Published in final edited form as:

Proc IPDPS (Conf). 2013 May ; 2013: 103–114. doi:10.1109/IPDPS.2013.11.

High-throughput Analysis of Large Microscopy Image Datasets on CPU-GPU Cluster Platforms

George Teodoro[1], **Tony Pan**[1], **Tahsin M. Kurc**[1], **Jun Kong**[1], **Lee A. D. Cooper**[1], **Norbert Podhorszki**[2], **Scott Klasky**[2], **Joel H. Saltz**[1]

[1]Center for Comprehensive Informatics, Emory University, Atlanta, GA

[2]Scientific Data Group, Oak Ridge National Laboratory, Oak Ridge, TN

Abstract

Analysis of large pathology image datasets offers significant opportunities for the investigation of disease morphology, but the resource requirements of analysis pipelines limit the scale of such studies. Motivated by a brain cancer study, we propose and evaluate a parallel image analysis application pipeline for high throughput computation of large datasets of high resolution pathology tissue images on distributed CPU-GPU platforms. To achieve efficient execution on these hybrid systems, we have built runtime support that allows us to express the cancer image analysis application as a hierarchical data processing pipeline. The application is implemented as a coarse-grain pipeline of stages, where each stage may be further partitioned into another pipeline of fine-grain operations. The fine-grain operations are efficiently managed and scheduled for computation on CPUs and GPUs using performance aware scheduling techniques along with several optimizations, including architecture aware process placement, data locality conscious task assignment, data prefetching, and asynchronous data copy. These optimizations are employed to maximize the utilization of the aggregate computing power of CPUs and GPUs and minimize data copy overheads. Our experimental evaluation shows that the cooperative use of CPUs and GPUs achieves significant improvements on top of GPU-only versions (up to 1.6×) and that the execution of the application as a set of fine-grain operations provides more opportunities for runtime optimizations and attains better performance than coarser-grain, monolithic implementations used in other works. An implementation of the cancer image analysis pipeline using the runtime support was able to process an image dataset consisting of 36,848 4Kx4K-pixel image tiles (about 1.8TB uncompressed) in less than 4 minutes (150 tiles/second) on 100 nodes of a state-of-the-art hybrid cluster system.

Keywords

Image Segmentation Pipelines; GPGPU; CPUGPU platforms

I. Introduction

Analysis of large datasets is a critical, yet challenging component of scientific studies, because of dataset sizes and the computational requirements of analysis applications.

Sophisticated image scanner technologies developed in the past decade have revolutionized biomedical researchers' ability to perform high resolution microscopy imaging of tissue specimens. With a state-of-the-art scanner, a researcher can capture color images of up to 100Kx100K pixels rapidly. This allows a research project to collect datasets consisting of thousands of images, each of which can be tens of gigabytes in size. Processing of an image consists of several steps of data and computation intensive operations such as normalization, segmentation, feature computation, and classification. Analyzing a single image on a workstation can take several hours, and processing a large dataset can take a very long time. Moreover, a dataset may be analyzed multiple times with different analysis parameters and algorithms to explore different scientific questions, to carry out sensitivity studies, and to quantify uncertainty and errors in analysis results. These requirements create obstacles to the utilization of microscopy imaging in research and healthcare environments and significantly limit the scale of microscopy imaging studies.

The processing power and memory capacity of graphics processing units (GPUs) have rapidly and significantly improved in recent years. Contemporary GPUs provide extremely fast memories and massive multi-processing capabilities, exceeding those of multi-core CPUs. The application and performance benefits of GPUs for general purpose processing have been demonstrated for a wide range of applications [1]. As a result, *hybrid* systems with multi-core CPUs and multiple GPUs are emerging as viable high performance computing platforms for scientific computation [2]. This trend is also fueled by the availability of programming abstractions and frameworks, such as CUDA[1] and OpenCL[2], that have reduced the complexity of porting computational kernels to GPUs. Nevertheless, taking advantage of hybrid platforms for scientific computing still remains a challenging problem. An application developer needs to be concerned about efficient distribution of computational workload not only across cluster nodes but also among multiple CPU cores and GPUs on a hybrid node. The developer also has to take into account the relative performance of application operations on CPUs and GPUs. Some operations are more suitable for massive parallelism and generally achieve higher GPU-vs-CPU speedup values than other operations. Such performance variability should be incorporated into scheduling decisions. On top of these challenges, the application developer has to minimize data copy overheads when data have to be exchanged between application operations. These challenges often lead to underutilization of the power of hybrid platforms.

In this work, we propose and evaluate parallelization strategies and runtime runtime support for efficient execution of large scale microscopy image analyses on hybrid cluster systems. Our approach combines the coarse-grain dataflow pattern with the bag-of-tasks pattern in order to facilitate the implementation of an image analysis application from a set of operations on data. The runtime supports hierarchical pipelines, in which a processing component can itself be a pipeline of operations, and implements optimizations for efficient coordinated use of CPUs and GPUs on a computing node as well as for distribution of computations across multiple nodes. The optimizations studied in this paper include data locality conscious and performance variation aware task assignment, data

[1] http://nvidia.com/cuda
[2] http://www.khronos.org/opencl/

prefetching, asynchronous data copy, and architecture aware placement of control processes in a computation node. Fine-grain operations that constitute an image analysis pipeline typically involve different data access and processing patterns. Consequently, variability in the amount of GPU acceleration of operations is likely to exist. This requires the use of performance aware scheduling techniques in order to optimize the use of CPUs and GPUs based on speedups attained by each operation.

We evaluate our approach using image datasets from brain tumor specimens and an analysis pipeline developed for study of brain cancers on a state-of-the-art hybrid cluster, where each node has multi-core CPUs and multiple GPUs. Experimental results show that coordinated use of CPUs and GPUs along with the runtime optimizations results in significant performance improvements over CPU-only and GPU-only deployments. In addition, multi-level pipeline scheduling and execution is faster than a monolithic implementation, since it can leverage the hybrid infrastructure better. Applying all of these optimizations makes it possible to process an image dataset at 150 tiles/second on 100 hybrid compute nodes.

II. Application Description

The motivation for our work is the in silico studies of brain tumors [3]. These studies are conducted to find better tumor classification strategies and to understand the biology of brain tumors, using complementary datasets of high-resolution whole tissue slide images (WSIs), gene expression data, clinical data, and radiology images. WSIs are captured by taking color (RGB) pictures of tissue specimens stained and fixated on glass slides. Our group has developed WSI analysis applications to extract and classify morphology and texture information from images, with the objective of exploring correlations between tissue morphology, genomic signatures, and clinical data [3]. The WSI analysis applications share a common workflow which consists of the following core stages: 1) image preprocessing tasks such as color normalization, 2) segmentation of micro-anatomic objects such as cells and nuclei, 3) characterization of the shape and texture features of the segmented objects, and 4) machine-learning methods that integrate information from features to classify the images and objects. In terms of computation cost, the preprocessing and classification stages (stages 1 and 4) are inexpensive relative to the segmentation and feature computation stages (stages 2 and 3). The current implementation of the classification stage works at image and patient level and includes significant data reduction prior to the actual classification operation which decreases data and computational requirements. The segmentation and feature computation stages, on the other hand, may operate on hundreds to thousands of images with resolutions ranging from 50K×50K to 100K×100K pixels and on 10^5 to 10^7 micro-anatomic objects (e.g., cells and nuclei) per image. Thus, we target the segmentation and feature computation stages in this paper.

The segmentation stage detects cells and nuclei and delineates their boundaries. It consists of several component operations, forming a dataflow graph (see Figure 1). The operations in the segmentation include morphological reconstruction to identify candidate objects, watershed segmentation to separate overlapping objects, and filtering to eliminate candidates that are unlikely to be nuclei based on object characteristics. The feature computation stage derives quantitative attributes in the form of a feature vector for the entire image

or for individual segmented objects. The feature types include pixel statistics, gradient statistics, edge, and morphometry. Most of the features can be computed concurrently in a multi-threaded or parallel environment.

III. Application Parallelization for High Throughput Execution

The workflow stages described in the previous section were originally implemented in MATLAB. To create high performance versions of the segmentation and feature computation stages, we first implemented GPU-enabled versions, as well as C++-based CPU versions, of individual operations in those stages (Section III-A). We then developed a runtime middleware and methods that combine the bag-of-tasks and coarse-grain dataflow patterns for parallelization across multiple nodes and within each CPU-GPU node (Section III-B). Finally, we incorporated a set of runtime optimizations to reduce computation time and use CPUs and GPUs in a coordinated manner on each compute node (Section III-C).

A. GPU-based Implementations of Operations

We used existing implementations from OpenCV [4] library or from other research groups, or implemented our own if no efficient implementations were available. The Morphological Open operation, for example, is available in OpenCV [4] and is implemented using NVIDIA Performance Primitives (NPP) [5]. The Watershed operation, on the other hand, has only a CPU implementation in the OpenCV library. We used the GPU version developed by Körbes et. al. [6] for this operation. A list of core operations supported by our current implementation along with the sources of the CPU/GPU implementations is presented in Table I.

Several of the methods we developed in the segmentation stage have irregular data access and processing patterns. The *Morphological Reconstruction* (MR) [7] and *Distance Transform* algorithms are used as building blocks in a number of these methods. These algorithms can be efficiently executed on a CPU using a queue structure. In these algorithms, only the computation performed in a subset of the elements (active elements) from the input data domain effectively contribute to the output. Therefore, to reduce the computation cost, the active elements are tracked using a container, e.g., a queue, so that only those elements are processed. When an element is selected for computation, it is removed from the set of active elements. The computation of the active element involves its neighboring elements on a grid, and one or more of the neighbors may be added to the set of active elements as a result of the computation. This process continues until stability is reached; i.e., the container of active elements becomes empty. To port these algorithms to the GPU, we have implemented a hierarchical and scalable queue to store elements (pixels) in fast GPU memories along with several optimizations to reduce execution time. We refer the reader to the following manuscripts [8], [9] for implementation details. The queue-based implementation resulted in significant performance improvements over previously published GPU-enabled versions of the MR algorithm [10]. Our implementation of the distance transform results in a distance map equivalent to that of Danielsson's algorithm [11].

The *connected component labeling* operation (BWLabel) was implemented using the union-find pattern [12]. Conceptually, the BWLabel with union-find first constructs a forest where

each pixel is its own tree. It then merges trees corresponding to adjacent pixels by inserting one tree as a branch of the other. The trees are merged only when the adjacent pixels have the same mask pixel value. During a merge, the roots of two trees are compared by the label values. The root with the smaller label value remains the root, while the other is *grafted* onto the new root. After all the pixels have been visited, pixels belonging to the same component are on the same label tree. The label can then be extracted by flattening the trees and reading the labels. The output of this operation in the computation pipeline, shown in Figure 1, is a labeled mask containing all of the segmented objects.

The operations in the feature computation stage consist of pixel and neighborhood based transformations that are applied to the input image (Color deconvolution, Canny, and Gradient) and computations on individual objects (e.g., nuclei) segmented in the segmentation stage. The feature computations on objects are generally more regular and compute intensive than the operations in the segmentation stage. This characteristics of the feature computation operations lead to better GPU acceleration [13].

B. Parallelization on Distributed CPU-GPU machines

The image analysis application encapsulates multiple processing patterns. First, each image can be partitioned into rectangular tiles, and the segmentation and feature computation steps can be executed on each tile independently. This leads to a bag-of-tasks style processing pattern. Second, the processing of a single tile can be expressed as a hierarchical coarse-grain dataflow pattern. The segmentation and feature computation stages are the first level of the dataflow structure. The second level is the set of fine-grain operations that constitute each of the coarse-grain stages. This formulation is illustrated in Figure 1.

The hierarchical representation lends itself to a separation of concerns and enables the use of different scheduling approaches at each level. For instance, it allows for the possibility of exporting second level operations (*fine-grain operations*) to a local scheduler on a hybrid node, as opposed to describing each pipeline stage as a single monolithic task that should entirely be assigned to a GPU or a CPU. In this way, the scheduler can control tasks at a finer granularity, account for performance variations across the finer grain tasks within a node, and assign them to the most appropriate device.

In order to support this representation, our implementation is built on top of a Manager-Worker model, shown in Figure 2, that combines the bag-of-tasks style execution with the coarse-grain dataflow execution pattern. The application Manager creates stage instances, each of which are represented by a tuple, (*input data chunk, processing stage*), and builds the dependencies among them to enforce the correct pipeline execution. This dependency graph is not completely known prior to execution, thus is built dynamically at runtime. For instance, after the segmentation of a tile, the feature extraction for that particular chunk of data is only dispatched for computation if a certain number of objects were segmented. Since stage instances may be created as a consequence of the computation of other stage instances, it is possible to create loops as the dependency graph may be reinstantiated dynamically.

The granularity of tasks assigned to the application Worker nodes is equal to stage instances, i.e., a segmentation or feature computation stage instance in our case. The tasks are scheduled to the Workers using a demand-driven approach. Stage instances are assigned to the Workers for execution in the same order the instances are created, and the Workers continuously request work as they finalize the execution of the previous instances (see Figure 2). In practice, a single Worker is able to execute multiple application stages concurrently, and the sets of Workers shown in Figure 2 are not necessarily disjoint. All the communication among the processes is done using MPI.

Since a Worker (see Figure 3) is able to use all CPU cores and GPUs in a node concurrently, it may ask for multiple stage instances from the Manager in order to keep all computing devices busy. The maximum number of stage instances assigned to a Worker at a time is a configurable value (*Window size*). The Worker may request multiple stage instances in one request or in multiple requests; in the latter case, the assignment of a stage instance and the retrieval of necessary input data chunks can be overlapped with the processing of an already assigned stage instance.

The Worker Communication Controller (WCC) module runs on one of the CPU cores and is responsible for performing any necessary communication with the Manager. All computing devices used by a Worker are controlled by a local Worker Resource Manager (WRM). When a Worker receives a stage instance from the application Manager and instantiates the pipeline of finer-grain operations in that pipeline instance, each of the fine-grain operation instances, (*input data, operation*), is dispatched for execution with the local WRM. The WRM maps the (*input data, operation*) tuples to the local computing devices as the dependencies between the operations are resolved. In this model of a Worker, one computing thread is assigned to manage each available CPU computing core or a GPU. The threads notify the WRM whenever they become idle. The WRM then selects one of the tuples ready for execution with operation implementation matching the processor managed by that particular thread. When all the operations in the pipeline related to a given stage instance are executed, a callback function is invoked to notify the WCC. The WCC then notifies the Manager about the end of that stage instance and requests more stage instances. During a stage instance destruction phase, it could also instantiate other stages instances as necessary.

C. Efficient Cooperative Execution on CPUs and GPUs

This section describes optimizations that address smart assignment of operations to CPUs and GPUs and data movement between those devices.

1) Performance Aware Task Scheduling (PATS)—The stage instances (Segmentation or Feature Computation) assigned to a Worker create many finer-grain operation instances. The operation instances need to be mapped to available CPU cores and GPUs efficiently in order to fully utilize the computing capacity of a node. Several recent efforts on task scheduling in heterogeneous environments have targeted machines equipped with CPUs and GPUs [14], [15], [16]. These works address the problem of partitioning and mapping tasks between CPUs and GPUs for applications in which operations (or tasks) achieve consistent speedups when executed on a GPU vs on a CPU. The previous efforts differ mainly in

whether they use off-line, on-line, or automated scheduling approaches. However, when there are multiple types of operations in an application, the operations may have different processing and data access patterns and attain different amounts of speedup on a GPU.

In order to use performance variability to our advantage, we have developed a strategy, referred here to as *PATS* (formerly *PRIORITY* scheduling) [13]. This strategy assigns tasks to CPU cores or GPUs based on an estimate of the relative performance gain of each task on a GPU compared to its performance on a CPU core and on the computational loads of the CPUs and GPUs. In this work, we have extended the PATS scheduler to take into account dependencies between operations in an analysis workflow.

The PATS scheduler uses a queue of operation instances, i.e., *(data element, operation)* tuples, sorted based on the relative speedup expected for each tuple. As more tuples are created for execution with each Worker and pending operation dependencies are resolved, more operations are queued for execution. Each new operation is inserted in the queue such that the queue remains sorted (see Figure 3). During execution, when a CPU core or GPU becomes idle, one of the tuples from the queue is assigned to the idle device. If the idle device is a CPU core, the tuple with the minimum estimated speedup value is assigned to the CPU core. If the idle device is a GPU, the tuple with the maximum estimated speedup is assigned to the GPU. The PATS scheduler relies on maintaining the correct relative order of speedup estimates rather than the accuracy of individual speedup estimates. Even if the speedup estimates of two tasks are not accurate with respect to their respective real speedup values, the scheduler will correctly assign the tasks to the computing devices on the node, as long as the order of the speedup values is correct.

Time based scheduling strategies, e.g., heterogeneous earliest finish time, have shown to be very efficient for heterogeneous environments. The main reason we do not use a time based scheduling strategy is that most operations in our case have irregular computation patterns and data dependent execution times. Estimating execution times for those operations would be very difficult. Thus, our scheduling approach uses relative GPU-vs-CPU speedup values, which we have observed are easier to estimate, have less variance, and lead to better scheduling in our example application.

Although we provide CPU and GPU implementations of each operation in our implementation, this is not necessary for correct execution. When an operation has only one implementation, CPU or CPU, the scheduler can restrict the assignment of the operation to the appropriate type of computing device.

2) Data Locality Conscious Task Assignment (DL)—The benefits of using a GPU for a certain computation are strongly impacted by the cost of data transfers between the GPU and the CPU before the GPU kernel can be started. In our execution model, input and output data are well defined as they refer to the input and output streams of each stage and operation. Leveraging this structure, we have extended the base scheduler in order to promote data reuse and avoid penalties due to excessive data movement. After an operation assigned to a GPU has finished, the scheduler explores the operation dependency graph and searches for operations ready for execution that can reuse the data already in the GPU

memory. If the operation speedups are not known, the scheduler always chooses to reuse data instead of selecting another operation that does not reuse data. For the case where speedup estimates for operations are available, the scheduler searches for tasks that reuse data in the dependency graph, but it additionally takes into consideration other operations ready for execution. Although those operations may not reuse data, it may be worthwhile to pay the data transfer penalties if they benefit more from execution on a GPU than the operations that can reuse the data. To choose which operation instance to execute in this situation, the speedup of the dependent operations with the best speedup (S_d) is compared to that of the operation with the best speedup (S_q) that does not reuse the data. The dependent operation is chosen for execution, if $S_d \geq S_q \times (1 - transferImpact)$. Here *transferImpact* is a real value between 0 and 1 and represents the fraction of the operation execution time spent in data transfer. We currently rely on the programmer to provide such an estimate, but we plan to automate this step in a future work.

3) Data Prefetching and Asynchronous Data Copy—Data locality conscious task assignment reduces data transfers between CPUs and GPUs for successive operations in a pipeline. However, there are moments in the execution when data still have to be exchanged between these devices because of scheduling decisions. In those cases, data copy overheads can be reduced by employing pre-fetching and asynchronous data copy. New data can be copied to the GPU in parallel to the execution of the computation *kernel* on a previously copied data [17]. In a similar way, results from previous computations may be copied to the CPU in parallel to a *kernel* execution. In order to employ both data prefetching and asynchronous data copy, we modified the runtime system to perform the computation and communication of pipelined operations in parallel. The execution of each operation using a GPU in this execution mode involves three phases: *uploading*, *processing*, and *downloading*. Each GPU manager thread in the WRM pipelines multiple operations through these three phases. Any input data needed for another operation waiting to execute and the results from a completed operation are copied from and to the CPU in parallel to the ongoing computation in the GPU.

IV. Experimental Evaluation

A. Experimental Setup

We have evaluated the runtime system, optimizations, and application implementation using a distributed memory hybrid cluster, called Keeneland [2]. Keeneland is a National Science Foundation Track2D Experimental System and has 120 nodes in the current configuration. Each computation node is equipped with a dual socket Intel X5660 2.8 Ghz Westmere processor, 3 NVIDIA Tesla M2090 (Fermi) GPUs, and 24GB of DDR3 RAM (See Figure 4). The nodes are connected to each other through a QDR Infiniband switch.

The image datasets used in the evaluation were obtained from brain tumor studies [3]. Each image was partitioned into tiles of 4K×4K pixels. The codes were compiled using “gcc 4.1.2”, “-O3” optimization flag, OpenCV 2.3.1, and NVIDIA CUDA SDK 4.0. The experiments were repeated 3 times. The standard deviation in performance results was not observed to be higher than 2%. The input data were stored in the Lustre filesystem.

B. Performance of Application Operations on GPU

This section presents the performance gains on GPU of the individual pipeline operations. Figure 5 shows the performance gains achieved by each of the fine-grain operations as compared to the single core CPU counterparts. The speedup values in the figure represent the performance gains (1) when only the computation phase is considered (computation-only) and (2) when the cost of data transfer between CPU and GPU is included (computation+data transfer). The figure also shows the percentage of the overall computation time spent in an operation on one CPU core.

The results show that there are significant variations in performance gains among operations, as expected. The most time consuming stages are the ones with the best speedup values – this is in part because of the fact that we have focused on optimizing the GPU implementations of those operations to reduce overall execution time. The feature computation stage stands out as having better GPU acceleration than the segmentation stage. This is a consequence of the former's more regular and compute intensive nature.

This performance evaluation indicates that the task scheduling approach should take into consideration these performance variations to maximize performance on hybrid CPU-GPU platforms. We evaluate the performance impact on pipelined execution of using PATS for scheduling operations in Section IV-C.

C. Cooperative Pipeline Execution using CPUs and GPUs

This section presents the experimental results when multiple CPU cores and GPUs are used together to execute the analysis pipeline. In these experiments, two versions of the application workflow are used: (i) *pipelined* refers to the version described in Section II, where the operations performed by the application are organized as a hierarchical pipeline; (ii) *non-pipelined* that bundles the entire computation of an input tile as a single monolithic task, which is executed either by CPU or GPU. The comparison between these versions is important to understand the performance impact of pipelining application operations.

Two scheduling strategies were employed for mapping tasks to CPUs or GPUs: (i) FCFS which does not take performance variation into consideration; and, (ii) PATS that uses the expected speedups achieved by an operation in the scheduling decision. When PATS is used, the speedup estimates for each of the operations are those presented in Figure 5.

The results for the various configurations are presented in Figure 6, using the three images. In all cases, the CPU speedup using 12 cores is about 9. The sub-linear speedups are a result of the application's high memory bandwidth requirements. The 3-GPU execution achieved about 1.8× speedup on top of the 12 CPU cores version for all images. The coordinated use of CPUs and GPUs improved performance over the 3-GPU executions. We should note that only up to 9 CPU cores are used for computation in the multi-device experiments, because 3 cores are dedicated to GPU control threads. In the non-pipelined version of the application, potential performance gains by using CPUs and GPUs together are limited by load imbalance. If a tile is assigned to a CPU core near the end of the execution, the GPUs will sit idle waiting until the CPU core finishes. This reduces the benefits of cooperative use of computing devices. The performance of PATS for the non-pipelined version is similar

to FCFS. In this case, PATS is not able to make better decisions than FCFS, because the non-pipelined version bundles all the internal operations of a stage in the analysis pipeline as a single task, hence the performance variations of the operations are not exposed to the runtime system.

The CPU-GPU execution of the pipelined version of the application with FCFS (3 GPUs + 9 CPU cores - FCFS pipelined) also improved the 3-GPU execution, reaching similar performance to that of the non-pipelined execution. This version of the application requires that the data are copied to and from a GPU before and after an operation in the pipeline is assigned to the GPU. This introduces a performance penalty due to the data transfer overheads, which are about 13% of the computation time as shown in Figure 5, and limits the performance improvements of the pipelined version. The advantage of using the pipelined version in this situation is that load imbalance among CPUs and GPUs is reduced. The assignment of computation to CPUs or GPUs occurs at a finer-grain; that is, application operations in the second level of the pipeline make up the tasks scheduled to CPUs and GPUs, instead of the entire computation of a tile as in the non-pipelined version. Figure 6 also presents the performance of the PATS scheduling for the pipelined version of the application. As is seen from the figure, processing of tiles using PATS is about 1.33× faster than using FCFS with the non-pipelined or pipelined version of the application. The performance gains result from the ability of PATS to better assign the application internal operations to the most suited computing devices.

Figure 7 presents the percent of tasks that PATS assigned to the CPUs or GPUs for each pipeline stage. As is shown, the execution of components with lower speedups are mostly performed using the CPUs, while the GPUs are kept occupied with the operations that achieve higher speedups. For reference, when using FCFS with the pipelined version, about 62% of the tasks for each operation are assigned to GPUs and the rest to CPUs regardless of performance variations between the operations.

D. Data Locality Conscious Scheduling/Data Prefetching

This section evaluates the performance impact of the data locality conscious task assignment (DL) and data prefetching and asynchronous data download (Prefetching) optimizations. Figure 8 presents the performance improvements with these optimizations for both PATS and FCFS. For reference, the GPU-only and CPU-only performance for each of the images are the same presented in the last section (Figure 6). As is shown in the figure, the pipelined version with FCFS and DL is able to improve on the performance of the non-pipelined version by about 1.1× for all input images. When Prefetching is used in addition to FCFS and DL (“3GPUs + 9 CPU core - pipelined FCFS + DL + Prefetching”), there are no significant performance improvements. The main reason is that DL already avoids unnecessary CPU-GPU data transfers; therefore, Prefetching will only be effective in reducing the cost of uploading the input tile to the GPU and downloading the final results from the GPU. These costs are small and limit the performance gains resulting from Prefetching.

Figure 8 also shows the performance results for PATS when DL and Prefetching are employed. The use of DL improves the performance of PATS as well, but the gains achieved

(1.04×) with DL are smaller than those in FCFS. The estimated speedups for the operations are available, thus PATS will check whether it is worthwhile to download the operation results to map another operation to the GPU. The number of upload/downloads avoided by using DL is also smaller than when FCFS is used, which explains the performance gain difference. Prefetching with DL results in an additional 1.03× performance improvement. This optimization was more effective in this case, because the volume of data transferred between the CPU and the GPU is higher than when FCFS with DL is employed.

E. Impact of Worker Request Window Size

This section analyzes the effect of the demand-driven window size between Manager and Workers (i.e., the number of pipeline stage instances concurrently assigned to a Worker) on the CPU-GPU scheduling strategies utilized by the Worker. In this experiment, we used 3 GPUs and 9 CPU cores (with 3 CPU cores allocated to the GPU manager threads) with FCFS and PATS. The window-size is varied from 12 until no significant performance changes are observed.

Figure 9 presents the execution times. The performance of FCFS is impacted little by variation in the window size. The performance of PATS, on the other hand, is limited for small window sizes. In the scenario where the window size is 12, FCFS and PATS tend to make the same scheduling decisions, because only a single operation will be usually available when a processor requests work. This makes the decision trivial and equal for both strategies. When the window size is increased, however, the scheduling decision space becomes larger, providing PATS with opportunities to make better task assignments. As is shown in the figure, with a window size of 15, PATS already achieves near its best performance. This is another good property of PATS, since very large window sizes can create load imbalance among Workers.

The profile of the execution (% of tasks processed by GPU) as the window size is varied as is displayed in Figure 10. As the window size increases, PATS changes the assignment of tasks, and operations with higher speedups are more likely to be executed by GPUs. FCFS profile is not presented in the same figure, but its profile is similar to PATS with a window size of 12 for all configurations.

F. Sensitivity to Inaccurate Speedup Estimation

In this section, we empirically evaluate the sensitivity of PATS to errors in the GPU-vs-CPU speedup estimation of operations. For the sake of this evaluation, we intentionally inserted errors in the estimated speedup values of the application operations in a controlled manner. The estimated speedup values of operations with lower speedups that are mostly scheduled to the CPUs (Morph. Open, AreaThreshold, FillHoles, and BWLabel) were increased, while those of other operations were decreased. The changes were calculated as a percentage of an operation's original estimated speedup, and the variation range was from 0% to 100%.

The execution times for different error rates are presented in Figure 11. The results show that PATS is capable of performing well even with high errors and error rates in speedup estimations. For instance, when 60% estimation error is used, the performance of the pipeline is only 10% worse than the initial case (0% speedup estimation error). At 70%

and 80% errors, the performance of PATS is more affected, because stages with lower GPU speedups (such as AreaThreshold, FillHoles, and BWLabel), which were previously executed on the CPU, are now scheduled for execution on a GPU. Nevertheless, PATS still performs better than FCFS, because the operations in the feature computation stage are not miss-ordered. To emulate 100% estimation error, we set to 0 the speedups of all substages that in practice have higher speedups, and double the estimated speedups of the other stages that in reality have lower speedup values. This forces PATS to preferably assign operations with low speedups to GPU and the ones with high speedup to CPU. Even with this level of error, the execution times are only about 10% worse than those using FCFS.

G. Multi-node Scalability

This section presents the performance evaluation of the cancer image analysis application when multiple computation nodes are used. The evaluation was carried out using 340 glioblastoma brain tumor WSIs, which were partitioned into a total of 36,848 4K×4K tiles. Similarly to other experiments, the input data tiles were stored as image files in the Lustre filesystem. Therefore, the results presented in this manuscript represent real end-to-end executions of the application, which include overheads for reading input data.

The strong scaling evaluation is presented in Figure 12. First, Figure 12(a) shows the execution times for all configurations of the application when the number of computing nodes is varied from 8 to 100. All the application versions achieved improvements as the number of nodes increase, and the comparison of the approaches shows that the cooperative CPU-GPU execution resulted in speedups of up to 2.7× and 1.7× on top of the “12-CPU cores non-pipelined” (CPU-only) version, respectively, for PATS and FCFS. As shown in the results figure, PATS with optimizations achieved the best performance for all number of computing nodes.

Figure 12(b) also presents the parallelization efficiency for all versions of the application. As may be noticed, the parallelization efficiency reduces in different rates for the application versions as the number of nodes increase. For instance, the efficiency on 100 nodes is about 85% for the CPU-only version of the application, while it is nearly 70% for the CPU-GPU cooperative executions. The main bottleneck for better parallelization efficiency is the I/O overhead of reading image tiles. As the number of nodes increases, I/O operations become more expensive, because more clients access the file system in parallel. The strategies that use cooperative CPU-GPU execution have smaller efficiency simply because they are faster and, consequently, require more I/O operations per unit of time. If only the computation times were measured, the efficiency for those versions would increase to about 93%. Even with the I/O overheads, the application achieved good scalability and was able to process the entire set of 36,848 tiles in less than four minutes when 100 nodes were employed, using a total of the 1,200 CPU cores and 300 GPUs in cooperation and PATS. This represents a huge improvement in data processing capabilities. Currently, as discussed in Section VI, we are evaluating efficient I/O mechanisms to improve the performance of this component of the application on large scale machines.

V. Related Work

The use of hybrid computing architectures with hardware accelerators is growing in HPC leadership supercomputing systems [2]. The appropriate utilization of hybrid systems, however, typically requires complex software instruments to deal with a number of peculiar aspects of the different processors available. This challenge has motivated a number of languages and runtime frameworks [15], [14], [16], [18], [19], [20], [21], [22], [23], [24], [25], [26], specialize libraries [4], and compiler techniques [27].

The Mars [15] and Merge [14] projects evaluate the cooperative use of CPUs and GPUs to speed up MapReduce computations. Mars has examined the benefits of partitioning Map and Reduce tasks between CPU and GPU statically. Merge has extended that approach with dynamic distribution of work at runtime. The Qilin [16] system has further proposed an automated methodology to map computation tasks to CPUs and GPUs. The Qilin strategy is based on having a profiling phase, where performance data of the target application is collected. This data is used to build a performance model to estimate the best work division. Neither ther of these solutions (Mars, Merge, and Qilin), however, are able to take advantage of distributed systems.

PTask [28] provides OS abstractions for execution of tasks, called ptasks, on GPU equipped machines. It treats GPUs as first class resources and provides methods for scheduling tasks to GPUs, fairness, and isolation. Applications are represented using a data-flow model. Our approach differs from Ptask in several ways. A task in our framework is dynamically bound to a GPU or CPU during execution. ptasks in the PTask framework, on the other hand, cannot be re-targeted. Our PATS scheduler employs a different priority metric, the relative performance of (data, oper) on GPU vs CPU, for mapping tasks to processors. These optimizations are targeted at improving throughput. We also allow for cyclic and dynamic dependency graphs, hierarchical pipelines, and multi-node execution. We believe PTask and our system could co-exist and benefit from each other.

Efficient execution of applications on distributed CPUGPU equipped platforms has been an objective of several projects [23], [24], [25], [26], [22], [29], [30]. Ravi et al. [24], [26] proposes techniques for automatic translation of generalized reductions to CPU-GPU environments via compile-time techniques, which are coupled with runtime support to coordinate execution. The runtime system techniques employ auto-tuning approaches to dynamically partition tasks among CPUs and GPUs. The work by Hartley et al. [25] is contemporary to Ravi's and proposes runtime strategies for divisible workloads. The OmpSs [29] supports efficient asynchronous execution of dataflow applications automatically generated by compiler techniques from serial annotated code. OmpSs statically binds the tasks to one of the available processors, while our approach dynamically schedules tasks to processors during execution according to the performance benefits and device suitability of tasks.

DAGuE [23] and StarPU [18], [31] are frameworks that support execution of regular linear algebra applications on CPU-GPU machines. These systems represent an application as a DAG of operations and ensure that dependencies are respected. They offer different

scheduling policies, including those that prioritize computation of critical paths in the dependency graph in order to maximize parallelism. They include support for multi-node execution and assume that the application DAG is static and known before execution. This structure fits well with regular linear algebra applications, but is a limitation that prevents their use in irregular and dynamic applications such as ours. In our application, the dependency graph representing the application must be dynamically built during the execution, as the computation of the next stage of the analysis pipeline may depend on the results of the current stage. For instance, the entire feature computation should not be computed, if nuclei are not found in the segmentation stage. Additionally, neither of the previous solutions allow for the representation of the application as a multi-level pipeline, which is key to achieving high performance with fine-grain task management.

Our work targets a scientific data analysis pipeline, including the GPU/CPU implementations of several challenging irregular operations. In addition, we develop support for execution of applications that can be described as a multilevel pipeline of operations, where coarse-grain stages are divided into fine-grain operations. In this way, we can leverage variability in the amount of GPU acceleration of fine-grain operations that was not possible in the previous works. We also investigate a set of optimizations that include data locality aware task assignment. This optimization dynamically groups operations that present good performance according to the current set of tasks ready to execute on a machine, instead of doing it statically prior to execution as in our previous work [13]. Data prefetching and asynchronous data transfer optimizations are also employed in order to maximize computational resource utilization.

VI. Conclusions and Future Directions

Hybrid CPU-GPU cluster systems offer significant computing and memory capacity to address the computational needs of large scale scientific analyses. In this paper, we have developed an image analysis application that can fully exploit such platforms to achieve high-throughput data processing rates. We have shown that significant performance improvements are achieved when an analysis application can be assembled as pipelines of fine-grain operations, as compared to bundling all internal operations in one or two monolithic methods. The former allows for exporting application processing patterns more accurately to the runtime environment and empowers the middleware system to make better scheduling decisions. Performance aware task scheduling coupled with function variants enable efficient coordinated use of CPU cores and GPUs in pipelined operations. Performance gains can further be increased on hybrid systems through such additional runtime optimizations as locality conscious task mapping, data prefetching, and asynchronous data copy. Employing a combination of these optimizations, our application implementation has achieved a processing rate of about 150 tiles per second when 100 nodes, each with 12 CPU cores and 3 GPUs, are used. These levels of processing speed make it feasible to process very large datasets and would enable a scientist to explore different scientific questions rapidly and/or carry out algorithm sensitivity studies.

The current implementation of the classification step (step 4 in Section II) clusters images into groups based on average feature values per image. The average feature values can be

computed by maintaining on each compute node a running sum of the feature values of segmented objects for each image. The partial sums on the nodes can then be accumulated in a global sum operation, and the average feature values per image can be computed. Thus, the amount of data transferred from the feature computation step to the classification step is relatively small. However, there are cases when the output from a stage, or even from an operation, needs to be staged to disk. For example, studying the sensitivity to input parameters and algorithm variations of output from the segmentation stage would require us to execute multiple runs. It might not be possible, due to time and resource constraints, to maintain output from a run in memory until all the runs have been completed. The output from a stage in a single run or multiple runs may also need to be stored on disk for inspection or visualization at a later time. As a future work, in order to support the I/O requirements in such cases, we are developing an I/O component based on a stream-I/O approach, drawing from filter-stream networks [32], [33], [34], [35], [36] and data staging [37], [38]. This implementation facilitates flexibility. The I/O processes can be placed on different physical processors in the system. For example, if a system had separate machines for I/O purposes, the I/O nodes could be placed on those machines. Moreover, the implementation allows us to leverage different I/O sub-systems. In addition to POSIX I/O in which each I/O process writes out its buffers independent of other I/O nodes, we have integrated ADIOS [39] for data output. ADIOS is shown to be efficient, portable, and scalable on supercomputing platforms and for a range of applications. We are in the process of carrying out initial performance evaluations of the I/O component.

Acknowledgment

This work was supported in part by HHSN261200800001E from the National Cancer Institute, R24HL085343 from the National Heart Lung and Blood Institute, by R01LM011119-01 and R01LM009239 from the National Library of Medicine, RC4MD005964 from National Institutes of Health, and PHS UL1RR025008 from the Clinical and Translational Science Awards program. This research used resources of the Keeneland Computing Facility at the Georgia Institute of Technology, which is supported by the National Science Foundation under Contract OCI-0910735.

REFERENCES

1. NVIDIA. GPU Accelerated Applications. 2012. [Online]. Available: http://www.nvidia.com/object/gpu-accelerated-applications.html
2. Vetter JS, Glassbrook R, Dongarra J, Schwan K, Loftis B, McNally S, Meredith J, Rogers J, Roth P, Spafford K, Yalamanchili S. Keeneland: Bringing Heterogeneous GPU Computing to the Computational Science Community. Computing in Science and Engineering. 2011; 13
3. Cooper LAD, Kong J, Gutman DA, Wang F, Cholleti SR, Pan TC, Widener PM, Sharma A, Mikkelsen T, Flanders AE, Rubin DL, Meir EGV, Kurc TM, Moreno CS, Brat DJ, Saltz JH. An integrative approach for in silico glioma research. IEEE Trans Biomed Eng. 2010; 57 (10) :2617–2621. [PubMed: 20656651]
4. Bradski G. The OpenCV Library. Dr. Dobb's Journal of Software Tools. 2000
5. NVIDIA. NVIDIA Performance Primitives(NPP). Feb 11, 2011. [Online]. Available: http://developer.nvidia.com/npp
6. Körbes, A; Vitor, GB; de Alencar Lotufo, R; Ferreira, JV. Advances on watershed processing on GPU architecture. Proceedings of the 10th International Conference on Mathematical Morphology, ser. ISMM'11; 2011;
7. Vincent L. Morphological grayscale reconstruction in image analysis: Applications and efficient algorithms. IEEE Transactions on Image Processing. 1993; 2 :176–201. [PubMed: 18296207]

8. Teodoro G, Pan T, Kurc TM, Cooper L, Kong J, Saltz JH. A Fast Parallel Implementation of Queue-based Morphological Reconstruction using GPUs. Emory University, Center for Comprehensive Informatics Technical Report CCI-TR-2012-2. Jan. 2012
9. Teodoro G, Pan T, Kurc T, Kong J, Cooper L, Saltz J. Efficient Irregular Wavefront Propagation Algorithms on Hybrid CPU-GPU Machines. Parallel Computing. 2013
10. Karas, P. *MEMICS*, ser. OASICS. Vol. 16. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik; Germany: 2010. Efficient Computation of Morphological Greyscale Reconstruction.
11. Danielsson P-E. Euclidean distance mapping. 14. Computer Graphics and Image Processing. 1980 :227–248.
12. Oliveira VMA, de Alencar Lotufo R. A Study on Connected Components Labeling algorithms using GPUs. SIBGRAPI. 2010
13. Teodoro, G; Kurc, TM; Pan, T; Cooper, LA; Kong, J; Widener, P; Saltz, JH. Accelerating Large Scale Image Analyses on Parallel, CPU-GPU Equipped Systems. 26th IEEE International Parallel and Distributed Processing Symposium (IPDPS); 2012; 1093–1104.
14. Linderman MD, Collins JD, Wang H, Meng TH. Merge: a programming model for heterogeneous multi-core systems. 43. SIGPLAN Not. 2008; (3) :287–296.
15. He B, Fang W, Luo Q, Govindaraju NK, Wang T. Mars: A MapReduce Framework on Graphics Processors. Parallel Architectures and Compilation Techniques. 2008
16. Luk, C-K; Hong, S; Kim, H. Qilin: Exploiting parallelism on heterogeneous multiprocessors with adaptive mapping. 42nd International Symposium on Microarchitecture (MICRO); 2009;
17. Jablin, TB; Prabhu, P; Jablin, JA; Johnson, NP; Beard, SR; August, DI. Automatic CPU-GPU communication management and optimization. Proceedings of the 32nd ACM SIGPLAN conference on Programming language design and implementation, ser. PLDI '11; 2011; 142–151.
18. Augonnet, C; Thibault, S; Namyst, R; Wacrenier, P-A. Starpu: A unified platform for task scheduling on heterogeneous multicore architectures. Euro-Par '09: Proceedings of the 15th International Euro-Par Conference on Parallel Processing; 2009; 863–874.
19. Diamos, GF; Yalamanchili, S. Harmony: an execution model and runtime for heterogeneous many core systems. Proceedings of the 17th international symposium on High performance distributed computing, ser. HPDC '08; New York, NY, USA. 2008; ACM; 197–200.
20. Teodoro G, Sachetto R, Sertel O, Gurcan M, W. M. Catalyurek U, Ferreira R. Coordinating the use of GPU and CPU for improving performance of compute intensive applications. IEEE Cluster. 2009 :1–10.
21. Sundaram, N; Raghunathan, A; Chakradhar, ST. A framework for efficient and scalable execution of domain-specific templates on GPUs. IPDPS '09: Proceedings of the 2009 IEEE International Symposium on Parallel and Distributed Processing; 2009; 1–12.
22. Teodoro, G; Hartley, TDR; Catalyurek, U; Ferreira, R. Run-time optimizations for replicated dataflows on heterogeneous environments. Proc. of the 19th ACM International Symposium on High Performance Distributed Computing (HPDC); 2010; 13–24.
23. Bosilca, G; Bouteiller, A; Herault, T; Lemarinier, P; Saengpatsa, N; Tomov, S; Dongarra, J. Performance Portability of a GPU Enabled Factorization with the DAGuE Framework. 2011 IEEE International Conference on Cluster Computing (CLUSTER); sept. 2011; 395 –402.
24. Ravi, V; Ma, W; Chiu, D; Agrawal, G. Compiler and runtime support for enabling generalized reduction computations on heterogeneous parallel configurations. Proceedings of the 24th ACM International Conference on Supercomputing. ACM; 2010; 137146
25. Hartley TDR, Saule E, Çatalyürek ÜV. Automatic dataflow application tuning for heterogeneous systems. HiPC. IEEE. 2010 :1–10.
26. Huo, X; Ravi, V; Agrawal, G. Porting irregular reductions on heterogeneous CPU-GPU configurations. 18th International Conference on High Performance Computing (HiPC); dec. 2011; 1 –10.
27. Lee, S; Min, S-J; Eigenmann, R. OpenMP to GPGPU: a compiler framework for automatic translation and optimization. PPoPP '09: Proceedings of the 14th ACM SIGPLAN symposium on Principles and practice of parallel programming; 2009; 101–110.

28. Rossbach, CJ; Currey, J; Silberstein, M; Ray, B; Witchel, E. Ptask: operating system abstractions to manage gpus as compute devices. Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles, ser. SOSP '11; New York, NY, USA. 2011; ACM; 233–248.

29. Bueno, J; Planas, J; Duran, A; Badia, R; Martorell, X; Ayguade, E; Labarta, J. Productive Programming of GPU Clusters with OmpSs. 2012 IEEE 26th International Parallel Distributed Processing Symposium (IPDPS); may 2012; 557 –568.

30. Teodoro G, Hartley T, Catalyurek U, Ferreira R. Optimizing dataflow applications on heterogeneous environments. Cluster Computing. 2012; 15 :125–144.

31. Augonnet, C, Aumage, O, Furmento, N, Namyst, R, Thibault, S. StarPU-MPI: Task Programming over Clusters of Machines Enhanced with Accelerators. In: Träff, S. B. Jesper Larsson; Dongarra, J, editors. The 19th European MPI Users' Group Meeting (EuroMPI 2012), ser. LNCS. Vol. 7490. Springer; Vienna, Autriche: 2012.

32. Arpaci-Dusseau, RH, Anderson, E, Treuhaft, N, Culler, DE, Hellerstein, JM, Patterson, DA, Yelick, K. IOPADS '99: Input/Output for Parallel and Distributed Systems. Atlanta, GA: May, 1999 Cluster I/O with River: Making the Fast Case Common.

33. Plale B, Schwan K. Dynamic Querying of Streaming Data with the dQUOB System. IEEE Trans. Parallel Distrib. Syst. 2003; 14 (4) :422–432.

34. Kumar VS, Sadayappan P, Mehta G, Vahi K, Deelman E, Ratnakar V, Kim J, Gil Y, Hall MW, Kurc TM, Saltz JH. An integrated framework for performance-based optimization of scientific workflows. HPDC. 2009 :177–186. [PubMed: 22068617]

35. Tavares T, Teodoro G, Kurc T, Ferreira R, Guedes D, Meira WJ, Catalyurek U, Hastings S, Oster S, Langella S, Saltz J. An Efficient and Reliable Scientific Workflow System. IEEE International Symposium on Cluster Computing and the Grid. 2007; 0 :445–452.

36. Teodoro G, Tavares T, Ferreira R, Kurc T, Meira J, Wagner, Guedes D, Pan T, Saltz J. A run-time system for efficient execution of scientific workflows on distributed environments. International Journal of Parallel Programming. 2008; 36 :250–266. [PubMed: 22582009]

37. Docan C, Parashar M, Klasky S. Dataspaces: an interaction and coordination framework for coupled simulation workflows. HPDC. 2010 :25–36.

38. Abbasi H, Wolf M, Eisenhauer G, Klasky S, Schwan K, Zheng F. Datastager: scalable data staging services for petascale applications. Cluster Computing. 2010; 13 (3) :277–290.

39. Lofstead JF, Klasky S, Schwan K, Podhorszki N, Jin C. Flexible IO and integration for scientific codes through the adaptable IO system (ADIOS). CLADE. 2008 :15–24.

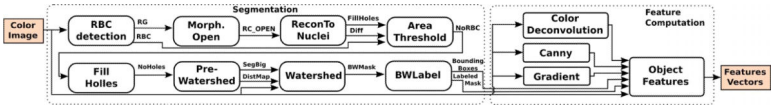


Figure 1.

Pipeline for segmenting nuclei in a whole slide tissue image and computing their features. The input to the pipeline is an image or image tile. The output is a set of features for each segmented nucleus.

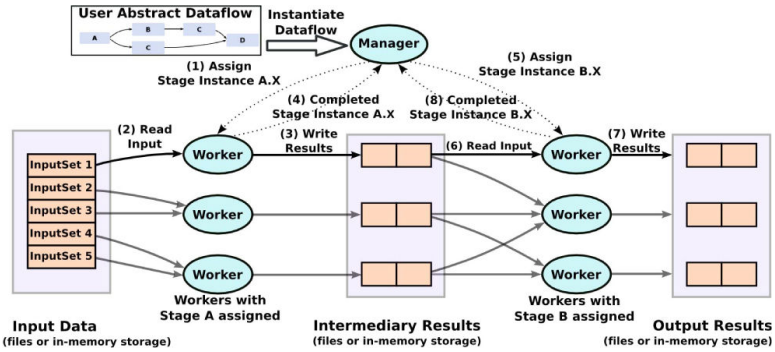


Figure 2.
Overview of the multi-node parallelization strategy.

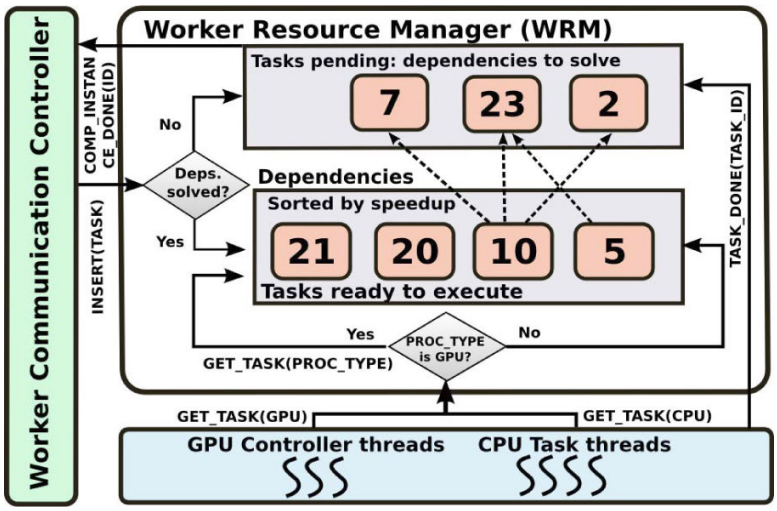


Figure 3.

A Worker is a multi-thread process. It uses all the devices in a hybrid node via the local Worker Resource Manager, which coordinates the scheduling and mapping of operation instances assigned to the Worker to CPU cores and GPUs.

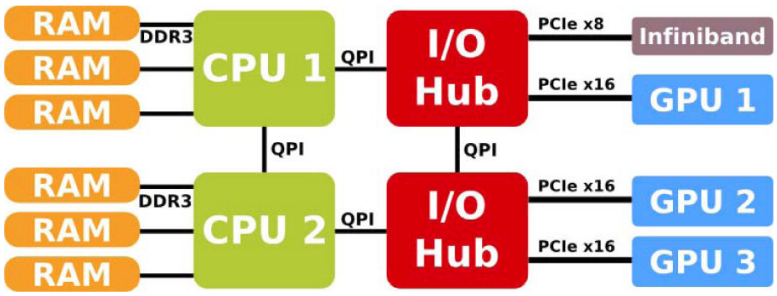


Figure 4.
Architecture of a Keeneland node.

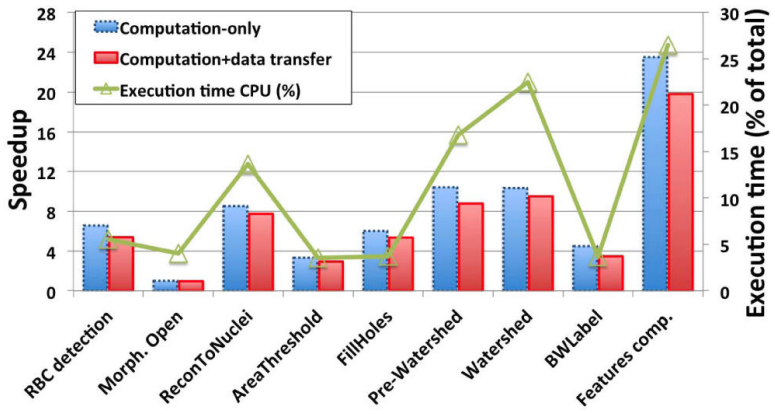


Figure 5.
Evaluation of the GPU-based implementations of application components (operations).

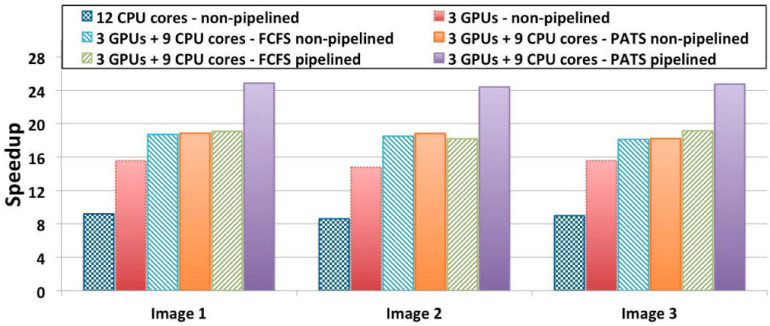


Figure 6.

Application scalability when multiple CPUs and GPUs are used via the PATS and FCFS scheduling strategies.

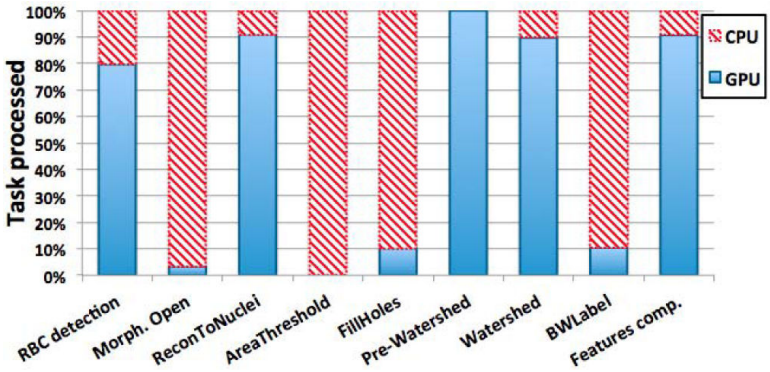


Figure 7.
Execution profile (% of tasks processed by CPU or GPU) using PATS per pipeline stage.

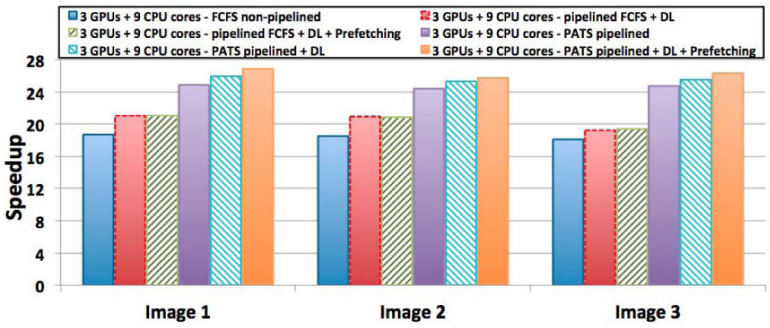


Figure 8.
Performance impact of data locality conscious mapping and asynchronous data copy optimizations.

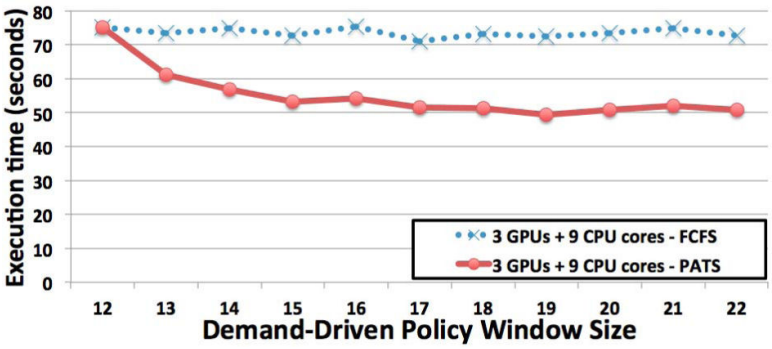


Figure 9.
Performance impact of request window size.

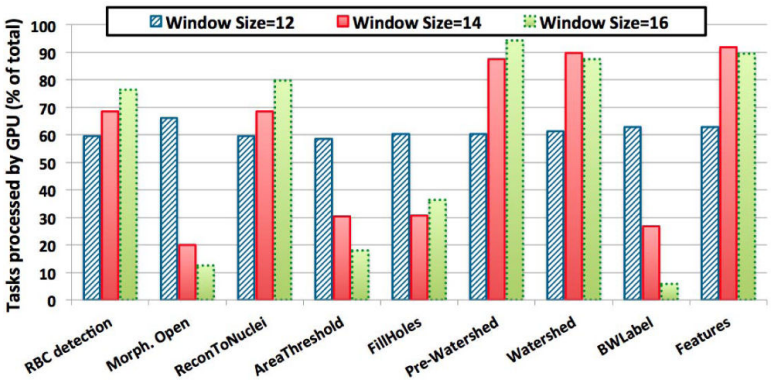


Figure 10.
Execution scheduling profile for different window sizes and the PATS strategy.

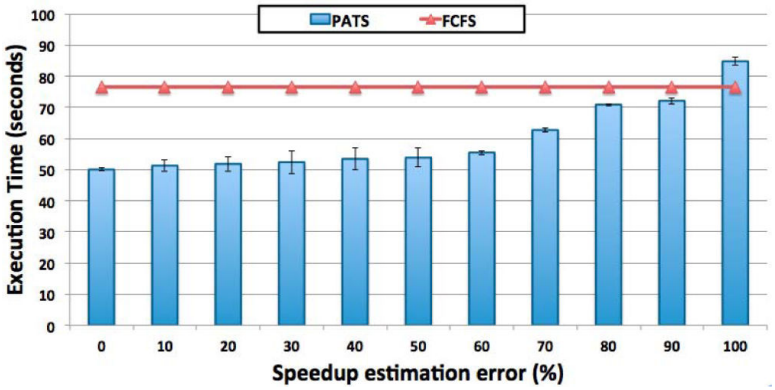
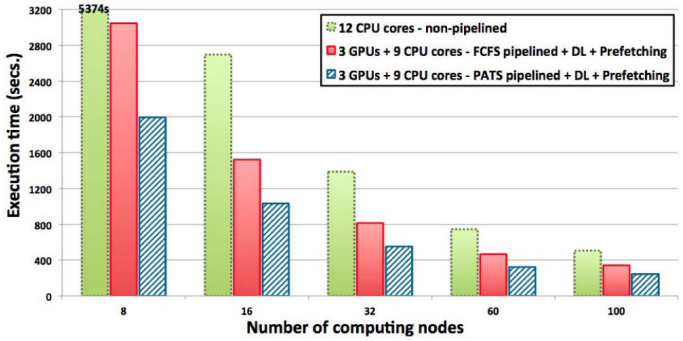
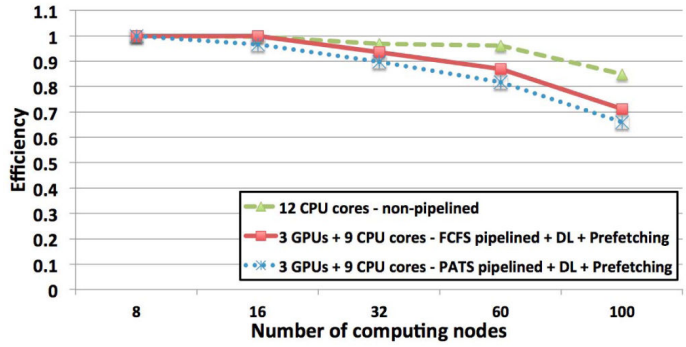


Figure 11.
Performance of PATS when errors in speedup estimation for the pipeline operations are introduced.



(a) Execution times.



(b) Parallelization efficiency.

Figure 12.
Multi-node scalability: strong scaling evaluation.

Table I Sources of CPU and GPU implementations of operations in the segmentation and feature computation stages.

Pipeline operation	**CPU source**	**GPU source**
RBC detection	OpenCV and Vincent [7] Morph. Reconstruction (MR)	Implemented
Morph. Open	OpenCV (by a 19x19 disk)	OpenCV
ReconToNuclei	Vincent [7] MR	Implemented
AreaThreshold	Implemented	Implemented
FillHolles	Vincent [7] MR	Implemented
Pre-Watershed	Vincent [7] MR and OpenCV for distance transformation	Implemented
Watershed	OpenCV	Korbes [6]
BWLabel	Implemented	Implemented
Features comp.	Implemented. OpenCV(Canny)	Implemented. OpenCV(Canny)