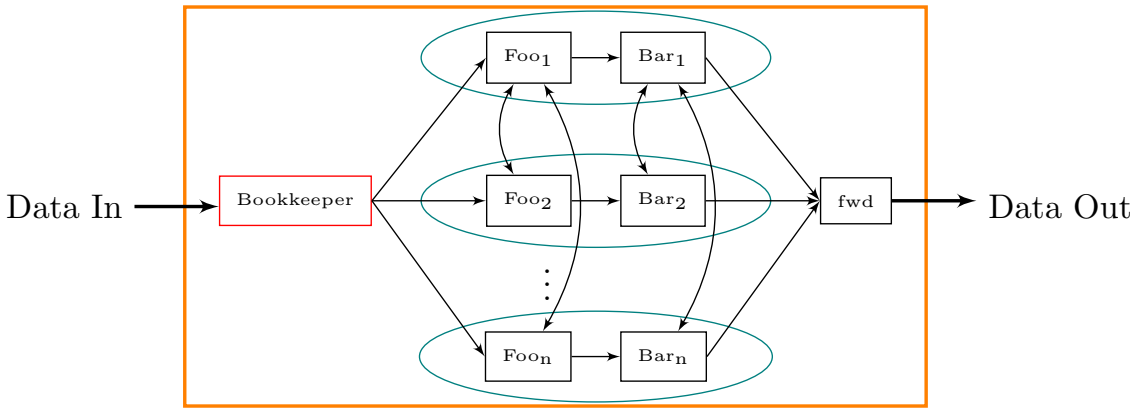

Figure 4.5: HTGS execution pipeline task.

Each task graph within the execution pipeline is an exact copy of the original task graph including memory edges and bookkeepers. The rules within the bookkeepers are shared and synchronously accessed, which prevents race conditions where multiple bookkeepers are trying to update the global state simultaneously. Each task graph copy is assigned a pipeline ID to distinguish which pipeline the graph belongs too. This pipeline ID is passed along to all its underlying tasks during the initialization phase. The identifier is used to distinguish between the different task graph copies. If there are CUDA tasks within these copies, then each CUDA task is bound to a separate accelerator. For example, in a system with three CUDA graphics cards, the execution pipeline would spawn three copies of the task graph and assign each copy to a separate graphics card. The copy would bind all of its tasks to a separate graphics card, including its memory managers. The memory managers in the copy would allocate memory for its specified device and tasks would schedule work for their assigned graphics card. In addition, every task have their own pool of threads, allowing for all task graph copies to operate across all devices concurrently.

The bookkeeper within the execution pipeline task uses decomposition strategies, which can create halo regions depending on the algorithm. It is possible that if a memory manager from one pipeline allocates memory for a device within the halo region, then that memory could end up in different pipeline, which is bound to a different device. During release, that memory needs to make its way back to the memory manager that allocated the memory. To support this kind of behavior, the execution pipeline builds a connectivity graph between nodes within the copies. This is achieved by providing every task access to every other task's input queues. In the case for the halo region, the memory manager from one pipeline would receive memory from another pipeline and then be able to pass the memory to the appropriate pipeline that allocated that memory. The only component needed is to store the pipeline ID within the memory. Additionally, this design allows for other more advanced scheduling behavior. For example, if there is a load imbalance within the decomposition strategy, then one pipeline could be configured to steal work from another pipeline. This allows the idle pipeline to continue processing work even if there is a load imbalance.

4.4 Algorithm Design Methodology for HTGS

The methodology for using the HTGS model to implement an algorithm consists of five steps: (1) Algorithm representation, (2) Dataflow interpretation, (3) Task graph design, (4) Task graph implementation, and (5) Task graph refinement and optimizations. The first three steps are pictorial representations of an algorithm, *the white board stage*. Preparing the algorithm for HTGS requires understanding the parallelism and data dependencies within the computation workflow. Traditional sequential algorithms will most likely not map well into HTGS. Using HTGS requires a well defined parallel algorithm whose computational kernels are laid out as a series of modular functions that processes data concurrently.

Using the parallel algorithm representation, map the computational functions into nodes and connect edges based on data dependencies to formulate a dataflow graph. Then, using the dataflow graph interpret the graph as an HTGS task graph where dependencies are handled by bookkeepers, input and output parameters are mapped to data objects, computational functions are tasks, and edges connect the components. Each edge represents the input/output type between tasks. The types map to the definition of the

data objects that contain the data needed between one or more tasks. When memory is a concern, then memory edges are annotations between tasks where one task allocates the data, and the other frees.

Next, using an implementation of the HTGS model, the HTGS task graph is created in code. The traversal strategies when producing data for the HTGS task graph must follow the definitions assigned to memory rules to avoid deadlock when memory is not released at the appropriate times. After validation of the implementation, revisit the HTGS task graph to understand the bottlenecks and customize thread configurations based on computationally intensive tasks. Additional optimizations can also be applied such as redesigning the graph to use co-processors or identifying better approaches for scheduling to improve data locality.

Using this approach, the HTGS model provides a concrete method for representing an algorithm that maps to the parallel program. The design decisions for modularizing the algorithm, scheduling data, and representing memory are explicit and defined within the model as separate entities. This exposes these representations to the programmer to assist in understanding the code and finding points of contention that require further optimization. Additionally, the code maps back to the analysis, so programmers not familiar to the parallel code can observe the implementation at a much higher level of abstraction. In Chapters 6, 7, and 8, we will demonstrate this design methodology. In the next chapter, we present the details of the C++ implementation of the HTGS model.

Chapter 5

HYBRID TASK GRAPH SCHEDULER C++ IMPLEMENTATION

The HTGS model provides the methodology to utilize high performance fat nodes that consist of multiple CPUs and GPUs. The C++ HTGS API is the code to implement and execute the model. The API is designed using a templatized object-oriented approach where a task is a customizable interface that is implemented to define the four phases of a task; initialize, execute, terminate, and shutdown. The API is written as a series of C++ header files that are included in projects to construct and execute HTGS task graphs and uses the C++11 standards for threading, thread safety, and data structures.

The API is split into two components; (1) the *core* API and (2) the *user* API.

5.1 Core API

The core API implements low-level HTGS components that task graphs operate with such as parent base classes, connectors (edges), thread safe queues, task schedulers, rule managers, and memory managers.

Parent Base Classes

Parent classes are used to strip the template types to allow children to be stored within container data structures, even if the children's underlying template types are different. This is fundamental to the functionality of task graphs to store its vertices and edges. Type safety is maintained during task graph construction through functions that ensure the input and output types of connected edges are matching. These edges are managed by *Connectors*.

Connector

The connector is an object that represents an edge between two or more tasks by managing an input/output queue. Each connector has a single template type that defines the type of queue. Additionally, connectors maintain the number of active tasks that are producing data for the edge. If there are no more active tasks producing data, then the edge is no longer giving data to the consumer task. This is used to identify when an edge is closed to terminate tasks.

Thread Safe Queue

Data is stored in queues between tasks. Each task contains one or more threads producing/consuming to/from these queues. To maintain thread safety each queue implements a monitor-based implementation that uses a mutex to conditionally wait for data to be available. This mutex is used to ensure two or more threads cannot enter the critical section when adding or removing data from the queues. There are two types of queues that are supported for connectors; a traditional thread safe blocking queue or a priority blocking queue. The priority blocking queue is enabled through compile-time directive, which are used to specify an ordering for data. By default the queue uses a FIFO style of data production/consumption.

Task Scheduler and Task Scheduler Runtime Thread

Every task is bound to one or more threads. Each thread that is spawned is managed by a task scheduler runtime thread. The task scheduler is responsible for interacting with the task interface to initialize, consume data, produce data, and terminate the task. The task scheduler is also responsible for managing the input and output connectors. A thread is bound to a separate instance of the task scheduler and underlying task interface. If there is a pool of threads assigned to a specific task, then each thread within the pool with have a new copy of that task. This allows each task to easily allocate local reusable memory from within the task across all of the threads. Figure 5.1 shows the call graph for the task scheduler runtime thread and how it interacts with the task interface.

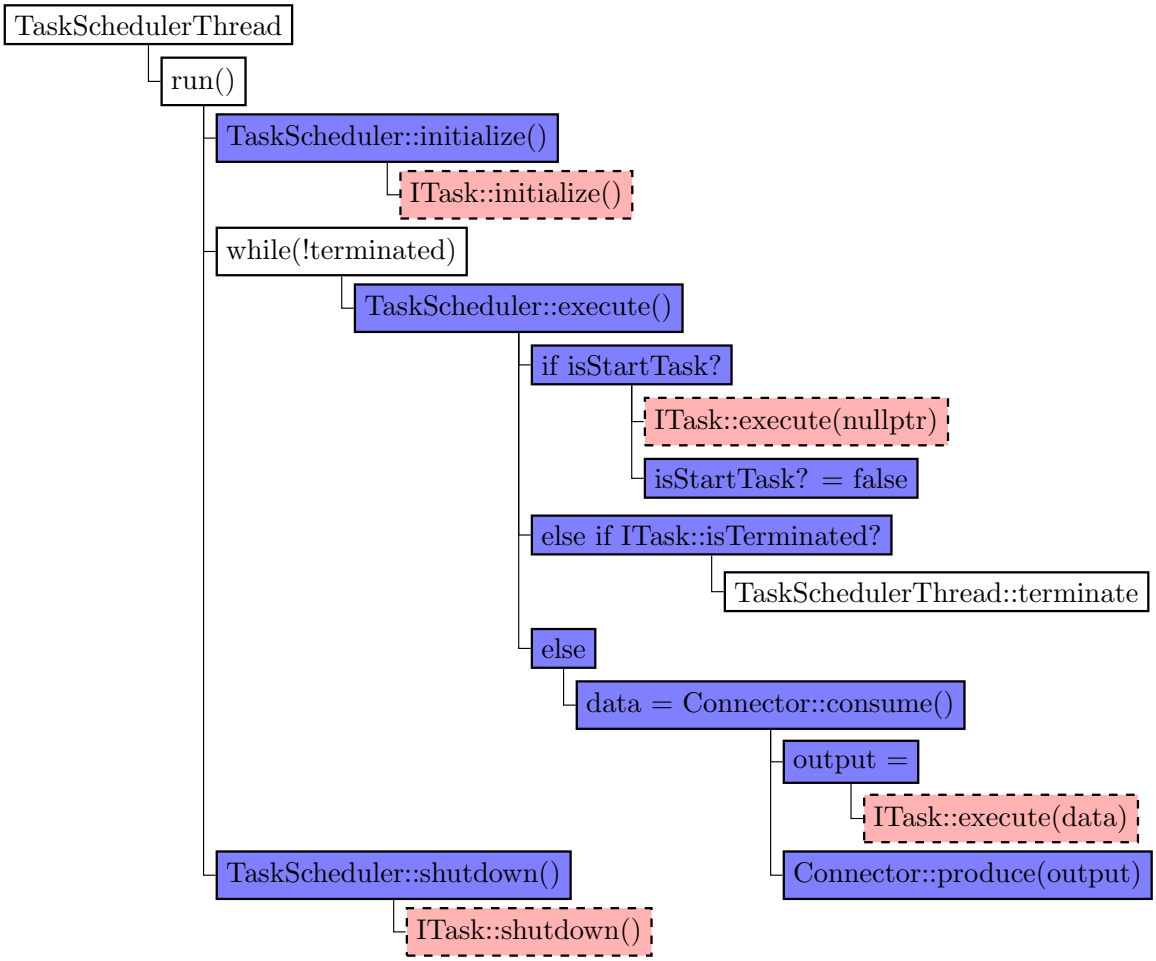

Figure 5.1: HTGS task scheduler thread call graph.

Rule Manager

The rule manager provides the functionality for bookkeeper edges. A bookkeeper has one input and multiple outputs. Each output is represented by a rule manager, which shares the same input type as the bookkeeper, but has a separate output type that matches the rules that are managed by the rule manager. When data enters a bookkeeper, the data is passed to one or more rule managers that are added to the bookkeeper. The rule manager then sends the data to each rule that is added to the rule managers. Each rule has its own mutex, which is locked by the rule manager to ensure no other threads are processing that rule at the same time. The rule is an interface implemented by the programmer that processes data, updates state, and decides when

to produce data, see Figure 4.2.

Memory Manager

The memory manager is a task that implements the functionality for interacting with a memory edge. The memory edge is separate from the standard edge that connects two tasks together. Instead, the memory manager is used as the intermediary between two tasks where one is allocating memory and the other is releasing memory. Each memory edge is named, which is used by the allocating and releasing tasks to send data to that named edge. This method allows for a single task to have multiple memory edges each with their own names.

The API presents two implementations of the memory manager edge; (1) Traditional and (2) CUDA. The traditional has the basic behavior for memory management without any extra functionality. The CUDA memory manager inherits the functionality of the traditional memory manager, except adds CUDA device binding during initialization. The extra initialization step of the CUDA memory manager allows for allocation on a specific CUDA device.

There are three modes of operation for a memory manager; (1) Static, (2) Dynamic, and (3) User Managed. These modes are specified when the memory edge is added to a graph. Additionally, the edge contains an allocator that defines how memory is allocated and freed, a memory pool size, which specifies the amount of memory data produced by the edge, and which tasks are getting and releasing memory. Additionally, every memory manager has a specific data type that specifies the type of data that it is producing, which must match the allocator.

The three modes of operation control how/when memory is allocated and freed by the memory manager.

1. **Static**

   The static memory manager recycles memory when it is released. All of the memory for the memory pool is allocated only once when the memory manager is initialized and is freed when the memory manager is shutdown. This memory manager is ideal for accelerator memory allocation due to the synchronous behavior of dynamic allocation. However, this method assumes that the memory allocated never needs to be resized, see Figure 5.2

2. **Dynamic**

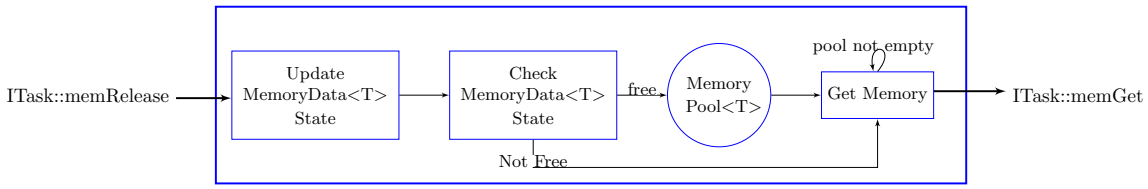

Figure 5.2: HTGS static memory manager, where T is the type of memory.

The dynamic memory manager allocates memory when the task receives the memory and is freed when the memory manager adds the memory back into the memory pool. This method allows for dynamic allocation during execution and is used to specify different memory sizes based on the task data requirements, see Figure 5.3

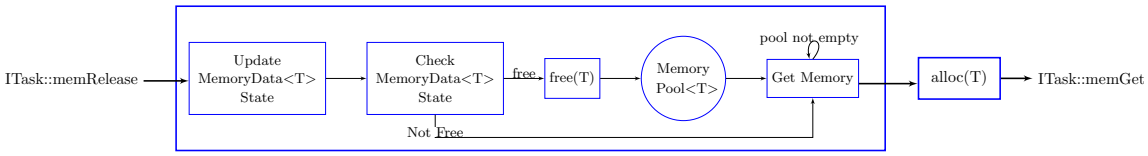

Figure 5.3: HTGS dynamic memory manager, where T is the type of memory.

3. **User Managed**

   The user managed memory manager is used to help keep track of the number of elements allocated/freed for the programmer. The user managed memory manager keeps track of how many data elements are allocated/freed by the programmer. The memory allocation, freeing, and release rules are up to the programmer to define and are separate from the memory manager. When memory enters the user managed memory manager, the memory data is immediately added into the memory pool without any consideration on locality or reuse, see Figure 5.4.

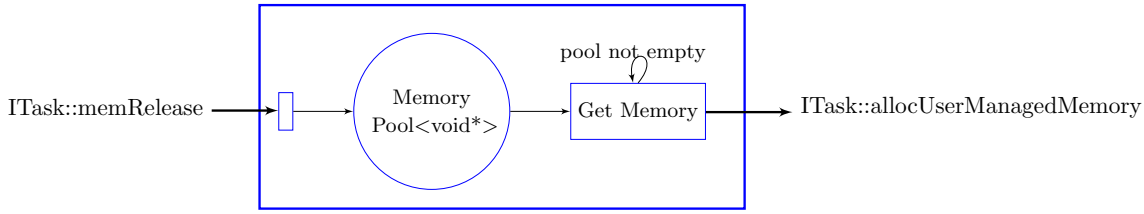

Figure 5.4: HTGS user managed memory manager. The type is *void* * as there is no memory that is allocated or freed by this manager, but rather is managed entirely by the programmer.

The core API is mainly used by the user API, but can also be updated to include new functionality to expand or build upon the HTGS model. For example, creating a new type of memory manager to support OpenCL functionality. In most cases the programmer will only use the user API.

5.2 User API

The user API contains C++ classes and interfaces that are used to construct and execute HTGS task graphs. Each of these components are built using the core API to enable future customization. The user API is used by programmers to implement and execute algorithms designed by the HTGS model.

Data

Data is represented as an interface that is used to store the various parameters that are required by tasks. All data that flows within HTGS task graphs are stored using instances of this interface. Additionally, the data interface holds onto meta data that is used to customize the ordering of tasks, which is enabled when priority blocking queues are activated. Figure 5.5 shows an example implementation of the data interface to store input and output data for summing two numbers.

Figure 5.5: Example data implementations for adding two numbers and returning the sum.

```
#include <htgs/api/IData.hpp>

class InputData : public htgs::IData
{
public:
  InputData(int x, int y) : x(x), y(y) {}
  int getX() const { return x; }
  int getY() const { return y; }
private:
  int x, y;
};

class OutputData : public htgs::IData
{
public:
  OutputData(int result) : result(result) {}
  int getResult() const { return result; }
private:
  int result;
};
```

Task

Tasks in the HTGS API are interfaces that implements the task phases of the HTGS model. These phases are defined as virtual functions that are called from the task scheduler within the core API, as shown in Figure 5.1. The virtual functions are generalized and allows for a multitude of implementations, such as the HTGS bookkeeper, execution pipeline, or CUDA tasks. Figure 5.6 shows an example task implementation.

Each task has a specific input and output type that is defined during task definition or creation, which is used to connect tasks within task graphs. If one task is producing

Figure 5.6: Example task implementation that adds two values and produces the sum.

```
#include <htgs/api/ITask.hpp>
#include "InputData.hpp"
#include "OutputData.hpp"

// Creates the add task; consumes InputData, produces OutputData
class AddTask : public htgs::ITask<InputData, OutputData>
{
public:
  void initialize() {}
  void shutdown() {}
  bool isTerminated(
        std::shared_ptr<htgs::BaseConnector> inputConnector) {
    return inputConnector->isInputTerminated();
  }
  virtual AddTask *copy() {return new AddTask(); }
  virtual void executeTask(std::shared_ptr<InputData> data) {
    int sum = data->getX() + data->getY();
    this->addResult(new OutputData(sum));
  }
};
```

data for another task, then the output and input types of those tasks must match, respectively. This design provides compile-time error checking for graph creation and type safety.

Creating/implementing a task consists of four parameters that customizes the scheduling behavior of the task: (1) Number of threads, (2) Is the task a start task? (3) Whether to poll for data, and (4) Timeout period for polling. These parameters are used by the HTGS runtime system and the task scheduler. In addition, each task holds onto meta-data that is used to bind memory edges to each task. These are managed using hash maps that provide lookup times for getting or releasing memory from a

named edge. If a task is declared as a *start task*, then the task will begin executing as soon as the task has finished the initialization phase, which is used to immediately begin producing data for memory managers.

If a task is specified to have more than one thread, then the HTGS runtime will create copies of that task along with a copy of the task scheduler managing each task instance. These copies are all bound to the same input and output connectors, allowing the tasks to operate with a thread pool. The tasks use a single instruction multiple data (SIMD) style of programming for the multiple threads. Local memory allocation for the thread is done during initialization. Data is passed in as a parameter to the execute function, which is consumed from the input connector of the task by the task scheduler. With this behavior the task implementation focuses on processing one independent data instance at a time and can assume all dependencies have been satisfied for the data that it has received. The task produces data using the *addResult* function to insert data onto the output connector of the task. In addition, all memory allocation within the HTGS API is wrapped in smart/shared pointers that automatically free once all references to the pointer has expired. This design enables for zero-copy of data objects as only the shared pointers are distributed among the tasks. Data copying is only done using memory that must be sent/received between address spaces, such as memory allocated using *malloc*.

Task Graph

The HTGS task graph is implemented as a C++ class to store tasks (vertices) and connectors (edges). These components are added into a task graph through functions that create edges, bookkeepers, and memory edges. Each task graph has an input and output task that produces/consumes data entering/leaving the task graph. Using this approach, the graph becomes a black box where data enters a graph, flows through its tasks, and then produces potentially transformed data after being processed within the graph. The scheduling behavior of the graph is defined by its vertices and edges, where each vertex produces data to be consumed by the next task. This procedure is started by inserting data into the graph, or by producing data within one of the tasks, acting as a *start task*.

Tasks that are added into a graph are kept track of using history objects that are defined in the core API. The history objects describe the order in which tasks are added and connected within the graph. These objects are used to aid in duplicating a task

graph for execution pipelines. When a copy is created, the copy is a mirror images of the original graph including vertices, edges, and memory edges. The copy function is highly optimized to process the copy as efficiently as possible, making use of hash maps to speedup lookup times when acquiring the mapping between tasks and task schedulers.

In addition, the task graph object provides functionality to create a dot file representation of the graph, which is used to create a visual representation of the graph. Figures 5.7 and 5.8 shows the code for creating a task graph and visualization of that graph, respectively.

Figure 5.7: Example task graph creation.

```
#include <htgs/api/TaskGraph.hpp>
#include "AddTask.hpp"
int main() {
  AddTask *addTask = new AddTask();

  // Task graph; consumes InputData and produces OutputData
  auto taskGraph = new htgs::TaskGraph<InputData, OutputData>();

  // AddTask is consuming and producing the data
  taskGraph->addGraphOutputProducer(addTask);
  taskGraph->addGraphInputConsumer(addTask);

  // Indicate that we will be producing data for the graph
  taskGraph->incrementGraphInputProducer();
  taskGraph->writeDotToFile("output.dot");
}
```

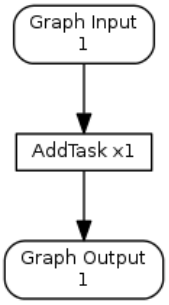

Figure 5.8: Task graph visualization for add task task graph from Figure 5.7 The graph input and graph output show the number of active connections for that edge. The task shows the number of threads that will be bound to that task.

The task graph visualization shows the created graph, including the number of threads that will be spawned for the tasks by the HTGS runtime, the number of active connections for connectors, and (if enabled) profiling information for each task. Profiling details are enabled using the compile-time directive *PROFILE*. When enabled, all tasks within the graph will monitor various runtime behavior such as compute time, wait time, and the maximum size its input queues receive. These values can be visualized by generating the dot file representation of the task graph after the graph has finished executing.

Runtime

The runtime is a class implemented to execute a task graph. It is responsible for creating threads, which are bound to task schedulers. If a task has more than one thread specified, then the runtime will duplicate the task such that each thread will be responsible for a separate instance of the task. The runtime is launched asynchronously, allowing for interaction with the task graph, such as producing and consuming data for a task graph as it is executing. Once the runtime has begun executing, then all of the tasks within the graph will wait for data to begin processing. Figure 5.9 shows one example implementation of using the runtime for handling input and output data from the task graph in Figure 5.7. Figure 5.10 shows the execution output.

Figure 5.9: Example runtime usage for handling input and output data from the task graph in Figure 5.7.

```
#include <htgs/api/Runtime.hpp>
...
auto runtime = new htgs::Runtime(taskGraph);
runtime->executeRuntime();

for (int i = 0; i < 5; i++) {
  auto inputData = new InputData(i, i);
  taskGraph->produceData(inputData);
}
taskGraph->finishedProducingData();

while(!taskGraph->isOutputTerminated()) {
  auto data = taskGraph->consumeData();
  std::cout << "Result: " << data->getResult() << std::endl;
}
taskGraph->waitForRuntime();
delete runtime;
```

Figure 5.10: Example runtime output from Figure 5.7.

```
Result: 0
Result: 2
Result: 4
Result: 6
Result: 8
```

Bookkeeper

The bookkeeper is implemented using the task interface. Data that is consumed by a bookkeeper is forwarded to all rule managers that have been added to the bookkeeper.

These rule managers are constructed when a task is connected to the bookkeeper within a task graph, with a rule as an intermediary. Each rule manager acts as its own edge and produces data based on the programmer-defined rules that are associated with that rule manager. The bookkeeper itself, has no output data, but changes the basic behavior of a task through its use of rule managers.

Rules

Rules are interfaces implemented to determine when to produce data for a task based on dependencies and other customizable behavior. A rule manager produces data for the rule, which is managed by a bookkeeper. The main function that is implemented for the rule is *applyRule*. This function has two parameters; (1) the data that is updating the rule and (2) the pipeline ID that the rule is being used for. The pipeline ID is used to identify which pipeline the rule is intended for, which is used in domain decomposition rules for execution pipelines. A C++ mutex is associated with each rule to ensure one instance of a rule that is shared among two bookkeepers is accessed synchronously.

CUDA Task

The CUDA task is an interface that implements the task interface, which modifies the execution and initialization functions to apply to a CUDA device. These functions attach the task scheduler's thread to a specific CUDA device, which is then bound to the task scheduler's thread. This allows for all memory allocation and execution within the task to be sent to the specified CUDA device. The parameters for the CUDA task is an array of CUDA contexts, CUDA device IDs, and the number of GPUs. These parameters are used within execution pipelines to bind a separate instance of the CUDA task to different CUDA devices.

Execution Pipeline

The execution pipeline is a task that implements the task interface to provide scaling capabilities, particularly for multi-GPU configurations. This task encapsulates an entire task graph and creates copies of that task graph to distribute data among the copies using domain decomposition rules. Any task within the encapsulated graph is copied, including Bookkeeper tasks, which will share the underlying rules among other copies. CUDA tasks are bound to separate devices, one per copy. The domain decomposition rules are defined using the rule interface, which is added to the

execution pipeline task. These rules use the pipeline ID parameter for the *applyRule* function to indicate, which pipeline the rule is intended.

Memory Allocator

The memory allocator is an interface used for memory edges to indicate how to allocate and release memory.

Memory Release Rule

Memory release rules define the state of memory, which indicates if memory is ready to be released/recycled. The release rule is attached to memory when a task gets memory from a memory edge. This rule is used to define the locality of data based on scheduling decisions. Ideally, the memory should only be released when the memory is no longer needed; however, due to memory limits of devices and data access patterns of algorithms, memory may require additional load/stores.

Custom Edge

Custom edge is an interface that bridges the gap between the core API and the user API. It is used to provide special functionality for an edge, which is used to define how the edge is added to a task graph. This is used to help copy the edge when a task graph is copied. This interface has been used to recreate the bookkeeper and memory edges to demonstrate the additional customization to task graphs without having to edit the core API.

The user API contains the primary functionality of the HTGS model. In the next section, we present a *Hello World* example of using the HTGS C++ API by computing the Hadamard product between two matrices.

5.3 Hello World – Hadamard Product

The Hadamard product computes the element-wise matrix multiplication between two matrices that share the same dimension, $A \circ B = C$, as shown in Algorithm 1.

The Hadamard product is an embarrassingly parallel algorithm where each step of the algorithm can be done independently. When sending data between tasks in an HTGS task graph, there is some overhead, so the amount of computation per element of data transmitted between tasks should have some amount of computational complexity. Adding complexity within tasks is done by processing the Hadamard product with block

Algorithm 1 Hadamard Product

```
1: function HADAMARDPRODUCT(A, B, C)
                                   ▷ Matrix dimensions: A^{n×n} B^{n×n} C^{n×n}
2:     for each row do
3:         for each col do
4:             C[row][col] = A[row][col] * B[row][col];
5:         end for
6:     end for
7: end function
```

decomposition, as shown in Figure 5.11.

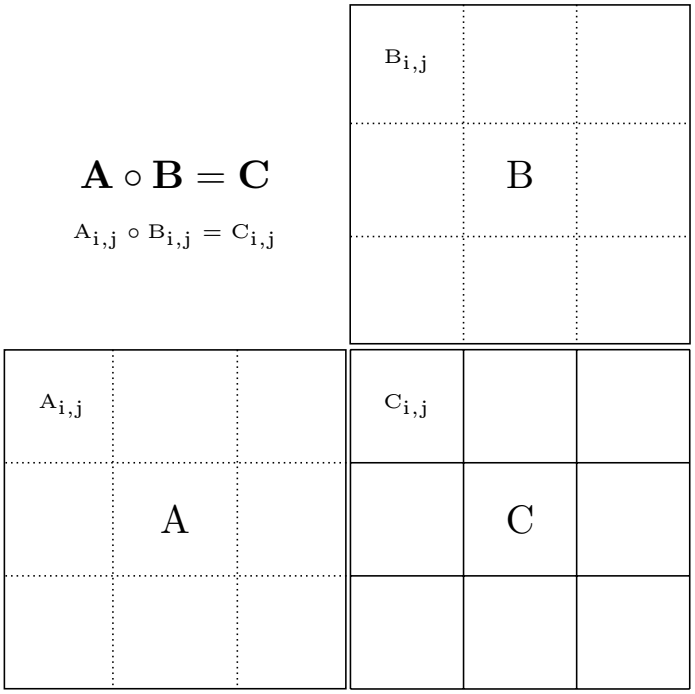

Figure 5.11: Hadamard product block decomposition.

Using the block decomposition variation of the algorithm, we transform the Hadamard product into a dataflow graph, see Figure 5.12. For this implementation, we assume the data has been saved to disk and distributed into separate files, where each file represents a block

of either matrix A or B

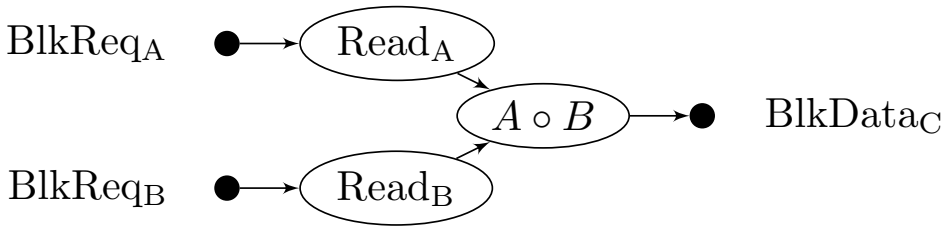

Figure 5.12: Hadamard product block decomposition dataflow graph.

Using the dataflow graph, we analyze the various data requirements that need to be created for the task graph. First, we need a data object to represent the read requests. The block read request should request the row, column block that is to be read. Next, there are two reads required to compute one Hadamard product, so a data object is needed to encapsulate both reads into a single object. Finally, the result of the Hadamard product produces a block of data that represents the result matrix data.

From the dataflow graph, we create a task graph representation, as shown in Figure 5.13. In this example, we merged the read task into a single read task. A flag within the block request data is used to indicate whether to read from A or B. The read operation could be represented as two separate tasks, as will be seen in matrix multiplication, so it is used as an alternate example for reading data. Once data is read, matrix data is sent to a bookkeeper. The bookkeeper uses the LoadRule to store the blocks of matrices that have been loaded and produces work for the Hadamard product for matrices whose row, column block have been loaded.

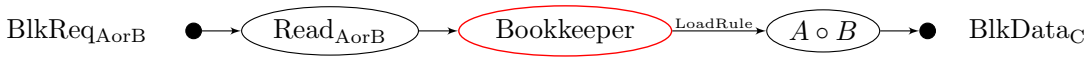

Figure 5.13: Hadamard product block decomposition task graph.

Each task in this task graph can have one or more threads processing data. This enables the read task to pipeline with the Hadamard product task, overlapping the I/O of reading a matrix block with computation. There are two main concerns with this design. First, what is the ideal block size to use, and second, what if the matrix being operated on cannot fit into memory.

There two factors that are involved with identifying the correct block size to be used. First, there is a small overhead involved with passing data between tasks, and second, we need to have enough data flowing in the graph to enable pipelining. To demonstrate these components, Figure 5.14 is a plot showing the impact of block size versus runtime for 4096×4096 sized matrices.

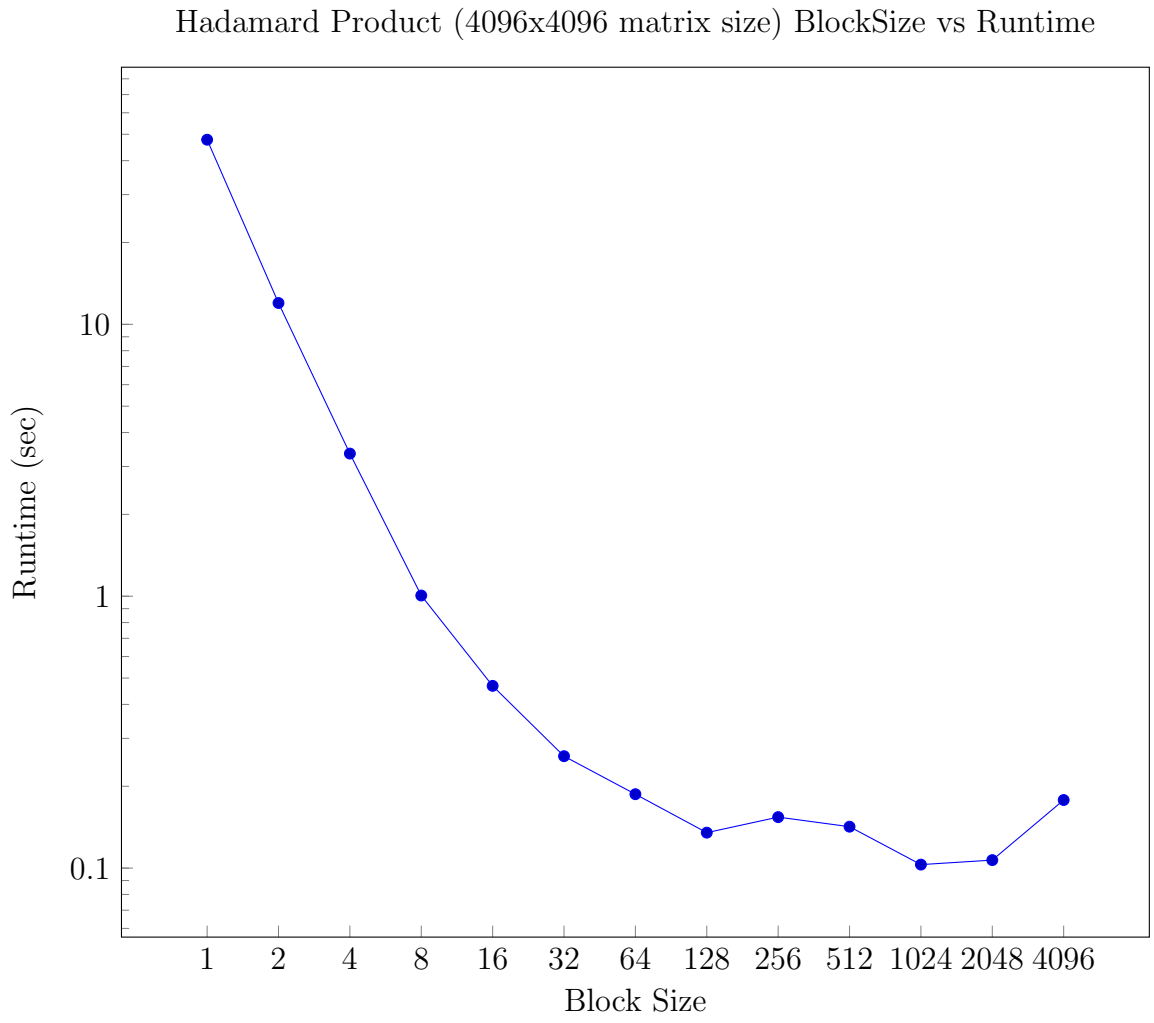

Figure 5.14: Hadamard product block decomposition block size impact on runtime.

The results show that using a small block size impacts performance as there is not enough computation to justify sending data between tasks. Using a block size that is the size of the entire matrix also impacts performance as it prevents pipelining. Therefore,

using a block size that produces enough data enables better pipelining.

The other concern, memory, is due to the original task graph operating without waiting. Using a memory manager, we can connect two tasks with a memory edge to allow one task to throttle another task. Figure 5.15 shows the final task graph that annotates the graph from Figure 5.13. This new graph throttles the read and Hadamard products using three memory edges. The final memory edge that is consumed by the Hadamard product is managed by the main thread that is interacting with the task graph.

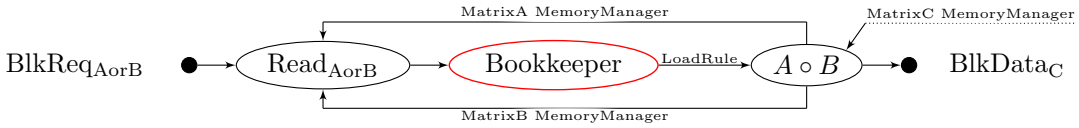

Figure 5.15: Hadamard product block decomposition task graph with memory managers.

Using the task graph from Figure 5.15, we can implement the Hadamard product, which can operate on matrices that exceed the limits of the CPU, using the HTGS C++ API. The source code for this example is available at `https://github.com/usnistgov/HTGS-Tutorials/tree/master/tutorial2/hadamard-product`.

Chapter 6

CASE STUDY 1: IMAGE STITCHING

We describe below a high performance hybrid CPU-GPU implementation that accelerates the Fourier-based stitching of 2D optical microscopy images to less than 1 min (end-to-end execution times) (Blattner *et al.* 2014). It stitches a 59×42 grid of images in 43 s while an optimized single-threaded reference implementation takes nearly 10 min for the same workload. Using the same grid of images on the popular ImageJ/Fiji image stitching plugin, the execution time exceeds 3.6 h, which uses the same mathematical algorithm as our implementation for computing relative displacements. Using this research as a baseline, we present the HTGS implementation of the same algorithm. We show that we obtain similar performance as the hybrid CPU-GPU implementation, but with $\approx 43\,\%$ fewer lines of code (Blattner *et al.* 2015).

6.1 Problem Description

Image Stitching comes up in Optical Microscopy because of a scale mismatch between the dimensions of a plate being examined and the microscope's field of view. For example, the region of interest in a plate is measured in *cm* (e.g., $2\,\mathrm{cm} \times 2\,\mathrm{cm}$) whereas the field of view is at least an order of magnitude smaller ($< 1\,\mathrm{mm} \times 1\,\mathrm{mm}$). To image a plate, a microscope scans the plate as it travels under the optical column and generates overlapping partial images or tiles. Software then assembles the tiles into a single image. Szeliski discusses several algorithms for finding the proper alignment of image tiles (Szeliski 2006).

The algorithm in this study is a Fourier approach, which is based on Kuglin and Heins' phase correlation image alignment method, and is used to calculate the relative displacements between neighboring images (Kuglin & Hines 1975a). The algorithm is commonly found in microscopy image stitching because of its ability to line up a wide variety of features within images in the Fourier space. For example, the popular image stitching plugin found in Fiji (Fiji 2012), created by Stephan Preibish (Preibisch, Saalfeld, & Tomancak 2009), uses the same alignment method to determine image translations. Fiji is used as an image processing package and is the tool of choice for many biologists. The package is built on top of the ImageJ library and acts as a distribution of ImageJ (ImageJ 2012). More details about the algorithm is found in Section 6.3.

This study uses imaging data acquired by biologists at the National Institute of Standards and Technology (NIST) using an Olympus IX71 microscope with a 10x lens and an infrared camera. The images form a grid of 59×42 tiles. Each tile is a 1040×1392 16-bit gray-scale image; its size is $2.76\,\mathrm{MB}$ and covers an area of $896.44\,\mu\mathrm{m} \times 669.76\,\mu\mathrm{m}$. The size of the dataset is $6.68\,\mathrm{GB}$. The grid and tile sizes are fixed during a particular experiment, but can vary between experiments.

6.2 Contributions

First, we present and compare five implementations of Fourier-based image stitching and detail a hybrid CPU-GPU implementation which achieves end-to-end processing times of $43\,\mathrm{s}$ for a grid of nearly 2500 tiles (59×42) on a machine with one high-end GPU card. Such execution times transform image stitching into quasi-interactive tasks and are two orders of magnitude better than those of ImageJ/Fiji (Fiji 2012) which takes nearly

$3.6\,\mathrm{h}$ for the same workload. These execution times also compare favorably with published timing results for similar problems using GPUs (K. U. Venkataraju et al. 2009).

Next, we present and compare the HTGS implementation of image stitching that acquires similar performance as the hybrid CPU-GPU approach. The hybrid CPU-GPU implementation uses the hybrid pipeline workflow model, which requires a significant amount of programmer effort to implement. Using the HTGS model and API, the number of lines of code is reduced by $\approx 43\,\%$, while maintaining similar performance as the manually coded hybrid workflow implementation. This showcases the low overhead of scheduling with the HTGS API.

Both the hybrid CPU-GPU and HTGS implementations takes advantage of coarse grain parallelism in the image stitching computation and organizes it into a pipeline of functional stages: reading, computing, and bookkeeping (managing dependencies). Each stage consists of one or more CPU threads, some of which interact with GPUs. The pipeline overlaps various computations that take place on CPU or GPU cores with data transfers between disk, main memory, and memory on the graphics cards.

6.2.1 Organization

The remaining sections are organized as follows: section 6.3 discusses alternative approaches underlying image stitching algorithms; section 6.4 describes in detail the Fourier-based image stitching algorithm used in this study; section 6.5 presents the implementations that were developed and discusses their performance; section 6.6 presents the HTGS implementation; and section 6.7 discusses and compares the HTGS and hybrid workflow implementations.

6.3 Image Stitching Algorithm

The two main approaches for automatically stitching images are feature-based alignment techniques (Barnea & Silverman 1972; B. Ma et al. 2007) and direct methods (Kuglin & Hines 1975a; Jing, Chang-shun, & Wu-ling 2009). In our study, we are using a direct method, a version of Kuglin and Hines' phase correlation image alignment method (Kuglin & Hines 1975a) that is modified to use normalized correlation coefficients as described by Lewis (Lewis 1995). This method uses Fast Fourier Transforms (FFTs) to compute Fourier Correlation Coefficients and then uses these correlation coefficients

to determine image displacements. Figures 6.2 & 6.3 give pseudo-code listings of the correlation functions. In our context, the Fourier-based approach is advantageous because it is simple, has predictable performance, and lends itself to parallelism. Furthermore, it is more robust as it does not depend on feature detection and, as such, effectively stitches a wide variety of data.

We elected to not use feature-based alignment because it cannot always handle the large uncertainties present in microscopy imaging. For example, one approach in feature-based alignment is the sequential similarity detection algorithm (Barnea & Silverman 1972). This algorithm tends to be fast, but does not guarantee finding the maximum correlation surface. This is particularly problematic with microscopy images as there can be multiple narrow extrema in the correlation surface. As such, the image generated will have incorrect alignment if the alignment is not within a few pixels of a maximum; this result will be detrimental to biologists trying to analyze the stitched image. Despite the difficulties with the feature-based approach, the literature reports on systems that are feature-based and that work for some microscopy image stitching. For example, the AutoStitch software (AutoStitch 2012) implements Brown and Lowe's scale-invariant feature transform (Brown & Lowe 2007). Ma et al. report using AutoStitch to process microscopy images (B. Ma et al. 2007). We plan to run the AutoStitch software on our data set in a future experiment.

Cooper, Huang, and Ujaldon implement another feature-based algorithm (Cooper, Huang, & Ujaldon 2011). Their implementation is aimed at clusters and optimizes for both the CPU and GPU. It uses a combination of FFT-based normalized cross-correlation and feature detection. Their implementation is targeted at images that are much larger than the ones used in this study (e.g., 16K $\times$ 16K and 23K $\times$ 62K). Their results showed that the GPU was not an ideal candidate for image stitching as their implementation could not overcome the latency involved with transferring data to/from graphics memory as well as handle the memory limitations on the GPU for handling very large images.

6.4 Computation

Fourier-based image stitching operates in three phases:

1. Compute relative displacements for all adjacent image pairs. These displacements form an over-constrained system that one can represent as a directed graph where

vertices are images and edges relate adjacent images. The over-constraint in the system is due to absolute displacements of images being equivalent to path summations in the graph that must be path invariant.

2. Select a subset of the relative displacements or adjust them to remove the over-constraint through a fitting or optimization technique (e.g., least squares).

3. Apply the computed translations to tiles and compose them into a single large image.

This work focuses on the first phase of the algorithm, namely the *relative displacements computation* phase, as it is the more compute-intensive one. The second phase is much lighter from a computational load point of view while the third phase can be carried out on demand as part of visualizing the stitched image.

Figure 6.1 shows the data-flow graph for computing the relative displacement between two adjacent images, i and j (east-west or north-south). The steps are outlined below. The algorithms in Figures 6.2 and 6.3 give the corresponding pseudo-code listings.

1. Read the two image files, F_i and F_j into image objects I_i and I_j.

2. Compute the 2D Fourier transforms of the two images (FFT$_i$, FFT$_j$).

3. Compute the image pair's Normalized Correlation Coefficient (NCC$_{ij}$). This is the element-wise normalized conjugate multiplication of two complex vectors.

4. Compute the 2D inverse Fourier transform of the normalized correlation coefficient (NCC$_{ij}^{-1}$).

5. Reduce the inverse transform to its maximum (max$_{ij}$), identify the index of this maximum, and map this index back to image coordinates (x, y).

   Fourier transforms are periodic in nature. As such, the overlap distances, x and y, are ambiguous and can be interpreted as either x or $(w - x)$ and as either y or $(h - y)$.

6. Compute the four Cross-Correlation Factors (CCF$_{ij}^{1,\cdots 4}$). Each cross-correlation factor corresponds to one overlap mode, (x or $w - x$) and (y or $h - y$).

7. Find CCF$_{ij}^{\max}$ and identify its corresponding displacement $(x, y)_{ij}$.

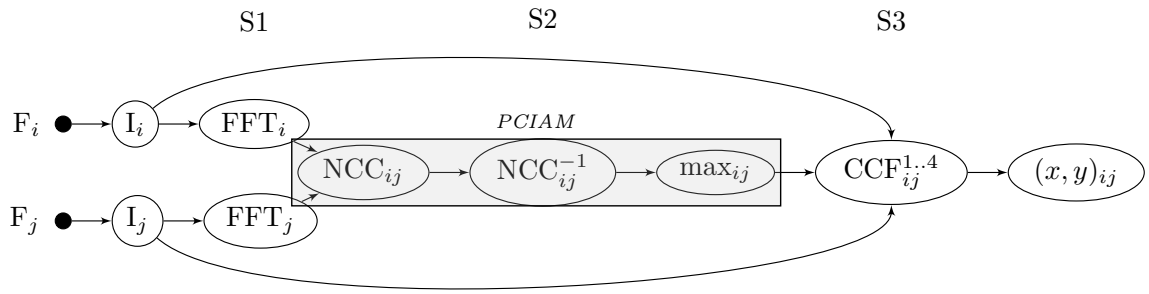

Figure 6.1: Data Flow of Computation for Two Adjacent Images

The computation for the whole grid repeats the pair-wise computation for all adjacent image pairs in the grid as listed in Algorithm 6.4.

The image stitching algorithm is compute-bound and is dominated by Fourier transform computations. Table 6.1 shows the count and complexity of operations as well as the sizes of the operands in these operations; in the table, n and m denote the grid size while h and w give the size of the partial images. Processing an $n \times m$ grid performs $(3nm - n - m)$ forward and backward 2-D Fourier transforms on double complex numbers. The cost of each transform is $Ohw \log(hw)$ when h and w have a special form, a power of small prime numbers (e.g., 2, 3, 5, & 7) or a product of such powers, and the FFT library uses a divide and conquer approach to take full advantage of the recursive formulation of FFT. For optical microscopy, there is no guarantee that the partial images will have such *nice* dimensions and the cost of these transforms may be substantially higher. The image stitching computation also includes a large number of vector multiplications and reductions; these operations can become comparatively expensive unless they are implemented using SSE intrinsics.

For the class of problems under consideration (plates of nearly 2500 images), the relative displacement computation exhibits a high degree of coarse-grain parallelism: computing the forward transforms of all images (FFTs), computing the normalized correlation coefficients of all adjacent image pairs (NCCs), computing the inverse transforms of all NCCs, etc. However, this computation is not embarrassingly parallel because of data dependencies and memory size limits.

- There are two sets of computed entities with multiple dependencies, NCC_{ij} and $CCF_{ij}^{1..4}$. A parallel implementation must explicitly handle these data dependencies

function PCIAM(I_i, I_j) ▷ Disp. of 2 adj. images
  $\texttt{FFT}_i \leftarrow \text{FFT\_2D}(I_i)$ ▷ Forward FFTs
  $\texttt{FFT}_j \leftarrow \text{FFT\_2D}(I_j)$
  ▷ Normalized Correlation Coeff.
  $\texttt{fc} \leftarrow \texttt{FFT}_i \quad .\times \overline{\texttt{FFT}_j}$ ▷ elt wise op
  $\texttt{NCC}_{ij} \leftarrow \texttt{fc./|fc|}$ ▷ normalize
  ▷ Find max in Inverse FFT
  $\texttt{NCC}_{ij}^{-1} \leftarrow \text{IFFT\_2D}(\texttt{NCC}_{ij})$
  $[\texttt{max}, \texttt{y}, \texttt{x}]_{ij} \leftarrow \text{MAX}(|\texttt{NCC}_{ij}^{-1})$
  ▷ Consider four combinations
  $c_1 \leftarrow \text{CCF}(I_i[\texttt{y} : \texttt{H}, \texttt{x} : \texttt{W}], I_j[\texttt{0} : \texttt{H} - \texttt{y}, \texttt{0} : \texttt{W} - \texttt{x}])$
  $c_2 \leftarrow \text{CCF}(I_i[\texttt{y} : \texttt{H}, \texttt{W} - \texttt{x} : \texttt{W}], I_j[\texttt{0} : \texttt{H} - \texttt{y}, \texttt{0} : \texttt{x}])$
  $c_3 \leftarrow \text{CCF}(I_i[\texttt{H} - \texttt{y} : \texttt{H}, \texttt{x} : \texttt{W}], I_j[\texttt{0} : \texttt{y}, \texttt{0} : \texttt{W} - \texttt{x}])$
  $c_4 \leftarrow \text{CCF}(I_i[\texttt{H} - \texttt{y} : \texttt{H}, \texttt{W} - \texttt{x} : \texttt{W}], I_j[\texttt{0} : \texttt{y}, \texttt{0} : \texttt{x}])$

  $\texttt{return max}([c_1, \texttt{x}, \texttt{y}], [c_2, \texttt{W} - \texttt{x}, \texttt{y}],$
    $[c_3, \texttt{x}, \texttt{H} - \texttt{y}], [c_4, \texttt{W} - \texttt{x}, \texttt{H} - \texttt{y}])$
end function

Figure 6.2: Relative Displacement of Adjacent Images

across CPU and GPU threads.

- An implementation must also manage memory because the problem does not fit into main memory, let alone GPU memory. Each transform takes up nearly $22\,\mathrm{MB}$ in RAM. This results in a total of $53.5\,\mathrm{GB}$ just for the forward transforms of the grid! Such a size is well beyond the capacity of most machines. This constraint is substantially more severe with GPUs where even high end GPUs are limited to $6\,\mathrm{GB}$.

The challenges in developing parallel implementations lie in exploiting the available coarse grain parallelism by scheduling computations on the available computing resources (CPU and GPU cores) as early as possible without violating any of the data dependency constraints and memory size limits.

function CCF(I_i, I_j) ▷ Cross correlation factor

$I_i \leftarrow I_i - \text{MEAN}(I_i)$ ▷ Center both vectors
$I_j \leftarrow I_j - \text{MEAN}(I_j)$

$N \leftarrow I_i.I_j$ ▷ dot product
$D \leftarrow |I_i|.|I_j|$ ▷ product of norms

`return` N/D ▷ **`double`**
end function

Figure 6.3: Fourier Cross Correlation Coefficients

for each $I \in$ Grid of Tiles **do**
$\texttt{translations\_west}[I] \leftarrow \text{PCIAM}(I, I\#\texttt{west})$
$\texttt{translations\_north}[I] \leftarrow \text{PCIAM}(I\#\texttt{north}, I)$
end for

Figure 6.4: Grid Relative Displacements

6.5 Implementations

We describe the implementations that we have developed below. Our evaluation machine has the following hardware specifications:

- two Intel Xeon E-5620 CPUs (2.4 GHz, quad-core with hyper threading),
- 48 GB of RAM (4 GB DDR3 1333 MHz modules), and
- two NVIDIA Tesla C2070 cards with 6 GB of GDDR5 memory each with ECC turned off; the cards attach to the motherboard at PCIe Gen. 2.0 x16 slots.

Our software stack is as follows:

- Ubuntu Linux 12.04/x86_64, kernel v. 3.2.0,
- Libc6 v. 2.15, libstd++6 v. 4.6, BOOST v. 1.48 (boost 2012),

Table 6.1: Operation Counts & Complexities

Operation	Op. Count	Op. Cost	Opd. Size
Read	$n \times m$	$h \times w$	2 bytes
FFT-2D	$n \times m$	$hw \log(hw)$	16 bytes
$\otimes$	$2nm - n - m$	$h \times w$	16 bytes
FFT-2D^{-1}	$2nm - n - m$	$hw \log(hw)$	16 bytes
/max	$2nm - n - m$	$h \times w$	16 bytes
CCF$^{1..4}$	$2nm - n - m$	$h \times w$	4 bytes

- GCC version 4.6.3 with `-O3` optimization, and
- NVIDIA CUDA and cuFFT v. 5.0.

6.5.1 Reference Implementations

We developed two reference implementations: (1) a sequential CPU-only version and (2) a simple GPU version that is almost a direct port of the sequential CPU version. We label these two implementations, *Simple-CPU* and *Simple-GPU*.

The reference CPU-only sequential implementation reads images files using libTIFF4 v. 3.9.5 (libTIFF 2012) and uses FFTW3 v. 3.3 (Frigo & Johnson 2005a; Frigo & Johnson 2005b) to compute Fourier transforms. We explicitly coded the functions for the element-wise vector multiplication and the max reduction with SSE intrinsics because the compiler being used (GCC v. 4.6.3 with `-O3` optimization) was not generating such code.

This implementation used a strategy of freeing memory as early as possible: it freed an image's transform memory as soon as the relative displacements of its eastern and southern neighbors were computed. For this purpose, this implementation supported multiple traversal orders of the grid (row, column, diagonal, and their chained counterparts). The chained-diagonal traversal order gave the best performance because it allowed memory to be freed earlier than the other traversal orders. Consequently, the chained-diagonal traversal order became the default.

This reference implementation computes the relative displacements for the 59×42 grid on the evaluation machine in $20.5\,\mathrm{min}$ with 84% of this time spent on Fourier transforms when using FFTW in its *estimate* planning mode. FFTW is an auto-tuning

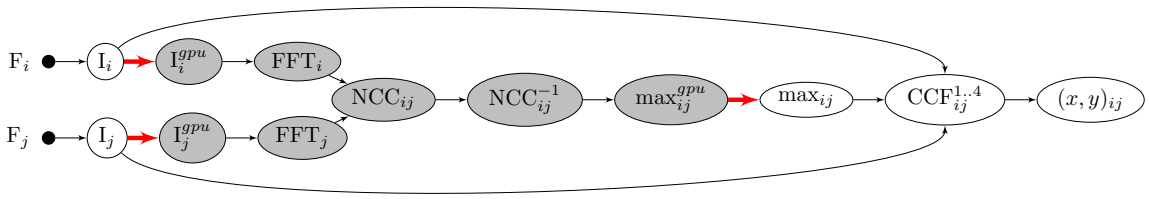

Figure 6.5: Data Flow in Sequential GPU Implementation

library which operates in two modes, *planning* and *execution*. It first generates a plan, based on the problem and machine characteristics, which it then executes. When using FFTW's *exhaustive* planning mode, the sequential execution time, excluding planning cost, drops to $10.6\,\mathrm{min}$. Exhaustive planning is expensive; it takes nearly $10\,\mathrm{min}$ to generate the plan in exhaustive mode. However, a plan can be saved and used by multiple runs to amortize the cost of planning.

We used the CPU-only sequential implementation to develop a simple multi-threaded implementation. This implementation uses spatial domain decomposition and a thread-variant of the SPMD (Single Program Multiple Data) approach to handle coarse-grained parallelism. The best execution times were $4.5\,\mathrm{min}$ with 8 threads when using FFTW's estimate planning mode and $96\,\mathrm{s}$ with 16 threads when using exhaustive planning.

The first GPU implementation, *Simple-GPU*, is almost a direct port of the CPU sequential version. Figure 6.5 illustrates the dataflow underlying this implementation. In this figure, entities that reside in GPU memory are shaded in gray while CPU entities are not shaded. This dataflow differs from the one shown in Figure 6.1 by having two *copy* operators, shown as thick arrows, that copy (1) image data from CPU to GPU memory (step 1.5) and (2) the maximum reduction result from GPU to CPU memory (step 5.5). The augmented dataflow consists of the steps below that execute on the CPU or are orchestrated in sequence on the GPU by the CPU.

1. Read an image file from disk.

1.5 Copy the image data from host to device memory.

2. Compute the image's transform on the GPU by using NVIDIA's cuFFT library. The resulting transform is kept in GPU memory for later use with neighboring images.

3. Invoke the NCC kernel on the device for every pair of neighboring images whose transforms are available. This kernel uses fast shared memory as well as other GPU kernel optimizations.

4. Compute the inverse FFT from the NCC result on the GPU by using NVIDIA's cuFFT library.

5. Invoke our parallel reduction kernel on the GPU for finding the max element and its index. The parallel reduction routine is an adaptation of NVIDIA's parallel reduction for finding the sum of a vector's elements.

5.5 Copy the single index value back to the host.

6. Compute the four Cross-Correlation Factors.

7. Find the max CCF and identify its corresponding displacement.

The implementation allocates a pool of buffers in GPU memory for FFT transforms and keeps track of these buffers on the CPU to get around the limited amount of memory available in the GPU.

Table 6.2: Profile of Reference Sequential Implementations

		CPU		**GPU**	
Function	**Count**	**T (ms)**	**%**	**T (ms)**	**%**
read	2478	3.51	1.13	3.91	1.59
copy	2478	4.31	1.39	6.58	2.68
FFT	2478	75.18	24.35	50.96	20.75
FFT^{-1}	4855	82.22	52.17	48.57	38.75
NCC	4855	21.16	13.42	9.40	7.50
max	4855	5.87	3.73	3.88	3.10
CCF	19420	1.49	3.77	2.10	6.69
GPU copy	2478			6.06	2.47

The reference GPU implementation is single threaded, executes CUDA memory copies synchronously, and invokes all kernels on the default stream. Nevertheless, it includes several features that were put in place to improve performance:

- The implementation uses NVIDIA's cuFFT library (NVIDIA Corp. 2012a) to compute FFTs, but does so without padding the data. The partial images in our data set are 1392×1040 with prime factors of 2, 3, and 29 and 2, 5, and 13 respectively. These sizes are definitely not ones favored by many FFT implementations. A comparison of computing FFTs on the CPU versus the GPU reveals that the GPU only gains a factor of ≈ 1.5x over FFTW using exhaustive planning!

- It has a custom-written GPU kernel for computing the normalized cross correlation. This kernel uses shared memory and maximizes the occupancy of the graphics card. This kernel runs ≈ 2.3x faster than the corresponding CPU function.

- It has a custom-written max reduction kernel. This kernel implements a variant of the parallel reduction kernel (Harris 2012) that is distributed with NVIDIA's "GPU Computing SDK" (NVIDIA Corp. 2012b). Our reduction kernel uses the same optimizations and obtains a performance increase of ≈ 1.5x over the CPU.

- The last step on the GPU reduces an image to a single value, the index of the maximum in its NCC. The implementation transfers this single value from graphics to host memory, thereby opportunistically minimizing the volume of transferred data.

- The implementation copies image data to the GPU only once per image. It frees an image tile's GPU memory only when all the neighboring tiles have been computed. In order to do so, the implementation maintains a data structure on the CPU side to keep track of a tile's state; this data structure includes a reference count that is initialized to the number of times the tile will be used for computing relative displacements.

The reference GPU implementation stitches the 59×42 grid in $9\,\mathrm{min}\ 16\,\mathrm{s}$, a mere 1.15x speedup over the reference CPU-only sequential implementation when run with FFTW's exhaustive planning mode. Table 6.2 compares the execution profiles of both reference sequential implementations, *Simple-CPU* and *Simple-GPU*.

As expected, porting the CPU code directly onto the GPU was not advantageous considering that we obtained a speedup of only 1.15x. We used NVIDIA's visual profiler (NVIDIA Corp. 2012c) to analyze the GPU serial implementation and noticed an overhead of ≈ 0.03 to 0.25 s between tile computations. Figure 6.6 shows this overhead

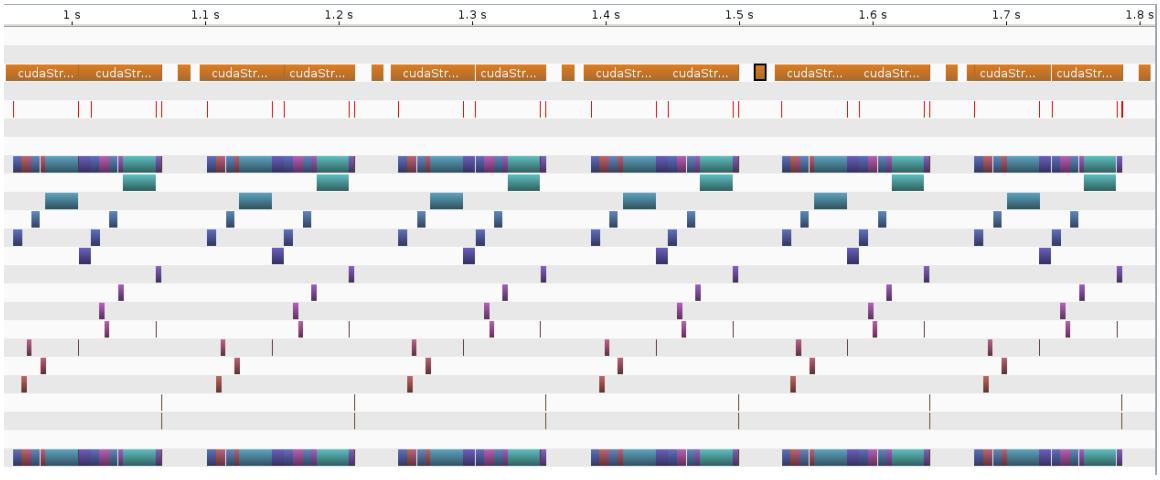

Figure 6.6: CUDA Profile of Reference GPU Implementation

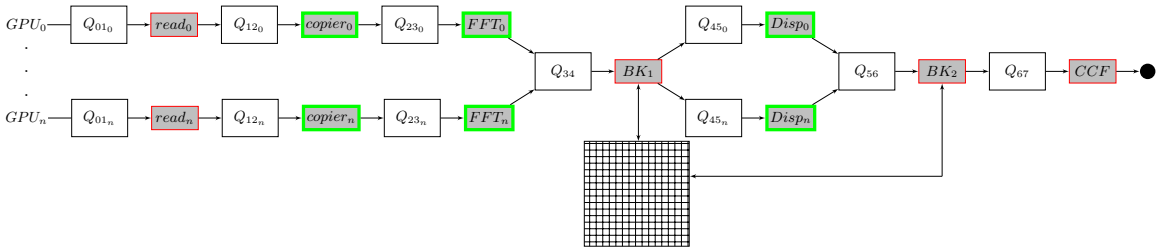

Figure 6.7: Pipelined GPU Structure

in a 1 s interval. The different colors represents the different kernels being executed on the GPU. The large gaps in this visual profile also indicate that the GPU is not being fully utilized and synchronization is preventing not only the GPU to have large wait periods, but the CPU as well.

The major factors contributing to this overhead are synchronously invoking kernels, waiting for CPU reads and computations, and copying between CPU and GPU memories. Each of these uses valuable cycles and keeps the GPU unoccupied. To overcome these problems, we decided to restructure the code with the goals of (1) overlapping data transfers with GPU computations and (2) overlapping CPU tasks such as reading and computing the CCFs. We developed a workflow system based on these constraints.

6.5.2 Pipelined GPU Implementation

The pipelined GPU implementation, *Pipelined-GPU*, is an implementation of a hybrid CPU-GPU workflow system that organizes the image stitching computation into a pipeline of six producer-consumer stages with the option of having multiple threads per stage. It structures the flow of data in the pipeline to guarantee keeping the GPU busy as soon and as often as possible and to also keep many of the CPU cores busy at the same time. The implementation partitions the grid into equal parts and instantiates one execution pipeline per GPU to take advantage of multiple GPUs when available.

Figure 6.7 shows the structure of the pipeline. Each stage has an input and an output queue; the threads of a stage consume from its input queue and produce for its output queue. These queues can be considered as having monitor implementations to prevent race conditions. Images are processed in the pipeline as follows:

- One *reader* thread per GPU reads image tiles. Each image tile is initialized with its row/column index, and which GPU it is bound to.
- One *copy* thread per GPU then copies each tile to the GPU and invokes a kernel to transform the 16-bit image into a buffer of double complex numbers. GPU memory for the input and output buffers are allocated here using our memory manager.
- Tiles enter the FFT stage which computes forward FFTs using invocations of cuFFT.
- The first bookkeeping stage gathers FFT computations from all execution pipelines and manages the state of tiles and resolves dependencies. Tiles are grouped into *ready* pairs and are sent to the next stage. The pairs is sent to the appropriate GPU.
- Pairs of adjacent tiles (north-south or east-west) enter this stage which invokes the NCC computation, the inverse transform, and the maximum reduction. This stage also copies the index of the maximum to the host. GPU memory of pairs that exist on the spatial boundary of the grid will be automatically copied using peer to peer copies which copies GPU memory without going through host CPU memory.
- In the second bookkeeper stage, pairs of images are gathered and a reference count is decremented. Once the reference count of an image reaches zero, then the memory for that image can be released into the memory manager.

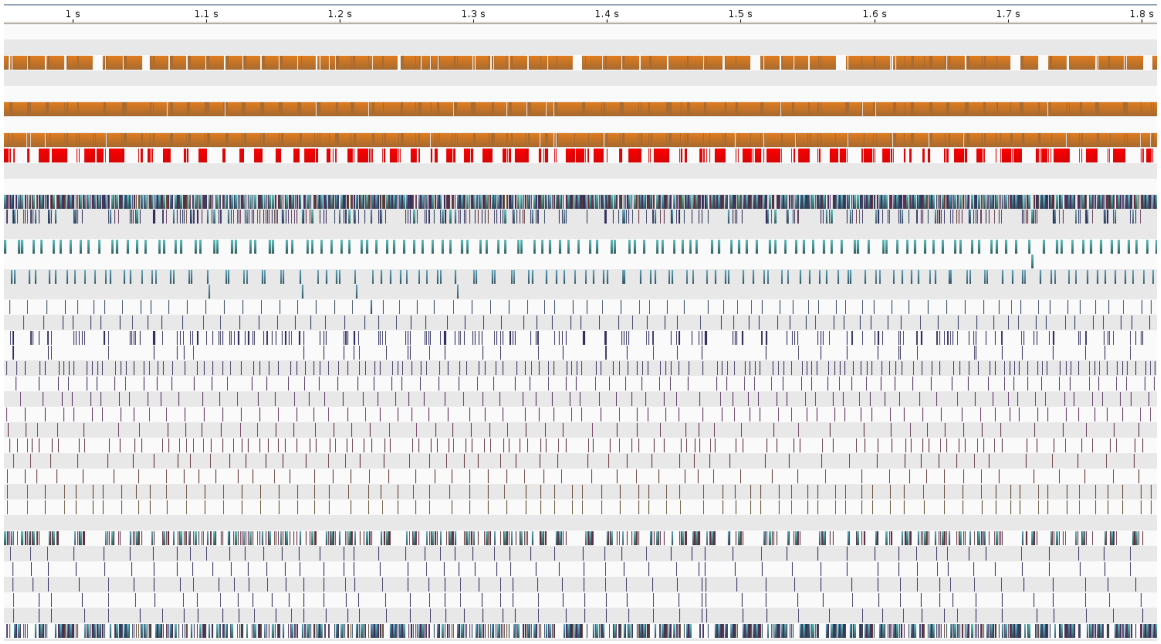

Figure 6.8: CUDA Profile of Pipelined workflow GPU Implementation

- In this stage, a pool of CPU threads translate the index of the max value into image coordinates and compute the four $\mathrm{CCF}_{ij}^{1,\cdot 4}$ values. This yields the final x and y relative displacement for the image pair.

Each stage in the GPU-side of the pipeline generates its own streams. Unfortunately, CUFFT kernels are resource hungry (e.g., registers) and, as such, cannot run concurrently with other kernels on NVIDIA's Fermi architecture. Despite this drawback, having a separate stream per stage enables computations on the GPU to overlap with the memory transfers. This is illustrated in the visual profile of Figure 6.8 of a $1\,\mathrm{s}$ interval. Comparing this profile with that of the previous version's profile, we can identify a number of major differences. First, the pipelined workflow profile shows many GPU kernels being executed in no particular order and all the computations are overlapping with PCI express memory transfers. In contrast, the reference GPU implementation's profile have many gaps in time. The pipeline workflow implementation enables for full occupancy of all GPUs and as seen in the profile, it keeps things very busy. (Figure 6.6).

As with the GPU reference implementation, the pipelined version maintains a data structure that holds state information for each tile. However, it includes additional variables

due to the multiple stages and threads within the pipeline. This new data structure is maintained by two bookkeeper threads to minimize the number of synchronizations. Both bookkeepers are extremely light weight threads, which are only responsible for processing the state of image tiles as they flow through the pipeline. The first bookkeeper thread advances pairs of neighboring tiles to the next stage when ready. The second bookkeeper thread decrements a reference count for each image tile, which is used to determine when an image tile can be freed. Using this system, we maximize the usage of each image tile as to only free it when all neighboring tiles have been computed. This effectively caches the tile's FFT in GPU memory until it is ready to be freed.

The overall runtime compared to the CPU single threaded version when using a single GPU and our workflow system achieves a speedup of 14.8x and can process the 59×42 in 43 seconds; this is a speedup of almost 13x with respect to the reference GPU implementation. Our test machine has two GPUs and can process the grid in 25 seconds resulting in a speedup of 25.5x. It improves on the serial GPU implementation by a factor of 22.2x. Compared to the commonly used Fiji image stitching application and using one or two GPUs gains speedups of 301.4x and 518.4x respectively. The Fiji image stitching application is using an identical algorithm to ours and is implemented in Java with multi-threading on the CPU. Comparing the Fiji application with our reference CPU and GPU implementations we see speedups of 20.4x and 23.2x respectively. Gaining the extra order of magnitude is attributed to our workflow system effectively scheduling tasks on all available resources and overlapping all disk and PCI-express I/O.

CUFFT is capable of batching multiple FFTs into a single execution; this can boost performance for doing multiple FFTs of the same dimension. We developed a micro-benchmark to analyze the difference in computation times between batching FFTs and computing FFTs individually. The results of the micro-benchmark showed that executing a batch of 2x2 image tiles can obtain a performance increase of ≈ 9 times for our images.

An implementation of the pipeline, which batched FFT computations, was developed. However, its results showed that the performance was the same as the original pipelined version. We attribute this to the additional waiting times that must be incurred to group images into batches.

6.6 HTGS Microscopy Image Stitching

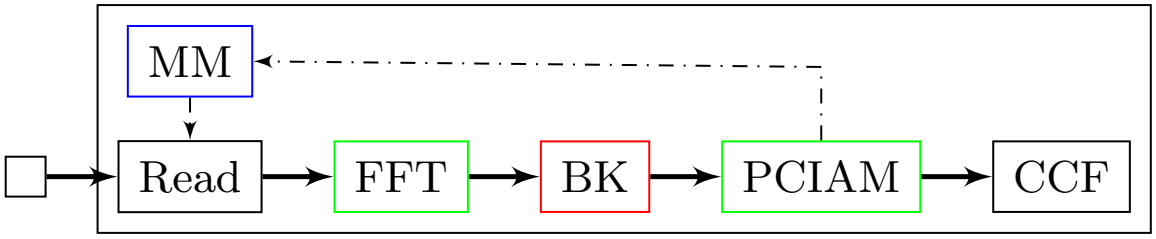

Figure 6.9: Hybrid image stitching task graph (machine with 1 GPU).

As shown in Figure 6.9, hybrid image stitching consists of six tasks. The six tasks are listed below:

1. **MM** manages CUDA FFT memory.
2. **Read** loads an image from disk.
3. **FFT** copies an image to the GPU and computes the forward fast Fourier transform on the GPU.
4. **BK** identifies when two neighboring images have their FFTs computed.
5. **PCIAM** computes the phase correlation image alignment method between two tiles on the GPU and copies the single scalar back to the CPU.
6. **CCF** computes the cross correlation factors on the CPU.

There is one dependency that requires the FFTs of two neighboring tiles to be computed before processing the PCIAM function. When an image's FFT is available, the FFT can be used in computations with its four cardinal neighbors. To avoid unnecessary FFT computations, the memory manager uses a reference count to keep FFTs in memory. The reference count refers to the number of times an image's FFT is used with its four neighbors (three for boundary cases, and two for the corners).

The task graph in Figure 6.9 will execute on one GPU only. To scale to multiple GPUs, the task graph is partitioned into two task graphs; (1) a GPU task graph and (2) a

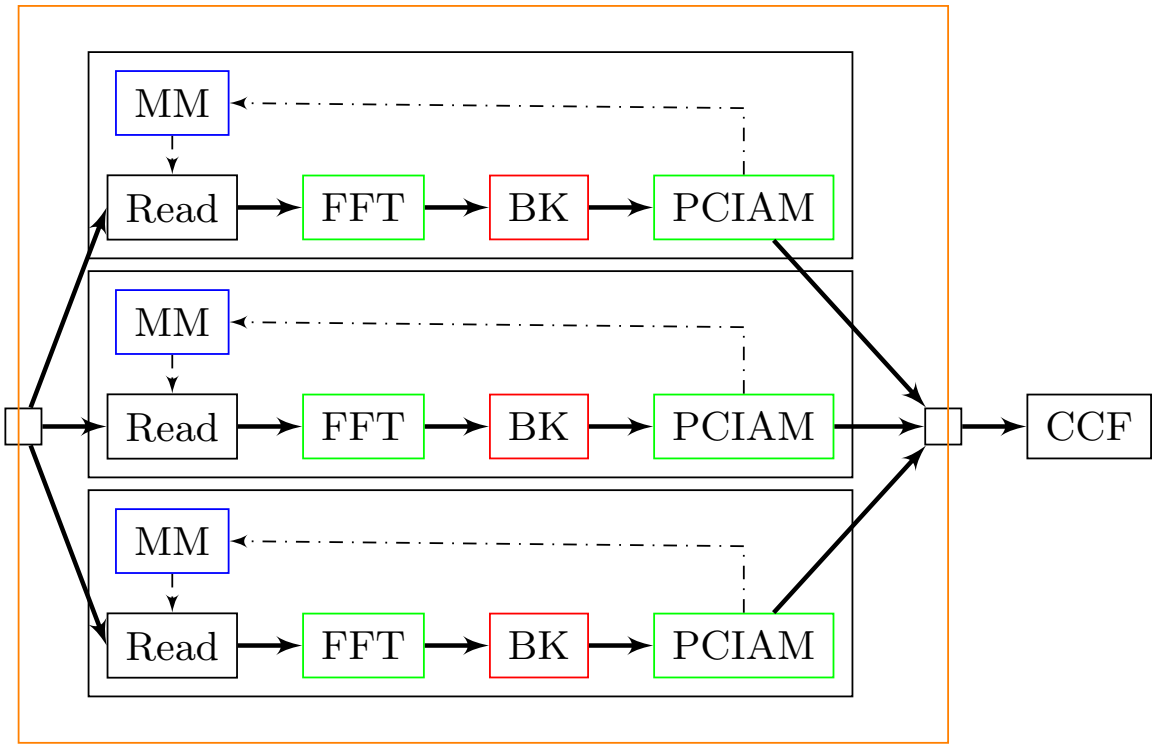

Figure 6.10: Hybrid image stitching with execution pipeline (3 GPUs).

container task graph to hold an execution pipeline task and the CCF task. The GPU task graph is added to the execution pipeline as shown in Figure 6.10.

The execution pipeline copies the GPU task graph. The number of copies generated is specified by the programmer and each copy is bound to a separate GPU. The image tile grid is decomposed evenly such that each copy processes a different non-overlapping region. The CCF task remains outside of the execution pipeline and uses a pool of CPU threads to process CCFs.

6.7 Discussion

Table 6.3 summarizes the timing results of the hybrid workflow implementation and indicates that we have achieved an order of magnitude performance improvement. Our evaluation machine for the hybrid pipeline workflow implementation uses two quad-core Intel Xeon CPUs, 48 GB of RAM, and two NVIDIA Tesla C2070 GPUs.

We started with two reference implementations that computed the relative displacements of each image pair in roughly 10 minutes for both the CPU and GPU. After profiling each implementation, we determined that parallelizing the program in a pipeline fashion was optimal in order to overlap disk, the PCI express, and computation on the GPU and CPU. From this implementation, we were able to obtain a speedup of 13x and compute the 59×42 grid in 43 s on a single Tesla C2070. By partitioning the grid and setting up one execution pipeline per GPU, we were able to obtain nearly a 1.7x speedup by going from one GPU to two, ultimately computing the entire grid in 25 seconds, which is a 22.2x speedup from our reference implementations.

Table 6.3: Runtime and speedup results of the reference and hybrid pipeline workflow implementations.

Implementation	Time	Speedup	Effective Speedup	Threads	GPUs
Fiji IS plugin	3.6 hr	–	–	16	–
Reference CPU	10.6 min	–	20.4	–	–
MT CPU	1.6 min	6.6	135	16	–
Simple GPU	9.3 min	1.05	2.3	–	1
Pipelined GPU	43 s	14.8	301.4	9	1
Pipelined GPU	25 s	25.5	518	11	2

Table 6.4 compares our novel HTGS-based implementation of hybrid microscopy image stitching with the implementation without HTGS (Blattner *et al.* 2014). Each test case is repeated 50 times using a grid of 42×59 images (6.6 GB) and the average end-to-end run-time is reported. The machine used has two Intel Xeon E5-2650 v3 CPUs (40 logical cores) and three NVIDIA Tesla K40 GPUs. The implementation is written in C++ and uses the C++11 standard for threading, CUDA 7.5 for GPU kernel invocations, and CuFFT 7.5 for FFT computations.

Table 6.4 shows that using HTGS without execution pipelines reduces the code size by $43.4\,\%$ compared to the original hybrid workflow. Including the execution pipeline enables the hybrid workflow to scale to multiple GPUs and obtains a performance improvement of 2.1x with three GPUs at the cost of *ten additional lines of code*. Execution from two to three GPUs shows little performance improvements due to hardware limitations within the

Table 6.4: Runtime results of the HTGS Prototype for hybrid microscopy image stitching.

HTGS	Exec Pipeline	GPUs	Runtime (s)	Lines of Code
✗	✗	1	17.278	1232
✗	✗	2	9.721	1232
✗	✗	3	8.301	1232
✓	✗	1	17.232	697
✓	✓	1	17.235	707
✓	✓	2	9.537	707
✓	✓	3	8.102	707

PCI express. The lack of PCI express lanes on the Xeon E5-2650 v3 socket was unable to feed all three GPUs. Additionally, the third GPU was incapable of residing on the same PCI express bus as the first two GPUs, which prevented the third GPU to do GPU-direct peer-to-peer PCI express transfers. Peer-to-peer transfers allows data to be shared between GPUs on the same PCI express bus without the need to copy data back to the CPU. For the third GPU, any data that resided on another GPU would have to first be copied back to the CPU, and then copied onto the third GPU. This additional overhead impacted the performance when adding the third GPU.

The results compared with they hybrid workflow implementation and HTGS using 3 GPUs show a $42.6\,\%$ reduction in code size, while maintaining the same relative performance. Hybrid workflows are effective at parallelizing an algorithm, hiding data motion, and keeping processors busy. HTGS reduces the effort required to represent hybrid workflows in image stitching, while maintaining the performance of manually creating a hybrid workflow. HTGS also provides a framework for representing algorithms and tools for complex, data-intensive applications that require very high performance.

Our results demonstrate a highly effective mechanism for structuring the image

stitching problem and could be utilized for a variety of other problems that are not embarrassingly parallel. We will demonstrate two numerical linear algebra routines using the HTGS model and API and how it compares with modern high performance implementations.

Chapter 7

CASE STUDY 2: MATRIX MULTIPLICATION

Matrix multiplication is a well studied algorithm that has many parallel characteristics. Implementing matrix multiplication using HTGS provides insights into approaching other linear algebra problems that share similar data access behaviors as matrix multiplication.

The product of two matrices, of dimensions M, N, and P, $C^{M\times P} = A^{M\times N} \times B^{N\times P}$, is computed by multiplying row entries from matrix A by column entries from matrix B, then adding their products into matrix C. To exploit parallelism and pipelining, we split matrix C into square blocks. A sub-matrix $C_{i,j}$ is computed by multiplying and adding the horizontal and vertical slices of $A_{i,1:k}$ and $B_{1:k,j}$, such that $C_{i,j} = \sum_{k=1}^{N} A_{i,k} \times B_{k,j}$, as shown in Figure 7.1.

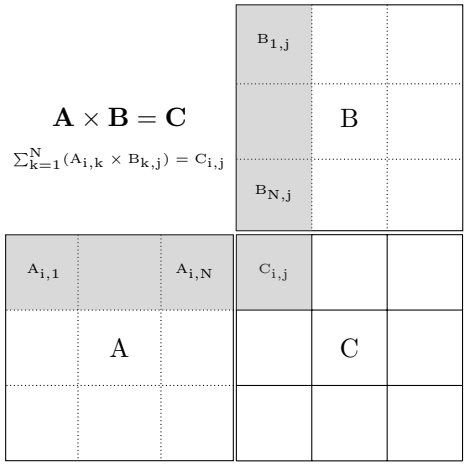

Figure 7.1: Block matrix multiplication.

Within the algorithm there are two primary computational routines. First, the matrix multiplication (GEMM) between two blocks of matrices, which produces a partial result of C, and second the accumulation (Acc) of the partial results. Both GEMM and Acc calls can be done in parallel assuming the data has been loaded and multiplied, respectively. Algorithm 2 provides the pseudo code for block matrix multiplication. This implementation traverses the blocks along the shared dimension of A and B. Using this data traversal pattern computes the sub-result of C as quickly as possible; however, each of the blocks from A and B will have to be reused for other regions of C. This data access pattern provides excellent performance if the sub-result of C is to be released/written as quickly as possible, but will require A and B to persist in memory longer. To maximize the instruction level parallelism, the GEMM function from line 6 in Algorithm 2 is replaced with a function call to an optimized basic linear algebra routine (BLAS), such as from OpenBLAS.

Algorithm 2 Block Matrix Multiplication

1: **function** BLOCK-GEMM($A, B, C, blksize$)
▷ Matrix dimensions: $A^{M\times N}$ $B^{N\times P}$ $C^{M\times P}$
2: $b = blksize$
3: **for** $I = 1 : M$ in steps of b **do**
4: **for** $J = 1 : P$ in steps of b **do**
5: **for** $K = 1 : N$ in steps of b **do**
6: *GEMM(*$A_{(I:I+b,\, K:K+b)}$,
$B_{(K:K+b,\, J:J+b)}$,
$C_{(I:I+b,\, J:J+b)}$*)*
7: ***end for***
8: ***end for***
9: ***end for***
10: ***end function***

11: ***function*** GEMM*(A, B, C)*
▷ *Matrix dimensions:* $A^{m\times n}$ $B^{n\times p}$ $C^{m\times p}$
12: ***for each*** $i \in m$ ***do***
13: ***for each*** $j \in p$ ***do***
14: $sum = C_{i,j}$
15: ***for each*** $k \in n$ ***do***
16: $sum = sum + A_{i,k} \times B_{k,j}$
17: ***end for***
18: $C_{i,j} = sum$
19: ***end for***
20: ***end for***
21: ***end function***

From the algorithmic analysis of the matrix multiplication, we formulate the dataflow representation, shown in Figure 7.2. Two nodes are defined to represent the matrix multiplication and accumulate routines. The matrix multiplication node processes two blocks and produces one partial result. The accumulate node sums all partial results until each block has been fully accumulated.

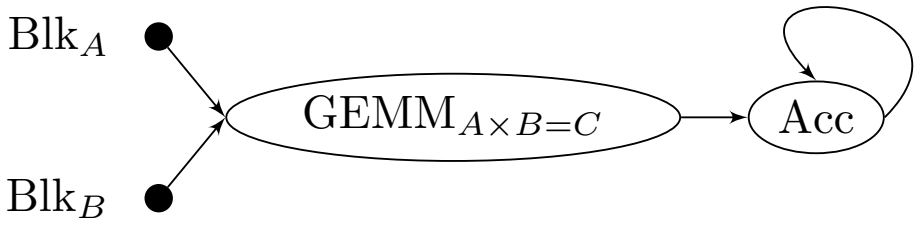

Figure 7.2: Matrix multiplication dataflow.

Using the dataflow graph, we have identified two dependencies that must be satisfied to process the GEMM and Acc nodes. First, two blocks of A and B must be loaded into memory, and second, the Acc node requires to accumulate the results of the GEMM node until all partial results have been accumulated. Using this, we formulate the matrix multiplication task graph, shown in Figure 7.3.

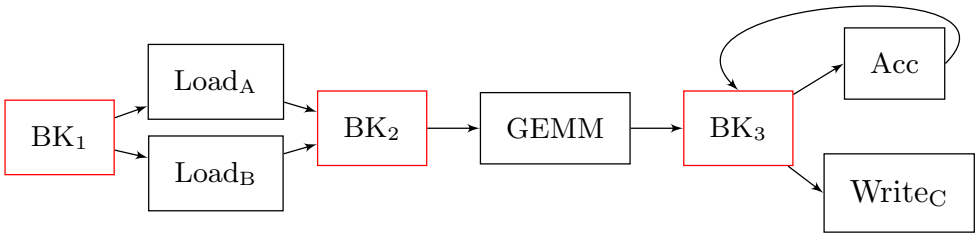

Figure 7.3: Matrix multiplication task graph.

In this implementation, we specify two separate loading tasks for A and B to process each load independently. The first bookkeeper (BK_1) distributes the data between the two load tasks based on the input into the task graph. The second bookkeeper (BK_2) initiates the GEMM computation after the appropriate blocks for A and B are loaded. The third

bookkeeper (BK_3) gathers the data produced by the GEMM task and produces for the Acc task to accumulate the partial results for C. When a sub-block of C has been fully accumulated, then BK_3 produces for the $\mathrm{Write}_\mathrm{C}$ task to write the result to disk.

7.1 Matrix Multiplication on the CPU Results

We implemented the task graph from Figure 7.3 using the HTGS C++ API and executed it on the CPU. The hardware configuration for this implementation uses two Intel Xeon E5-2650 v3 CPUs (40 logical cores) and 128 GB of DDR4 RAM. The system uses OpenBLAS v.0.2.18 DGEMM routine for the GEMM task, which is configured to use 1 CPU thread. As a baseline, we ran the problems as a one-off function call from OpenBLAS configured with 20 CPU threads. HTGS is configured with 20 and 10 threads for the GEMM and Acc thread pools, respectively.

Table 7.1: Matrix multiplication OpenBLAS vs HTGS Runtime for $16k^2$ and $32k^2$ matrices in memory.

Test Case	Matrix Sizes	Block Size	Runtime (s)
OpenBLAS	16384^2	N/A	16.3
HTGS	16384^2	2048^2	15.1
HTGS	16384^2	4096^2	17.6
HTGS	16384^2	8192^2	35.5
OpenBLAS	32768^2	N/A	125.4
HTGS	32768^2	2048^2	120.7
HTGS	32768^2	4096^2	120.3
HTGS	32768^2	8192^2	137.5

Table 7.1 shows the runtimes for two problem sizes 16384^2 and 32768^2, which gains $8\,\%$ and $4\,\%$ in runtime using 2048^2 and 4096^2 block sizes, respectively. These results show the minimal overhead that HTGS has for scheduling matrix blocks within the matrix multiplication task graph. The matrices in this experiment were pre-loaded into RAM. Next, the same matrices are stored on an SSD and the files are memory mapped. Table 7.2 shows the runtimes for this experiment.

Table 7.2: OpenBLAS vs HTGS Runtime for $16k^2$ and $32k^2$ matrices on disk.

Test Case	Matrix Sizes	Block Size	Runtime (s)
OpenBLAS	16384^2	N/A	32.2
HTGS	16384^2	2048^2	28.0
HTGS	16384^2	4096^2	24.4
HTGS	16384^2	8192^2	42.0
OpenBLAS	32768^2	N/A	296.0
HTGS	32768^2	2048^2	204.4
HTGS	32768^2	4096^2	170.9
HTGS	32768^2	8192^2	156.2

Comparing Tables 7.1 and 7.2, OpenBLAS percent decrease when adding the disk is $49.4\,\%$ and $57.6\,\%$ for $16k^2$ and $32k^2$, respectively. HTGS, on the other hand, has a percent decrease of $38.1\,\%$ and $23.0\,\%$ for $16k^2$ and $32k^2$, respectively, comparing the optimal block sizes between the two experiments. This demonstrates the effectiveness of HTGS to overlap computation with data transfer costs. Identifying the optimal block size is important to obtain good performance. Figure 7.4 and 7.5 show HTGS processing matrices on disk at varying block sizes and thread configurations. The best block size provides sufficient data to be sent throughout the graph to ensure optimal pipelining and parallelism.

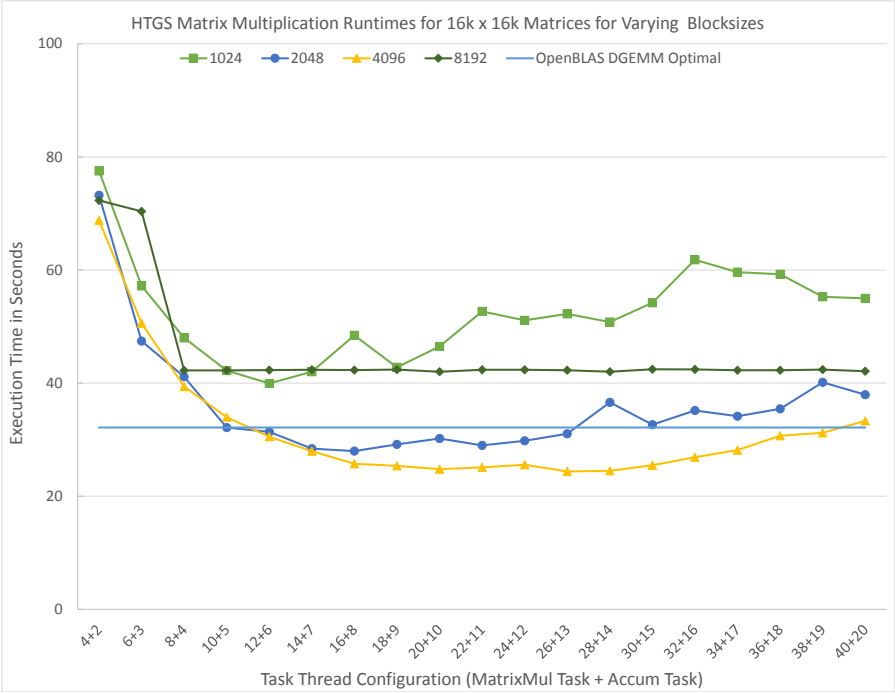

Figure 7.4: Runtimes for $16k^2$ matrices on disk at varying block sizes and thread configurations.

The results thus far assume that the matrix fits in CPU memory. To accommodate larger matrices, it is necessary to analyze the memory behavior of matrix multiplication in more depth to identify a traversal that keeps sub-matrices local to the processing. By doing so, the sub-matrices can reside in memory for as long as needed until the next sub-matrix needs be loaded from disk. In the next section, we analyze this behavior and identify the optimal scheduling strategy using the GPU.

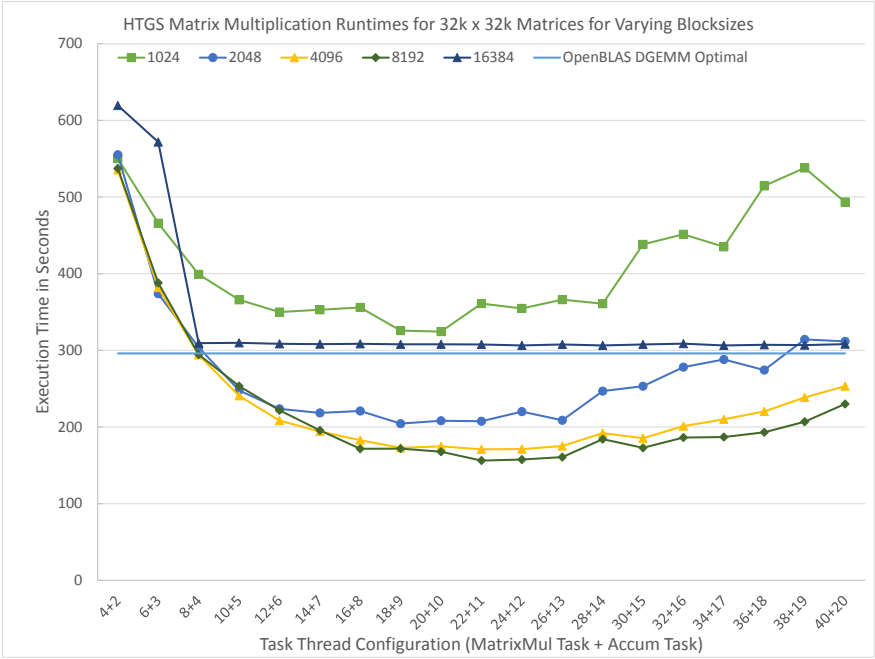

Figure 7.5: Runtimes for $32k^2$ matrices on disk at varying block sizes and thread configurations.

7.2 Matrix Multiplication on the GPU using HTGS

Implementing matrix multiplication on the GPU for large matrices has one primary challenge, memory. The amount of memory on the CPU surpasses the available memory on GPUs, i.e. 128 GB of RAM for a CPU versus 12 GB of RAM for the NVIDIA Tesla K40. The traversal strategy shown in Figure 7.1 computes a single block of the result matrix; however, the blocks from the input matrix need to be reused for each block of the result matrix, resulting in memory to persist longer. With a matrix size of $32k^2$, the amount of RAM used is 8 GB of RAM per matrix, or 24 GB for the two input matrices and one output matrix. With this traversal strategy and matrix size forces the GPU to copy sub-matrices to/from the CPU multiple times to fully compute $A \times B$, which will impact the overall utilization of the GPU.

Switching the computation from an inner traversal to an outer, as shown in Figure 7.6, changes the data access pattern that can reduce the overall memory requirements for processing the matrix multiplication. This traversal allows for A and B to be released

as soon as one traversal of a row/column has been completed. The new traversal strategy generates an entire copy of the result matrix, which needs to be accumulated for each row/column of the input matrices. To save on memory costs on the GPU, this step is offloaded to the CPU. This strategy assumes that the GPU can hold at least one entire column of blocks for A and one row of blocks for B, and the CPU can hold at least one copy of the result matrix.

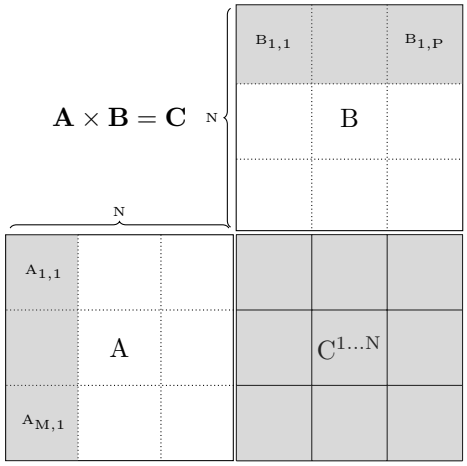

Figure 7.6: Matrix multiplication GPU data traversal.

This strategy is implemented into the existing CPU code by changing the traversal behavior when inserting blocks into the task graph. The GPU task graph implementation is made based on the original CPU task graph from Figure 7.3. The majority of the graph remains the same such as the matrix multiplication bookkeeper rule and load tasks. To incorporate the GPU into the graph required three modifications. First, the matrix multiplication function in the GEMM task is modified to use cuBLAS. Second, a memory manager edge is added for A, B, and C matrices to throttle the graph to ensure the blocks reside in GPU memory until they are ready to be released. Third, copy tasks are added to the graph to copy memory to/from the CPU and GPU. These GPU-based operations are encapsulated into a GPU task graph and inserted into an execution pipeline task. The resulting task graph is shown in Figure 7.7 for one pipeline and Figure 7.8 for 2 pipelines. The execution pipeline features a decomposition rule that sends a matching row/column from A and B to each GPU using round robin scheduling.

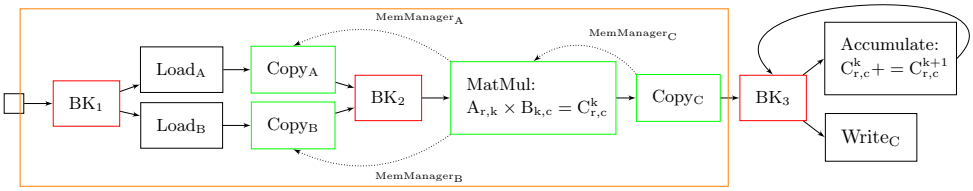

Figure 7.7: Matrix multiplication GPU task graph, 1 pipeline.

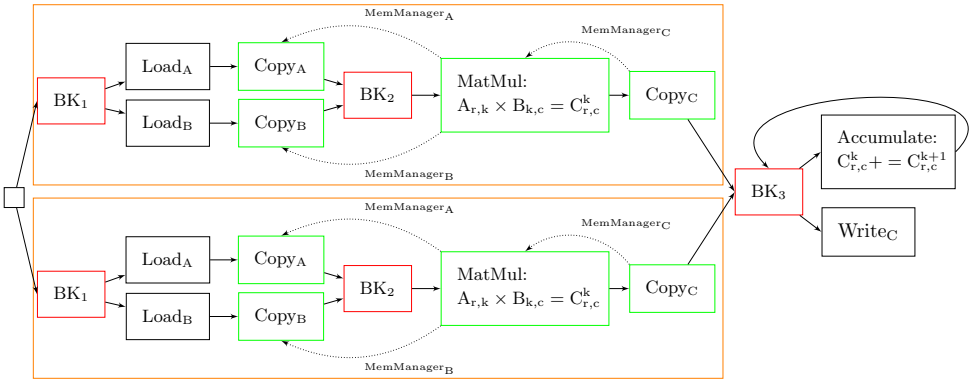

Figure 7.8: Matrix multiplication GPU task graph, 2 pipelines.

7.3 Matrix Multiplication on the GPU Results

We implemented the task graph from Figure 7.7 using the HTGS C++ API and executed it on the CPU and GPU. The hardware configuration uses two Intel Xeon E5-2650 v3 CPUs (40 logical cores), 128 GB of DDR4 RAM, and 4 NVIDIA Tesla K40 GPUs, each with 12 GB of GDDR5 RAM. The matrix multiplication task, uses the cuBLAS DGEMM routine. For comparison purposes, the cuBLAS-XT DGEMM routine is used as a one-off function call. The matrices in this experiment are pre-loaded onto the CPU, and both HTGS and cuBLAS-XT are responsible for managing the data transfers to/from the GPUs to process the matrix multiplication. The results using a block size of 1024^2 and multiple GPUs is presented in Table 7.3. Figures 7.9 and 7.10 show the runtimes of cuBLAS-XT and HTGS across multiple block sizes and number of GPUs.

Table 7.3: Matrix multiplication cuBLAS-XT vs HTGS Runtime for $16k^2$ and $32k^2$ matrices on GPUs.

Test Case	Matrix Sizes	Block Size	GPUs	Runtime (s)
cuBLAS-XT	16384^2	1024^2	1	18.7
cuBLAS-XT	16384^2	1024^2	2	10.9
cuBLAS-XT	16384^2	1024^2	3	8.3
cuBLAS-XT	16384^2	1024^2	4	7.5
HTGS	16384^2	1024^2	1	13.0
HTGS	16384^2	1024^2	2	7.0
HTGS	16384^2	1024^2	3	6.3
HTGS	16384^2	1024^2	4	5.5
cuBLAS-XT	32768^2	1024^2	1	139.0
cuBLAS-XT	32768^2	1024^2	2	77.9
cuBLAS-XT	32768^2	1024^2	3	55.2
cuBLAS-XT	32768^2	1024^2	4	48.9
HTGS	32768^2	1024^2	1	84.9
HTGS	32768^2	1024^2	2	45.7
HTGS	32768^2	1024^2	3	34.9
HTGS	32768^2	1024^2	4	30.7

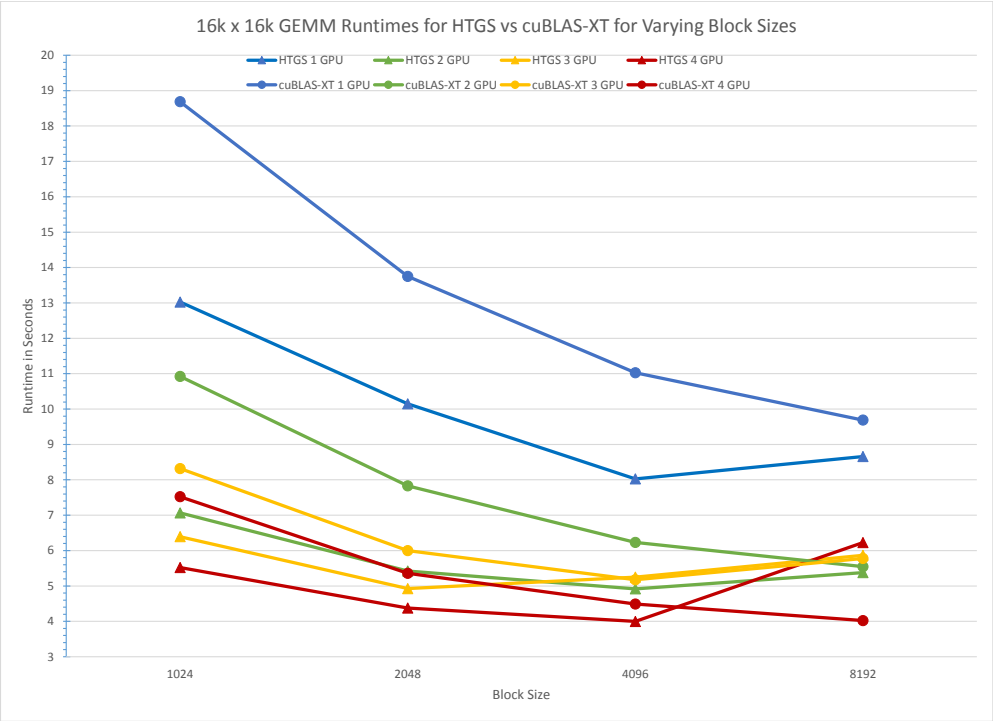

Figure 7.9: Runtimes for $16k^2$ matrices on the GPU with varying block sizes.

The results show that HTGS has less overhead when processing smaller blocks and is capable of using all GPUs, while overlapping PCIe transfers with compute. This is most apparent in observing the gradual slope of HTGS compared with cuBLAS-XT as the block size increases. In all test cases, HTGS improves upon cuBLAS-XT in runtime for both $16k^2$ and $32k^2$; however, as the block size increases, cuBLAS-XT obtains similar performance to HTGS. One significant improvement upon cuBLAS-XT is for smaller block sizes. For example, 1024^2 block sizes only require 256 MB and 512 MB of memory to process the input matrices for $16k^2$ and $32k^2$, respectively. HTGS processes these small blocks efficiently using only 2 GPUs, which obtained higher performance than cuBLAS-XT with 4 GPUs. HTGS further improves its performance for 1024^2 block sizes when adding 4 GPUs with HTGS, resulting in an additional 1.49x speedup compared to 2 GPUs for $32k^2$ matrices. This aspect showcases the high utilization and minimal overhead of HTGS in

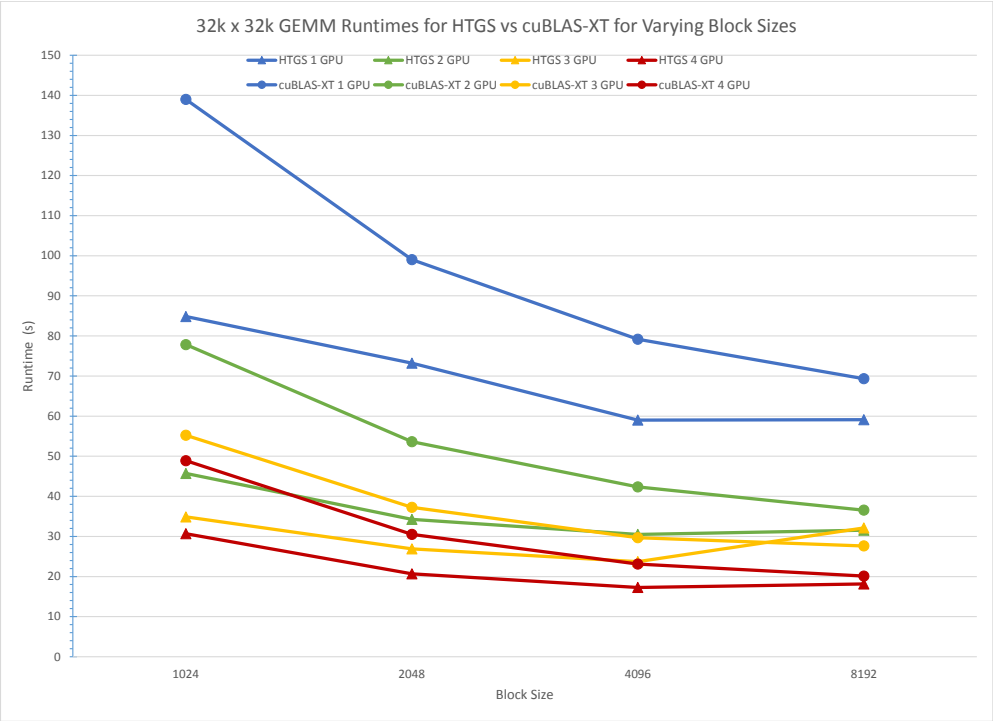

Figure 7.10: Runtimes for $32k^2$ matrices on the GPU with varying block sizes.

multi-GPU computation.

In this implementation, there is a load imbalance in the round robin scheduling. When the number of blocks is not evenly divisible by the number of GPUs, then near the tail end of the computation, one or more GPUs will become idle. This is most apparent for $16k^2$ matrix sizes with $8k^2$ block size and using 4 GPUs. One possible solution is to incorporate the CPU to process the tail end of the matrix multiplication on the CPU.

7.4 Discussion

The HTGS implementation of matrix multiplication implementation was done with a modest effort. This includes the parallel algorithm identification through literature

review, dataflow design, CPU task graph design/implementation, and GPU task graph design/implementation. With the low effort, we were still able to obtain similar or better performance compared to highly optimized implementations of matrix multiplication from OpenBLAS and cuBLAS-XT. Most noticeable are the improvements with obtaining performance while having to deal with disk I/O or PCI express transfers in both the CPU and GPU implementations, respectively. These improvements shows off two of the fundamental designs philosophies within the HTGS model, (1) explicitly handling data locality and (2) overlapping computation with data motion.

Chapter 8

CASE STUDY 3: LU DECOMPOSITION

LU Decomposition (LUD) is the process of representing a matrix as a lower and upper triangular matrix, $A = LU$. LUD can be used to help solve systems of linear equations, inverting a matrix, and has many applications. For example, to solve a system of linear equations, assuming a matrix is positively definite and given A = LU, solve for x in the equation $Ax = b$. First, solve the equation $Ly = b$ for y; then solve the equation $Ux = y$ for x. The cost of solving the system of linear equations with LUD is $2n^3/3+2n^2$, alternatively, computing A^{-1} to solve $x = A^{-1}b$ costs $2n^3 + 2n^2$, making solving systems of linear equations with LUD about 3 times faster.

The primary algorithm to compute $A = LU$ is Gaussian elimination, see line 12 in Algorithm 3. Despite the savings from LUD, the Gaussian elimination algorithm is mostly sequential and has little room for parallelization. However, using a block decomposition strategy transforms the LUD computation into a series of matrix multiplications, which becomes the dominant operation and exposes the algorithm to better parallelism. In the block LUD algorithm there are three steps; (1) Gaussian elimination, (2) Factor, and (3) Update. This process iterates using blocks, computing Gaussian eliminations along the diagonal of the matrix, such that after processing the diagonal of a block, that block is used to factor along the rows and columns of that block. The rows and columns are then used to update the remainder of the matrix. The updated blocks are then used to process the next diagonal, as shown in Figure 8.1.

This algorithm assumes there is no pivoting required, such that A is positively definite. Algorithm 3 shows the pseudo code for block LUD (Golub & Van Loan 1996). Transforming the algorithm in this way modularizes the approach into the three independent computational steps, which can be executed concurrently assuming data

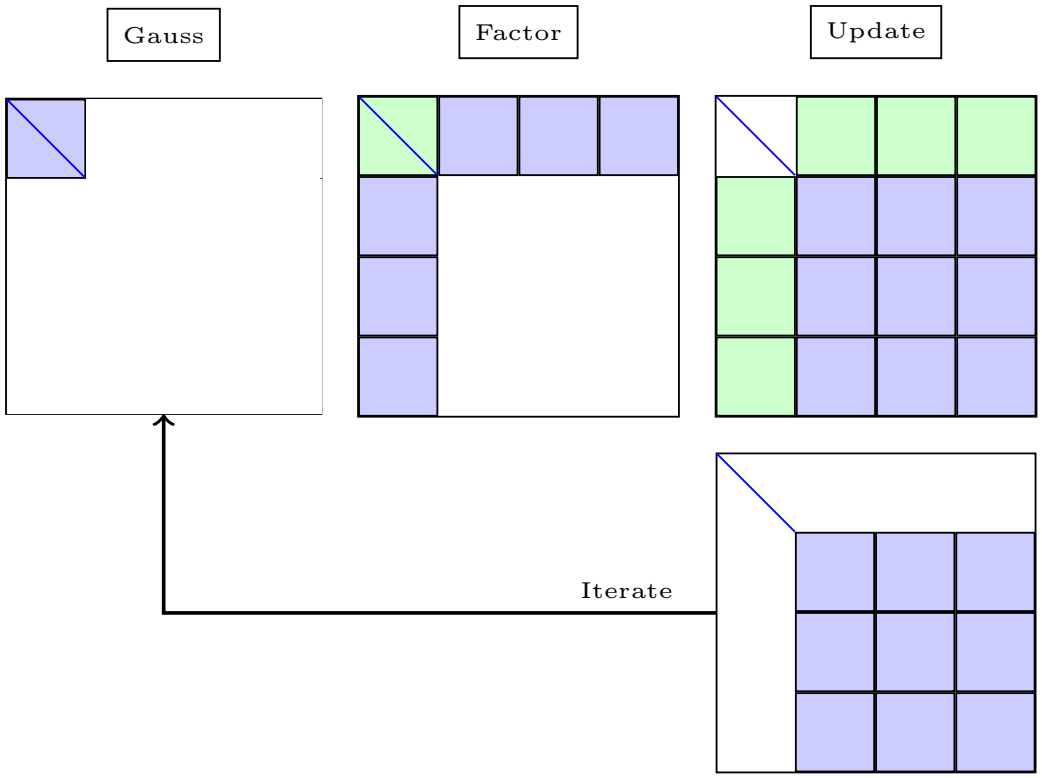

Figure 8.1: Block LU decomposition.

dependencies are satisfied. Using this algorithm, we design the dataflow representation, shown in Figure 8.2.

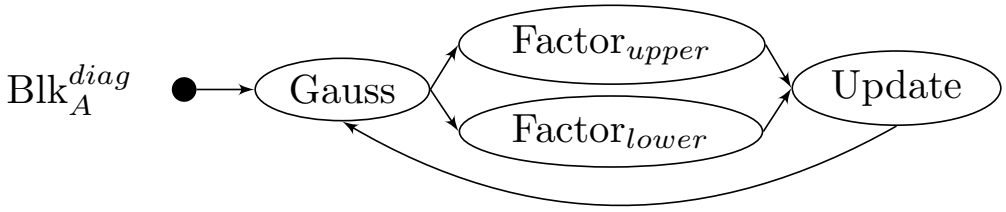

Figure 8.2: LU decomposition dataflow.

The LUD dataflow represents the data dependencies between all of the computational steps; however, it must wait for the update phase to finish processing to avoid factoring blocks prior to receiving updates. Adding state updates for this data dependency allows for

Algorithm 3 Block LU Decomposition

1: **function** BLOCK-LU($A, blksize$)
▷ A is non-zero and diagonally dominant
▷ $A_{i,j}$ is overwritten with $L_{i,j}$ for $i > j$
▷ $A_{i,j}$ is overwritten with $U_{i,j}$ if $j \geq i$
▷ Matrix dimensions: $A^{N \times N}$
2: $k = 1$
3: **while** $k \leq N$ **do**
4: $\mu = min(N, k + blksize - 1)$
5: $L, U =$ ***GaussElim(*** $A_{(k:b,k:b)}$ ***)***
6: *Solve:* $LZ = A_{(k:b,b+1:N)}$
7: *Solve:* $WU = A_{(b+1:N,k,b)}$
8: $A_{(b+1:N,b+1:n)} = A_{(b+1:N,b+1:n)} - W \times Z$
9: $k = b + 1$
10: ***end while***
11: ***end function***

12: ***function*** GAUSSELIM*(A)*
▷ *Matrix dimensions:* $A^{n \times n}$
13: ***for*** $k = 1 : n - 1$ ***do***
14: $rows = k + 1 : n$
15: $A_{(rows,k)} = A_{(rows,k)} / A_{(k,rows)}$
16: $A_{(rows,rows)} = A_{(rows,rows)} - A_{(rows,k)} \times A_{(k,rows)}$
17: ***end for***
18: ***end function***

the Gaussian elimination to start the next diagonal while the update phase is still processing the previous diagonal. This design is represented in the HTGS task graph.

In Figure 8.3, the dataflow representation is converted into a task graph representation. Each compute node is mapped to a task, bookkeepers are added to manage the state of the computation, and edges connect each task based on data dependencies.

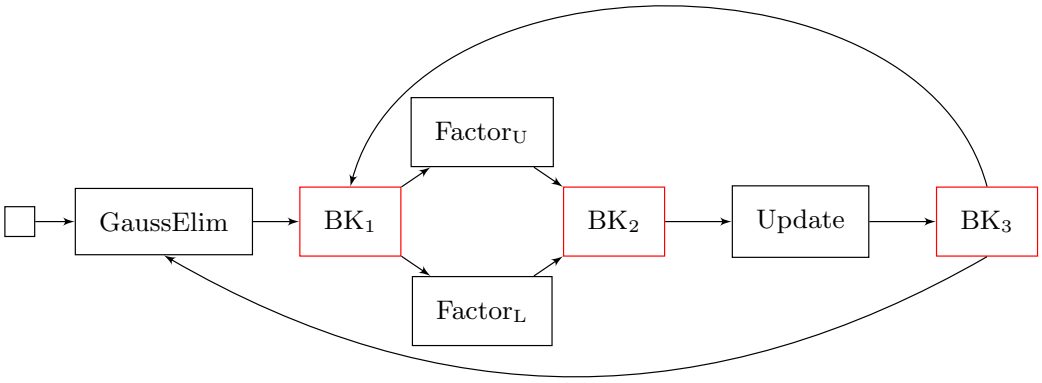

Figure 8.3: Block LU decomposition task graph on the CPU.

All of the tasks within this graph execute concurrently and the bookkeepers produce work when dependencies are satisfied. BK_1 updates the state from BK_3 and $\mathrm{GaussElim}$. Using this state, BK_1 determines when the upper or lower factor tasks are ready to receive work. BK_2 is responsible for gathering the factored blocks and produces work for the update routine. BK_3 holds onto two rules. First, BK_3 produces data for the $\mathrm{GaussElim}$ task if the block received is along the diagonal and that block has been fully updated. Second, BK_3 produces data to allow BK_1 to update the factoring state. BK_3 contains the rules that enable the Gaussian elimination task to begin processing the next diagonal block as soon as it has been updated, even if the entire matrix has not been updated. This design allows for concurrent execution that emphasizes processing Gaussian eliminations as quickly as possible.

8.1 Block LUD CPU Results

Using the task graph from Figure 8.3, we implement block-LUD using the HTGS C++ API and for the CPU. The hardware configuration for this implementation uses two Intel Xeon E5-2650 v3 CPUs and 128 GB of DDR4 RAM. We used the OpenBLAS v.0.2.18 library for the tasks. DGEMM for the Update task, DGETRF for the $\mathrm{GaussElim}$ task, and DTRSM for the $\mathrm{Factor_U}$ and $\mathrm{Factor_L}$ tasks. The matrices used are positively definite, so no pivoting is required. As a comparison, we ran OpenBLAS DGETRF as a one-off call with 20 threads. HTGS is configured with 1 thread for the $\mathrm{GaussElim}$ task, 10 threads for both $\mathrm{Factor_U}$ and $\mathrm{Factor_L}$ tasks, and 20 threads for the Update task.

Table 8.1 shows the runtime for the HTGS implementation of block LUD. The results show that deciding on the block size for the problem significantly alters the performance gains, particularly when increasing the number of unknowns. We compare these results with Table 8.2, which shows the OpenBLAS one-off call of DGETRF for the same matrices. This shows that OpenBLAS has been optimized more efficiently than this initial LUD implementation.

Table 8.1: Block LU decomposition HTGS CPU runtimes.

Test Case	Unknowns	Block Size	Runtime (s)	GFlops
HTGS	10000	250	1.8	352.5
HTGS	10000	500	2.8	221.4
HTGS	20000	250	11.5	432.0
HTGS	20000	500	11.49	432.4
HTGS	40000	250	101.4	392
HTGS	40000	500	82.9	479.7
HTGS	50000	250	208.0	373.2
HTGS	50000	500	157.6	492.3
HTGS	60000	250	346.7	386.9
HTGS	60000	500	282.7	474.5
HTGS	70000	250	566.0	376.4
HTGS	70000	500	441.4	482.7

The behavior of the HTGS implementation is attributed to the poor utilization of the CPU compared to that of OpenBLAS due to the small block sizes used for the update routine. Increasing the block size causes the Gaussian elimination task to overwhelm the computation, resulting in worse performance than the smaller block sizes. Additionally, the smaller block sizes utilized the CPU far less than using a larger block size. Therefore, the task graph needs to increase the amount of computation done within the Update task, while maintaining the small block size for the Gaussian Elimination.

Table 8.2: LU decomposition OpenBLAS CPU runtimes.

Test Case	Unknowns	Block Size	Runtime (s)	GFlops
OpenBLAS	10000	N/A	1.7	366.7
OpenBLAS	20000	N/A	11.1	446.7
OpenBLAS	40000	N/A	77.2	514.3
OpenBLAS	50000	N/A	149.9	518.0
OpenBLAS	60000	N/A	257.5	520.8
OpenBLAS	70000	N/A	404.7	526.4

8.2 Block+Panel LUD

The block+panel LUD enables the matrix multiplication within the Update task to get better utilization, while also keeping the block size small for the Gaussian elimination. This procedure is similar to the task scheduler approaches from PLASMA and MAGMA. (Buttari *et al.* 2009) and (Dongarra *et al.* 2014). Figure 8.4 shows the block+panel LUD procedure. In this algorithm, both the Gaussian elimination and factor tasks use blocks, and the update routine operates using panels.

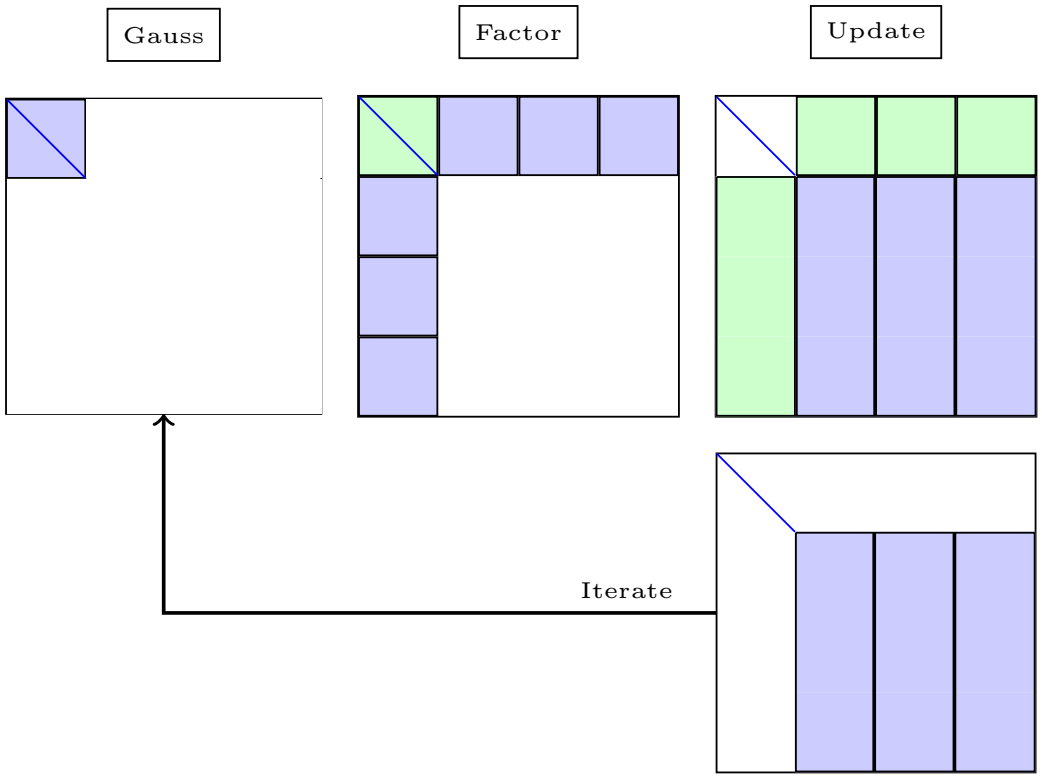

Figure 8.4: Block+Panel LU decomposition.

Using the block+panel modification, we adapt the original task graph from Figure 8.3 into a new task graph, which is shown in Figure 8.5. The new graph adds a new bookkeeper between the $\mathrm{Factor_L}$ and $\mathrm{BK_3}$, which collects blocks that are factored along the lower diagonal. Once all the blocks for a panel have been factored, then a panel is constructed for $\mathrm{BK_3}$. $\mathrm{BK_3}$ collects these panels and produces data for the Update task as they become available. The Update processes each update and sends the results to $\mathrm{BK_4}$. $\mathrm{BK_4}$ then decomposes the panels into blocks, which are sent to the $\mathrm{GaussElim}$ or $\mathrm{BK_1}$ tasks. The primary difference between the original, block-based approach and the block+panel approach is the different data output types of the tasks. Some tasks consume block data and produces panel data. This modification only required altering a few of the tasks and rules, whereas the core computational functions remained the same.

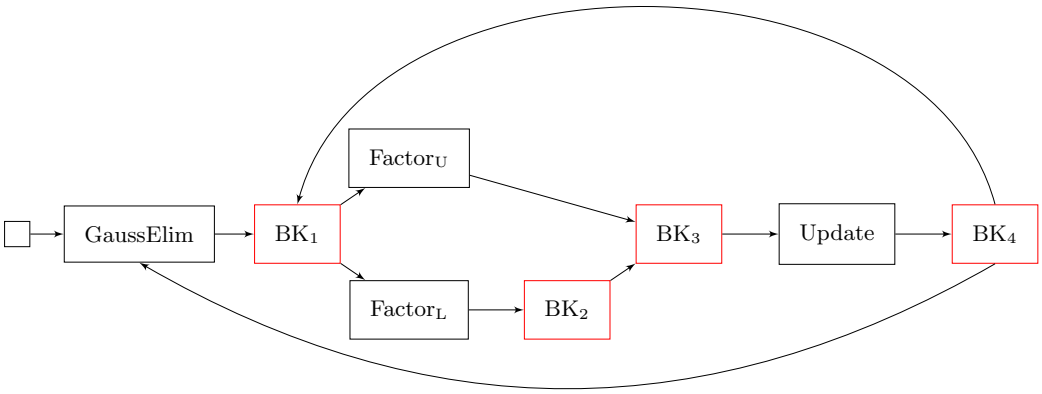

Figure 8.5: Block+Panel LU decomposition on the CPU.

8.3 Block+Panel LUD CPU Results

Using the task graph from Figure 8.5, we implement the block+panel LUD using the HTGS C++ API and execute it on the same hardware as the block LUD approach. Table 8.3 shows the runtimes for the block+panel LUD.

The results show that using the block+panel approach enables HTGS to acquire similar or better performance to that of OpenBLAS on the same matrices ranging by only a few percent difference. Adding the block+panel optimization to the original task graph requires minimal code modification at a high level of abstraction. Next, we look at porting the LUD algorithm to the GPU.

Table 8.3: Block+Panel LU decomposition HTGS CPU runtimes.

Test Case	Unknowns	Block Size	Runtime (s)	GFlops
HTGS	10000	250	2.0	308.0
HTGS	10000	500	3.7	167.2
HTGS	20000	250	10.8	461.6
HTGS	20000	500	12.8	389.8
HTGS	40000	250	81.8	486.0
HTGS	40000	500	79.1	502.1
HTGS	50000	250	159.3	487.2
HTGS	50000	500	152.8	508.2
HTGS	60000	250	273.5	490.4
HTGS	60000	500	257.7	520.4
HTGS	70000	250	432.5	492.5
HTGS	70000	500	404.0	527.2

8.4 LUD on the GPU

Implementing LUD on the GPU requires special consideration for memory due to the difference in capacity between the GPU and CPU memories, particularly when matrices will fit in the large CPU memory, but not within the GPU memory. Using the block LUD approach, additional memory copies are required to process the matrix, often requiring recopying data that had already been on the GPU, in particular for the Update task. Using the block LUD algorithm, we implement a GPU variant that adds copy tasks, memory

manager tasks, and execute tasks for the GPU. In LUD, tasks such as Gaussian elimination and factoring are not compute intensive, so these tasks remain on the CPU. The update task is the dominant operation in LUD and uses the matrix multiplication routine, so the GPU is a prime candidate for this task. To help minimize the number of memory copies needed to process each update, the lower diagonal is left in GPU memory until all of the updates have been processed for the active diagonal. The upper diagonal must be copied multiple times to update along each of the diagonals resulting in $\sum_{i=1}^{m-1}(m-i)^2$ total copies, where m is the number of blocks along the width of the matrix.

Using the block LUD task graph from Figure 8.1, we transform the task graph by adding copy routines, memory managers, and transform the Update task to execute on the GPU. Figure 8.6 shows the resulting task graph.

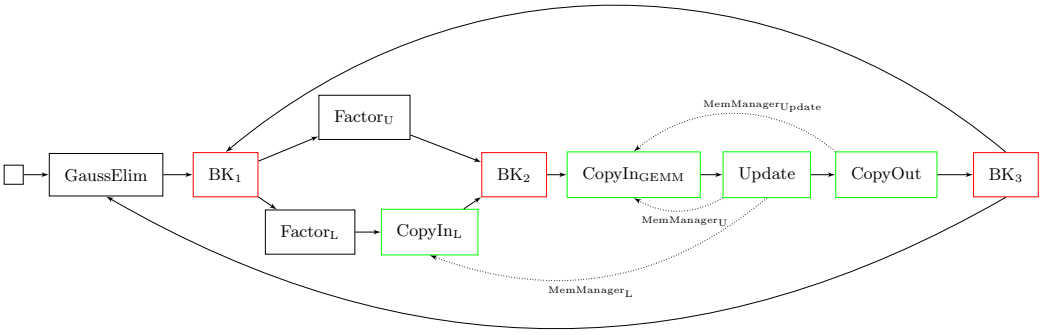

Figure 8.6: Block LU decomposition task graph on the GPU.

Three new tasks and three memory edges are added for the GPU version of block LUD. $\mathrm{CopyIn_L}$ is responsible for copying the lower diagonal matrices that have been factored onto the GPU. The lower diagonal matrices will reside in GPU memory until all updates have been applied for the active diagonal. $\mathrm{CopyIn_{GEMM}}$ copies the upper factored matrix and the result matrix for the Update GPU task. The Update task computes the matrix multiplication on the GPU, and the $\mathrm{CopyOut}$ task copies the result matrix from the CPU to the GPU. The memory for the upper diagonal and result matrices are used only once. As shown in Figure 8.6, the $\mathrm{CopyIn_L}$ task is placed before the $\mathrm{BK_2}$ bookkeeper. Having this task separate from the $\mathrm{CopyIn_{GEMM}}$ allows a different thread to be processing this copy. This is done to ensure that the lower diagonal remains in GPU memory until it is

no longer needed for the update routine for the current diagonal and to ensure the task will not have to wait for memory that is being requested in the $\mathrm{CopyIn}_{\mathrm{GEMM}}$ task.

8.5 Block LUD GPU Results

Using the task graph from Figure 8.6, we implement block-LUD using the HTGS C++ API for the GPU. The hardware configuration for this implementation uses two Intel Xeon E5-2650 v3 CPUs, 128 GB of DDR4 RAM, and one Tesla K40. CUDA and cuBLAS v7.5 is used for the GPU tasks. MAGMA v2.0 is used as a comparison, which contains GPU implementations of DGETRF.

Tables 8.4 and 8.5 show the runtimes on the GPU at varying block sizes and number of unknowns. The results show that the block LUD HTGS approach poorly utilized the Tesla K40 for problem sizes between 10000 and 40000 unknowns, reaching a peak of 383 Gflops for 40000 unknowns compared to the CPU achieving 520 Gflops for the same problem size. For larger problems, such as between 50000 and 70000 unknowns, the GPU implementation achieved 600 and 765 Gflops, respectively, showing higher utilization. These results demonstrate the impact of having enough data and computation per data when scheduling work on HTGS task graphs. Smaller block sizes are unable to keep the GPU busy for the block LUD approach. If the block size is too large and the number of unknowns is too small, then there will be insufficient data to fill the pipeline. These decisions must be balanced based on how the tasks interact with the data and the underlying performance of the task's kernel when operating on that data. The problem size also affects the performance, so finding the ideal block size based on the problem size is key to acquiring performance in LUD, particularly on the GPU for HTGS.

Table 8.4: Block LU decomposition HTGS GPU runtimes 10000 to 40000 unknowns.

Test Case	Unknowns	Block Size	Runtime (s)	GFlops
HTGS	10000	500	5.2	129.7
HTGS	10000	1000	4.8	139.7
HTGS	10000	2000	11.9	53.6
HTGS	10000	2500	21.8	29.0
HTGS	20000	500	38.3	138.5
HTGS	20000	1000	17.6	289.5
HTGS	20000	2000	35.2	142.7
HTGS	20000	2500	58.2	85.9
HTGS	40000	500	269.6	147.6
HTGS	40000	1000	112.8	353.6
HTGS	40000	2000	104.0	383.4
HTGS	40000	2500	150.3	265.2

Table 8.5: Block LU decomposition HTGS runtimes 50000 to 70000 unknowns.

Test Case	Unknowns	Block Size	Runtime (s)	GFlops
HTGS	50000	500	515.6	151.1
HTGS	50000	1000	219.2	354.7
HTGS	50000	2000	125.6	620.5
HTGS	50000	2500	159.7	487.2
HTGS	60000	500	918.2	146.1
HTGS	60000	1000	112.8	353.9
HTGS	60000	2000	199.5	673.7
HTGS	60000	2500	243.1	552.7
HTGS	70000	500	1419.9	150.2
HTGS	70000	1000	599.7	355.4
HTGS	70000	2000	278.5	765.6
HTGS	70000	2500	302.3	705.8

The block LUD approach on the GPU was able to acquire performance gains compared to that of the CPU for larger numbers of unknowns, shown in Table 8.5. We ran the same problem on the same hardware using the MAGMA library as a one-off call for DGETRF to compute the LUD. Table 8.6 shows the runtimes using one GPU.

Table 8.6: LU decomposition MAGMA one GPU runtimes.

Test Case	Unknowns	Block Size	Runtime (s)	GFlops
MAGMA	10000	N/A	3.5	571.1
MAGMA	20000	N/A	8.5	814.8
MAGMA	40000	N/A	45.9	915.3
MAGMA	50000	N/A	85.4	935.6
MAGMA	60000	N/A	143.9	948.3
MAGMA	70000	N/A	223.0	965.2

The results show that MAGMA does an excellent job utilizing the GPU and in all tests out performed the block LUD implementations from HTGS. To better understand the performance gap between HTGS and MAGMA, we use the HTGS profiling tools to visualize the task graph and identify how each task performed. One of the metrics that the profiling tools has is the ability to view the maximum queue size within each $\mathrm{Connector}$. This shows the maximum number of data elements that were residing in the queue throughout the entire execution. Figure 8.7 shows the profile graph.

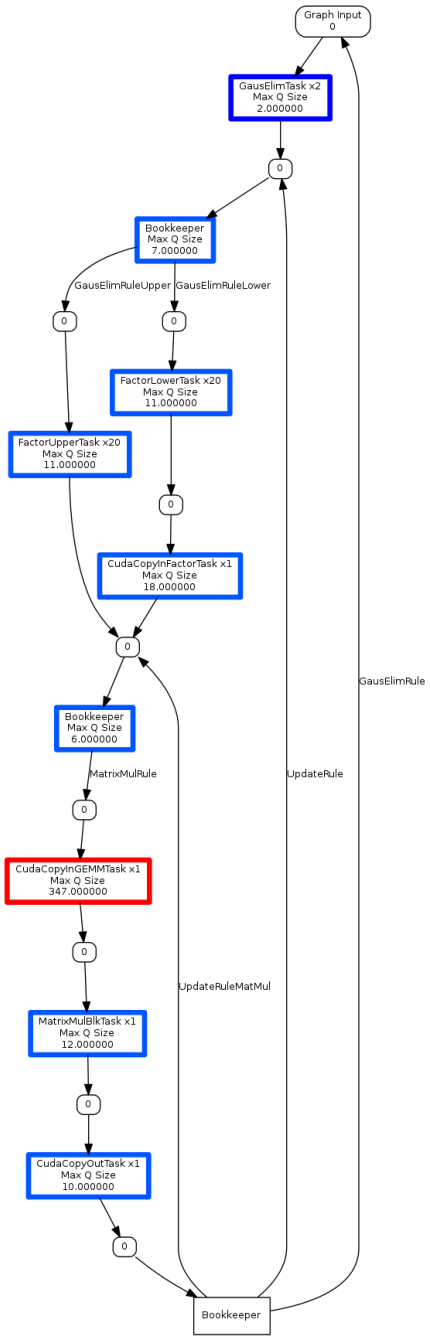

Figure 8.7: HTGS Block LU decomposition Max Q size profile for 70000 unknowns with 2000 block size on the GPU.

The profile graph from Figure 8.7 color codes the nodes based on the overall impact factor. Cooler blue colors have less of an impact than hot red colors. From the profile, we clearly see that the $\mathrm{CopyIn}_{\mathrm{GEMM}}$ task has the highest impact, reaching a maximum queue size of 1130. All of the other tasks do not go beyond 15 to 35 elements in their queues. This pin-points the issue to the copy task for matrix multiplication, which is impacting the utilization of the task graph.

The system we used has four Tesla K40 GPUs. An execution pipeline task must also be incorporated to scale the graph to these GPUs. Using the block LUD approach is challenging to implement for multi-GPU, particularly when trying to optimize the data locality Using the block+panel LUD implementation simplifies the memory management as the panels can be used to easily identify when an entire panel is ready to be freed and executed.

8.6 Block+Panel LUD on the GPU

Using the task graph from Figure 8.5, we implement a block+panel approach for the GPU. One of the main factors that we want to resolve from the block LUD approach is to improve the locality of data on the GPU by reducing the number of GPU data transfers required. In the previous version, the matrix being updated was continuously being copied to/from the GPU. With the panel-based approach, we have the opportunity to leave panels from the update region in GPU memory, which can be reused during an update for another diagonal. To accommodate this behavior, we incorporate a sliding window to represent the memory that is reusable. If panels are outside of the window, then those matrices will need to be copied to/from the GPU multiple times; however, once the update matrix fits entirely within the window, then no additional memory copies will be required for those updates. Figure 8.8 shows the sliding window approach for the Update task. Green blocks and panels represent factored matrices, blue panels are within the window and require updates, and red panels are outside of the window and will need to be copied multiple times to be updated until they fit inside of the window.

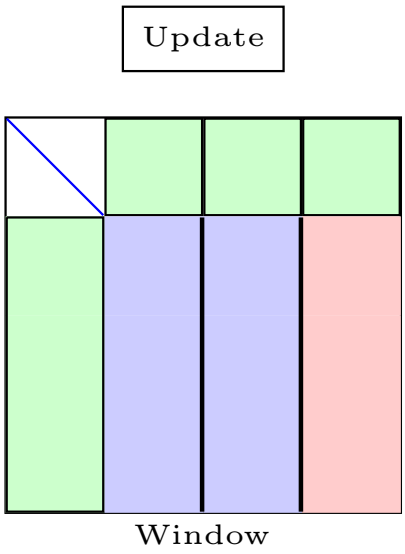

Figure 8.8: Window update.

To support multi-GPU execution, the main graph is partitioned into a sub-graph that is primarily GPU-based. Figure 8.9 presents the block+panel GPU task graph with a sliding window and execution pipeline.

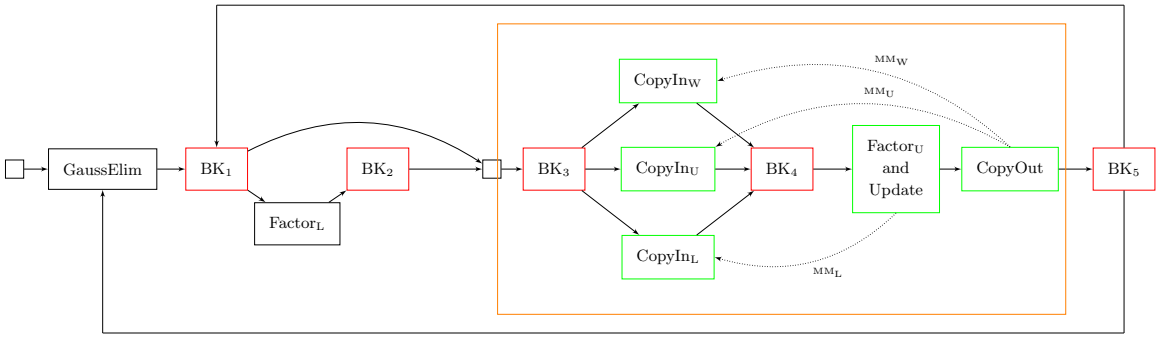

Figure 8.9: Block+Panel LU decomposition HTGS task graph for GPU with execution pipeline and sliding window.

Based on the block+panel task graph from Figure 8.5, the GPU version incorporates three main components; (1) a representation for the sliding window, (2) tasks for copying data in/out of the graph, and (3) memory edges to manage the locality of data. In addition,

we have merged the $\mathrm{Factor_U}$ and Update task into one task to prevent additional copies to/from the GPU for the factored region on the upper diagonal.

In the new task graph, there are now five bookkeepers to manage the state and distribute data. $\mathrm{BK_1}$ produces panels that are not factored. $\mathrm{BK_2}$ produces factored panels from the lower diagonal. These panels are collected by $\mathrm{BK_3}$, which are distributed to one of three CPU to GPU copy tasks. $\mathrm{CopyIn_L}$ copies the lower diagonal onto the GPU, $\mathrm{CopyIn_U}$ copies the upper diagonal for panels that are not within the sliding window, and $\mathrm{CopyIn_W}$ copies panels that are within the sliding window. The $\mathrm{CopyIn_W}$ task holds onto a cache of panels that are within the window. The cache is used to check if a panel has already been copied. For panels that are already within the cache, the data received from $\mathrm{BK_3}$ is updated with meta-data that describes the original cached panel and is immediately sent to the next task without a copy to the GPU. This process allows for the panels that are within the sliding window to only be copied once, and released as soon as all computations are done for that panel. The data that is copied in the $\mathrm{CopyIn_U}$ task is recopied each time until the un-factored matrix fits fully into the window.

The window size is specified to the $\mathrm{BK_3}$ task, which is used to determine the tasks that are in or out of the window. $\mathrm{BK_4}$ collects the panels and produces for the $\mathrm{Factor_UandUpdate}$ task, which is responsible for factoring the upper diagonal and updating the un-factored panel. The $\mathrm{CopyOut}$ task copies the results back from the GPU if one of two conditions are satisfied; (1) the panel is outside of the sliding window or (2) the panel is ready to be factored. $\mathrm{BK_5}$ produces work for two tasks. First, $\mathrm{BK_1}$ consumes data from $\mathrm{BK_5}$ to update the state and determine when a panel is ready to be factored. Second, $\mathrm{BK_5}$ produces work for the $\mathrm{GaussElim}$ task only if the panel being processed is ready to be factored.

The $\mathrm{BK_3}$, $\mathrm{CopyIn_W}$, $\mathrm{CopyIn_U}$, $\mathrm{CopyIn_L}$, $\mathrm{BK_4}$, $\mathrm{Factor_UandUpdate}$, and $\mathrm{CopyOut}$ tasks are all based on GPU computation, so these tasks are added into a separate GPU task graph. This new GPU task graph is encapsulated into an execution pipeline task, which is added into the main graph. The execution pipeline task has two data distribution rules. First, if the panel received is factored for the lower diagonal, then it is broadcast to all pipelines. This allows every pipeline to have its own copy of the factored lower diagonal panel. The second decomposition rule distributes the un-factored upper diagonal panels among pipelines using round robin scheduling that is generated during initialization. Because the round robin scheduling is decided at initialization time

simplifies the distribution and ensures that un-factored panels will always be sent to the same pipeline no matter what diagonal is being processed at the time. BK_3 is unaware of this decomposition, so the data that is received for a pipeline is assumed to be on the correct GPU based on these decomposition rules. Each of these un-factored panels are processed independently, allowing completely independent computation across multiple GPUs. Additionally, the multi-GPU execution allows for the window size to be multiplied by the number of GPUs, aggregating the memories across all GPUs.

There are three memory managers in this task graph. MM_L holds onto the memory for the lower diagonal panels, and contains rules that ensure the panel is available until all computation is done for that panel. MM_W allocates memory for the sliding window, which also ensures the panel will be available until all computation is done. MM_U allocates memory for data outside of the window, which will be recycled as soon as it is released. Each of these memory managers are within the execution pipeline task and are bound to their designated GPU address space based on the their pipeline ID.

8.7 Block+Panel LUD on the GPU Results

Using the task graph from Figure 8.9, we implement block+panel LUD using the HTGS C++ API for multiple GPUs. The hardware configuration for this implementation uses two Intel Xeon E5-2650 v3 CPUs, 128 GB of DDR4 RAM, and four Tesla K40s. CUDA and cuBLAS v7.5 is used for the GPU tasks. MAGMA v2.0 is used as a comparison, which contains a multi-GPU implementations of DGETRF.

Table 8.7 shows the runtimes for HTGS at optimal block sizes using one to four GPUs. The results show three interesting factors: (1) Having enough computation for a task; (2) The impact of data locality on the GPU; and (3) The scalability across multiple GPUs. The memory requirements for these problem sizes vary and impact whether the GPU is out-of-core on in-core. The Tesla K40 has 12 GB of RAM, the problem sizes for 10000, 20000, 40000, 50000, 60000, and 70000 use 0.745, 2.98, 11.9, 18.6, 26.8, and 36.5 GB of RAM, respectively. For problem sizes larger than 40000 unknowns, the GPU has insufficient memory to hold the entire matrix, forcing panels to be processed outside of the sliding window.

Table 8.7: Block+Panel LU decomposition HTGS GPU runtimes (optimal block sizes).

Test Case	Unknowns	Block Size	Wall Time (s)	Compute Time (s)	GFlops	GPUs	Relative Wall Speedup
HTGS	10000	250	2.1	1.6	392.2	1	1x
HTGS	10000	250	2.2	1.3	482.9	2	0.95x
HTGS	10000	250	2.6	1.32	469.9	3	0.8x
HTGS	10000	250	3.2	1.5	407.7	4	0.66x
HTGS	20000	500	8.4	7.9	630.7	1	1x
HTGS	20000	250	6.1	5.1	975.5	2	1.38x
HTGS	20000	250	5.5	4.1	1215.3	3	1.53x
HTGS	20000	250	5.6	3.8	1227.7	4	1.5x
HTGS	40000	500	51.4	50.8	782.2	1	1x
HTGS	40000	500	28.2	27.2	1465.2	2	1.82x
HTGS	40000	500	21.6	20.1	1976.5	3	2.38x
HTGS	40000	500	19.4	17.5	2275.8	4	2.65x
HTGS	50000	2000	112.8	122.3	634.7	1	1x
HTGS	50000	500	49.1	48.0	1615.9	2	2.3x
HTGS	50000	500	36.1	34.7	2237.6	3	3.12x
HTGS	50000	500	31.0	29.1	2669.9	4	3.64x
HTGS	60000	2000	213.0	212.4	631.4	1	1x
HTGS	60000	750	91.0	90.0	1490.3	2	2.34x
HTGS	60000	500	57.3	55.9	2287.8	3	3.72x
HTGS	60000	500	47.5	45.5	2948.6	4	4.48x
HTGS	70000	2500	332.1	331.6	642.4	1	1x
HTGS	70000	1000	170.7	169.7	1254.7	2	1.95x
HTGS	70000	500	94.6	93.1	2287.8	3	3.5x
HTGS	70000	500	70.3	68.4	3116.1	4	4.72x

For these problems, the single GPU was forced to load panels multiple times for updates. However, using the sliding window with multiple GPUs allowed for these

problems to become entirely in-core. The impact in performance when comparing 1 GPU versus 2, 3, or 4 GPUs showed super linear speedup, which is a direct result in the improved locality for problems that would then fit into the larger window. For smaller problems, such as 10000 unknowns, the problem size was too small to get significant performance due to low utilization. Increasing the block size for 10000 unknowns forced the pipeline to be less effective with overlapping the PCIe with computation.

As a comparison, MAGMA v2.0 is executed using the same matrices from the HTGS execution. The results are presented in Table 8.8. In comparison, with Table 8.7, results show that MAGMA and HTGS have very similar performance for problems that are in-core for the GPU, such as 70000 unknowns with 4 GPUs. For out-of-core matrices, MAGMA does a better job at scheduling, for example 70000 unknowns with 1 GPU. Although MAGMA is showing much better performance for these larger problems, the results demonstrate that, with a modest effort, HTGS was capable of obtaining performance comparable to a highly tuned and optimized implementation that is a feature within MAGMA.

Table 8.8: LU decomposition MAGMA multi-GPU runtimes.

Test Case	Unknowns	Block Size	Wall Time (s)	Compute Time (s)	GFlops	GPUs	Relative Wall Speedup
MAGMA	10000	N/A	3.5	1.1	571.1	1	1x
MAGMA	10000	N/A	3.4	1.0	617.3	2	1.03x
MAGMA	10000	N/A	3.4	1.0	577.7	3	1.03x
MAGMA	10000	N/A	3.6	1.2	520.2	4	0.97x
MAGMA	20000	N/A	8.5	6.1	814.8	1	1x
MAGMA	20000	N/A	6.4	4.0	1243.7	2	1.33x
MAGMA	20000	N/A	5.9	3.6	1398.3	3	1.44x
MAGMA	20000	N/A	6.3	3.8	1301.0	4	1.35x
MAGMA	40000	N/A	45.9	43.4	915.3	1	1x
MAGMA	40000	N/A	26.9	24.5	1619.9	2	1.7x
MAGMA	40000	N/A	21.3	18.9	2108.4	3	2.15x
MAGMA	40000	N/A	18.6	16.2	2459.7	4	2.47x
MAGMA	50000	N/A	85.4	83.0	935.6	1	1x
MAGMA	50000	N/A	48.9	46.5	1670.2	2	1.75x
MAGMA	50000	N/A	36.1	33.8	2298.4	3	2.37x
MAGMA	50000	N/A	30.8	28.4	2737.9	4	2.77x
MAGMA	60000	N/A	143.9	141.4	948.3	1	1x
MAGMA	60000	N/A	80.1	77.6	1728.9	2	1.8x
MAGMA	60000	N/A	59.3	56.9	2358.0	3	2.43x
MAGMA	60000	N/A	47.7	45.3	2959.7	4	3.02x
MAGMA	70000	N/A	223.0	220.6	965.2	1	1x
MAGMA	70000	N/A	124.7	122.3	1741.4	2	1.79x
MAGMA	70000	N/A	91.7	89.3	2384.0	3	2.4x
MAGMA	70000	N/A	74.7	72.3	2946.9	4	2.99x

Figures 8.10 and 8.11 show HTGS vs MAGMA end-to-end wall time for 60000 and 70000 unknowns and varying block sizes. These figures show that picking the best block

size is significant for gaining performance with LUD. MAGMA algorithmically determines the block size using an auto tuning approach prior to executing. The optimal block size evenly balances the data transfers, Gaussian elimination, and matrix multiplication, which results in evenly distributing the work and overlapping data motion with computation. Within the current design of HTGS, problems that are out-of-core for the GPU have significant overhead resulting in low utilization. Adding GPUs to these problems results in super linear speedup as the total number of PCI express transfers are reduced due to the problem becoming more in-core. These speedups are visualized within the two figures showing the poor performance with using 1 GPU versus adding more GPUs. For problems that are in-core, HTGS performs as good as or better than the MAGMA version assuming the optimal block size is used.

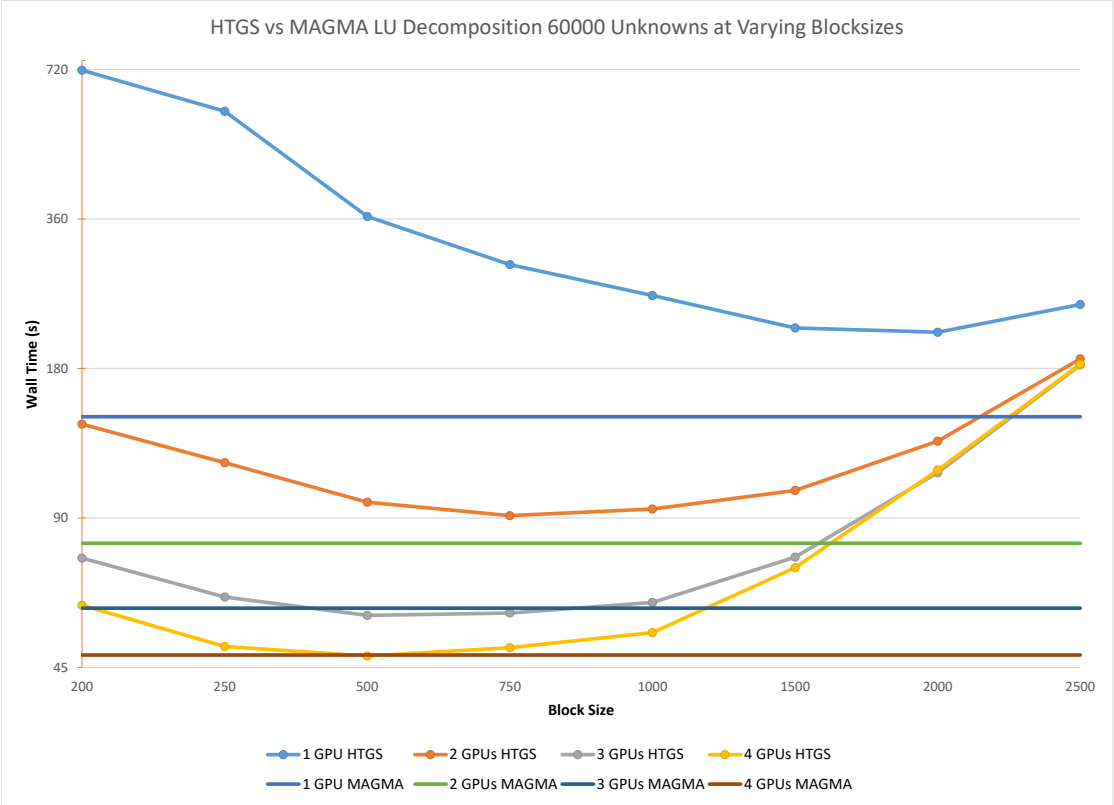

Figure 8.10: LU Decomposition HTGS vs MAGMA on GPUs for 60000 unknowns at varying block sizes.

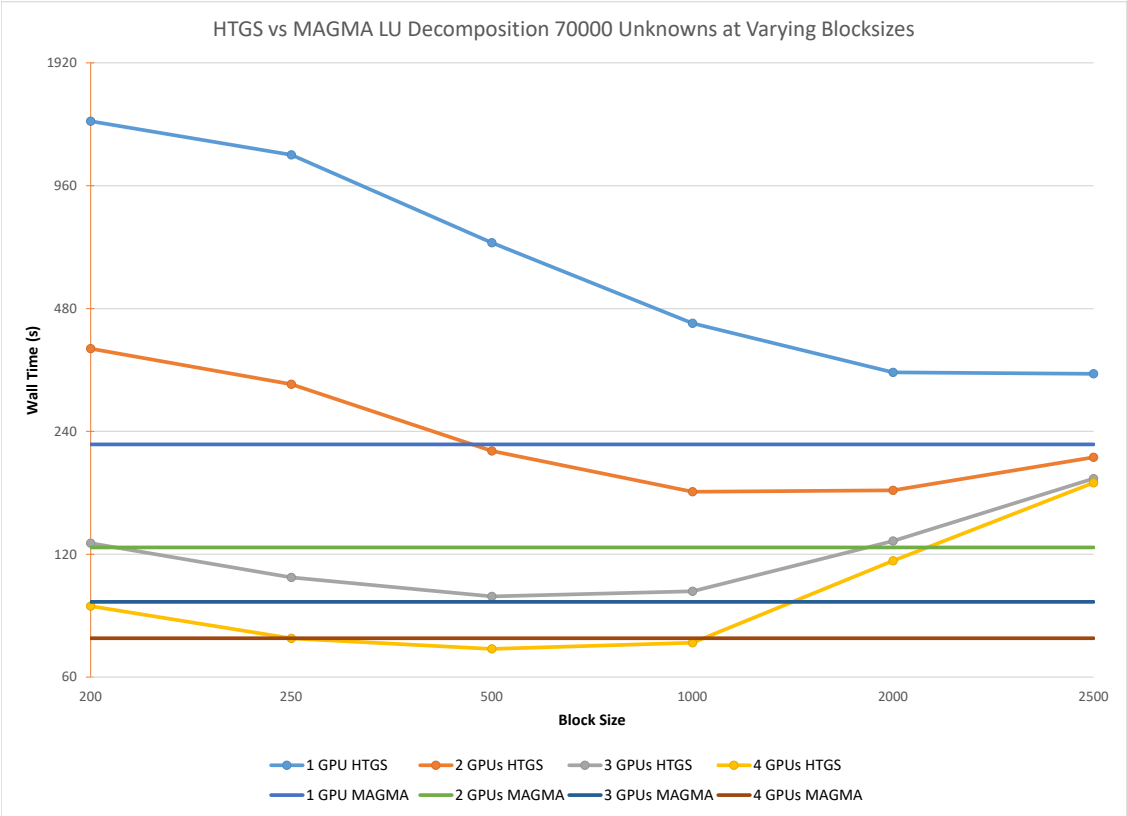

Figure 8.11: LU Decomposition HTGS vs MAGMA on GPUs for 70000 unknowns at varying block sizes.

8.8 Discussion

The HTGS implementations are done with a modest effort, and are able to get similar or better performance to that of OpenBLAS and MAGMA for LUD. One of the most significant components that this case study shows is the adaptability of HTGS for representing an algorithm and refining that implementation to improve data locality, utilization, and domain decomposition at a very high level of abstraction. With the HTGS model and API, making these modifications did not require significant code change and non of underlying computational functions were altered.

There are still optimizations that can be done to the HTGS GPU implementation. MAGMA has a clear advantage over HTGS for out-of-core problems. One of the main

differences between LUD in MAGMA and HTGS is the lack of CPU utilization in the HTGS implementation. The Gaussian elimination and factor tasks have very low CPU utilization, so the CPU is mostly idle during the HTGS GPU computation. MAGMA utilizes an alternate approach that was able to distribute work across both the CPU and GPU within the update routine, resulting in higher CPU utilization. One optimization would be to remove the $\mathrm{CopyIn_U}$ task and force the GPU to operate only within the sliding window. Any panels that are outside of the window would be sent to a CPU task to factor the upper diagonal and update the panel. Adding this component would allow for fewer memory copies, higher utilization of the CPU, and process the remainder of matrices in-core on the GPU as soon as the window slides far enough.

Chapter 9

CONCLUSIONS

In this thesis, we developed the Hybrid Task Graph Scheduler model and API that combines elements from dataflow semantics and task graph schedulers with the aim to improve programmer productivity when implementing and optimizing parallel algorithms to fully utilize fat nodes. With case studies we have demonstrated the proposed model and obtained full utilization for such nodes with a modest effort. Thus, the HTGS model was effective at exposing the parallel attributes of algorithms, representing the algorithms at a high level of abstraction. With this high abstraction, HTGS enabled improved productivity by identifying bottlenecks, scheduling behavior to assist with data locality, and scalability with multiple accelerators. HTGS obtained good or better performance compared to best of breed implementations for three algorithms; image stitching, matrix multiplication, and LU decomposition.

Image Stitching

Reduced the total lines of code by $42.6\,\%$, and obtained similar performance. Incorporating execution pipelines into the task graph required only 10 additional lines of code, enabling scalability across multiple GPUs.

Matrix Multiplication

HTGS improved upon both OpenBLAS and cuBLAS-XT for matrix multiplication. In addition, HTGS was able to effectively overlap data motion with computation. For problems loaded from disk, OpenBLAS had a decrease of $49.4\,\%$ and $57.6\,\%$ in performance compared with problems executing from RAM for $16k^2$ and $32k^2$ matrices, respectively. HTGS, on the other hand, had a percent decrease of $38.1\,\%$ and $23.0\,\%$ for the same problem sizes, respectively. The overlapping of computation

with data motion is inherent to the HTGS model, which results in improved utilization. On the GPU we observed an increase of performance between HTGS and cuBLAS-XT for $16k^2$ and $32k^2$ matrices across all block sizes. For 1024^2 block sizes the HTGS GPU version of matrix multiplication improves upon cuBLAS-XT by 1.64x, 1.70x, 1.58x, and 1.59x for 1, 2, 3, and 4 GPUs, respectively. The small block sizes processed the $32k^2$ matrices using less than 1 GB of memory. Despite the low memory footprint, HTGS effectively kept the GPUs busier than cuBLAS-XT.

LU Decomposition

LU decomposition is a complex algorithm to schedule and optimize. In this implementation, we started with a block-based approach that achieved good performance on the CPU. We transformed that implementation with minimal effort into a block+panel design that achieved better performance than OpenBLAS. On the GPU, the block-based approach had low utilization. Using the HTGS API, we profiled the GPU graph and visually identified the bottleneck that was causing an impact on pipelining. Based on this analysis, we identified a more suitable approach for scheduling data by using the block+panel with a sliding window to improve the locality of data. With the sliding window approach, multi-GPU systems aggregated their memories enabling the sliding window to expand, reducing the number of data transfers required to process large matrices. The results showed similar or better performance compared to MAGMA for matrices within the sliding window. For matrices that were beyond the sliding window, MAGMA had significant gains. The HTGS model and API enabled rapid prototyping allowing for the development of these approaches at a high level of abstraction and provided the tools necessary to understand the impact that the implementations had on performance.

Thus, through these case studies, we validated our thesis statement regarding the use of the Hybrid Task Graph Scheduler model to improve programmer productivity when implementing and optimizing parallel algorithms to fully utilize single fat nodes consisting of many-core CPUs and multiple accelerators.

9.1 Future Work

In the HTGS GPU implementation of LU decomposition, we observed poor performance for out-of-core computation compared to MAGMA. This is attributed to the additional overhead of transferring memory to/from the GPU multiple times for the same panels of data. To reduce the number of memory transfers, it is possible to implement a new variant of the block+panel approach and incorporate the CPU into the update routines. This would use the window to operate on the GPU, any panels outside of the window would be processed on the CPU. Eventually the entire matrix will fit into the window allowing the GPU to only copy each panel once. This approach should improve utilization for the CPU, which is mostly idle in the current version.

Profiling task graphs visually is extremely useful to identify bottlenecks, as shown in Chapter 8. In addition, these visualizations can be used to pinpoint issues such as incorrectly formed task graphs or deadlocks. Improving the profiling and debugging into a real-time visualization that can be monitored during execution would be further useful to understand the behavior of tasks in any moment of time during execution. Due to the light-weight nature of the tasks, this component should be achievable allowing for further productivity of understanding the performance and scheduling behavior of task graphs.

In the current version of the HTGS model and C++ API implementation we focus entirely on scalability for desktop super computers through the use of execution pipelines. Execution pipelines provides an excellent and easy to use interface to scale to multiple accelerators. Using this approach, it would be possible to use execution pipelines to scale to clusters. Instead of binding a pipeline per GPU, the execution pipeline would bind tasks to one pipeline per node in a cluster of computers. The primary challenge with this is managing dependencies and state. Bookkeepers within the execution pipeline's task graph share their rules among their pipelines through synchronization. Scaling this to tens or hundreds of nodes in a cluster would be detrimental to performance. One technique to assist with this would be to incorporate a new type of bookkeeper that provides a spatial decomposition to only lock the rules along shared boundaries.

Deadlock occurs in HTGS when memory release rules do not line up with the scheduling behavior within the HTGS task graph. These issues are currently left up to the programmer to identify and fix. Traditional dataflow graphs annotates its edges with production and consumption rates that indicate when a node is ready to be activated by the

scheduler. Combining this aspect by annotating the connectors within HTGS task graphs could be used to detect or pinpoint possible locations for deadlocks prior to execution.

As hierarchical memories are incorporated onto these fat nodes, HTGS will be in an excellent position to address the performance optimizations required to effectively utilize these complex volatile and non-volatile memories. Adaptations to the HTGS memory managers could provide one high-level mechanism to incorporate the hierarchy into an addressable space for tasks, which could use a caching mechanisms to retrieve memory with rules such as least recently used caching As shown in memory release rules, these caching rules could be customized to apply ensure locality-sensitive data is more readily available for out-of-core computation on accelerators.

Appendices

Appendix A

HTGS DOCUMENTATION

The full HTGS documentation is available on-line at `https://pages.nist.gov/HTGS/doxygen/index.html`

REFERENCES

[1] Abadi, M.; Agarwal, A.; Barham, P.; Brevdo, E.; Chen, Z.; Citro, C.; Corrado, G. S.; Davis, A.; Dean, J.; Devin, M.; Ghemawat, S.; Goodfellow, I.; Harp, A.; Irving, G.; Isard, M.; Jia, Y.; Jozefowicz, R.; Kaiser, L.; Kudlur, M.; Levenberg, J.; Mané, D.; Monga, R.; Moore, S.; Murray, D.; Olah, C.; Schuster, M.; Shlens, J.; Steiner, B.; Sutskever, I.; Talwar, K.; Tucker, P.; Vanhoucke, V.; Vasudevan, V.; Viégas, F.; Vinyals, O.; Warden, P.; Wattenberg, M.; Wicke, M.; Yu, Y.; and Zheng, X. 2015. TensorFlow: Large-scale machine learning on heterogeneous systems. Software available from tensorflow.org.

[2] Ang, J. A.; Barrett, R. F.; Benner, R. E.; Burke, D.; Chan, C.; Cook, J.; Donofrio, D.; Hammond, S. D.; Hemmert, K. S.; Kelly, S. M.; Le, H.; Leung, V. J.; Resnick, D. R.; Rodrigues, A. F.; Shalf, J.; Stark, D.; Unat, D.; and Wright, N. J. 2014. Abstract machine models and proxy architectures for exascale computing. In *Proceedings of the 1st International Workshop on Hardware-Software Co-Design for High Performance Computing*, Co-HPC '14, 25–32. IEEE Press.

[3] Augonnet, C.; Thibault, S.; Namyst, R.; and Wacrenier, P.-A. 2011. StarPU: A Unified Platform for Task Scheduling on Heterogeneous Multicore Architectures. *Concurrency and Computation: Practice and Experience* 23(2):187–198.

[4] AutoStitch. 2012. AutoStitch. `www.cs.bath.ac.uk/brown/autostitch/autostitch.h` Last access: 2012-12-19.

[5] B. Ma et al. 2007. Use of AutoStitch for automatic stitching of microscope images. *Micron* 38(5):492–499.

[6] Barnea, D. I., and Silverman, H. F. 1972. A class of algorithms for fast digital image registration. *Computers, IEEE Transactions on* C-21(2):179–186.

[7] Blattner, T.; Keyrouz, W.; Chalfoun, J.; Stivalet, B.; Brady, M.; and Zhou, S. 2014. A hybrid CPU-GPU system for stitching large scale optical microscopy images. In *43rd International Conference on Parallel Processing (ICPP)*, 1–9.

[8] Blattner, T.; Keyrouz, W.; Halem, M.; Brady, M.; and Bhattacharyya, S. S. 2015. A hybrid task graph scheduler for high performance image processing workflows. In *2015 IEEE Global Conference on Signal and Information Processing (GlobalSIP)*, 634–637.

[9] Blattner, T. 2013. A Hybrid CPU/GPU Pipeline Workflow System. Master's thesis, University of Maryland Baltimore County.

[10] 2012. BOOST C++ library. `boost.org`. Last access: 2012-07-12.

[11] Brown, M., and Lowe, D. G. 2007. Automatic panoramic image stitching using invariant features. *Int. J. Comput. Vision* 74(1):59–73.

[12] Budimlic, Z.; Chandramowlishwaran, A.; Knobe, K.; Lowney, G.; Sarkar, V.; and Treggiari, L. 2009. Multi-core implementations of the concurrent collections programming model. *CPC09: 14th International Workshop on Compilers for Parallel Computers*.

[13] Budimlic, Z.; Burke, M.; Cave, V.; Knobe, K.; Lowney, G.; Newton, R.; Palsberg, J.; Peixotto, D.; Sarkar, V.; Schlimbach, F.; and Tacsirlar, S. 2010. Concurrent collections. *Sci. Program.* 18(3-4):203–217.

[14] Buttari, A.; Langou, J.; Kurzak, J.; and Dongarra, J. 2009. A class of parallel tiled linear algebra algorithms for multicore architectures. *Parallel Computing* 35(1):38 – 53.

[15] Cavé, V.; Zhao, J.; Shirako, J.; and Sarkar, V. 2011. Habanero-java: The new adventures of old x10. In *Proceedings of the 9th International Conference on Principles and Practice of Programming in Java*, PPPJ '11, 51–61. ACM.

[16] Chamberlain, B.; Callahan, D.; and Zima, H. 2007. Parallel programmability and the chapel language. *Int. J. High Perform. Comput. Appl.* 21(3):291–312.

[17] Cooper, L.; Huang, K.; and Ujaldon, M. 2011. Parallel automatic registration of large scale microscopic images on multiprocessor cpus and gpus. In *2011 IEEE International Symposium on Parallel and Distributed Processing Workshops and Phd Forum (IPDPSW)*, 1367–1376. IEE.

[18] Dean, J., and Ghemawat, S. 2008. Mapreduce: Simplified data processing on large clusters. *Commun. ACM* 51(1):107–113.

[19] Deelman, E.; Singh, G.; hui Su, M.; Blythe, J.; Gil, Y.; Kesselman, C.; Mehta, G.; Vahi, K.; Berriman, G. B.; Good, J.; Laity, A.; Jacob, J. C.; and Katz, D. S. 2005. Pegasus: a framework for mapping complex scientific workflows onto distributed systems. *SCIENTIFIC PROGRAMMING JOURNAL* 13:219–237.

[20] Dennis, J. B. 1974. First version of a data flow procedure language. In Robinet, B., ed., *Programming Symposium*, volume 19 of *Lecture Notes in Computer Science*. Springer Berlin Heidelberg. 362–376.

[21] Dongarra, J.; Gates, M.; Haidar, A.; Kurzak, J.; Luszczek, P.; Tomov, S.; and Yamazaki, I. 2014. Accelerating numerical dense linear algebra calculations with gpus. *Numerical Computations with GPUs* 1–26.

[22] 2012. Fiji Is Just ImageJ. `fiji.sc`. Last access: 2012-07-18.

[23] Foley, D. 2014. Nvlink, pascal and stacked memory: Feeding the appetite for big data. *Nvidia. com*.

[24] Frigo, M., and Johnson, S. G. 2005a. The design and implementation of FFTW3. *Proceedings of the IEEE* 93(2):216–231. Special issue on "Program Generation, Optimization, and Platform Adaptation".

[25] Frigo, M., and Johnson, S. G. 2005b. The design and implementation of FFTW3. *Proceedings of the IEEE* 93(2):216–231. Special issue on "Program Generation, Optimization, and Platform Adaptation".

[26] Gansner, E. R., and North, S. C. 2000. An open graph visualization system and its applications to software engineering. *SOFTWARE - PRACTICE AND EXPERIENCE* 30(11):1203–1233.

[27] Gautier, T.; Besseron, X.; and Pigeon, L. 2007. Kaapi: A thread scheduling runtime system for data flow computations on cluster of multi-processors. In *Proceedings of the 2007 International Workshop on Parallel Symbolic Computation*, PASCO '07, 15–23. New York, NY, USA: ACM.

[28] Gautier, T.; Lima, J. V. F.; Maillard, N.; and Raffin, B. 2013. Xkaapi: A runtime system for data-flow task programming on heterogeneous architectures. In *Proceedings*

of the 2013 IEEE 27th International Symposium on Parallel and Distributed Processing, IPDPS '13, 1299–1308. Washington, DC, USA: IEEE Computer Society.

[29] Golub, G. H., and Van Loan, C. F. 1996. *Matrix Computations (3rd Ed.)*. Baltimore, MD, USA: Johns Hopkins University Press.

[30] Grossman, M.; Sbîrlea, A. S.; Budimlić, Z.; and Sarkar, V. 2011. Cnc-cuda: Declarative programming for gpus. In *Proceedings of the 23rd International Conference on Languages and Compilers for Parallel Computing*, LCPC'10, 230–245. Springer-Verlag.

[31] Harris, M. 2012. Optimizing parallel reduction in CUDA. `docs.nvidia.com/cuda/samples/6_Advanced/reduction/doc/reduction.pdf`. Last access: 2012-12-19.

[32] 2012. ImageJ. `rsbweb.nih.gov/ij` & `fiji.sc`. Last access: 2011-12-16.

[33] Intel. 2015. Intel® Xeon Phi™ Product Family. (Date last accessed: 2015-06-26).

[34] Isard, M.; Budiu, M.; Yu, Y.; Birrell, A.; and Fetterly, D. 2007. Dryad: Distributed data-parallel programs from sequential building blocks. In *Proceedings of the 2Nd ACM SIGOPS/EuroSys European Conference on Computer Systems 2007*, EuroSys '07, 59–72. New York, NY, USA: ACM.

[35] JCuda. 2015. JCuda - Java bindings for the CUDA runtime and driver API. (Date last accessed: 2015-06-26) http://www.jcuda.org/jcuda/JCuda.html.

[36] Jing, Z.; Chang-shun, W.; and Wu-ling, L. 2009. An image mosaics algorithm based on improved phase correlation. In *Proceedings of 2009 International Conference on Environmental Science and Information Application Technology*, 383–386. IEEE.

[37] K. U. Venkataraju et al. 2009. Assembling large mosaics of electron microscope images using gpus. `www.cs.utah.edu/sci/publications/kannanuv09/Venkataraju_SAAHPC09.pdf`

[38] Kathleen Knobe, C. D. O. 2005. Tstreams: A model of parallel computation (preliminary report).

[39] Kuglin, C. D., and Hines, D. C. 1975a. The phase correlation image alignment method. In *Proceedings of the 1975 IEEE International Conference on Cybernetics and Society*, 163–165.

[40] Kuglin, C. D., and Hines, D. C. 1975b. the phase correlation image alignment method. In *proceedings of the 1975 ieee international conference on cybernetics and society*, 163–165.

[41] Kukanov, A., and Voss, M. J. 2007. The foundations for scalable multi-core software in intel threading building blocks. *Intel Technology Journal* 11(4):309–322.

[42] Kurzak, J.; Ltaief, H.; Dongarra, J.; and Badia, R. M. 2010. Scheduling dense linear algebra operations on multicore processors. *Concurrency and Computation: Practice and Experience* 22(1):15–44.

[43] Lee, E. A., and Parks, T. 1995. Dataflow process networks. In *Proceedings of the IEEE*, 773–799.

[44] Lewis, J. P. 1995. Fast normalized cross-correlation. `www.idiom.com/~zilla/Work/nvisionInterface/nip.pdf`.

[45] 2012. libTIFF. `libtiff.org`. Last access: 2012-07-11.

[46] Lorenz, D.; Philippen, P.; Schmidl, D.; and Wolf, F. 2012. Profiling of openmp tasks with score-p. In *2012 41st International Conference on Parallel Processing Workshops*, 444–453.

[47] Luk, C.-K.; Hong, S.; and Kim, H. 2009. Qilin: Exploiting parallelism on heterogeneous multiprocessors with adaptive mapping. In *Proceedings of the 42Nd Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO 42, 45–55. New York, NY, USA: ACM.

[48] NVIDIA Corp. 2012a. CUFFT Library. `developer.nvidia.com/cufft`. Last access: 2012-04-10.

[49] NVIDIA Corp. 2012b. GPU Computing SDK. `developer.nvidia.com/cuda-downloads`. Last access: 2012-12-19.

[50] NVIDIA Corp. 2012c. NVIDIA visual profiler. `developer.nvidia.com/nvidia-visual-profiler`. Last access: 2012-12-19.

[51] NVIDIA CUDA. 2011. Cuda c programming guide version 4.1.

[52] NVIDIA-NVLink. 2016. NVIDIA NVLink High-Speed Interconnect. (Date last accessed: 2016-09-14).

[53] NVIDIA. 2015. Tesla Accelerated Computing. (Date last accessed: 2015-06-26).

[54] NVIDIA. 2016a. NVIDIA visual profiler. (Date last accessed: 2016-07-14).

[55] NVIDIA. 2016b. The NVIDIA DGX-1. (Date last accessed: 2016-07-13).

[56] OpenMP Architecture Review Board. 2013. OpenMP application program interface version 4.0.

[57] Pino, J.; Bhattacharyya, S.; and Lee, E. 1995. A hierarchical multiprocessor scheduling system for dsp applications. In *Signals, Systems and Computers, 1995. 1995 Conference Record of the Twenty-Ninth Asilomar Conference on*, volume 1, 122–126 vol.1.

[58] Plimpton, S. J., and Devine, K. D. 2011. Mapreduce in mpi for large-scale graph algorithms. *Parallel Comput.* 37(9):610–632.

[59] Preibisch, S.; Saalfeld, S.; and Tomancak, P. 2009. Globally optimal stitching of tiled 3d microscopic image acquisitions. *Bioinformatics* 25(11):1463–1465.

[60] Qian, C.; Ding, Z.; and Sun, H. 2013. A performance visualization method for openmp tasks. In *High Performance Computing and Communications 2013 IEEE International Conference on Embedded and Ubiquitous Computing, 2013 IEEE 10th International Conference*, 735–741.

[61] Reinders, J. 2007. *Intel Threading Building Blocks*. Sebastopol, CA, USA: O'Reilly & Associates, Inc., first edition.

[62] Sane, N. 2011. *Rapid Prototyping of High Performance Signal Processing Applications*. Ph.D. Dissertation, University of Maryland, College Park.

[63] Schlimbach, F. 2014. Intel Concurrent Collections for C++ for Windows and Linux. date last accessed: 2015-09-28 https://software.intel.com/en-us/articles/intel-concurrent-collections-for-cc.

[64] Shvachko, K.; Kuang, H.; Radia, S.; and Chansler, R. 2010. The hadoop distributed file system. In *Mass Storage Systems and Technologies (MSST), 2010 IEEE 26th Symposium on*, 1–10.

[65] Sodani, A.; Gramunt, R.; Corbal, J.; Kim, H. S.; Vinod, K.; Chinthamani, S.; Hutsell, S.; Agarwal, R.; and Liu, Y. C. 2016. Knights landing: Second-generation intel xeon phi product. *IEEE Micro* 36(2):34–46.

[66] Szeliski, R. 2006. Image alignment and stitching: a tutorial. *Found. Trends. Comput. Graph. Vis.* 2(1):1–104.

[67] Teodoro, G.; Hartley, Timothy, D.; Catalyurek, U.; and Ferreira, R. 2012. Optimizing dataflow applications on heterogeneous environments. *Cluster Computing* 15(2).

[68] Tomov, S.; Nath, R.; Ltaief, H.; and Dongarra, J. 2010. Dense linear algebra solvers for multicore with GPU accelerators. In *Proc. of the IEEE IPDPS'10*, 1–8. Atlanta, GA: IEEE Computer Society. DOI: 10.1109/IPDPSW.2010.5470941.

[69] Tomov, S.; Dongarra, J.; and Baboulin, M. 2010. Towards dense linear algebra for hybrid GPU accelerated manycore systems. *Parallel Computing* 36(5-6):232–240.

[70] TOP500. 2016. TOP500 supercomputer sites. (Date last accessed: 2016-06-20).

[71] YarKhan, A.; Kurzak, J.; Luszczek, P.; and Dongarra, J. 2016. Porting the plasma numerical library to the openmp standard. *International Journal of Parallel Programming* 1–22.

[72] YarKhan, A.; Dongarra, J.; and Kurzak, J. 2007. QUARK User's Guide: QUeueing And Runtime for Kernels. Technical Report 1, Innovative computing lab, University of Tennessee, Knoxville.

[73] Zaharia, M.; Chowdhury, M.; Franklin, M. J.; Shenker, S.; and Stoica, I. 2010. Spark: Cluster computing with working sets. In *Proceedings of the 2nd USENIX Conference on Hot Topics in Cloud Computing*, HotCloud'10, 10. USENIX Association.