J Sign Process Syst (2017) 89:457–467
DOI 10.1007/s11265-017-1262-6

A Hybrid Task Graph Scheduler for High Performance Image Processing Workflows

Timothy Blattner[1,2] · **Walid Keyrouz**[1] · **Shuvra S. Bhattacharyya**[3,4] · **Milton Halem**[2] · **Mary Brady**[1]

Received: 14 July 2016 / Revised: 11 April 2017 / Accepted: 22 June 2017 / Published online: 19 July 2017

Abstract Designing applications for scalability is key to improving their performance in hybrid and cluster computing. Scheduling code to utilize parallelism is difficult, particularly when dealing with data dependencies, memory management, data motion, and processor occupancy. The Hybrid Task Graph Scheduler (HTGS) improves programmer productivity when implementing hybrid workflows for multi-core and multi-GPU systems. The Hybrid Task Graph Scheduler (HTGS) is an abstract execution model, framework, and API that increases programmer productivity when implementing hybrid workflows for such systems. HTGS manages dependencies between tasks, represents CPU and GPU memories independently, overlaps computations with disk I/O and memory transfers, keeps multiple GPUs occupied, and uses all available compute resources. Through these abstractions, data motion and memory are explicit; this makes data locality decisions more accessible. To demonstrate the HTGS application program interface (API), we present implementations of two example algorithms: (1) a matrix multiplication that shows how easily task graphs can be used; and (2) a hybrid implementation of microscopy image stitching that reduces code size by $\approx$ 43% compared to a manually coded hybrid workflow implementation and showcases the minimal overhead of task graphs in HTGS. Both of the HTGS-based implementations show good performance. In image stitching the HTGS implementation achieves similar performance to the hybrid workflow implementation. Matrix multiplication with HTGS achieves 1.3x and 1.8x speedup over the multi-threaded OpenBLAS library for 16k $\times$ 16k and 32k $\times$ 32k size matrices, respectively.

Keywords Heterogeneous architectures · Task graph · Hybrid workflows · Dataflow · Image processing · Matrix multiplication

1 Introduction

Hybrid clusters currently play a prominent role in high performance computing; they make up four of the top ten fastest supercomputers as of June 2016 [21]. These petascale clusters consist of nodes that contain one or more CPUs with one or more co-processors or accelerators (Intel Xeon Phi [11] or NVIDIA Tesla [17]); these accelerators scale up the compute capabilities of individual nodes while staying within energy budgets. Clusters are approaching the exascale level; 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 [9]. Programming these exascale machines for performance will be challenging. It will require programmers to maximize parallelism and minimize data motion while maximizing the arithmetic density of computations applied to the data [1];

✉ Timothy Blattner
timothy.blattner@nist.gov

[1] Information Technology Laboratory, National Institute of Standards and Technology, Gaithersburg, MD 20899, USA

[2] Center for Hybrid Multicore Productivity Research, University of Maryland Baltimore County, Baltimore, MD 21250, USA

[3] Department of Electrical and Computer Engineering, University of Maryland, College Park, MD 20742, USA

[4] Department of Pervasive Computing, Tampere University of Technology, Tampere, Finland

this task is further complicated by accelerators which effectively have their own memory domains that are distinct from those of CPUs.

This paper builds on our previous work [6] and formalizes our approach to developing a scalable implementation of image stitching for large optical microscopy images. It presents the Hybrid Task Graph Scheduler (HTGS), which is designed to aid in building hybrid workflows for high performance image processing. HTGS presents a model and framework, which makes it easier to overlap data motion with computation, maximize processor occupancy when running on hybrid computers, and manage memory usage to stay within physical memory limitations. Ignoring these aspects of a system has a detrimental impact on performance.

An earlier version of this work [7] demonstrated a prototype of the hybrid task graph scheduler for image processing workflows in Java. This paper expands on that work by presenting the full release version of the HTGS API in C++, which we have recently made available in open source form [5], and its performance for image stitching compared to a similar hybrid workflow implementation in C++. Additionally, we present an HTGS-based implementation of matrix multiplication for in-core matrices, which achieves up to 1.8x speedup over the standalone OpenBLAS library for 32k $\times$ 32k size matrices.

In the following sections, we first introduce image stitching for microscopy. Next, we present an overview of our previous work with hybrid pipeline workflows and the performance gains attained and some related works to task scheduling and workflow systems. Then, we present the hybrid task graph scheduler model and framework, followed by two examples to illustrate the functionality of HTGS in microscopy image stitching and matrix multiplication. We will then present the conclusions.

1.1 Image Stitching for Microscopy

Image stitching addresses the scale mismatch between the dimensions of (1) an automated microscope's field of view and (2) the plate under study. To image a plate, the microscope acquires a grid of overlapping images as its motorized stage moves the plate. Image stitching reconstructs the full image by first computing the relative positions of adjacent image tiles and then using these distances to compose the image mosaic.

Image stitching applications for microscopy must address two challenges, algorithmic and scalability. Optical microscopy can generate images with few distinguishable features in the overlap region that can be used to guide stitching; this occurs often in early phases of live cell experiments. This lack of distinguishable features makes it difficult to determine the relative positioning of two adjacent tiles and rules out a large class of stitching algorithms with good performance characteristics. The scalability challenge is due to the large number of tiles in a grid (e.g., 100s–1000s); the resulting data sets are large and require a lot of processing. This challenge is further amplified by long-running experiments which image a plate periodically (e.g., every hour) to study phenomena such as cell growth over several days (e.g., 1–2 weeks). Image stitching must reconstruct a plate's image in a fraction of the imaging period to enable computationally steerable experiments, which allow researchers to analyze the acquired images and, if need be, intervene in the experiments.

There is a large body of literature on general image stitching. Szeliski provides an excellent entry point into this literature [20]; Szeliski discusses many algorithms for finding the proper alignment of image tiles and classifies them into two categories: feature-based alignment techniques [3, 16] and direct methods [12, 13].

Our work [6] uses a direct method, Kuglin and Hines' phase correlation image alignment method [13], that is modified to use normalized correlation coefficients as described by Lewis [14, 15]. This method uses Fast Fourier Transforms (FFTs) to compute Fourier Correlation Coefficients and then uses these correlation coefficients to determine image displacements. It is very similar to the one used by Preibisch, Saalfeld, and Tomanak in their ImageJ/Fiji plugin for image stitching [19]. We chose the Fourier-based approach because (1) it has predictable parallelism and (2) it is more robust for optical microscopy as it does not depend on feature detection which is inherently scene-dependent. The image stitching computation can be summarized in the following formulas:

$$\begin{aligned}\mathrm{NCC}_{ij} &= \mathcal{F}(I_i) \otimes \mathcal{F}(I_j)\\ \max_{ij} &= \mathrm{maxloc}(\mathcal{F}^{-1}(NCC_{ij}))\\ d_{ij} &= \max(\mathrm{CCF}^{1..4}(I_i, I_j, \max_{ij}))\end{aligned}$$

where $\mathcal{F}$ and $\mathcal{F}^{-1}$ are the forward and inverse Fourier transform operators, $\otimes$ is the normalized correlation coefficient operator, maxloc returns the index of the largest value in a vector, and $\mathrm{CCF}^{1..4}$ produces the four cross correlation coefficients and relative displacements of an image pair.

The image stitching algorithm is compute-bound and is dominated by Fourier transform computations, both forward and backward. For an $n \times m$ grid, there are $(3nm - n - m)$ such transforms. There is also a similar number of vector multiplications and reductions; these operations can become comparatively expensive if they do not take advantage of hardware vector instructions. For the large image grids under consideration (100s–1000s), image stitching exhibits a high degree of coarse-grained parallelism: all the forward transforms are independent of each other and can be computed in parallel; the NCC and maxloc computations exhibit a similar parallelism. However, the image stitching algorithm is not embarrassingly parallel because of data dependencies and size limitations imposed by physical memory. Therefore, a scalable parallel implementation must ensure that the dependencies of NCC_{ij} and $\mathrm{CCF}_{ij}^{1..4}$ are satisfied before these computations can proceed. Furthermore, the implementation must manage memory to stay within physical memory limits because the problem may not fit into a machine's main memory (e.g., 16, 32, or even 64 GB). Otherwise, the virtual memory system will result in excessive swapping with the ensuing dramatic fall in performance. Memory size limits are even more constraining in the case of accelerators where even high-end accelerators are limited to no more than 16 GB. The challenges in developing scalable parallel implementations lie in exploiting the coarse-grained parallelism available in image stitching and taking advantage of computing resources (CPUs and GPUs) while satisfying the constraints noted above.

In our previous work, we reported 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 workflows, which are designed to keep GPUs and CPUs busy while overlapping data movement with computations.

1.2 Hybrid Pipeline Workflows

In our previous work, we started with a CPU-only sequential implementation. This was then ported to the GPU (*Simple-GPU*) by replacing the compute-bound functions with equivalent CUDA kernels (hand-coded or from NVIDIA's CUDA libraries); *Simple-GPU* copied images to the GPU before operating on them. This resulted in a 14% speedup compared to the sequential CPU-only implementation. Data motion between co-processors and CPUs dominated the performance. By contrast, by scheduling the CPU/GPU invocations as a hybrid workflow (*Pipeline-GPU*) that properly manages memory and provides CPU threading to overlap computations with data motion, we improved the *Simple-GPU* implementation by 24x and scaled with multiple GPUs. The *Pipeline-GPU* implementation delivered 500x speedup w.r.t. ImageJ/Fiji, which is already multi-threaded to take advantage of multi-core computing. Additionally, we implemented a CPU-only version of microscopy image stitching using the hybrid pipeline workflow, which delivered 170x speedup w.r.t. ImageJ/Fiji.

Developing hybrid workflows can be complex and time consuming. We simplify the implementation of hybrid workflows by using the proposed HTGS, which includes a runtime system to schedule the task graphs on the hybrid compute resources (i.e., CPUs and GPUs). HTGS helps developers build task graphs that handle dependencies, manage memory in multiple native address spaces (CPU or GPU), overlap data motion with computations, and scale to multi-GPU systems through execution pipelines. Every task created through HTGS exposes the computational resources *to the programmer* and binds tasks to the specific hardware (CPU/GPU).

Workflows have been studied by a number of research groups: Concurrent Collections [10], Intel Threading Building Blocks [18], and Spark [24]. One example is the hybrid task scheduler StarPU [2], which uses work stealing to overlap CPU and accelerator computation. This is achieved by representing both memories as a unified address space and requires the programmer to implement a kernel for each architecture. The method is convenient, but requires careful consideration of data access patterns, which can result in inefficient data transfers. HTGS differs from this approach by explicitly representing each of the address spaces for the underlying architectures separately (i.e., CPU and GPU).

Kokkos [8] is a programming model that targets high performance computing platforms. It presents an alternative approach to utilizing GPU accelerated systems, which executes on GPU and CPU cores concurrently through high-level policies that express the parallel components of an algorithm as a series of execution patterns. Memory is also explicitly represented with scratch pads shareable among thread groups, and larger memory spaces that are attached to volatile or non-volatile memory that are addressed based on memory layouts and traits. Using the threading and memory model, locality-driven computation can be expressed at a high level of abstraction and defined for a specific abstract machine model. One major difference between HTGS and Kokkos is its method for expressing an algorithm. HTGS is explicit in representing an algorithm as a task graph. The task graph design maps directly into implementation and back, whereas in Kokkos some of these high-level decisions are lost in implementation.

2 Hybrid Task Graph Scheduler

The Hybrid Task Graph Scheduler (HTGS) provides an abstract execution model, a framework based on the model, and an API to assist in transforming an algorithm into a task graph and execute the core computational kernels concurrently using coarse-grained parallelism on compute nodes with multiple CPUs and GPUs. In the HTGS framework, a task graph consists of vertices and edges: vertices define state, computational, or scalable tasks; edges connect tasks to define data dependencies or state maintenance. HTGS represents an algorithm's components explicitly as a task graph and provides a separation of concerns by modularizing the components. This design improves productivity when optimizing and expanding an algorithm to incorporate improvements into an algorithm's implementation such as on accelerators or high-level scheduling decisions to improve data locality.

The HTGS model combines two paradigms, dataflow and task scheduling. Dataflow semantics represent an algorithm at a high-level of abstraction as a dataflow graph, similar to signal processing dataflow graphs (e.g., see [4]). Task scheduling provides the threading model. Traditionally, task scheduling assigns a thread pool to a task queue to process tasks concurrently; this queue is ordered based on inter-task data dependencies. HTGS alters this threading model by defining a thread pool for each task; the threads then operate on data sent to the task.

There are five steps in the HTGS design methodology, as shown in Fig. 1. The first three steps are pictorial *whiteboard* stages; the remaining steps are coding and revisiting of the pictorial stages. First, an algorithm is represented as a parallel algorithm where computational kernels are ideally laid out as a series of modular functions. Second, the parallel algorithm is represented using a dataflow interpretation, which maps computational functions to nodes and edges are data dependencies. This step exposes the concurrency and formulates a pipelined workflow system for the algorithm.

Third, using the HTGS framework, a mapping of the dataflow graph onto the targeted platform is designed. The HTGS framework provides abstractions for this *mapped dataflow graph* that make it easier for developers to experiment with and iteratively optimize this mapping when compared to conventional methods for designing CPU/GPU implementations. Such an iterative approach to design optimizations is important given the complexity of modern high performance platforms for image processing. The HTGS framework provides novel models, methods, and supporting APIs as the foundation for such an iterative system design process. Fourth, the HTGS task graph is implemented using an application programmer interface (API) that builds the HTGS framework. Finally, the HTGS task graph is refined and optimized; for example, modifying thread configurations of tasks, which improves data production and consumption rates, or altering the scheduling behavior and memory management to improve data locality.

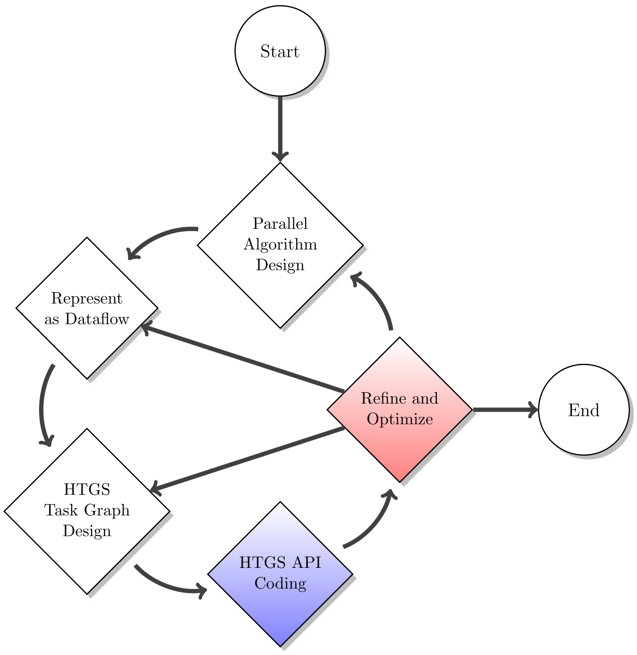


Figure 1 HTGS design methodology.

The HTGS framework defines components for building HTGS task graphs: tasks, bookkeepers, memory managers, and execution pipelines. HTGS specifies a task to have four phases with corresponding *phase* methods: (1) *Initialize* is called when a CPU thread has attached to the task; it allocates local task-level memory and/or binds the task and thread to an accelerator; (2) *Execute* consumes one input data object and produces zero or more output data objects; (3) *Terminate?* identifies if a task is ready to terminate; (4) *Shutdown* is called when the task is terminating; it is used to free local task-level memory and/or unbind from accelerators. These phase methods are used to customize the behavior of a task. Furthermore, HTGS specifies that every task has one input type, one output type, and a pool of threads. If the size of the thread pool is greater than one, then the task is copied, where each copy is bound to a separate thread. The threads share the same thread-safe input and output queues. The HTGS framework uses this task specification to build specialized tasks to assist in managing dependencies (bookkeeper), memory (memory manager), and scalability (execution pipelines).

Bookkeeper tasks manage task dependencies and advance the state of the computation. For example, image stitching uses a bookkeeper to detect that the FFTs of an adjacent pair of images are available before computing their relative displacements, as shown in Listing 1. A bookkeeper consumes data and forwards that data according to programmer-defined rules. A rule uses the data to update the state of the computation and determine when to produce data to the rule's output task. All rules in a bookkeeper have the same input type, but may have different output types. By convention, the rules should not be computationally intensive as they are accessed synchronously. Figure 2 shows the bookkeeper task.

An HTGS memory manager is a special-purpose task, dedicated to managing memory, that connects two tasks with a memory edge where one task allocates memory and the other *will eventually* release it. This edge is distinct from

```
void applyRule(std::shared_ptr<FFTData> data) {

  ...

  fftDataHolder->set(row, col, data);

  // Check northern neighbor
  if (fftDataHolder->has(row-1, col) &&
    northState->has(row, col) == false) {

    // Mark north is now done
    northState->set(row, col, true);

    // Get neighbor FFT data
    auto neighbor = fftDataHolder->get(row, col);

    auto output = new PCIAMData(data, neighbor)

    // Produce data along bookkeeper edge
    // to process phase correlation
    this->addResult(output);
  }

  // ... Check West, South, and East
}
```

Listing 1 Representative bookkeeper rule in image stitching.

traditional data edges in a task graph as it only passes memory data between the two tasks. The memory manager acts as an intermediary between the two connected tasks and updates the state of memory that is sent or received along its memory edge. This manager uses its memory rules to define and update the state of a memory buffer and to determine when the buffer can be released. These rules express the locality of the memory and ensure it is not released until it is no longer needed. For example, a memory manager can use a rule to decrement a reference count and to free a memory buffer when the count reaches zero. Furthermore, the memory manager allocates a buffer pool to be used for the memory edge at initialization. If the pool is empty and a task requests memory from the edge, then the requesting task will wait until memory becomes available based on the release rules of the memory, which is processed by a memory manager task.

Scalability in HTGS is further enhanced via *execution pipeline* tasks. Thread pools and accelerator tasks (e.g., CUDA NVIDIA GPU tasks) already build parallelism into HTGS task graphs. Execution pipeline tasks replicate and execute task graphs on systems with multiple GPUs, thereby utilizing all available GPUs concurrently. A developer takes advantage of this feature by encapsulating all or part of a graph into an execution pipeline task. During initialization, the execution pipeline task dynamically creates mirror-copies of the sub-graph and binds each copy to a separate accelerator. For example, in a graph with CUDA GPU tasks and CUDA memory managers, each accelerator gets its own CUDA task and memory manager: the accelerator-specific GPU tasks will schedule CUDA kernels on their own GPU; each memory manager task will manage the memory of its own GPU. When data enters an execution pipeline task, the data is distributed among the sub-graph copies using a bookkeeper with user-defined decomposition rules.

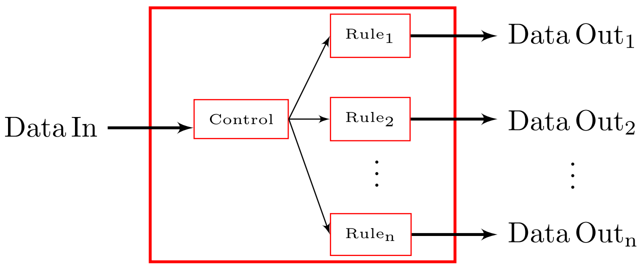


Figure 2 HTGS bookkeeper task with rule interface for scheduling management.

3 Microscopy Image Stitching with HTGS

Section 1.1 describes the image stitching algorithm in detail. This section uses the HTGS methodology to provide a detailed description of an HTGS-based implementation of image stitching. We first represent image stitching as a parallel algorithm. This algorithm can be split into two parts depending on whether it operates on single images or adjacent image pairs. The algorithm computes the Fourier transforms of images and uses these transforms as input to the phase correlation image alignment method (PCIAM) to compute the maximum intensity location of an adjacent image pair. The maximum intensity location is then used in conjunction with the image data of the image pairs to compute the cross correlation factors (CCF), which gives the relative displacement between the neighbors. Therefore, the primary data dependencies are (1) the image data and (2) the forward Fourier transforms. A parallel implementation can compute forward transforms concurrently as soon as their corresponding images are loaded. Similarly, an implementation can compute the maximum intensity of image pairs concurrently as soon as the needed forward transforms are available. This leads us to the dataflow representation for a single pair of image tiles shown in Fig. 3.

Based on the dataflow representation, we use the HTGS framework to build the HTGS task graph, shown in Fig. 4.

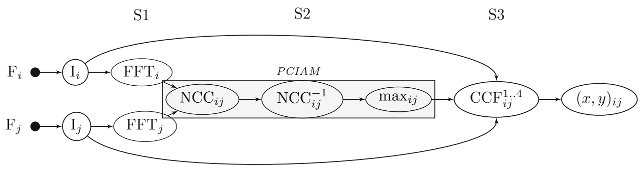


Figure 3 Dataflow graph for an adjacent pair computation.

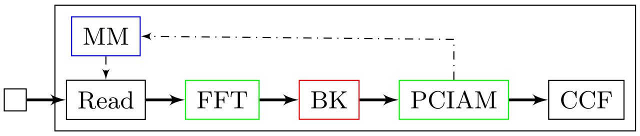


Figure 4 Hybrid image stitching task graph (machine with 1 GPU).

First, the *Read* task loads images from the file system and produces image data for the *FFT* task to compute fast Fourier transforms. This latter task's green color indicates that it will execute on the GPU and requires it to copy the image data from the CPU to the GPU. Next, the bookkeeper task, *BK*, enables the task graph to switch single images to pairs of images. *BK* collects forward transforms and dispatches an image pair to the *PCIAM* task when it detects that the transforms of an adjacent image pair are available, presented in Listing 1. *PCIAM* executes on the GPU, as indicated by its green color, and computes the maximum intensity location of an image pair using their forward transforms. It also copies the single scalar maximum intensity location from the GPU to the CPU. Last, the *CCF* task uses a thread pool on the CPU to compute the cross correlation factors of the image pair and outputs the relative displacement as the largest CCF value.

The task graph includes a memory manager (*MM*) between *Read* and *PCIAM* to manage a GPU's limited memory. *MM* allocates memory for the forward transforms on the GPU. It initializes a release count that is used in its memory rules; this count is the number of times an image's data is processed: 2, 3, and 4 for images along the corners, outside edges, and inner regions, respectively. The traversal of the image grid impacts the amount of memory required to process the grid. Using a chained-diagonal approach reduces the needed memory to $1 + \min(\text{gridWidth}, \text{gridHeight})$; this value is used to specify the size of the memory pool for the *MM* task.

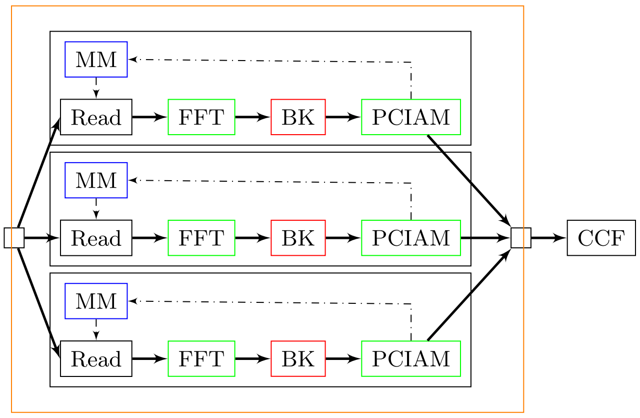


Figure 5 Hybrid image stitching with execution pipeline (3 GPUs).

This task graph computes the relative displacements for an image grid on a single GPU. To scale it to multiple GPUs, we partition it into two task graphs: (1) a GPU graph and (2) a container graph that holds an execution pipeline and the *CCF* task. The execution pipeline task encapsulates the GPU graph and duplicates the graph; one for each GPU as shown in Fig. 5. The number of copies is specified by the programmer, which is ideally the number of GPUs on the system, and each pipeline is bound to a separate GPU. The programmer adds a decomposition rule to the execution pipeline task that decomposes the image tile grid evenly such that each GPU graph copy processes a different non-overlapping region. The CCF task remains outside of the execution pipeline as it processes data on the CPU and stores the resulting relative displacements between a pair of images.

3.1 Microscopy Image Stitching Results

Table 1 compares the HTGS-based implementation of hybrid microscopy image stitching with the original implementation that did not use HTGS [6]. Each test case was repeated 50 times using a grid of 42 × 59 images (6.75 GB) and the average end-to-end run-time is reported. The evaluation 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 C++11 threads, CUDA 7.5 for GPU kernels, and cuFFT 7.5 for FFT computations.

Table 1 shows that using HTGS without execution pipelines reduces the code size by 43.4% compared to the original hybrid workflow [6]. 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 only *ten* additional lines of code. There is little performance improvement when using 3 GPUs instead of 2 due to hardware limitations within the PCI express (PCIe) bus. The lack of PCIe lanes on the Xeon E5-2650 v3 socket saturated the bus and could not keep all three GPUs busy. Additionally, the third GPU was on a

Table 1 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

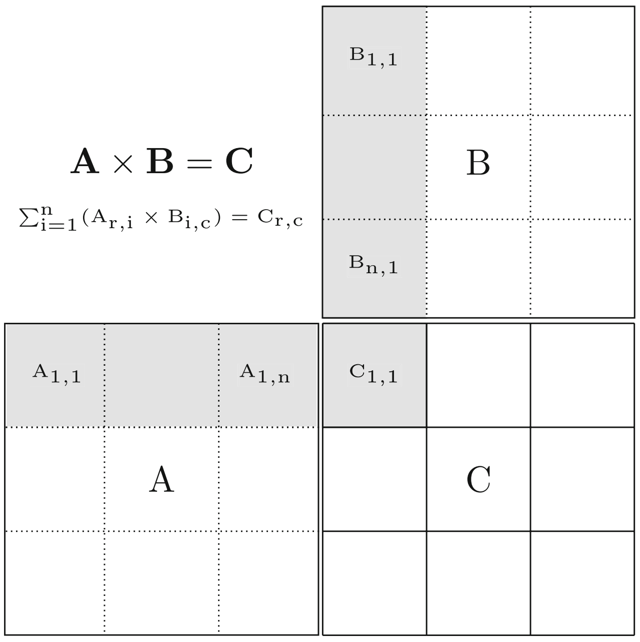

$\mathbf{A} \times \mathbf{B} = \mathbf{C}$
$\sum_{i=1}^{n}(A_{r,i} \times B_{i,c}) = C_{r,c}$
$B_{1,1}$
B
$B_{n,1}$
$A_{1,1}$
$A_{1,n}$
A
$C_{1,1}$
C

Figure 6 Matrix multiplication with block decomposition.

different PCIe bus from the first two GPUs; this prevented the third GPU from using GPU-direct peer-to-peer PCIe transfers. Such transfers copy data between GPUs on the same PCI express bus without first copying the data back to the CPU. This additional overhead impacted the performance when adding the third GPU.

The results compared to the original implementation and HTGS using 3 GPUs show a 42.6% reduction in code size, while maintaining the same relative performance, showing off the minimal overhead of HTGS and its ability to overlap data motion with computation.

4 Matrix Multiplication with HTGS

Matrix multiplication is one of the typical algorithms discussed in the parallel programming literature. This section presents the use of HTGS for parallelizing matrix multiplication. Our goals in presenting this example are twofold: (1) emphasize the utility of HTGS in optimizing end-to-end performance at an application level and (2) highlight the role of HTGS in experimenting with parallel signal processing software that weaves the invocations of compute kernels and I/O operations. Given the complexity of parallel software and interactions between memory management and computation in high-performance platforms, effective experimentation and iterative design optimization are critical to maximizing performance and retargeting designs efficiently across platforms.

```
// Instantiate tasks and graph
. . .
htgs::TaskGraph tgraph;

// Build graph
tGraph->setGraphConsumer(BK1);
tGraph->addRule(BK1, LoadA, sendARule);
tGraph->addRule(BK1, LoadB, sendBRule);

tGraph->addEdge(LoadA, BK2);
tGraph->addEdge(LoadB, BK2);
tGraph->addRule(BK2, MatMul, loadRule);

tGraph->addEdge(MatMul, BK3);
tGraph->addRule(BK3, Accumulate, accRule);
tGraph->addEdge(accTask, BK3);

tGraph->addRule(BK3, WriteC, outputRule);
tGraph->addGraphProducer(WriteC);
```

Listing 2 HTGS matrix multiplication graph construction.

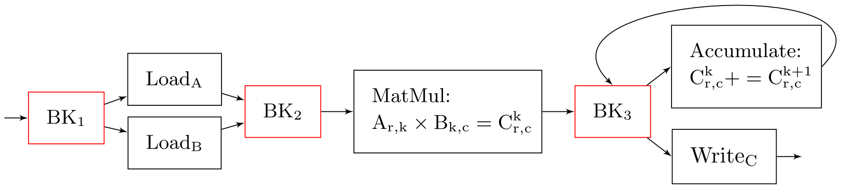
Given two matrices, $A(m, n)$ and $B(n, p)$, the elements of their product, matrix $C(m, p)$, are can be computed as the dot product of a row of A and a column of B: $c_{r,c} = A_{r,:} \cdot B_{:,c}$. To exploit parallelism and pipelining, we split matrix C into square blocks and compute its elements as the *dot-products* of horizontal and vertical bands of A and B as shown in Fig. 6.

Figure 7 shows a task graph that implements matrix multiplication. This graph consists of eight tasks and contains three dependencies, and takes as input square blocks.

The first bookkeeper, *BK*$_1$, distributes data to two load tasks, *Load*$_A$ and *Load*$_B$, which load blocks from $A_{r,:}$ and $B_{:,c}$ and sends these blocks to the second bookkeeper, *BK*$_2$. *BK*$_2$ advances the state of the computation by identifying which blocks of A and B have been loaded and when it can schedule the multiplication of two block matrices. Next, *MatMul* multiplies its two input block matrices $A_{r,k}$ and $B_{k,c}$ for one of the n contributions to $C_{r,c}$. *MatMul* sends the $C_{r,c}^{k}$ contribution to the third bookkeeper *BK*$_3$, which accumulates its input in $C_{r,c}$. When *BK*$_3$ detects that the *Accumulate* has accumulated all n contributions, it sends the final result $C_{r,c}$ to the *Write*$_C$ task.

Listing 2 shows pseudo-code for building the graph using the HTGS API. In this implementation, each task specializes

Figure 7 Matrix multiplication task graph.


BK_1
$Load_A$
$Load_B$
BK_2
MatMul:
$A_{r,k} \times B_{k,c} = C_{r,c}^{k}$
BK_3
Accumulate:
$C_{r,c}^{k} += C_{r,c}^{k+1}$
$Write_C$

```
htgs::Runtime runtime(tGraph)
runtime->executeRuntime();

// Produce data for graph
for (int row : matrixA)
  for (int col : matrixA)
    tGraph->produceData(
            MatrixARequest(row, col));

for (int col : matrixB)
  for (int row : matrixB)
    tGraph->produceData(
            MatrixBRequest(row, col));

// Indicate done adding data
tGraph->finishedProducingData();

// Process output from graph
while (!tGraph->isOutputTerminated()) {
  auto data = tGraph->consumeData();
  if (data != nullptr) { ... }
}
```

Listing 3 HTGS matrix multiplication graph execution and interaction.

the ITask base class, which implements the task's four phase methods. The tasks are added to a task graph as edges, rules, the graph's consumer, or its producer; the *consumer* and *producer* of the graph are the entry and exit tasks of the graph. Each rule edge has a bookkeeper and user-defined rules. These rules decide when data is produced along the edge based on the state of the computation.

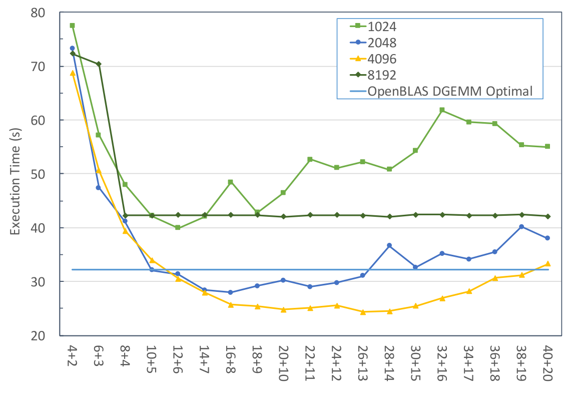
The listing directly corresponds to the task graph of Fig. 7 and provides conceptual continuity between analysis and implementation. Coding the graph in such a manner provides the programmer the following benefits: (1) identifying the performance bottlenecks and debugging of parallel code at a high level of abstraction, (2) instrumenting the code as per the conceptual model, and (3) sustaining the separation of concerns. Such an approach is consistent with optimizing performance as per the Roofline Model [22].

Listing 3 executes the task graph from Listing 2 using the HTGS runtime environment. The runtime spawns threads and binds them to tasks within the graph. Each thread immediately begins processing data as soon as data is available in its input queue. The data is processed in a First-In First-Out (FIFO) order. This mechanism customizes the traversal strategy for processing the matrices by using an inner-product, as shown in Fig. 6. This traversal is chosen to initiate the accumulation and write tasks as quickly as possible; the write bandwidth is often a significant bottleneck for traditional I/O subsystems, such as hard disk drives. The operations on the block matrices within the task graph remain the same and do not depend on whether the algorithm uses an inner or outer product. BK_1 consumes the graph's input data and distributes the data to the $Load_A$ and $Load_B$ tasks; this task is very fast as the send rules for BK_1 are light-weight by design. The $Write_C$ task produces data for the graph, which can then be used for additional post-processing in the main thread.

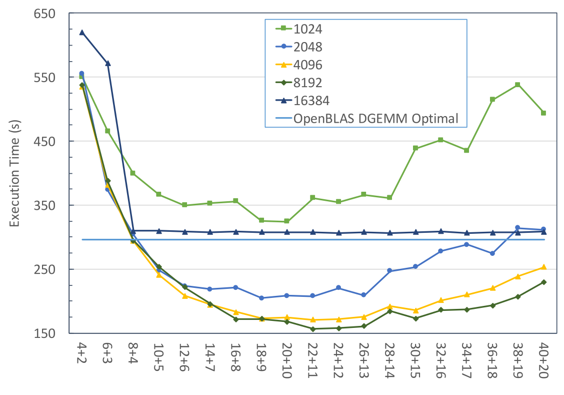
4.1 Matrix Multiplication Results

Our implementation of block-oriented matrix multiplication, shown in Fig. 7 and Listings 2 and 3, uses the double precision general matrix-matrix multiplication (DGEMM) operation from the OpenBLAS library [23]. This implementation configures OpenBLAS to use one thread only for this kernel. Each task in the graph is bound to one thread, except for the *MatMult* and *Accumulate* tasks, which uses n and $n/2$ threads, respectively. Therefore, the total number of threads spawned is $6 + n + n/2$.

The hardware configuration used for the evaluation is a system with two Intel Xeon E5-2650 v3 CPUs (40 logical cores) and 128 GB of DDR4 RAM. First, we ran OpenBLAS DGEMM as a one-off function call, configured with varying threads from 1 to 40. The runtimes leveled off once we reached the number of physical cores and the optimal performance was found to be the number of logical cores (40). Next, we ran the HTGS implementation at varying



(a) Runtimes for 16k × 16k matrices.



(b) Runtimes for 32k × 32k matrices.

Figure 8 DGEMM comparison—OpenBLAS and HTGS (block sizes and thread configurations).

block-sizes and compared the performance with the optimal one-off call of DGEMM.

Figure 8a and b show the results for two matrix sizes: 16k × 16k and 32k × 32k. Each reported runtime is the average runtime over 25 runs with varying HTGS task thread configurations and block sizes. The matrices are stored on disk and read using memory mapped files.

The results show that as the number of threads increases, the execution time goes down and then tapers off as HTGS uses more threads than logical cores. HTGS data points that are above the *Optimal OpenBLAS DGEMM* line indicate worse performance than OpenBLAS DGEMM. Interestingly, HTGS improves upon OpenBLAS's DGEMM by ≈ 1.3x and ≈ 1.8x for the 16k × 16k and 32k × 32k matrices, using a 26 + 13 thread configuration with 4k and 8k block sizes respectively. This improvement is a significant achievement for HTGS given the highly optimized nature of OpenBLAS and demonstrates the ability of HTGS to effectively overlap matrix multiplication computation with disk I/O with a modest effort. These results indicate that there is not a significant amount of overhead with scheduling data through the task graph and we are able to effectively pipeline the general matrix multiplication operation more efficiently than OpenBLAS's DGEMM for 16k × 16k and 32k × 32k matrix sizes.

5 Conclusions

The Hybrid Task Graph Scheduler (HTGS) improves programmer productivity for designing and implementing hybrid pipeline workflows to obtain performance on systems with multiple CPUs and GPUs. HTGS provides this productivity improvement by allowing the programmer to (1) represent an algorithm as a set of concurrent modular components that provide a separation of concerns and (2) optimize these components independently. This enables high-level modifications that can be used to alter the scheduling behavior and threading to improve data locality and parallelism, respectively. Using HTGS, these high-level abstractions are explicitly expressed programatically, which maps the task graph analysis phase directly to/from implementation.

We illustrate the use of the HTGS API using two algorithms. First, microscopy image stitching obtains similar performance to a manually-coded version, while reducing the code size by ≈ 43%. Second, matrix multiplication was developed with a modest effort and achieves 1.3x and 1.8x speedup over the multi-threaded OpenBLAS library for 16k × 16k and 32k × 32k size matrices, respectively. The performance of matrix multiplication and microscopy image stitching illustrates the applicability of HTGS task graphs to a broad class of algorithms and the performance gains of utilizing the HTGS model.

The source code for the HTGS API is available at https://pages.nist.gov/HTGS. Tutorials on how to use the API and the test cases presented are found at https://github.com/usnistgov/htgs-tutorials.

6 Future Work

Execution pipelines are a promising way to scale task graphs to multiple GPUs within a single compute node. In the future, we plan to extend this method to clusters and hybrid clusters. As such, using an *MPI* execution pipeline, task graphs may scale to clusters. Given that an execution pipeline in the HTGS methodology is just a task in the task graph, each execution pipeline may contain one or more additional execution pipelines within itself. This recursive nature enables execution pipelines to effectively map to hybrid clusters: for example, one execution pipeline per GPU, per node in a cluster, or a combination of both.

7 Disclaimer

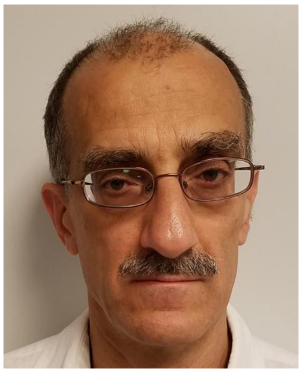
No approval or endorsement of any commercial product by NIST is intended or implied. Certain commercial software, products, and systems are identified in this report to facilitate better understanding. Such identification does not imply recommendations or endorsement by NIST, nor does it imply that the software and products identified are necessarily the best available for the purpose.

References

1. 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., & 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* (pp. 25–32). IEEE Press.
2. Augonnet, C., Thibault, S., Namyst, R., & 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.
3. Barnea, D.I., & Silverman, H.F. (1972). A class of algorithms for fast digital image registration. *IEEE Transactions on Computers*, *C-21*(2), 179–186. doi:10.1109/TC.1972.5008923.
4. Bhattacharyya, S.S., Deprettere, E., Leupers, R., & Takala, J. (Eds.) (2013). *Handbook of signal processing systems*, 2nd edn. Springer.
5. Blattner, T. (2016). HTGS application programming interface. https://pages.nist.gov/HTGS/. Last access: 2017-03-20.

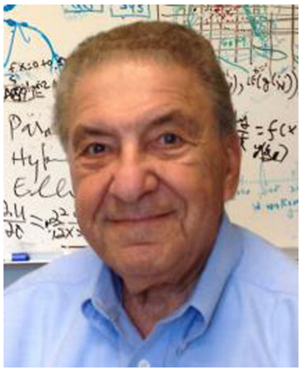
6. Blattner, T., Keyrouz, W., Chalfoun, J., Stivalet, B., Brady, M., & Zhou, S. (2014). A hybrid CPU-GPU system for stitching large scale optical microscopy images. In *43rd International conference on parallel processing (ICPP)* (pp. 1–9). doi:10.1109/ICPP.2014.9.
7. Blattner, T., Keyrouz, W., Halem, M., Brady, M., & 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)* (pp. 634–637).
8. Carter Edwards, H., Trott, C.R., & Sunderland, D. (2014). Kokkos. *Journal of Parallel and Distributed Computing*, *74*(12), 3202–3216. doi:10.1016/j.jpdc.2014.07.003.
9. Gardner, E. (2014). What public disclosures has Intel made about Knights Landing? https://software.intel.com/en-us/articles/what-disclosures-has-intel-made-about-knights-landing. Last access: 2015-05-21.
10. Grossman, M., Sbirlea, A.S., Budimlic, Z., & Sarkar, V. (2011). *CnC-CUDA: declarative programming for GPUs (pp. 230–245). No. 6548 in Lecture Notes in Computer Science*. Berlin: Springer.
11. Intel: Intel® Xeon Phi™ product family (2015). http://www.intel.com/content/www/us/en/processors/xeon/xeon-phi-detail.html. Last access: 2015-06-26.
12. Jing, Z., Chang-shun, W., & 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* (pp. 383–386). IEEE. doi:10.1109/ESIAT.2009.184.
13. Kuglin, C.D., & Hines, D.C. (1975). The phase correlation image alignment method. In *Proceedings of the 1975 IEEE international conference on cybernetics and society* (pp. 163–165).
14. Lewis, J. (1995). Fast normalized cross-correlation. http://scribblethink.org/Work/nvisionInterface/nip.pdf. Last access: 2017-03-31.
15. Lewis, J. (1995). Fast template matching. *Vision Interface*, *10*, 120–123.
16. Ma, B., Zimmermann, T., Rohde, M., Winkelbach, S., He, F., Lindenmaier, W., & Dittmar, K.E. (2007). Use of AutoStitch for automatic stitching of microscope images. *Micron*, *38*(5), 492–499. doi:10.1016/j.micron.2006.07.027.
17. NVIDIA: Tesla accelerated computing (2015). http://www.nvidia.com/object/tesla-supercomputing-solutions.html. Last access: 2015-06-26.
18. Pheatt, C. (2008). Intel^{TM} threading building blocks. *Journal of Computing Sciences Colleges*, *23*(4), 298–298.
19. Preibisch, S., Saalfeld, S., & Tomancak, P. (2009). Globally optimal stitching of tiled 3D microscopic image acquisitions. *Bioinformatics*, *25*(11), 1463–1465. doi:10.1093/bioinformatics/btp184.
20. Szeliski, R. (2006). Image alignment and stitching: a tutorial. *Found. Trends. Comput. Graph. Vis.*, *2*(1), 1–104. doi:10.1561/0600000009.
21. TOP500: TOP500 supercomputer sites (2016). http://www.top500.org/. Last access: 2016-06-20.
22. Williams, S., Waterman, A., & Patterson, D. (2009). Roofline: An insightful visual performance model for multicore architectures. *Communications of the ACM*, *52*(4), 65–76. doi:10.1145/1498765.1498785.
23. Xianyi, Z. (2016). OpenBLAS - an optimized BLAS library. http://www.openblas.net/. Last access: 2016-05-12.
24. Zaharia, M., Chowdhury, M., Franklin, M.J., Shenker, S., & Stoica, I. (2010). Spark: Cluster computing with working sets. In *Proceedings of the 2nd USENIX conference on hot topics in cloud computing, HotCloud'10* (p. 10). USENIX Association.

Timothy Blattner works as a research scientist at NIST. He acquired his Bachelor's degree in Computer Science from Marquette University, and his MS and PhD from the University of Maryland Baltimore County in 2016. His primary field of research is in high performance computing, in particular, developing high-level abstractions to take advantage of 'desktop super computers' that consist of multiple GPUs and multiple CPUs on a single node. Dr. Blattner developed a technique for utilizing these systems by transforming an algorithm into a hybrid pipeline workflow. He has formalized this approach into a model and API called the Hybrid Task Graph Scheduler (HTGS).



Walid Keyrouz is a research scientist at NIST, which he joined in 2011. His research interests are in High Performance Computing and Engineering Design. He has an undergraduate degree in Engineering from the American University of Beirut and MS and PhD from Carnegie Mellon University. His first exposure to parallel computing was during his Master's degree in the early 1980s and has used GPUs as compute engines since 2006.

Shuvra S. Bhattacharyya is a Professor in the Department of Electrical and Computer Engineering at the University of Maryland, College Park. He holds a joint appointment in the University of Maryland Institute for Advanced Computer Studies (UMIACS). He is also a part time visiting professor in the Department of Pervasive Computing at the Tampere University of Technology, Finland, as part of the Finland Distinguished Professor Programme (FiDiPro). He is an author of six books, and over 250 papers in the areas of signal processing, embedded systems, electronic design automation, wireless communication, and wireless sensor networks. He received the B.S. degree from the University of Wisconsin at Madison, and the Ph.D. degree from the University of California at Berkeley. He has held industrial positions as a Researcher at the Hitachi America Semiconductor Research Laboratory (San Jose, California), and Compiler Developer at Kuck & Associates (Champaign, Illinois). He has held a visiting research position at the US Air Force Research Laboratory (Rome, New York). He has been a Nokia Distinguished Lecturer (Finland) and Fulbright Specialist (Austria and Germany). He has received the NSF Career Award (USA). He is a Fellow of the IEEE.



Milton Halem is a Research Professor in the Computer Science and Electrical Engineering Department at the University of Maryland, Baltimore County. He acquired his Bachelor's degree in Mathematics from the City College of New York, and his PhD from the Courant Institute of Mathematical Sciences, New York University in 1968. Dr. Halem is Exec. Mgr. of the NSF funded Center for Hybrid Multicore Productivity Research. His current areas of research interest are high performance hybrid computer and storage technologies, machine learning, quantum computing, service oriented computing, remote sensing information systems, extreme events and Big Data Analytics. Prior to joining UMBC in 2003, Dr. Halem served as Assistant Director of the NASA Goddard Space Flight Center. He has more than 150 scientific publications and most noted for his ground breaking research in simulation studies of space observing systems and development of four dimensional data assimilation for weather and climate prediction. In 1999, Dr. Halem was awarded the honorary Doctoral degree from Dalhousie University, CA in recognition for his contributions to the field of computational science.

Mary Brady is the Manager of the Information Systems Group in NIST's Information Technology Laboratory (ITL). She received a M.S. degree in Computer Science from George Washington University, and a B.S. degree in both Computer Science and Mathematics from Mary Washington College. Her group is focused on developing measurements, standards, and underlying technologies that foster innovation throughout the information life cycle from collection and analysis to sharing and preservation.