Published in final edited form as:

Proc IPDPS (Conf). 2012 May ; 2012: 1093–1104. doi:10.1109/IPDPS.2012.101.

Accelerating Large Scale Image Analyses on Parallel, CPU-GPU Equipped Systems

George Teodoro, Tahsin M. Kurc, Tony Pan, Lee A.D. Cooper, Jun Kong, Patrick Widener, Joel H. Saltz

Center for Comprehensive Informatics Emory University Atlanta, GA 30322

Abstract

The past decade has witnessed a major paradigm shift in high performance computing with the introduction of accelerators as general purpose processors. These computing devices make available very high parallel computing power at low cost and power consumption, transforming current high performance platforms into heterogeneous CPU-GPU equipped systems. Although the theoretical performance achieved by these hybrid systems is impressive, taking practical advantage of this computing power remains a very challenging problem. Most applications are still deployed to either GPU or CPU, leaving the other resource under- or un-utilized. In this paper, we propose, implement, and evaluate a performance aware scheduling technique along with optimizations to make efficient collaborative use of CPUs and GPUs on a parallel system. In the context of feature computations in large scale image analysis applications, our evaluations show that intelligently co-scheduling CPUs and GPUs can significantly improve performance over GPU-only or multi-core CPU-only approaches.

Keywords

Image analysis; In Silico; Microscopy; CPU-GPU systems

I. Introduction

High-resolution images from advanced digital microscopy scanners provide rich information about the morphological and functional characteristics of biological systems. In the past decade, devices that can acquire high-resolution images from whole tissue slides and tissue microarrays have become more affordable, faster, and practical. Image scanning times have decreased from 6-8 hours per whole tissue slide to a few minutes with advanced scanners; and improvements in auto-focusing and slide holders have facilitated high throughput image generation from batches of slides with minimal manual intervention. As a result, it is rapidly becoming possible for even medium-size studies to routinely generate hundreds to thousands of whole slide images per day. Analysis of digital microscopy images, however, poses a variety of systems software and data management issues.

A typical whole pathology slide digitization may produce 20 billion pixels (at 40X magnification), resulting in an 8-bit, RGB color, uncompressed image 60 GB in size. In a typical analysis scenario, a single analysis type (e.g., feature extraction and region classification) for a single-layer image stack at 5X objective magnification may require approximately 10 CPU-hours. It is not feasible to process hundreds or thousands of images on a workstation.

Emerging heterogeneous high performance computing architectures offer viable platforms to address the computational challenges of analyzing large image datasets. General-purpose graphics processing units (GPGPUs) have become a popular implementation platform for scientific applications. HPC configurations consisting of multi-core CPUs and multiple GPUs provide an attractive alternative to traditional homogeneous computing clusters and are being deployed at national labs and supercomputing centers. Although the theoretical performance achieved by these hybrid systems is impressive, taking practical advantage of this computing power remains a very challenging problem.

In this paper, we describe techniques for dynamic smart work partitioning, load balancing, and performance-aware task grouping in order to make efficient collaborative use of available CPUs and GPUs in the context of computation of image and object features. Our evaluations show that intelligently co-scheduling CPUs and GPUs can significantly improve performance over GPU-only or multi-core CPU-only approaches and that performance can further be improved by carefully clustering operations and data elements to reduce scheduling and data copy overheads.

II. Application Description

Our work is motivated by imaging studies that quantify and characterize the micro anatomic morphology of imaged whole slide tissue specimens [1]. Such studies involve hundreds or thousands of high resolution microscopy images and execute many interrelated analysis pipelines on these images. A typical analysis pipeline for characterization of subcellular features consists of the following main steps (Figure 1): Segmentation of nuclei (and other microscopic objects), Feature extraction (i.e., computation of features per image and per nucleus), and Classification. In this paper we develop an implementation of the feature extraction step on machines with CPU-GPU nodes. In this section, we provide a brief overview of the entire analysis pipeline.

The *segmentation step* automatically identifies candidate regions of nuclei, removes red blood cells from the candidates, segments objects in those regions as candidate nuclei, excludes objects that are too large or too small, and separates adjacent nuclei. The output of the segmentation stage are bitmap masks and the polygonal boundaries of nuclei, which may be stored in a database for subsequent retrieval and feature extraction, or pipelined directly to the feature extraction step.

The *feature extraction step* calculates informative, quantitative descriptors for the whole image or individual segmented objects. A list of features computed in our current implementation are shown in Table I. The feature types include pixel statistics, gradient

statistics, Haralick features [2], edge, and morphometry. The first four features in the table are computed for the image and each nucleus, whereas the morphometry features are computed for nuclei only. The computation of most of the features can be performed independently. Some features require the same intermediate results, such as histograms or co-occurrence matrices, when computing the final feature values. At the end of the feature extraction stage, the computed features may be stored in a database or forwarded to the classification step.

The *classification step* utilizes common statistical approaches and machine learning techniques, including discriminant analysis and k-means clustering, to meaningfully label objects, regions, or images based on the features.

A. Application Composition and Implementation

We use a component-based implementation approach, in which the feature extraction step is expressed as a set of components. Each component performs a distinct feature computation. We employ the concept of generic functions [3], [4] that enables the runtime system to choose the appropriate function variants during the execution. In this concept, the application developer provides either a single function for all computing device types in the system or multiple variants of the function, each implemented for a different computing device type. In our case, two variants of each function (component) are provided: a CPU version and a GPU version.

An instance of a function with input data forms a task to be scheduled for execution. Multiple tasks can be dispatched and executed concurrently. The runtime system (Section III) decides the scheduling and mapping of tasks to efficiently utilize multiple computation nodes and computing devices (i.e., CPUs and GPUs). Low level issues, such as bandwidth and latency related to Non-Uniform Memory Access (NUMA), are hidden from the application developer, and internally handled by the runtime system as it maps tasks to the computing devices. Task mapping on a node is carried out on the fly using a sorted queue of tasks. The feature computation tasks are sorted in the queue based on each task's estimated GPU vs CPU-core speedup. New tasks are inserted into the queue such that the queue remains sorted. The scheduler assigns the tasks to idle computing devices dynamically either from the front of the queue (lower CPU/GPU speedup values) or the back of the queue (higher CPU/GPU speedup values). This approach is described in greater detail in Section III-B. This scheduling approach improves load balancing within each computing node, as the computing devices are able to consume tasks at different rates, and allows the runtime system to adequately take advantage of the heterogeneous environment.

B. GPU-enabled Feature Extraction Functions

The feature computations can be divided into two categories, as presented in Table I: (i) features for the entire image (“Per Image”); and, (ii) features that are calculated for each object, such as a nucleus, (“Per Nucleus”). These two categories exhibit different computation patterns and thus different opportunities for GPU acceleration.

The feature extraction operations performed for an entire image are regular and usually performed on each pixel of the image. With a high resolution image, the GPU

implementation of these operations can fully utilize the processing capacity of a GPU. We have leveraged OpenCV [5] GPU-based operations to implement the functions for the Pixel Intensity, Gradient Magnitude, and Edge features. But we have implemented the Haralick feature computation using CUDA directly. In our implementation, pixels are equally distributed among GPU threads, which independently build co-occurrence matrices for their respective partitions of the image. Subsequently, a reduction is performed to aggregate the co-occurrence matrices from each thread into a global matrix. The size of the co-occurrence matrix can impact performance, if it is too big for the fast shared memory of the GPU. Fortunately, the size of the matrix has a minimal impact on classification accuracy [6]. Thus, we have limited it to be 8×8 pixels.

Unlike the “Per Image” features, the “Per Nucleus” feature computations exhibit irregular computation patterns. The execution time of a “Per Nucleus” operation varies significantly depending on the input data. Instead of following the parallelization strategy used for the “Per Image” operations, we restructured the “Per Nucleus” computations into operations on a set of minimum bounding boxes, each of which contains a nucleus. The set of bounding boxes is determined in the segmentation step. Briefly, the segmentation operation identifies candidate regions of nuclei by examining intensity and color variations across image pixels. Once the regions have been identified, it carries out computations to determine the boundary of each candidate nucleus. The nucleus boundaries are stored as image masks, from which the polygon representations of the boundaries and bounding boxes are generated.

By restructuring the computations in this way, we can avoid unnecessary computation in areas that are not of interest (i.e., that do not have objects) and create a more compact representation of the data. Since each bounding box can be processed independently, a large set of fine grained tasks is created in this strategy. The feature computations are performed in 2 steps. First, one GPU block of threads are assigned to each bounding box. These threads collectively compute the intermediate results, i.e., the histograms and co-occurrence matrices of the corresponding nuclei. This approach takes advantage of a GPU's ability to dynamically assign thread blocks to GPU multiprocessors and, in this way, reduces load imbalance that may arise due to differences in the sizes and, consequently, computation costs of different nuclei. In the second step, the nuclear feature values are calculated from the intermediate results, which are now fixed sized per nucleus. One GPU thread is executed per nucleus in this step.

III. Execution on a Parallel Machine with CPU-GPU Nodes

The parallelization on a distributed, hybrid CPU-GPU system of the feature extraction step is done at two levels: (i) the inter-node level parallelization that scales the application execution to multiple nodes in the system; and (ii) the intra-node level parallelization that employs a scheduling strategy for efficient coordinated use of CPUs and GPUs available on each node.

A. Parallelization across Computing Nodes

An image to be processed is partitioned into tiles, and the tiles are distributed across the computation nodes. Each tile is padded with d pixels wide borders containing replicated

pixels from neighboring tiles, thus avoiding communication during concurrent processing. Here d is the maximum size of the co-occurrence matrix used for the Haralick features. The calculation of different “Per Image” features is performed based on the tile’s position in the original image and on the feature computed. The computation of the Haralick features [2], for instance, is based on a co-occurrence matrix built for each image, and the co-occurrence matrix calculation counts elements that co-occur within x pixels of distance. The value of x is limited by the padding size d .

Computation of a “Per Image” feature requires that partial results calculated for each tile within an image be reduced into a global result, which is then used to compute the feature. The strategy we employ for merging partial results from image tiles is made up of two phases. First, partial results computed by different devices (CPUs and GPUs) are reduced within the node during the execution of tasks. To efficiently perform this reduction, we make use of the atomic wait-free [7] instructions (e.g., *sync add and fetch(address, value)* for addition), which make it possible to reduce the partial results in a node without needing critical sections provided by traditional software based synchronization primitives — as is the case with mutexes and semaphores. In the second phase, the partial results computed by each node are merged using a collective reduce operation, and the features are finally calculated for the image. The computation of a feature from the global result is inexpensive compared to the computation of the partial and global results, and does not affect the application’s scalability.

B. Coordinated CPU-GPU Execution on a Node

Coordinated use of CPUs and GPUs on a computation node can provide high computation bandwidth, but this can only be achieved with appropriate mapping of processes to CPUs and GPUs. In this section we present a scheduling strategy and a set of optimizations to utilize multiple CPUs and GPUs in tandem and reduce data management overheads.

Each GPU on a node is assigned a control thread running on a CPU core, and the remaining available CPU cores are each assigned a task thread. Each Keeneland node used in our experiments has two six-core CPUs and three GPUs, all of which are connected through two I/O hubs as shown in Figure 3 [8]. The placement of the GPU control threads is done such that the thread responsible for managing GPU 1 is bound to CPU 1, while the threads that manage GPUs 2 and 3 are executed on CPU 2 to reduce control information exchange overheads.

We observed during the experimental evaluation that the concurrent use of multiple GPUs exposed significant runtime overheads in the CUDA API. The cost of using memory management operations (*cudaFree* and *cudaMalloc*) increased; so did the cost of invoking the feature computation functions (implemented as CUDA kernels). We also noticed that the cost of allocating memory by the application tasks for temporary purposes, which was inexpensive when using one GPU, increased rapidly as more GPUs were used. To address this problem, we reduced the number of CUDA memory allocation operations by reusing temporary memory allocations across tasks. Memory reuse is made possible by the introduction of a module to our runtime system to allow the programmer to allocate, query, and release data variables associated with each GPU. The increase in the cost of

kernel invocations due to use of multiple GPUs, unfortunately, could not be avoided, and, as discussed in the results section, is currently the main factor that impacts performance when three GPUs are used on a node in the testbed system.

1) Task Assignment—The feature extraction step creates many tasks to compute features for image tiles and nuclei. The tasks should be carefully scheduled across CPUs and GPUs to achieve optimal performance. Several recent efforts on task scheduling in heterogeneous environments have targeted machines equipped with CPUs and GPU accelerators [4], [9], [10]. These efforts address the problem of partitioning and mapping tasks between CPUs and GPUs for applications in which operations (or tasks) achieve consistent and data-independent speedups when executed on a GPU vs on a CPU. Given this characteristic of their target applications, the previous efforts differ mainly in scheduling strategies they employ: off-line, on-line, and automated scheduling of tasks. However, in applications that apply more than one type of operations on input data, it is likely that there will be variations across the operations in the amount of speedup attained by using a GPU. This is because GPU-vs-CPU speedup values significantly vary according to the computation patterns of data processing operations. Another source of performance variability is input data characteristics. The same operation may achieve different speedup values with different input datasets. In our image processing application, for instance, the GPU performance per image tile of a “Per Nucleus” operation is sensitive to the number of nuclei in a tile as well as the sizes of the nuclei.

A scheduling strategy should take into account performance variability in order to achieve more effective assignment of tasks to computing devices. We propose a strategy, referred to here as *PRIORITY*, which 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. The *PRIORITY* scheduler uses a queue of task tuples, *(operation, data element)*, sorted based on the relative speedup expected for each tuple. As more *(operation, data element)* tuples are created (for instance, by the segmentation phase or as a result of a query into a results database), the new tuples are inserted into the queue such that the queue remains sorted.

The performance gain (speedup) estimates are computed using a subset of the dataset in a profiling phase prior to the proper execution, where the performance of the feature computation operations is analyzed. For the “Per Nucleus” operations, the number of nuclei in a region impacts the relative speedup; a larger number of nuclei lead to better concurrent/multi-threaded execution opportunity in a GPU. Therefore, the profiling phase is conducted by varying the number of nuclei to be processed in a region in order to measure the speedup as a function of the number of nuclei. For the “Per Image” operations, the number of nuclei has less impact on the performance, and the type of operation performed is the most important attribute for speedup estimation. The information generated in the profiling phase is then consulted during the actual execution to determine the expected speedup for each respective *(operation, data element)* tuple.

During the proper 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. Figure 2 shows an overview of the scheduling and runtime system.

The PRIORITY scheduler is based on maintaining the correct relative order of speedups 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 remain unchanged. We should note that the addition of execution times in the scheduler could improve the performance of the scheduler. However, one would have to estimate both speedup and execution times, instead of only the speedup as required by the strategy proposed in this paper. We have used speedup estimates to reduce scheduling overheads. In our experience with the motivating application, the computation of execution time estimates is more complex than that of speedup estimates. Execution times for the “Per Nucleus” operations will depend on the characteristics of individual nucleus such as shape and size. Thus, estimating the execution times for a range of objects and operations would require more computation time to extract features from images, regions, and segmented objects (e.g., nuclei) that affect the execution times. The estimation of speedups, on the other hand, is less dependent on the characteristics of individual entities and based on the number of nuclei in a region and can be done in a profiling step as described above.

2) Grouping Data Elements and Tasks—Each tuple in the PRIORITY scheduler consists of an application function and a data element on which the function is applied. The minimum granularity of a function and that of a data element is determined by the application. In our case, atomic functions are the feature computation operations, and a data element is either an image tile or an object. Multiple functions and data elements, however, can be grouped together to reduce scheduling overheads and improve performance.

Consider the Haralick operation performed on each nucleus segmented in an image. If a tuple were created for each (*Haralick operation, nucleus*) pair, the priority queue might easily have millions of tuples. This would increase the cost of scheduling since the queue needs to be sorted as more objects (and tuples) are inserted into the queue. Moreover, scheduling and mapping individual objects to CPU cores or GPUs would likely not achieve good performance because of the relatively high cost of GPU kernel invocation and data copy. A grouping of data elements to form *chunks* of data elements can reduce these overheads. It would be beneficial to group data elements together based on the speedup estimates of individual tuples. That is, data elements in tuples that have similar speedup estimates and contain the same function, could be grouped together into chunks. This approach would reduce the CUDA kernel invocation overheads since the processing kernel is called only once for the chunk of data elements. However, the scheduling overheads might still be high because the groupings will need to be updated as new tuples are added to the priority queue. To reduce grouping updates, a grouping strategy could take advantage of the spatial or temporal proximity of data elements, e.g., nuclei in the same tile. We have employed this strategy in our implementation. Each tuple consists of (feature computation operation, all nuclei in an image tile), and the speedup estimate of the tuple is computed based in the number of nuclei in the corresponding image tile.

Another opportunity for reducing the scheduling overheads and improving computational performance is to cluster different functions that are applied to the same data element (or the same chunk of data elements) into a task group. This approach follows the GPU programming practice of maximizing the computation to data transfer ratio in order to amortize the data transfer cost. However, the grouping of functions should be done carefully when there are speedup variabilities among functions. If functions were grouped together randomly, it would be likely that the same task group would have functions that achieved good GPU speedups as well as those that did not. As a result, the task groups would not fully expose their respective members' performance differentials, on which our runtime system relies to map tasks, or task groups, to CPUs and GPUs.

We have implemented a strategy, called Performance Aware Task Grouping (PATG), which groups functions based on GPU speedup values. The operations are clustered in a step prior to execution, in which operations with high and low speedups are identified. In our current implementation, the clustering creates two types of task groups for the “Per Nucleus” operations; one group containing the functions with good GPU speedup values and the other group containing the remaining functions. The Per Image operations are affected less by the input and are always in the same group. Once a grouping is computed, the runtime system only needs to select the appropriate task group based on the number of nuclei in an image tile to be processed and assign the selected group of operations and the image tile to the appropriate computing device. The cost of this selection is very small and contributes very little to the overall execution time. Tuples in the sorted task queue take the form of *(task group, data element)*. When a tuple is assigned to a GPU, the data element is copied to the GPU memory and the functions in the task group are invoked one by one. As a result, the benefits of avoiding unnecessary data transfers are attained, with the restriction that only those tasks with similar performance are grouped and executed together. In the current implementation, we employed a simple grouping rule in which functions with speedups higher than 10 form one task group and those with speedups lower than 10 make up another task group.

Our approach does not merge CUDA kernels. If functions associated with the same input are grouped to reduce data transfer overheads, the functions in the task group are invoked in sequence one by one. Therefore, although the data transfer cost is reduced, the kernel invocation remains the same as before the grouping.

3) Load Balancing—A computation node may have multiple multi-core CPUs and multiple GPUs. Computational load needs to be distributed across these devices to exploit aggregate computation capacity. Addressing balanced distribution of load among computing nodes is a critical issue in the design of scalable parallel algorithms [11]. Moreover, the effects of computational load imbalance are accentuated on nodes with CPUs and GPUs because of the differences in performance characteristics of these devices.

The demand-driven assignment of task tuples to computing devices in the PRIORITY scheduling strategy aims to reduce load imbalance. However, towards the end of execution of a group of tuples, tuples with efficient performance on the GPU may be assigned to the CPU, which could potentially keep the CPU processing these tuples after the GPU is done

with the remaining ones. This is because the basic task assignment algorithm tries to keep all devices in use, even though an available task may not achieve good enough performance on a given device.

In order to alleviate this problem, a new step is added to the scheduler in order to verify whether a tuple chosen by the scheduling algorithm could result in a suboptimal task mapping. The CPU requesting the tuple is kept idle when such a situation is identified. Assuming a single GPU node, a potential approach to detecting suboptimal task assignment is to look at the estimated execution time of the tuple on the CPU. If the total estimated execution time on GPU of the remaining tuples in the queue is smaller than this value, the tuple is not assigned to the CPU. The reason behind this strategy is that even if all of the other tuples are assigned to the GPU, the CPU will still be processing this tuple when the GPU is done. This approach requires that the estimated execution times of tuples be computed, or be provided by the end user. We have employed a simple heuristic to approximate this approach. Let N be the number of GPUs on the node and the estimated speedup using GPU of a tuple is S . The heuristic compares $N \times S$ to the number of tuples remaining in the task queue. If the number of tuples is greater than $N \times S$, the tuple is assigned to the CPU, otherwise the scheduling of the tuple is delayed until a GPU becomes available.

IV. Experimental Results

We performed an experimental evaluation of the scheduling strategy and optimizations presented in this paper using a distributed memory, multi-CPU, multi-GPU system called, Keeneland [8]. Keeneland is a National Science Foundation Track2D Experimental System and has 120 nodes in the current configuration. Each of the nodes is equipped with a dual socket Intel X5660 2.8 Ghz Westmere processor, 3 GPUs NVIDIA Tesla M2070 (Fermi), and 24GB of DDR3 RAM (See Figure 3). The nodes are connected to each other through QDR Infiniband. The set of images used in the evaluation was obtained from the In Silico Brain Tumor Research Center [12]. Each image was partitioned into tiles of $4K \times 4K$ pixels. The codes were compiled using “gcc 4.1.2”, “-O3” optimization flag, OpenCV 2.3.0, and NVIDIA CUDA SDK 4.0. The experiments were repeated 3 times, and, unless stated, the standard deviation was not observed to be higher than 3%. For comparison purposes, we implemented a First Come First Served (FCFS) task scheduling strategy that does not take into account performance variability.

A. Impact of Input Data Characteristics on GPU Performance

In the first set of experiments, we examine the effect of input data characteristics on the performance of the feature extraction operations. We measure the speedup achieved by the GPU implementation over execution on a single CPU core with different input data. We used four different tiles in these experiments, selected according to the number of nuclei and tissue coverage, shown in Figure 4(a): Tile 1, Tile 2, Tile 3, and Tile 4 have 262, 2546, 9507, and 12324 nuclei with about 1/8, 1/3, 3/4, and 1/1 of tissue coverage, respectively.

The speedup values obtained for the feature extraction operations using different tile types are presented in Figure 4. The results show that the performance of the “Per Nucleus”

feature computations is dependent on the characteristics of the input data, because tiles with small tissue coverage and fewer nuclei have less parallelization capacity on a GPU. Despite this variation, the speedups for the Intensity and Gradient based features have smaller speedups on the GPU than those achieved by the Haralick feature computations. The main reason is the fact that the Haralick feature function has a longer execution time, which allows the GPU to be more efficiently employed as the cost to invoke the corresponding CUDA kernel is adequately amortized.

The speedup values obtained for the “Per Image” Haralick feature computations on the GPU are the best among all features, while the Sobel feature computation also achieves considerable gains. The Intensity and Gradient feature computations have smaller speedups than those in the “Per Nucleus” case. The discrepancy occurs because the nucleus based calculation has larger total work as it requires a histogram to be computed for each nucleus, and the features have to be calculated for all nuclei. In the “Per Image” case, however, a single histogram is created for the entire image, and subsequently the features are calculated only once. The “Per Nucleus” Intensity and Gradient feature calculations present more opportunities for parallelization on the GPU, resulting in higher speedups.

In summary, our experiments highlight two important performance characteristics for the GPU implementations: (i) the performance of the “Per Nucleus” computations is strongly data dependent, resulting in significant variations in efficiency across different input data, and (ii) there is also variation in performance across different types of feature computation functions in both “Per Nucleus” and “Per Image” feature computations.

B. Evaluating Multi-GPU Scalability

These experiments examine performance and overheads when multiple GPUs are utilized on a computation node. The Haralick feature computations were carried out in four angles using 200 tiles in the experiments. Three runtime strategies were investigated: Naïve, NUMA aware (NA), in which the placement of threads managing GPUs is performed according to what is described in Section III-B, and NUMA aware + avoiding CUDA memory management contention (NAMM), which implements NA placement and the reuse of temporary memory allocation among tasks. The term memory management contention here refers to the time spent in memory allocation and free operations, which are provided by the CUDA API. The kernel time, on the other hand, refers to the amount of time between when a kernel is invoked and when the control returns to the CPU. This time includes the proper computation time and the operations performed by the CUDA driver to startup the computation in the GPU and to return the control to the CPU. Therefore, the increase in kernel time is due to contention in the driver as well as communication overhead, which increases with the number of GPUs used. To address the memory management contention, we implemented modules to allocate, query, and release data variables associated with each GPU and reuse temporary memory allocations across tasks assigned to the same GPU. As is seen from Figure 5(a), all three strategies achieve similar performance for the single GPU execution case, with a slightly higher speedup for the NAMM strategy because of its smaller memory management costs.

In the two GPU case, however, the NUMA aware strategies (NA and NAMM), attain better performance due to their smaller data transfer overheads. Moreover, NAMM achieves almost the double of the performance gains of NA over the Naïve solution. In order to better understand the source of the performance gains, we show a breakdown of the execution times in three categories: Data transfer; Kernel time, which is the cost of invoking the CUDA *kernel* and its proper execution; and CUDA MM contention, which is the time spent in CUDA memory allocation and free operations. The breakdown, presented in Figure 5(b), shows that the NUMA aware policies have smaller data transfer times and that the cost of memory management per task increases when two GPUs are used. This leads NAMM to achieve better performance than NA.

A similar behavior in performance is observed when three GPUs are used. In this case, however, the cost of CUDA functions for memory management doubles compared to the 2-GPU case and there is a performance degradation even with NAMM. As is shown in Figure 5(b), this is the result of an increase in the data transfer times, which are smaller for the NUMA aware strategies, and in the kernel invocation times. The increases in the kernel invocation times contradicts the expected behavior that kernel invocation time remain constant regardless of the number of GPUs used. The kernel invocation time is smaller for the NUMA aware strategies. This can be attributed to the fact that the control threads have to traverse only a single I/O Hub or shared link in the NUMA aware strategies.

In order to validate these results, we also performed the same experiments using a very simple CUDA kernel that increments an array. Similar performance behaviors were observed with this simple kernel as well. Since the NVIDIA driver and programming toolkit codes are not open source, we could not further investigate what we suspect to be the main contention points: the shared Quick Path Interconnect (QPI) link among the CPUs and GPUs, and synchronization points in the CUDA driver as observed for the memory management case. Our experiments showed the NAMM strategy improves upon the performance of the Naïve strategy by 1.28 times for the 3-GPU configuration.

C. CPU-GPU Execution with Data Dependent Performance

This set of experiments evaluates the coordinated use of CPUs and GPUs when the performance of data processing functions varies based on the characteristics of input data as presented in Section IV-A. We focus in these experiments on the computation of the “Per Nucleus” Haralick features (see Table I), since the performance of this computation on the GPU is greatly affected by the number of nuclei in image tiles. In this way, we can isolate the effects of data dependent performance, and evaluate the proposed PRIORITY scheduling strategy in a controlled way.

The first set of results is presented in Figure 6. In the experiments 200 (4K×4K) input tiles were used, which fall into two classes: small and large tissue coverage (Tiles 1 and 2 in Figure 4(a), respectively) – the larger the tissue coverage is, the larger the number of nuclei is. These two categories of tiles were chosen to assess the performance of the scheduling techniques under controlled conditions with different mixes of tiles with small and large tissue coverage. Figure 6(a) presents the speedup values obtained by different scheduling configurations, when compared to the single CPU-core execution. As expected, the speedup

increases as the percentage of tiles with large tissue coverage is increased, since these tiles achieve better speedups when processed by the GPU. The “1 GPU” in the graphs is the case in which all processing is done on one GPU without using any CPU cores. The FCFS strategy improved the performance of the GPU-only version in most of the configurations, achieving an average speedup of 1.51 over GPU-only. FCFS, however, does not take into account performance variation arising from data and task characteristics. As is seen from the figure, the PRIORITY scheduling policy can achieve better efficiency, reaching an average speedup improvement of 1.87 in comparison to the GPU-only execution.

Although it achieves better performance, the PRIORITY scheduling is also impacted by load imbalance, which is the cause of the peaks and dips in the speedup curves. Figure 6(b) presents load imbalance as a percent of the execution time. FCFS has higher load imbalance, especially when the set of tiles has a small number of tiles with high tissue coverage. The load imbalance in PRIORITY follows the opposite pattern of its speedup curves, with an inversion of dips and peaks between the two curves.

In order to address the load imbalance issue, we employed the load balancing strategy presented in Section III-B3. The results are shown in Figure 6(c). The load balancing (LB) mechanism improved the performance of both scheduling policies across all experiments, clearly eliminating the dips in the PRIORITY speedup curves. Moreover, FCFS and PRIORITY with LB achieved additional speedups on top of the speedup values without LB, up to 1.22 and 1.35, respectively.

The next set of experiments looks at the performance of the scheduling techniques using sets of Tiles 1 and 4 in Figure 4(a). In this configuration, an extra stress is imposed on the scheduler, because the differences in GPU-CPU speedups with these tiles are higher. The load imbalance is also likely to increase as the CPU execution time for Tile 4 is much longer than for Tile 2. The results, presented in Figure 7, show that neither FCFS nor PRIORITY could sustain better performance than that achieved in the GPU-only case. This is mainly because the load imbalance is much higher, as shown in Figure 7(b). The performances of both policies are shown in Figure 7(c), when the load balancing mechanism is employed. PRIORITY with LB was able to consistently improve upon the GPU-only case, having average and maximum speedups of 1.77 and 3.53. FCFS with LB, on the other hand, slightly improved performance for a few cases only.

Lastly, in Figure 8, we present the performance of the scheduling strategies using sets of tiles consisting of Tiles 1 and 4 on a multi-CPU, multi-GPU configuration. The PRIORITY scheduler with LB using 9 CPU cores and 3 GPUs achieved an average speedup of 1.34 over the configuration with 3 GPUs only. It attained a maximum speedup of 1.78 over the 3-GPU configuration for the tile set with 20% of the tiles having large tissue coverage. FCFS, however, only showed performance gains over the 3-GPU configuration for the tile set with 0% of the tiles having large tissue coverage.

D. CPU-GPU Execution with Operation Dependent Performance

This section evaluates the cooperative use of CPUs and GPUs under operation-dependent performance variation only. For these experiments, all the “Per Nucleus” and “Per Image”

features presented in Table I are computed for 200 tiles. During each experiment, a single tile type is used to avoid the scheduler from employing optimizations for data-dependent performance variations.

Figure 9 presents the performance achieved by different groupings of tasks: *group all*, which puts all tasks on each tile into a single group, and performance-aware task grouping (*PATG*). As is shown in the figure, for the group all strategy, FCFS presents almost the same performance as PRIORITY, and, since all tasks have the same speedups, no better assignment can be done by PRIORITY. Moreover, the comparison of FCFS to the 3-GPU only configuration shows that the use of CPUs improves the performance for all types of tiles. An average speedup of 1.17 over the 3-GPU only configuration is observed.

The PRIORITY with PATG, on the other hand, performs much better than FCFS. Its performance is more than double the performance of the 3-GPU only configuration for the first two types of tiles, and provides strong gains for the other tile types. The speedups achieved by PRIORITY with PATG compared to the 3-GPU only configuration, are 1.82 on average (and a maximum speedup of 2.21) across all the tile types, with only 9 CPU-cores used in addition to the 3 GPUs.

E. Exploiting Data and Operation Dependent Performance Optimizations

In this section, we analyze the performance of the scheduling strategies when the optimizations for data- and operation-dependent performance variations are combined. As in the previous set of experiments, all features are computed, and we use two types of tiles: Tile 1 and Tile 4. The percentage of different tile types in the tile sets is varied to create different input data configurations.

The experimental results are presented in Figure 10. The cooperative use of CPUs and GPUs with FCFS or PRIORITY and group all strategy does not result in considerable gains over the 3-GPU only configuration. Because data performance variation exists, the PRIORITY scheduling with group all performed with the same performance as or better than the performance of FCFS. The performance of PRIORITY with PATG is much better than the other combinations, achieving an average, maximum, and minimum speedups of 1.72, 1.91, and 1.47, respectively, on top of the configuration with 3 GPUs only.

Figure 11 presents the performance of the different scheduling strategies and optimizations when entire whole slide images are used as input. Each of the images was digitized at 20 $\times$ magnification. As is seen from the figure, the performance of PRIORITY with PATG is superior to all other configurations. Moreover, PRIORITY with PATG has a speedup of up to 1.66 compared to the 3-GPU only configuration. The scheduling approaches that do not use PATG (FCFS + LB - 3 GPUs + 9 CPU-cores - group all and PRIORITY + LB - 3 GPUs + 9 CPU-cores - group all) achieve a maximum speedup of 1.05 on top of the 3-GPU only configuration.

F. Multi-node Scalability

This set of experiments measures the performance of our parallelization strategy in a distributed multi-node scenario. For this analysis, we performed a weak scaling experiment,

where the size of the input data and number of nodes used were increased proportionally. The input dataset for the baseline single node execution contained an image with 100 (4K×4K) tiles, and the subsequent executions in multiple nodes scaled this baseline dataset. This experiment is intended to exercise the worst case scalability scenario, where an arbitrarily large image is processed, and the reduction phase for features computation is performed among all computation nodes in the system.

Figure 12 presents the throughput (tiles processed per second) as the configuration of the application and the number of nodes are varied. The “Linear” curve refers to the linear throughput improvement based on the best single node execution. As is seen from the results, all versions of the feature extraction implementation achieve good speedups, with a near linear throughput increase as the number of nodes is increased. The results show that the gains achieved by our single node scheduling techniques are sustained in the distributed environment and that the optimizations we have proposed (PRIORITY + LB - 3 GPUs + 9 CPU-cores - PATG) lead to additional speedups of 1.88 and 1.80, respectively, in comparison to FCFS and the 3-GPU only configuration. Moreover, PRIORITY, FCFS, and the 3-GPU only schemes achieve 10344, 5727, and 5491 times higher throughput on 100 processors compared to the single CPU core execution, respectively.

V. Related work

The use of accelerators for general purpose computing has been rapidly increasing in the past few years, as a consequence of the massive computing power available at reduced cost and power consumption, as well as of advances in programmability realized by the introduction of CUDA [13] and OpenCL [14]. The efficient utilization of the computing capacity offered by these accelerators, however, is a challenging problem. Moreover, coupling of accelerators with multi-core CPUs has created a novel heterogeneous environment with deep memory hierarchies, including non-uniform memory access (NUMA) characteristics, and with multiple types of computing devices. These changes have motivated several research projects targeting different aspects of GPU-equipped high performance computing systems in various scenarios. As a result, a number of higher level languages and runtime systems [4], [9], [10], [15], [16], [17], [18], specialized libraries [5], [19], and compiler and auto-tuning techniques for GPUs [20], [21], [22] have been introduced.

Mars [9] and Merge [4] are examples of the efforts to evaluate the combined use of GPUs and CPUs for efficient execution of internal tasks of an application. Mars implements GPU support for a subset of the MapReduce API. The authors performed a preliminary evaluation of potential gains from using CPUs and GPUs together. In that analysis, Map and Reduce tasks were divided among CPU and GPU using a static offline task partition strategy, based on the speedups observed between different device types for each particular application. The Merge framework, on the other hand, proposes a parallel language, also based on MapReduce, and introduces a dynamic methodology for distributing work across heterogeneous computing cores during execution. Qilin [10] has extended the work proposed in Mars with an automated methodology for mapping computations into heterogeneous environments. The strategy adopted in Qilin consists of an adaptive

mapping step that is performed based on data collected during an offline training phase. This information is further used to estimate relative performance (speedup) between the CPU and the GPU for each subsequent execution of the application. Task partition is performed based on that estimation. This methodology improves task partitioning as it captures variations in speedups between CPUs and GPUs that may arise as the input data or application parameters varies.

More recently, Ravi et al. [23] have proposed and implemented strategies for dynamic partitioning of tasks among CPUs and GPUs, including multi-node task partitioning. The strategies were evaluated in the context of applications based on generalized reductions. Moreover, they have shown that their dynamic assignment, which employs a FCFS task partitioning approach very similar to our baseline scheduler, substantially outperforms static scheduling policies. The work by Teodoro et. al. [18] proposes dynamic runtime scheduling techniques to partition tasks. It evaluates the use of a distributed demand-driven task queue to map tasks across parallel machines, where data transfers and communication are performed concurrently in order to reduce communication overheads.

In contrast to previous research, we argue that the relative performance of GPUs and CPUs on internal tasks in many applications may vary based on both input data characteristics and the type of the computation performed. Our experimental evaluation shows that exploiting both dimensions is essential to achieve maximum performance in CPU-GPU equipped environments. Therefore, instead of partitioning tasks based on the overall performance of the application as Qilin, Merge, and Mars do, we analyze performance variability across the internal tasks of the application in order to classify them according to potential suitability for each available computing device. This methodology creates a broader scheduling approach, since we are not only interested in partitioning tasks among devices with minimal load imbalance, but we also perform a finer grain management of the computing devices and delay the decision of task assignment until runtime. This strategy may introduce overheads to application execution, but it provides us with the ability to improve cooperative use of CPUs and GPUs by using each of the devices for tasks which they can execute most efficiently. Our results have also shown that the load balancing mechanism is crucial to attain high performance in presence of heterogeneous irregular computations. The previous efforts addressed this problem with regular tasks, for which a few misassignments will have a small effect on the execution, and dynamic task scheduling without any other mechanism is able to achieve good performance. In our target application, however, this problem is more complicated, since a small number of misassignments are enough to impact the overall performance. Therefore, we have developed a more aggressive load balancing mechanism, which may even leave processors idle to improve overall performance.

Task grouping techniques have been long employed in high performance computing, for instance, to reduce data transfer, and scheduling overheads, e.g. in parallel loops [24], [25]. In this work, we propose a Performance Aware Task grouping (PATG) strategy that leads to improvement in performance on CPU-GPU equipped machines, because it provides the scheduling mechanism with a broader decision space.

Medical image processing is an active area that is continuously gaining importance in clinical treatment and, consequently, attracting attention from the scientific and industrial communities. The types of analyses involved in this discipline are typically very computationally demanding as a result of being applied to high resolution images. This computing power requirement is also aggravated by the increasing volumes of data due to the evolution of data acquisition technologies [26]. Various image processing algorithms are good candidates for parallel execution. This class of operations is also being ported to GPUs for efficient execution [26], [27]. In addition, there is a growing set of libraries, such as the NVIDIA Performance Primitives (NPP) [28] and OpenCV [5], which encapsulate the GPU-implementations of a number of image processing algorithms through high level and flexible APIs.

VI. Conclusions

We have investigated the use of large scale CPU-GPU systems to accelerate biomedical image analyses. Several optimizations targeting NUMA effects and cooperative execution in multi-CPU/-GPU environments have been proposed and evaluated. Our work has showed that (1) variations in performance (speed-up values) achieved by internal tasks of application when executed in the GPU may arise due to both input data characteristics and the type of operations performed; and (2) taking advantage of these variations in task scheduling is essential to maximize performance on CPU-GPU equipped machines. We also showed that the proposed performance variation aware scheduling strategy can further be improved by carefully grouping tasks and data elements. Although the task grouping strategy seems to be counter intuitive, it provides the scheduler with a broader decision space and reduces scheduling overheads. Due to the irregular nature of the computations in our application, load balancing emerged as a problem, which was addressed with addition of an extra load balancing mechanism to the scheduler.

Acknowledgments

This work was supported in part by Contracts HHSN261200800001E and N01-CO-12400 from the National Cancer Institute, R24HL085343 from the National Heart Lung and Blood Institute, by Grants R01LM011119-01 and R01LM009239 from the National Library of Medicine, RC4MD005964 from National Institutes of Health, NSF CNS-0615155, CNS-0403342, CCF-0342615, CNS-0406386, and ANI-0330612, NIH NIBIB BISTI P20EB000591, and PHS Grant 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. Cooper L, Kong J, Gutman D, Wang F, Cholleti S, Pan T, Widener P, Sharma A, Mikkelsen T, Flanders A, Rubin D, Van Meir E, Kurc T, Moreno C, Brat D, Saltz J. An Integrative Approach for In Silico Glioma Research. *IEEE Transactions on Biomedical Engineering*. 2010; 57
2. Haralick R. Statistical and structural approaches to texture. *Proceedings of the IEEE*. May. 1979 67 (5)
3. Millstein T. Practical predicate dispatch. *SIGPLAN Not*. Oct. 2004 39 :345–364.
4. Linderman MD, Collins JD, Wang H, Meng TH. Merge: a programming model for heterogeneous multi-core systems. *SIGPLAN Not*. 2008; 43 (3) :287–296.
5. Bradski G. The OpenCV Library. *Dr. Dobb's Journal of Software Tools*. 2000

6. Ruiz, A; Sertel, O; Ujaldon, M; Catalyurek, U; Saltz, J; Gurcan, M. Pathological image analysis using the GPU: Stroma classification for neuroblastoma. Proc. of IEEE Int. Conf. on Bioinformatics and Biomedicine; 2007;
7. Herlihy M. Wait-free synchronization. ACM Trans. Program. Lang. Syst. Jan. 1991 13 :124–149.
8. 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
9. He B, Fang W, Luo Q, Govindaraju NK, Wang T. Mars: A MapReduce Framework on Graphics Processors. Parallel Architectures and Compilation Techniques. 2008
10. Luk, C-K; Hong, S; Kim, H. Qilin: Exploiting Parallelism on Heterogeneous Multiprocessors with Adaptive Mapping. 42nd International Symposium on Microarchitecture (MICRO); 2009;
11. Cybenko G. Dynamic load balancing for distributed memory multiprocessors. J. Parallel Distrib. Comput. Oct. 1989 7 :279–301.
12. et al JHS. Multi-scale, integrative study of brain tumor: In silico brain tumor research center. Annual Symposium of American Medical Informatics Association 2010 Summit on Translational Bioinformatics (AMIA-TBI 2010). Mar. 2010
13. NVIDIA. NVIDIA CUDA SDK. 2007. [Online]. Available: <http://nvidia.com/cuda>
14. Khronos OpenCL Working Group. The OpenCL Specification, version 1.0.29. Dec 8. 2008
15. Augonnet, C; Thibault, S; Namyst, R; Wacrenier, P-A. StarPU: A Unified Platform for Task Scheduling on Heterogeneous Multicore Architectures. Proc. of the 15th Int. Euro-Par Conference on Parallel Processing; 2009;
16. Diamos, GF; Yalamanchili, S. Harmony: an execution model and runtime for heterogeneous many core systems. Proc. of the 17th International Symposium on High Performance Distributed Computing, ser. HPDC '08; 2008;
17. 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.
18. 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;
19. Tomov S, Nath R, Ltaief H, Dongarra J. Dense linear algebra solvers for multicore with gpu accelerators. Proc. of IPDPS'10. 2010
20. Lee, S; Min, S-J; Eigenmann, R. OpenMP to GPGPU: a compiler framework for automatic translation and optimization. PPOPP '09: Proc. of the 14th ACM SIGPLAN Symp. on Principles and Practice of Parallel Programming; 2009;
21. Rudy, G; Khan, MM; Hall, M; Chen, C; Jacqueline, C. A programming language interface to describe transformations and code generation. Proceedings of the 23rd International Conference on Languages and Compilers for Parallel Computing, ser. LCPC'10; Springer-Verlag. 2011;
22. Han T, Abdelrahman T. hiCUDA: High-Level GPGPU Programming. IEEE Transactions on Parallel and Distributed Systems. Jan. 2011 22 (1) :78 –90.
23. 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
24. Reinders, J. Intel threading building blocks. 1st ed.. O'Reilly & Associates, Inc.; Sebastopol, CA, USA: 2007.
25. Bacon DF, Graham SL, Sharp OJ. Compiler transformations for high-performance computing. ACM Comput. Surv. Dec. 1994 26 :345–420.
26. Scholl I, Aach T, Deserno TM, Kuhlen T. Challenges of medical image processing. Comput. Sci. Feb. 2011 26 :5–13.
27. Fialka, O; Cadik, M. FFT and Convolution Performance in Image Filtering on GPU. Proceedings of the conference on Information Visualization.; Washington, DC, USA. 2006; IEEE Computer Society; 609–614.

28. NVIDIA. NVIDIA Performance Primitives(NPP). Feb 11, 2011. [Online]. Available: <http://developer.nvidia.com/npp>

Author Manuscript

Author Manuscript

Author Manuscript

Author Manuscript

Figure 1.
Basic image analysis pipeline for classification of nuclei in imaged tissue samples.

Figure 2.
Runtime system and task assignment overview.

Figure 3.
Keeneland nodes architecture overview.

(a) Tiles with different tissue coverage.

(b) Performance of classes of feature computations.

Figure 4.
Evaluation of GPU-based implementations

(a) Speedups according to number of GPUs and app. version.

(b) Breakdown of the execution times per task.

Figure 5.
Multi-GPU scalability.

(a) Different scheduling strategies.

(b) Load imbalance.

(c) Scheduling w/ load balancing.

Figure 6.
Cooperative CPU-GPU execution: using tiles 1 and 2.

(a) Scheduling strategies performance.

(b) Load imbalance.

(c) Scheduling w/ load balancing.

Figure 7.
Cooperative CPU-GPU execution: using tiles 1 and 4.

Figure 8.
Cooperative CPU-GPU execution: the multi-gpu/-cpu scenario.

Figure 9.
CPU-GPU scheduling w/ operation based performance variation.

Figure 10.
CPU-GPU sched. w/ task and data based performance variation.

Figure 11.
CPU-GPU scheduling performance for whole slides.

Figure 12.
Multi-node scalability evaluation.

Table I TYPES OF FEATURES COMPUTED PER IMAGE AND PER NUCLEUS.

Class	Core operations	Features
Per Nucleus and Per Image		
Pixel Statistics	Histogram calculation	Mean, Median, Min., Max., 25%, 50%, and 75% quartile
Gradient Statistics	Gradient and Histogram calculation	Mean, Median, Min., Max., 25%, 50%, and 75% quartile
Haralick	Normalization pixel values and Co-occurrence matrix	Inertia, Energy, Entropy, Homogeneity, Max. prob., Cluster shade, prominence
Edge	Canny	Canny area
	Sobel	Sobel area
Per Nucleus only		
Morphometry	Pixel counting	Area
	Dist. among points	Perimeter
	Area	Equivalent diameter
	Area and Perimeter	Compactness
	Fitting ellipse	Major/Minor axis length
	Fitting ellipse	Orientation, Eccentricity
	Bounding box	Aspect ratio
	Convex hull	Convex area
	Connected components	Euler number