Biomedical Image Analysis on a Cooperative Cluster of GPUs and Multicores*

Timothy D. R. Hartley[1], Umit Catalyurek[1], Antonio Ruiz[2],
Francisco Igual[3], Rafael Mayo[3], Manuel Ujaldon[2]
[1] Departments of Biomedical Informatics and Electrical and Computer Engineering,
The Ohio State University, Columbus, OH, USA.
{hartleyt,umit}@bmi.osu.edu
[2] Computer Architecture Department, University of Malaga, Malaga, Spain.
{aruiz,ujaldon}@ac.uma.es
[3] Department of Computer Engineering and Computer Science,
University Jaume I, Castellon, Spain.
{figual,mayo}@icc.uji.es

ABSTRACT

We are currently witnessing the emergence of two paradigms in parallel computing: streaming processing and multi-core CPUs. Represented by solid commercial products widely available in commodity PCs, GPUs and multi-core CPUs bring together an unprecedented combination of high performance at low cost. The scientific computing community needs to keep pace with application models and middleware which scale efficiently to hundreds of internal processing units. The purpose of the work we present here is twofold: first, a cooperative environment is designed so that both parallel models can coexist and complement one another. Second, beyond the parallelism of multiple internal cores, further parallelism is introduced when multiple CPU sockets, multiple GPUs, and multiple nodes are combined within a unique multi-processor platform which exceeds 10 TFLOPS when using 16 nodes. We illustrate our cooperative parallelization approach by implementing a large-scale, biomedical image analysis application which contains a number of assorted kernels including typical streaming operators, co-occurrence matrices, convolutions, and histograms. Experimental results are compared among different implementation strategies and almost linear speed-up is achieved when all coexisting methods in CPUs and GPUs are combined.

*This work was supported by the US National Science Foundation (CNS-0643969, CNS-0403342, CNS-0615155, CFF-0342615), NIH NIBIB BISTI (P20EB000591), US Department of Energy (ED-FC02-06ER25775), Ministry of Education of Spain (TIC2003-06623, PR-2007-0014), Junta de Andalucía of Spain (P06-TIC-02109), and Fundación Caixa-Castellón/Bancaixa and the Univ. Jaume I (P1-1B2007-32).

ICS'08, June 7–12, 2008, Island of Kos Aegean Sea, Greece.

Categories and Subject Descriptors

C.1 [**Processor Architectures**]: Multiple Data Stream Architectures (Multiprocessors)—*Parallel Processors* ; I.3 [**Computer Graphics**]: Hardware Architecture—*Graphics Processors* ; I.4 [**Image Processing and Computer Vision**]: Applications

General Terms

Algorithms, Languages, Performance

1. INTRODUCTION

Biomedical applications are becoming a major focus due to the large possible benefit to the public welfare as well as to the scientific community. In particular, imaging applications are emerging as a new opportunity for innovation at the meeting point between medicine and computer science. These applications are challenging for several reasons. From the practical concerns of fitting into a clinician's standard workflow to the performance-centric difficulties in using disruptive architectures for the maximum application throughput, biomedical image analysis applications provide many opportunities for novelty.

Computer technology has made a tremendous impact on medical imaging technology. The recent availability of whole slide digital scanners has made research on pathological image analysis more attractive, by enabling quantitative analysis tools to decrease the evaluation time pathologists spend for each slide. This also reduces the variation in decision-making processes among different pathologists or institutions and introduces reproducibility.

The analysis of pathology images is particularly challenging due to the large size of the data. Since uncompressed image sizes can be 30 gigabytes for one slice of tissue, typical datasets for case studies can easily stretch to terabyte scale. Further, the computation required to analyze these images can be extensive; analysis can often take hours on a single CPU. Several research studies on different cancer types have been conducted to develop computational methods within this field [7, 12, 14, 15, 19, 24, 26]. Most of these approaches only tested randomly selected image tiles, while

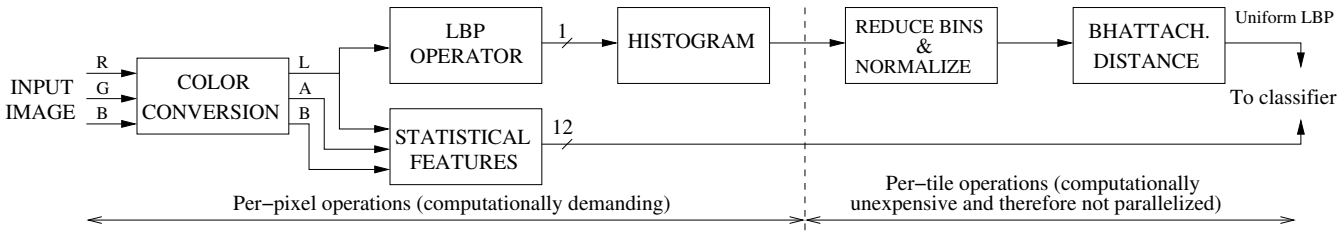

Figure 1: Flow chart for the stroma classification algorithm.

some [18] have recently extended their methods to processing whole-slide images, but without discussing the computational burden. One of the most recent results [3] involves a parallelization technique, but the focus of the study was not strictly one of execution time performance, and processing a relatively small image took almost half an hour on a 16-node configuration, which is still impractical for clinical application.

With respect to improving the processing time of scientific applications, the newest versions of programmable Graphics Processing Units (GPU) provide an ideal platform, since they allow extremely high floating point arithmetic throughput for applications which fit their architectural idiosyncrasies [16]. This fact has attracted many researchers and encouraged the use of GPUs in many fields [10] including data mining [11], image segmentation and clustering [13], numerical methods for finite element computations used in 3D interactive simulations [30], and nuclear, gas dispersion and heat shimmering simulations [31]. In an earlier work [22, 23], we have also leveraged GPU processing power and introduced techniques based on shaders and Cg to accelerate the feature extraction by a factor of 321x versus a Matlab version and 45x versus an equivalent C++ version running on a single CPU.

GPU manufacturers have responded to the wide acceptance and use of GPUs in general-purpose computing with the introduction of CUDA (Compute Unified Device Architecture) [5], a more general programming interface, and Tesla [28], a new computational node with multiple GPUs, reaching near supercomputer performance levels starting at $1500. Recent announcements from Nvidia (GeForce 9 Series) and ATI (FireStream) [9] have also responded to the scientific computing community's widespread call for double-precision, floating-point arithmetic which does not incur a large performance penalty compared to single-precision.

In this work, our goal is the efficient execution of large-scale biomedical image analysis applications on a cooperative cluster of GPUs and multi-core CPUs. The advent of multi-core CPUs means that more computation can occur on a single computer than ever before. Traditional SMPs are now becoming hybridized, such that multiple multi-core CPUs are resident in a single compute node. Furthermore, new architectures can support more than one GPU card per node. Hence, our target hardware architecture is a cluster of compute nodes with multiple multi-core CPUs and multiple GPUs, and this work presents methods to make full use of all of the computational power such a hardware system offers.

On the software side, our cooperative approach is enabled by software libraries and middleware which ease both the GPU programming and the parallelization computation at many granularities. Nvidia's CUDA provides easier access to the high computational performance available in modern era GPUs. Additionally, it provides capabilities beyond that of other programming methods with respect to applications which do not entirely fit into the more traditional graphics processing paradigm. DataCutter [2] is a powerful middleware tool for data-parallel application decomposition, transparent task replication, and task graph execution. Its use here allows us to leverage all of the parallelism inherent in the hardware architecture. By analyzing the application bottlenecks appropriately, we are able to hide much of the latency incurred by the hardware when analyzing very large data sets. Additionally, DataCutter enables the overlap of computation and communication, which are still fundamental issues in the GPU and multi-core era.

The rest of the paper is organized as follows. Section 2 presents the neuroblastoma image analysis application which provides the testbed for this new paradigm of cooperation between multi-core CPUs and GPUs. The specifications and properties of an example state-of-the-art, multi-socket, multi-core, multi-GPU cluster are presented in Section 3. Section 4 focuses on the specifics of the GPU programming with CUDA and that of the parallelization strategies. The experimental results are presented in section 5, and section 6 concludes.

2. PATHOLOGICAL IMAGE ANALYSIS

Neuroblastoma is a cancer of the sympathetic nervous system which mostly affects children. The prognosis of the disease is currently based on visual examination under a microscope of tissue slides by expert pathologists. These tissue slides are classified into different prognostic groups depending on the differentiation grade of the neuroblasts among other issues [25]. Manual examination by pathologists is an error-prone and very time consuming process which also may lead to inter- and intra-reader variations. Therefore, together with our collaborators, we are developing a computerized pathological image analysis system [3, 12, 22, 23, 24] to assist in the determination of the prognosis of the neuroblastoma by automatically characterizing stroma regions.

2.1 The algorithm

Figure 1 shows the flowchart of the image analysis algorithm for the classification of stromal development in neuroblastoma images [22, 23]. The image analysis occurs in four stages:

Phase 1: Color conversion. The RGB input image is converted into the LA*B* color space, which provides color

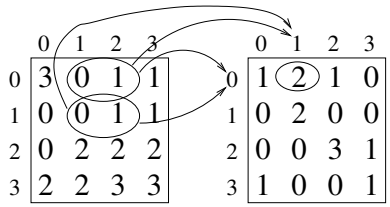

Figure 2: The computation of a co-occurrence matrix (right) from a 4x4 image (left) where pixel intensities are shown.

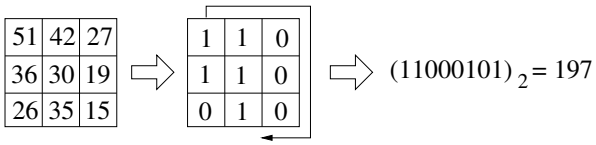

Figure 3: The LBP operator on a 3x3 grid.

perceptual uniformity and enables the use of Euclidean distance in feature calculation [17].

Phase 2: Statistical features. Four second order statistical features are extracted from each color channel: contrast, correlation, homogeneity and energy [29]. An intermediate data structure used during the calculation of those twelve features is the co-occurrence matrix [6], which represents how often a pixel with the intensity value i occurs in a specific spatial relationship to a pixel with the intensity value j (see Figure 2). The size of the co-occurrence matrix has a major impact on the workload, but only a marginal influence on the algorithm's classification accuracy [23]. Therefore we have selected a 4×4 co-occurrence matrix for our experiments, which yields the fastest execution times.

Phase 3: LBP operator. The local binary pattern feature (LBP) is extracted from the L (luminance) channel to become the thirteenth feature for texture analysis. Widely used in many applications such as facial expression recognition and content based image retrieval [27], LBP is a rotationally invariant operator defined within a 3×3 neighborhood of each pixel, where the eight neighbors are examined to see if their intensity is greater than that of center pixel p. The results form a binary number $b_1b_2b_3b_4b_5b_6b_7b_8$ where $b_i = 0$ if the intensity of the ith neighbor is less than or equal to p and 1 otherwise (see Figure 3).

Phase 4: Histogram. Since LBP feature values are in the range $0 \rightarrow 255$, a histogram is constructed with 256 bins for the entire image, with each bin accumulating the number of pixels which have that LBP feature value. These bins are then reduced into ten canonical classes and normalized between 0 and 1 to become the components of a ten-dimensional vector. The Bhattacharyya distance [21] is then measured between the LBP feature vector and a $(1, 1, 1, 1, 1, 1, 1, 1, 1, 1)$ vector to constitute the uniform LBP feature value used for the stroma classification. The subsequent classifier is a computationally inexpensive process, so we will not mention it further.

2.2 Major challenges

Input tissue samples are digitized at 40x magnification and stored in TIFF files using JPEG compression and the RGB color format. Each whole-slide image has a resolution of 110K×80K pixels in the worst case, which poses two major challenges:

- A single uncompressed image's size is well over the memory capacity of a single GPU. Additionally, providing balanced parallelization and effective use of cluster resources like disk I/O requires a smaller data granularity. In order to relieve memory usage and allow for an efficient parallelization of the computation across nodes, we decompose the image into 1K x 1K tiles.
- The algorithm takes several hours to run on a single CPU (see Table 5). This motivates us to study alternative platforms for its parallel execution. Our selection of multi-socket and multi-core CPUs combined with high-end GPUs will give us the opportunity for assessing scalability on both sides.

Tiling and parallelism are in fact tightly coupled, since the first strategy favors the latter. On a multi-processor system, each processor may independently perform the image analysis task on a subset of tiles and return a label indicating whether the particular image tile is stroma rich or stroma poor. Finally, labels are gathered on a central host and the classification map for the whole-slide image is constructed.

Apart from its implementation on a multi-processor system, our particular image analysis application is of great interest for evaluating the memory hierarchy and computational power of graphics architectures, because it meets a diverse number of features. For example, color conversion is a typical streaming operation with no data reuse. The LBP operator, on the other hand, exhibits a large degree of data reuse and locality, which typically favors more traditional, cache-based systems. Statistical features (through a co-occurrence matrix) and histogram construction both use extensive indirect array accesses such as those characterizing irregular computing and reduction operations often found in linear algebra. These two phases present undesirable features for both the CPU and the GPU. In our earlier work, we have investigated these in the context of Cg programming [23], while here we look at the trade-offs in the context of CUDA and a parallel implementation using cooperative cluster of multi-core, multi-socket CPUs and multiple GPUs.

3. GPU CLUSTER TESTBED

In this section we will present the specifications and properties of a typical, state-of-the-art, multi-socket, multi-core, multi-GPU cluster. For this purpose we picked the Ohio Supercomputer Center's BALE cluster [1]. The BALE cluster is comprised of a total of 71 Linux nodes, but our main focus is the newly added 16 visualization nodes, equipped with two dual-core AMD Opteron 2218 CPUs and two Nvidia Quadro FX 5600 GPUs each. The interconnection network for the BALE cluster is Infiniband. Figure 4 illustrates the architecture of the visualization nodes, and Table 1 shows the specifications of the CPU and GPU processors. Below we will go over some important specifications and features of the CPUs and GPUs used in our system.

On the CPU side, each node consists of two Opteron X2 2218 chips, which are dual-core processors running at 2.6 GHz. Each core in the system has a pair of 64 KB 2-way set associative L1 caches for holding data and instructions, and a 1 MB 4-way set associative L2 cache which is not shared among cores but are cache coherent. Each socket

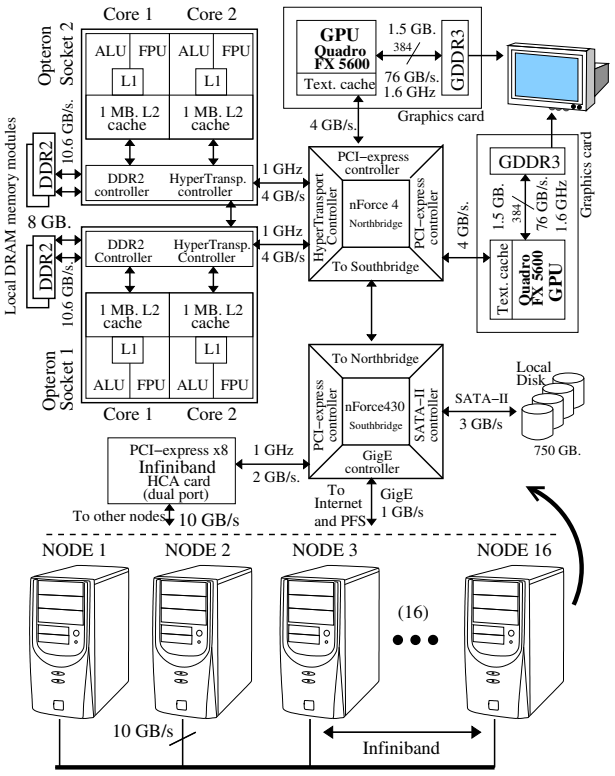

Figure 4: The BALE supercomputer at a glance.

includes its own dual-channel DDR2-667 memory controller as well as a single HyperTransport link to access the other socket's cache and memory. Each socket can thus deliver 10.6 GB/s for an aggregate memory bandwidth of 21.3 GB/s to each node's 8 GB of 667 MHz DDR2. The peak double-precision performance is 4.4 GFLOPS per core, 8.8 GFLOPS per socket, 17.6 GFLOPS per node. The peak single-precision performance per node is 35.2 GFLOPS, providing an aggregate single-precision performance of 563.2 GFLOPS for the 16 visualization nodes.

On the GPU side, each node has two Nvidia Quadro FX 5600 GPUs based on the G80 architecture. From a graphics viewpoint, the G80 can be seen as a 4-stage graphics pipeline for shading, texturing, rasterizing and coloring. As a parallel architecture, however, the G80 becomes a SIMD processor endowed with 128 cores, and CUDA is the programming interface to use it for general purpose computing. From the CUDA perspective, cores are organized into 16 multi-processors, each having a set of 32-bit registers, constants and texture caches, and a 16 KB memory shared by the eight cores inside each multi-processor. In any given cycle, each core executes the same instruction on different data, and communication between multi-processors is performed through global memory (see Figure 6).

4. TOOLS AND GPU IMPLEMENTATION

In order to give an example of the cooperative relationship between GPUs and multi-core CPUs, this section shows the details of the CUDA [5] implementation of the image analysis algorithm as well as the multi-node, multi-core and multi-GPU parallelization using DataCutter middleware [2]. We

Table 1: Hardware features of the CPU and GPU used on each node of the testbed cluster. The GLFLOPS listed are for 32-bit, single-precision floating-point arithmetic.

Hardware feature	CPU from AMD	GPU from Nvidia
Board	ASUS KFN32-D SLI	Quadro FX 5600
Processor model	Opteron X2 2218	G80
Processor speed	2.6 GHz	600/1350 MHz
Number of sockets	2	2
Number of cores	2	128
Peak processing	2 x 8.8 GFLOPS	2 x 330 GFLOPS
Memory size	8 GB	2 x 1.5 GB
Memory bus width	2 x 64 bits	2 x 384 bits
Memory speed	667 MHz	1600 MHz
Memory bandwidth	2 x 10.8 GB/s.	2 x 76.8 GB/s

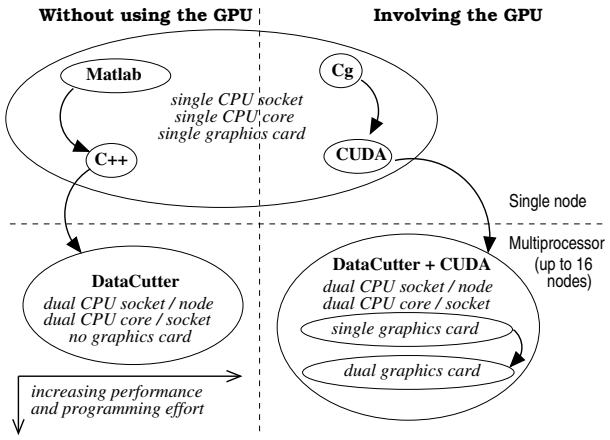

Figure 5: The programming approaches and their effect on performance, both single node and parallel.

have used Matlab [3], C++ and Cg implementations [23] as a baseline for our performance comparisons. Programming tools and paradigms are summarized in Figure 5.

4.1 CUDA

The CUDA programming interface consists of a set of C language library functions, and the CUDA-specific compiler generates the executable code for the GPU. Since CUDA is designed for generic computing, it does not suffer from excessive constraints when accessing memory (as Cg [4] does), but memory access times do vary for different memory types.

4.1.1 Computation Paradigm

CUDA exposes a model to the developer consisting of a collection of threads running in parallel. There are several other elements involved in the execution which bear mentioning, since they are unique to graphics processor programming with CUDA:

- A program is decomposed into **blocks** that run *logically* in parallel (physically only if there are resources available). Assembled by the developer, a block is a group of threads that is mapped to a single multi-processor, where they can share 16 Kbytes of memory (see Figure 6). All threads of concurrent blocks on a single multi-processor divide the resources available

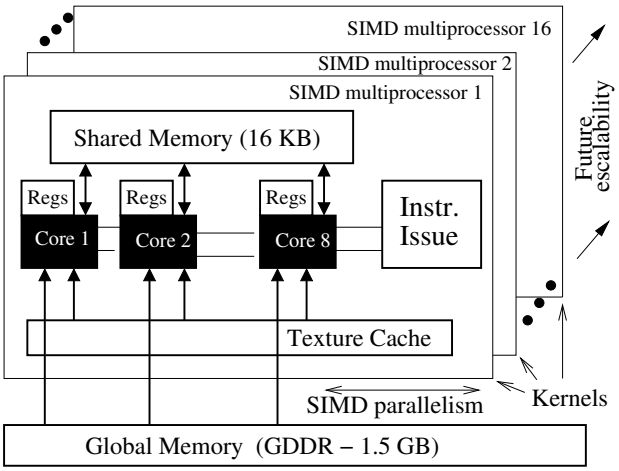

Figure 6: The CUDA hardware interface.

equally amongst themselves. The data is also divided amongst all of the threads in a SIMD fashion with a decomposition explicitly managed by the developer.

- A **warp** is a collection of threads that can actually run concurrently (with no time sharing) on all of the multi-processors. The size of the warp (32 threads on the G80) is less than the total number of available cores (128 on the G80) due to memory access limitations. The developer has the freedom to determine the number of threads to be executed (up to a limit intrinsic to CUDA), but if there are more threads than the warp size, they are time-shared on the actual hardware resources.

- A **kernel** is the actual code to be executed by each thread; the executable is shared among all of the threads in the system. Conditional execution of different operations on each multi-processor can be achieved based on a unique thread ID.

The CUDA documentation states that a single block should contain 128-256 threads to maximize execution efficiency, with a CUDA-imposed maximum of 512. Other hardware and software limitations are listed in Table 2, where we have ranked them according to their impact on the developer's implementation and overall performance based on our own experience.

4.1.2 *Memory and registers*

In the CUDA model, all of the threads can access all of the GPU memory, but, as expected, there is a performance boost when each thread reads data resident in the shared memory area, particularly when the data resides in several different memory banks (each bank can only support one memory access at a time, and therefore simultaneous accesses are serialized, hurting parallelism). The use of up to 16 KB of shared memory is explicit within a thread, which allows the developer to solve bank conflicts wisely. However, this type of optimization is often very difficult, but can also be very rewarding. Execution times may decrease by as much as 10x for vector operations and latency hiding may increase by up to 2.5x [8].

Table 2: Major hardware and software limitations with CUDA. Constraints are listed for the G80 GPU and categorized according to their difficulty of optimization and impact on the overall performance.

Parameter	Limitation	Impact
Multi-Processors per GPU	16	Low
Processors / Multi-Processor	8	Low
Threads / Warp	32	Low
Thread Blocks / Multi-Processor	8	Medium
Threads / Block	512	Medium
Threads / Multi-Processor	768	High
32-bit registers / Multi-Processor	8192	High
Shared Memory / Multi-Processor	16 Kbytes	High

When developing applications for GPUs with CUDA, the management of registers becomes important as a limiting factor for the amount of parallelism we can exploit. Each multi-processor contains 8,192 registers which will be split evenly among all the threads of the blocks assigned to that multi-processor. Hence, the number of registers needed in the computation will affect the number of threads able to be executed simultaneously, given the constraints outlined in Table 2. For example, if a kernel (and therefore a thread) consumes 16 registers, only 512 threads can be assigned to a single multi-processor, and this can be achieved by using 1 block with 512 threads, 2 blocks of 256 threads, and so on.

4.1.3 *Implementation of Image Analysis Code*

We have used a typical CUDA development cycle, which we will describe briefly. First, the code was compiled using the CUDA compiler and a special flag that outputs the hardware resources (memory and registers) consumed by the kernel. Using these values, we were able to analytically determine the number of threads and blocks that were needed to use a multi-processor with maximum efficiency. If a satisfactory efficiency could not be achieved, the code would need revision to reduce the memory footprint.

Due to the high floating point computation performance of the GPU, memory access becomes the bottleneck in many parts of our application. The input image tile (1K×1K×3 bytes) is much larger than the size of the multi-processor shared memory (16 KB), so we prioritize data structures like partial co-occurrence matrices (used in phase 2) and partial histograms (phase 4). However, in phase 1, even though the tile pixels are swept over without being reused, the execution time was lower when using shared memory (2.32 ms versus 2.77 ms - see Table 3). Also, in phase 3, the input pixels were moved to shared memory because the calculation of the LBP feature shows high data reuse.

In order to illustrate the progression of a typical CUDA implementation, we discuss the specific optimizations applied to each phase of the image analysis application below.

Phase 1: Color conversion.

As a departure point, we started using 24-bit `float3` data types for each color channel. However, extra performance can be obtained by padding the RGB input to match the expected data width of 32-bits, which simplifies all subsequent optimizations involving shared memory. This is called data *coalescing*, and for this phase it saved 35% of the computa-

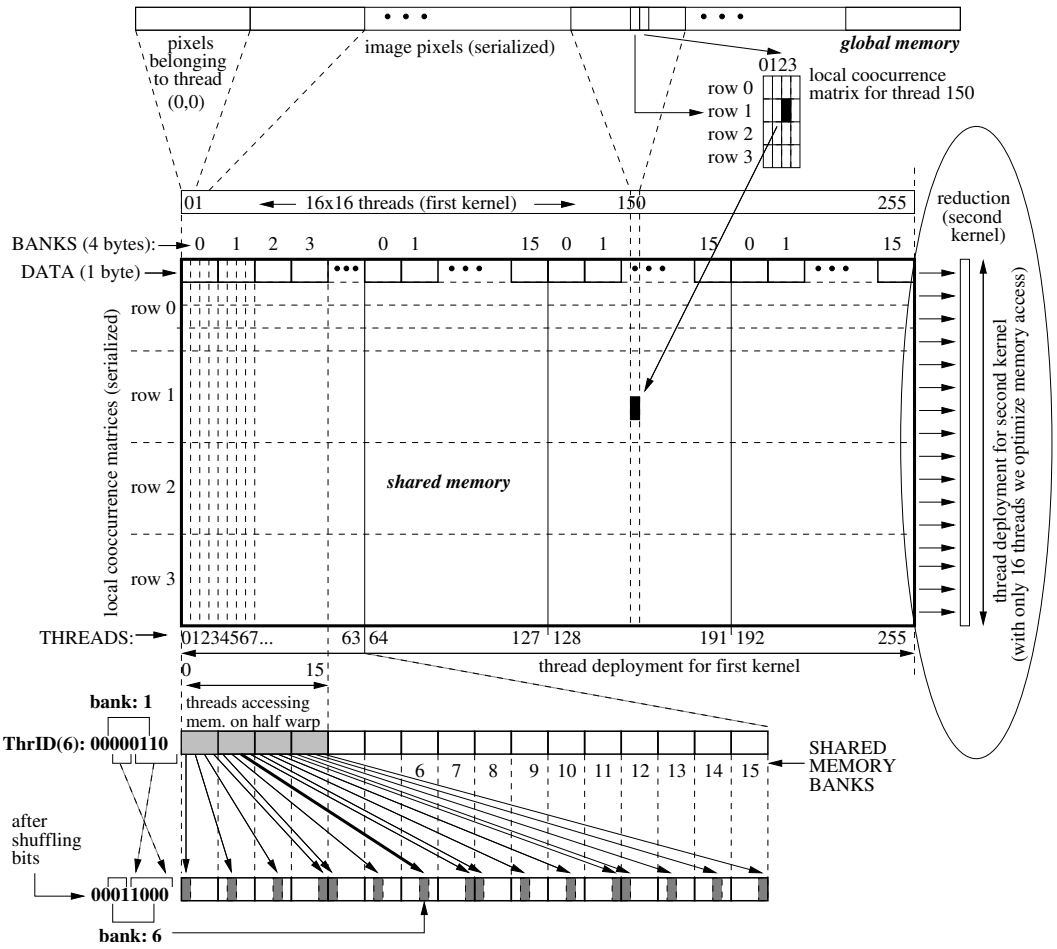

Figure 7: Phase 2: Local co-occurrence matrices in CUDA. Assigning banks in shared memory to each thread to avoid conflicts when computing co-occurrence matrices.

tion time at the expense of communication time (see code 1.11 in Table 3). Next, it was found that 8-bit `uchar` data types were sufficient for the precision of the application. As expected, this reduces the communication time by nearly a factor of 4.

We then used the special CUDA compilation flag to find that the color conversion kernel requires 13 registers and 1064 bytes of shared memory, leading to a maximum processor occupancy of 75% when allocating between 176 and 192 threads. However, we chose to allocate 256 threads instead, trading processor occupancy (from 75% down to 67%) for better load balance, since 256 is a multiple of 32 (maximum threads per warp) and a divisor of 768 (maximum threads per multi-processor) and 1024 (maximum pixels per image). The result was that the execution times improved slightly (see code 1.30 versus 1.40 in Table 3 - version 1.40 reports the minimum time obtained for all threads/block cases between 169 and 192, which turns out to be 169 threads). Unfortunately, maximum performance here is limited because each thread needs more than 10 registers (11 to be precise), which, as discussed earlier, prevents us from reaching the maximum occupancy of 768 threads within a multiprocessor. The optimal execution time for this phase was 2.98 ms for pixel transfer and 2.32 ms for computing the color conversion, as reflected in Table 3.

Phase 2: Statistical features.

This kernel requires 9 registers and 4132 bytes of shared memory, which allowed us to allocate 3 parallel blocks of 256 threads. This perfect usage of all 768 threads filled all of the G80 hardware resources for a 100% occupancy factor. Pixels are equally distributed among threads and local co-occurrence matrices are simultaneously computed within them. Finally, partial results are accumulated through a reduction operator.

It was found to be challenging to compute co-occurrence matrices in shared memory while avoiding conflicts accessing its 16 banks. With a grid of 256 threads arranged in a 16x16 grid, the naive thread deployment would force the 32 threads in a warp to access only 8 shared-memory banks, which would severely limit parallelism and performance. We found that by intelligently shuffling the active threads combined into a warp, the local matrix computation and the subsequent global matrix aggregation operation can proceed without forcing threads to wait for bank access (see Figure 7). This complex optimization solves all conflicts when accessing memory banks, reducing the execution time to 2.58 ms from 4.48 ms. Without using shared memory, a straightforward implementation consumes 15.40 ms instead (see Table 3).

Table 3: Major CUDA optimizations in the image analysis application. Execution times correspond to an isolated 1Kx1K tile on a single GPU. Phase 1 shows the times for CPU to GPU communication and actual GPU computation. Phases 2 and 3 show only GPU computation (no communication is required). Phase 4 shows the times for GPU to CPU communication and actual GPU computation. The fifth column in the table refers to conflicts arising when several threads simultaneously access the same bank of shared memory.

Analysis Phase	Code tag	Description / Optimizations	In shared memory	Conflicts solved	Execution time (millisecs.) Comm.	Comput.	Total
1:	1.10	Baseline version: Using float3 per color channel	All in global memory		8.49	3.71	12.20
RGB to	1.11	Coalescing (Alpha channel inserted) on float3	All in global memory		10.79	2.44	13.23
LA*B*	1.12	Replacing float3 by uchar (256 threads/block)	All in global memory		2.98	2.77	5.75
color	1.20, 1.30	**Using shared memory (256 threads/block)**	**Pixels**	**Unneeded**	**2.98**	**2.32**	**5.30**
conversion	1.40	Using between 169 and 192 threads/block	Pixels	Unneeded	2.98	2.43	5.41
2:	2.10	Baseline version: Using global memory	All in global memory		None	15.40	15.40
Statistical	2.20	Using shared memory for co-occurrence matrices	Co-oc. mat.	No	None	4.48	4.48
features	2.30	**Solving conflicts on shared memory banks**	**Co-oc. mat.**	**Yes**	**None**	**2.58**	**2.58**
3:	3.10	Baseline version: Special threads on grid borders	All in global memory		None	2.29	2.29
LBP	3.20, 3.30	**Blocks of 16x16 threads, 14x14 computing**	**Pixels**	**Unneeded**	**None**	**1.82**	**1.82**
operator	3.40	Blocks of 8x8 threads, 6x6 of them computing	Pixels	Unneeded	None	2.31	2.31
4:	4.10	Baseline version: Using global memory	All in global memory		4.02	2.08	6.10
Histogram	4.20	Using shared memory for local histograms	Local hist.	No	0.31	0.61	0.92
	4.30	**Solving inter-warp conflicts in mem. banks**	**Local hist.**	**Inter-warp**	**0.31**	**0.59**	**0.90**
Total	Σ(*.10)	Baseline version	All in global memory		12.51	23.48	35.99
GPU	Σ(*.20)	Involving shared memory	Enabled	No	3.29	9.23	12.52
time	Σ(*.30)	**Optimal version**	**Enabled**	**Most**	**3.29**	**7.31**	**10.60**
Total CPU time:		(1:) 880.31 ms. + (2:) 43.24 ms. + (3:) 156.28 ms. + (4:) 7.49 ms. = **1087.32**					

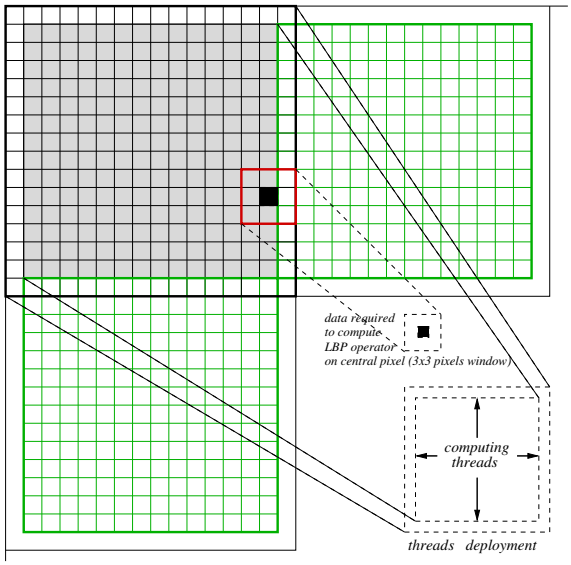

Figure 8: Phase 3: LBP operator in CUDA. We allocate more memory than computing threads and overlap two rows and columns of the external frame on neighbor blocks in order to cover the whole computational domain homogeneously.

Phase 3: LBP operator.

The computation of the LBP feature entails a convolution with a 3x3 mask, followed by a binary to decimal conversion (see Figure 3). Each thread requires 10 registers and each block of threads uses 296 bytes of shared memory. Due to the memory usage characteristics, we were able to allocate 256 threads in a 16 x 16 grid to reach 100% occupancy on the G80. Each thread reads a pixel from global memory and stores it in a shared memory data structure. Those threads located on the border of the grid are unable to compute, since they do not have access to their neighbor data (see Figure 8); they exit the kernel at this stage. The LBP for the border regions will be computed by the next block of threads, since we overlap the thread layout by two rows and two columns of pixels each time. By using this strategy, we incur 23% idle cycles and the associated redundant memory accesses, but the computation is more homogeneous and the threads are more lightweight. As compared to the heterogeneous version, where there are no idle cycles and no memory redundancy, the homogeneous version is faster by 1.25x, leading to the lowest execution time of 1.82 ms.

Since the LBP kernel is very regular, we can select different sizes of thread deployments, provided a square grid is used. Therefore, we investigated the effect of different thread/block ratios. We gathered values for grids of threads of sizes 20x20 (2.02 ms), 18x18 (1.90 ms), 16x16 (1.82 ms - optimal), 14x14 (1.89 ms), 12x12 (2.00 ms), 10x10 (2.16 ms), and 8x8 (2.31 ms - reported in Table 3 as version 3.40). A grid of 16x16 threads is a popular choice among expert programmers to maximize performance in CUDA; that said, our intention was more to quantify the penalty we may incur by making an inappropriate choice for thread allocation.

Phase 4: Histogram.

The implementation of this phase is based on a histogram kernel included with the CUDA library [20], where the global reduction is delegated to the CPU. We deemed this an appropriate solution, since the histogram is only computed once per tile, and does not incur a large execution time cost. The execution time for this last phase is 0.9 ms.

Table 3 summarizes all of the major optimizations performed within CUDA on each phase along with their execution times. Since the CUDA programming model does not allow the developer to assign different kernels to different multi-processors, the four phases are sequentially executed and the whole GPU is used during each phase independently.

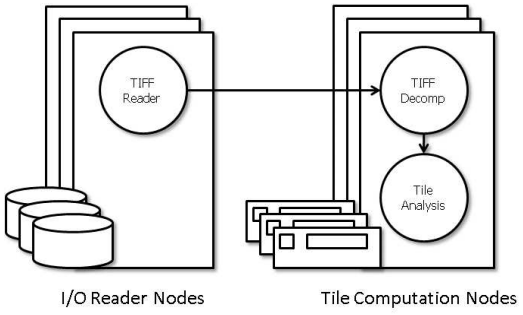

Figure 9: DataCutter layout of the image analysis application.

Overall, by using CUDA to implement the image analysis, we were able to reduce the execution time by a factor of 3-5 over the times obtained by Cg (see Table 5 for times on an entire image), by an additional factor of 3 when enabling shared memory, and by an extra 20% when solving memory conflicts as much as possible for an optimal execution.

4.2 DataCutter

We employed DataCutter middleware for the parallelization of the testbed image analysis computation. DataCutter [2] is a component-based middleware framework and provides a coarse-grained data flow system allowing combined use of task- and data-parallelism. Applications are decomposed into sequential tasks (*filters*) with data dependencies between them. Data flows from filter to filter through *logical streams* in a buffered, non-blocking manner. This buffering allows for the easy overlap of data communication and computation, which allows for the hiding of disk and internode communication latencies. The DataCutter runtime system supports efficient execution of filters on heterogeneous, multi-socket, and multi-core compute clusters. The runtime system performs all steps necessary to instantiate filters on the desired hosts, to connect all logical endpoints, and to call the filter's interface functions for processing work. Each filter executes within a separate POSIX thread, allowing for CPU, I/O and communication overlap. Data exchange between two filters on the same host is carried out by memory copy or pointer operations, while TCP sockets or MPI communications (in order to leverage low latency high bandwidth interconnects such as Infiniband) are employed for communication between filters on different hosts.

Our image analysis application is easily divided into three stages, each of which is implemented as a single filter type. These are: a *TIFF-Reader* filter that reads binary TIFF tiles, a *TIFF-Decompressor* filter that decompresses TIFF tiles and produces RGB images, and a *Tile-Analysis* filter where the real image analysis computation is carried out. A layout which includes one each of these three filter types constitutes a complete image analysis task graph, while multiple copies of each filter type allows for quick parallelization and efficient execution. Figure 9 shows the general layout of the system. One or more reader nodes (with one *TIFF-Reader* filter each) read the binary TIFF image tiles from the disk and write them to the stream leading to the *TIFF-Decompressor*. One or more *TIFF-Decompresser* filters are able to coexist on the same node and simply take the binary TIFF file and decompress it into the appropriate RGBA tuples. This decompressed image data is then written to the input stream to the *Image-Analysis* filter on the same node. When the *Tile-Analysis* filter makes use of the GPU, the ability for multiple *Tiff-Decompressor* filters to feed data to the GPU allows for full utilization of the GPU's throughput. Placing both the *TIFF-Decompressor* and the *Image-Analysis* filters on the same node has the benefit of saving memory bandwidth by leaving the largest representation of each TIFF tile in place and simply transferring the pointer.

The component-based programming model of DataCutter allows us to easily develop and deploy image analysis applications that utilize GPUs as co-processors. By simply replacing the C++ *Tile-Analysis* filters with filters using the GPU, we can quickly develop a parallel code that efficiently executes on a multi-socket, multi-core, multi-GPU cluster. The layout, the two *TIFF-Reader* and *TIFF-Decompressor* filters, and the entire parallelization system are reusable.

Table 4: Properties of the three slides used in our experiments.

Name	Resolution in pixels	Number of 1Kx1K tiles
SMALL	32,980 x 66,426	33 x 65 = 2,145
MEDIUM	76,543 x 63,024	75 x 62 = 4,659
LARGE	109,110 x 80,828	107 x 79 = 8,453

Table 5: Execution times for the image analysis application for different programming methods and hardware platforms. These times do not include disk I/O or decompression overheads. The image tile size is 1Kx1K pixels. The co-occurrence matrix size is 4x4. The values in the CUDA column represent running on one and two GPUs.

	On the CPU		On the GPU	
Image size	Matlab	C++	Cg	CUDA
SMALL	2h 57' 29"	43' 40"	1' 02"	27", 14"
MEDIUM	6h 25' 45"	1h 34' 51"	2' 08"	58", 32"
LARGE	11h 39' 28"	2h 51' 23"	3' 54"	1' 47", 58"

5. EXPERIMENTAL RESULTS

Our experiments were performed on the BALE cluster (see Section 3) using the 16 visualization nodes and an additional six compute nodes as reader nodes, in order to provide enough disk bandwidth to avoid the disk I/O bottleneck. Further, these reader nodes had their system file caches preloaded with several discarded experiments to ensure extremely high I/O from the upstream, *TIFF-READER* filters. We feel this is a suitable experimental setup to use; any production cluster designed to analyze this kind of image data with very high performance with multiple GPUs and a fast interconnect can reasonably be said to have parallel disks providing high I/O bandwidth.

In our experiments, we have used three different digitized pathology images. Table 4 summarizes their features.

The first set of experiments shows the single node CPU and GPU performance for the various implementations of the image analysis algorithm. Figure 10 shows the execution time and overhead time of each implementation when

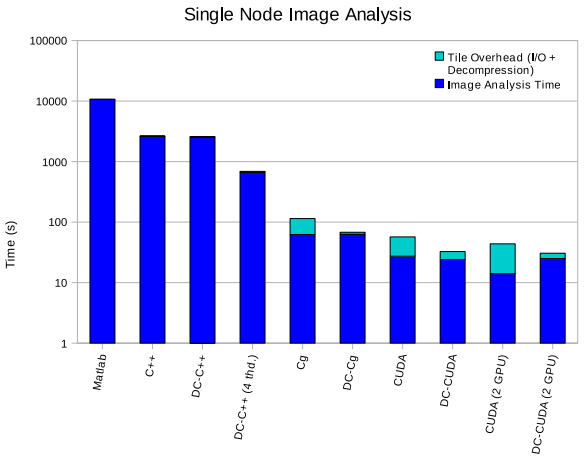

Figure 10: Execution time comparison of all implementations of the image analysis codes running on a single node using `SMALL` image.

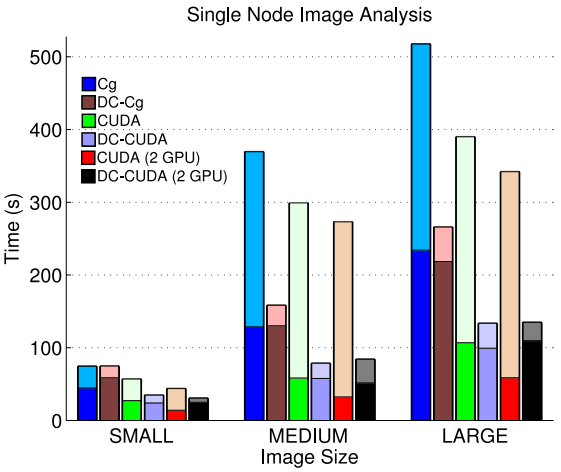

Figure 11: Execution time comparison of GPU and DataCutter implementations running on a single node using all three input images.

analyzing the `SMALL` image. The first four stacked bars represent CPU-only implementations, while the last six stacked bars bring one or two GPUs into the fold. Those bars with labels beginning with 'DC' are results from those implementations using DataCutter and those without the 'DC' label are the basic, serial implementations.

The execution time in Figure 10 (shown by the lower, darker portion of each bar) is due solely to the actual image analysis, while the overhead (shown by the upper, lighter portion of each bar) is caused by disk I/O, TIFF decompression, remote process invocation, and network latencies, where applicable. In this experiment, the most important thing to note is the two to three orders of magnitude of performance speedup when moving from the CPU-based solutions to the GPU-based solutions. While a large amount of performance is gained by moving away from Matlab, the C++ implementations are still far slower than those which are GPU-based. Additionally, the DataCutter versions of

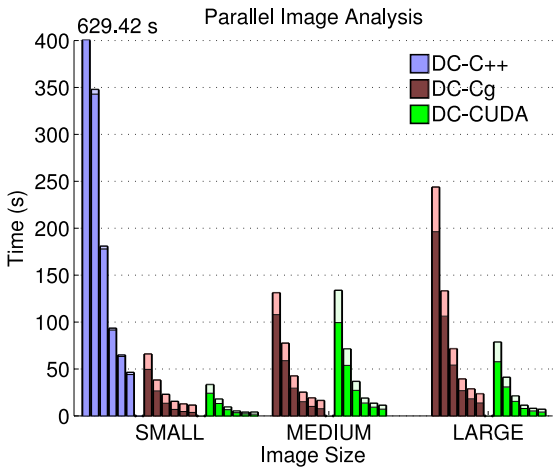

Figure 12: Parallel execution times of C++, Cg, and CUDA based DataCutter implementations using three input images while varying the number of nodes from 1 to 16.

the GPU-based image analysis algorithms are able to shorten the execution time for the entire image versus the non-DataCutter versions, since the decoupled, multi-threaded nature of DataCutter allows the image analysis to overlap with the TIFF tile decompression and the disk I/O. Unfortunately, the CUDA implementation of the image analysis algorithm is fast enough to cause the TIFF tile decompression stage to become a bottleneck when 2 GPUs are used. This stalling, incurred by the GPUs waiting for tiles to analyze, prevents both GPUs from being fully utilized. Since four C++ threads shows a clear advantage over running a single thread, all future C++ results will be comprised of the DataCutter version with four tile analysis threads per node.

Figure 11 shows the performance comparison of the GPU-based implementations of the analysis routine for our three images. The main point to take from this chart is that there is a linear relationship between the execution time and the overall size of the image under analysis in all of the implementations. Unfortunately, even when analyzing large images, making full use of 2 GPUs is hindered by the associated overheads; this being the case, we will not show 2 GPU results for the remainder of the experiments.

Figure 12 shows the scalability of our solution with respect to the number of nodes. The numbers of nodes involved in the image analysis are $1, 2, 4, 8, 12, 16$, and are represented by the bars in the figure from left to right, six in each color group. As in Figure 11, the lower, darker-colored portion of each bar represents the image analysis time, while the upper, lighter-colored portion of each bar shows the aggregated overhead. This type of image analysis computation scales extremely well, resulting in image analysis execution times which decrease nearly linearly with the number of nodes. Further, the total analysis times for the DataCutter/CUDA implementation are under four seconds for the `SMALL` image, under seven seconds for the `MEDIUM` image, and just over eleven seconds for the `LARGE` image, when running on sixteen nodes. Compared with the single node CPU Matlab

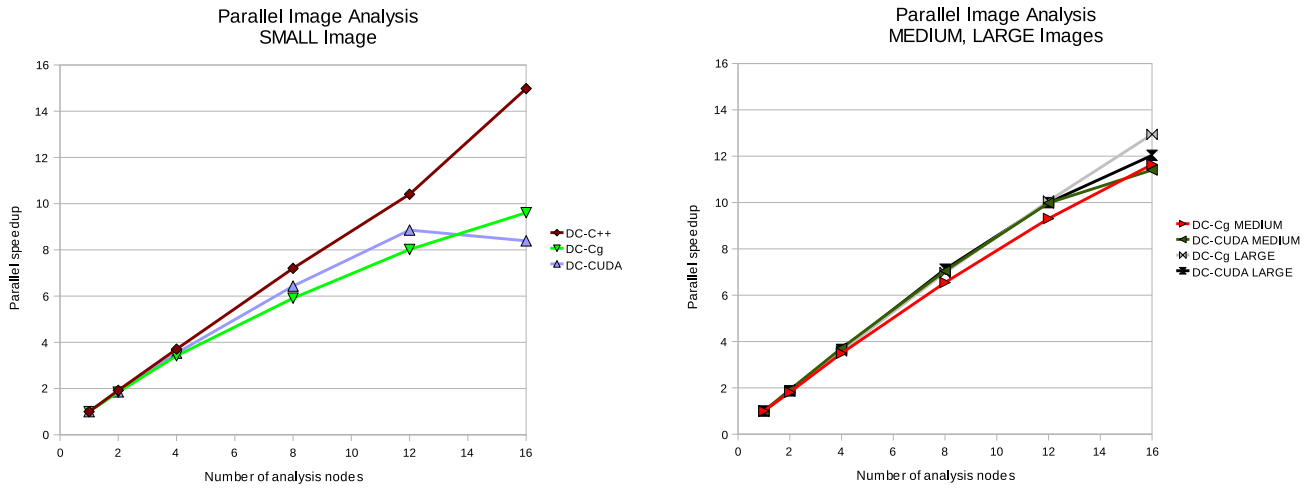

Figure 13: Parallel speedup results. Within DataCutter (DC) versions, we compare on the left a CPU-only case with two GPU-assisted ones for the `SMALL` image. On the right, we contrast the two GPU versions for the `MEDIUM` and `LARGE` images.

computation time of nearly three hours for the `SMALL` image and almost twelve hours for the `LARGE` image, this represents a tangible benefit of increased productivity. In the interest of increased chart legibility, we have cropped the single node C++ result. Its values are 629.42 seconds of image analysis time and 29.7 seconds of overhead. Additionally, since the main focus of this paper is the GPU results, and since the C++ result is shown to scale well in the worst-case (because it incurs the lowest proportional overhead), we have chosen to remove it from the figures showing results for the `MEDIUM` and `LARGE` images.

Figure 13 shows the parallel speedup of the execution time versus the number of nodes. As seen in the figures, there is nearly linear speedup since the tiles are able to be decompressed and processed entirely independently of each other. However, due to the small execution times in the GPU-based implementations, the various overheads (comprised of remote process startup, network, and TIFF decompression latencies) begin to become comparable in overall time to the total time spent per node processing the image tiles. For instance, on sixteen nodes, the CUDA implementation requires at most 1.80 seconds of computation to compute 2,145 tiles. However, despite concurrently running three *TIFF-Decompressor* filters, the decompression time alone for each node's allotment of 135 tiles could range from 0.3 seconds to 1.3 seconds.

Nearly linear speedup could be achieved in a production environment, however, since it is reasonable to assume that remote process invocation would only occur once for many images which are to be analyzed. Under these server-like circumstances, only the I/O system and network latencies would comprise the system overheads.

6. CONCLUSIONS

In this paper, we have presented design trade-offs and a performance evaluation of a sample biomedical image analysis application running on a cooperative cluster of CPUs and GPUs.

By implementing algorithms on GPUs using CUDA and using DataCutter to parallelize the computation within and across nodes, we establish a solid heterogeneous and cooperative multi-processor platform where all the granularities of parallelism inherent in the architecture and in the application are fully exploited: multi-node (using DataCutter for data partitioning across nodes), SMP and thread-level (using DataCutter to fully utilize the available on-node and on-chip hardware resources), SIMD (using CUDA to fully populate the 128 stream processors of the GPU with work), and finally, ILP (Instruction Level Parallelism, by setting up blocks of computational threads within the GPU execution).

Our experimental results show great success for our techniques, first by decreasing the execution time on a single CPU/GPU node by using different intra-node optimizations, and then extending those performance gains to inter-node parallelism for a scalable multi-processor execution. When analyzing the largest test image and including overheads, on the 16 node cluster configuration, the single GPU DataCutter-CUDA implementation is 31.3 times faster than the serial CUDA implementation. By using two GPUs per node, the single-node time to process the image is under one minute, if you ignore the overheads associated with disk I/O and tile decompression, proving that the CUDA method is extremely powerful. Additionally, the use of DataCutter to overlap the computation with disk I/O and tile decompression helps the GPU stay as busy as possible. This results in up to 12.94 speedup on 16 nodes using GPU-based DataCutter implementations.

GPUs are highly scalable and are evolving towards general-purpose architectures [10]; we envision biomedical image processing as one of the most exciting fields able to benefit from the use of GPUs. Additionally, new tools like CUDA [5] may assist non-computer scientists with a more friendly interface for adapting biomedical applications to GPUs. This computational power may then be combined with DataCutter to parallelize the computation across clusters of GPUs as outlined in this paper to provide real-time response to clinicians of all types.

7. ACKNOWLEDGEMENTS

We thank Metin Gurcan from the Department of Biomedical Informatics, Ohio State University, for providing us the input neuroblastoma images and Matlab code for the image analysis application. We also thank Dennis Sessanna and Don Stredney at the Ohio Supercomputer Center for support running our code on the BALE visualization cluster.

8. REFERENCES

[1] The BALE cluster at the Ohio Supercomputer Center. http://www.osc.edu/supercomputing/hardware.

[2] M. Beynon, K. T., U. Catalyurek, C. Chang, A. Sussman, and J. Saltz. Distributed processing of very large datasets with DataCutter. *Parallel Computing*, 27(11):1457–1478, October 2001.

[3] B. B. Cambazoglu, O. Sertel, J. Kong, J. Saltz, M. N. Gurcan, and U. V. Catalyurek. Efficient processing of pathological images using the grid: Computer-aided prognosis of neuroblastoma. In *Proceedings Intl. Workshop on Challenges of Large Applications in Distributed Environments*, 2007.

[4] The Cg language. Home page maintained by Nvidia. http://developer.nvidia.com/page/cg_main.html.

[5] CUDA. Home page maintained by Nvidia. http://developer.nvidia.com/object/cuda.html.

[6] L. Davis, S. Johns, and J. Aggarwal. Texture analysis using generalized co-occurrence matrices. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 3:251–259, 1979.

[7] A. N. Esgiar, R. N. G. Naguib, B. S. Sharif, M. K. Bennet, and A. Murray. Microscopic image analysis for quantitative measurement and feature identification of normal and cancerous colonic mucosa. *IEEE Transactions on Information Technology in Biomedicine*, 2:197–203, 1998.

[8] M. Fatica, D. Luebke, I. Buck, D. Owens, M. Harris, J. Stone, C. Phillips, and B. Deschizeaux. CUDA Tutorial at Supercomputing 2007.

[9] FireStream. GPU hardware solutions from AMD-ATI. http://ati.amd.com/products/streamprocessor/specs.html, 2008.

[10] GPGPU. General-purpose computation using graphics hardware. http://www.gpgpu.org.

[11] S. Guha, S. Krisnan, and S. Venkatasubramanian. Data visualization and mining using the GPU. *Tutorial at 11th ACM International Conference on Knowledge Discovery and Data Mining*, 2005.

[12] M. Gurcan, J. Kong, O. Sertel, B. Cambazoglu, J. Saltz, and U. Catalyurek. Computerized pathological image analysis for neuroblastoma prognosis. In *AMIA Annual Symposium*, 2007.

[13] M. Hadwiger, C. Langer, H. Scharsach, and K. Buhler. State of the art report on GPU-based segmentation. Technical Report TR-VRVIS-2004-17, VRVis Research Center, Vienna, Austria, 2004.

[14] K. J. Khouzani and H. S. Zadeh. Multiwavelet grading of pathological images of prostate. *IEEE Transactions on Biomedical Engineering*, 50(6):697–704, 2003.

[15] J. Kong, H. Shimada, K. Boyer, J. Saltz, and M. Gurcan. Image analysis for automated assessment of grade of neuroblastic differentiation. In *IEEE Intl. Symposium on Biomedical Imaging*, 2007.

[16] J. D. Owens, D. Luebke, N. Govindaraju, M. Harris, J. Kruger, A. E. Lefohn, and T. J. Purcell. A survey of general-purpose computation on graphics hardware. *Journal of Computer Graphics Forum*, 26:21–51, 2007.

[17] G. Paschos. Perceptually uniform color spaces for color texture analysis: An empirical evaluation. *IEEE Transactions on Image Processing*, 10:932–937, 2001.

[18] S. Petushi, F. U. Garcia, M. Habe, C. Katsinis, and A. Tozeren. Large-scale computations on histology images reveal grade-differentiating parameters for breast cancer. *BMC Medical Imaging*, 6(14), 2006.

[19] S. Petushi, C. Katsinis, C. Coward, F. Garcia, and A. Tozeren. Automated identification of microstructures on histology slides. In *IEEE Intl. Symposium on Biomedical Imaging*, 2004.

[20] V. Podlozhnyuk. Histogram calculation in CUDA, 2007.

[21] S. Ray. On a theoretical property of the bhattacharyya coefficient as a feature evaluation criterion. *Pattern Recognition Letters*, pages 315–319, 1989.

[22] A. Ruiz, O. Sertel, M. Ujaldón, U. Catalyurek, J. Saltz, and M. Gurcan. Stroma classification for neuroblastoma on graphics processors. *Intl. Journal of Data Mining and Bioinformatics*, 3(4), 2008.

[23] A. Ruiz, O. Sertel, M. Ujaldón, U. Catalyurek, J. Saltz, and M. Gurcan. Pathological image analysis using the GPU: Stroma classification for neuroblastoma. In *IEEE Intl. Conference on BioInformatics and BioMedicine (BIBM'07)*, November, 2007.

[24] O. Sertel, J. Kong, H. Shimada, U. Catalyurek, J. Saltz, and M. Gurcan. Computer-aided prognosis of neuroblastoma: classification of stromal development on whole-slide images. In *SPIE Medical Imaging*, San Diego, California, 2008.

[25] H. Shimada, I. M. Ambros, L. P. Dehner, J. Hata, V. V. Joshi, B. Roald, D. O. Stram, R. B. Gerbing, J. N. Lukens, K. K. Matthay, and R. P. Gastlebery. The Intl. neuroblastoma pathology classification (the Shimada system). *Cancer*, 86(2):364–372, 1999.

[26] M. A. Tahir and A. Bouridane. Novel round-robin tabu search algorithm for prostate cancer classification and diagnosis using multispectral imagery. *IEEE Transactions on Information Technology in Biomedicine*, 20:782–791, 2006.

[27] V. Takala, T. Ahanen, and M. Pietikainen. Block-based methods for image retrieval using local binary patterns. *Lecture Notes in Computer Science*, 3540:882–891, 2005.

[28] Nvidia Tesla GPU computing solutions for HPC. http://www.nvidia.com/object/tesla_computing_solutions.html

[29] M. Tuceryan and A. K. Jain. *Texture Analysis, in The Handbook of Pattern Recognition and Computer Vision (2nd Ed.)*. World Scientific Publishing Co, 1998.

[30] W. Wu and P. Heng. A hybrid condensed finite element model with GPU acceleration for interactive 3D soft tissue cutting: Research articles. *Computer Animation and Virtual Worlds*, 15(3-4):219–227, 2004.

[31] Zhao, Y., Y. Han, Z. Fan, F. Qiu, Y. Kuo, Kaufman, and K. A., Mueller. Visual simulation of heat shimmering and mirage. *IEEE Trans. on Visualization and Computer Graphics*, 13(1):179–189, 2007.