Run-time optimizations for replicated dataflows on heterogeneous environments

George Teodoro[†], Timothy D. R. Hartley[‡], Umit Catalyurek[‡], Renato Ferreira[†]

[†]Dept. of Computer Science
Universidade Federal de Minas Gerais, Brazil
{george, renato}@dcc.ufmg.br

[‡]Dept. of Biomedical Informatics,
Dept. of Electrical & Computer Engineering
The Ohio State University, USA
{hartleyt, umit}@bmi.osu.edu

ABSTRACT

The increases in multi-core processor parallelism and in the flexibility of many-core accelerator processors, such as GPUs, have turned traditional SMP systems into hierarchical, heterogeneous computing environments. Fully exploiting these improvements in parallel system design remains an open problem. Moreover, most of the current tools for the development of parallel applications for hierarchical systems concentrate on the use of only a single processor type (e.g., accelerators) and do not coordinate several heterogeneous processors. Here, we show that making use of all of the heterogeneous computing resources can significantly improve application performance. Our approach, which consists of optimizing applications at run-time by efficiently coordinating application task execution on all available processing units is evaluated in the context of replicated dataflow applications. The proposed techniques were developed and implemented in an integrated run-time system targeting both intra- and inter-node parallelism. The experimental results with a real-world complex biomedical application show that our approach nearly doubles the performance of the GPU-only implementation on a distributed heterogeneous accelerator cluster.

Categories and Subject Descriptors

C.1 [**Processor Architectures**]: Multiple Data Stream Architectures (Multiprocessors); I.3 [**Computer Graphics**]: Hardware Architecture—*Graphics Processors*

General Terms

Algorithms, Design, Performance

Keywords

GPGPU, run-time optimizations, filter-stream

1. INTRODUCTION

With the current advances in computer architecture, traditional distributed platforms are fast transforming into hierarchical environments, where each computing node may have multiple processors each having multiple cores. Moreover, the use of modern graphics processors (GPUs) as fast co-processors for general-purpose computation has become very popular, as these devices can significantly improve the performance of applications under certain conditions. With these two trends developing simultaneously, developers of large-scale or high-performance applications will need to design their applications for heterogeneous computing devices.

The problem discussed in this paper is that of efficiently executing applications in heterogeneous clusters of GPU-equipped multicore computers. The proposed techniques are evaluated in the context of the filter-stream programming model, a type of dataflow where applications are decomposed into filters that can be replicated on multiple nodes of a distributed system. These filters communicate using logical streams that are capable of delivering data from a instance of the source filter to an instance of the destination filter. We feel that such a model naturally exposes a large number of independent tasks which can be executed concurrently on multiple devices. Filters, in our runtime framework, are multi-threaded and can have different versions of their codes so they can execute on heterogeneous processors. The techniques are generalizable to other environments with similar capabilities.

An important facet of our work is that the performance of the GPU is data-dependent, meaning that the speedup the GPU can yield over the CPU depends on the type of computation and the input data. Moreover, the tasks generated by the applications are not known prior to execution. Therefore, the decision about where to run each task has to be made at run-time as the tasks are created. Therefore, our approach assigns tasks to each device based on the relative performance of that device in such a way as to optimize the overall execution time. Extending our previous work [38], we consider both intra-node as well as inter-node parallelism. We also present new techniques for on-line performance estimation and handling data transfers between the CPU and the GPU. Our main contributions are:

- A performance estimation module for task relative performance based on their input parameters.
- An algorithm for efficiently coordinating data transfers that automatically provides for asynchronous data copying back and forth between the CPU and the GPU;
- A novel stream communication policy for heterogeneous environments that efficiently coordinates the use of CPUs and GPUs in clusters equipped with multi-core and accelerator processors;

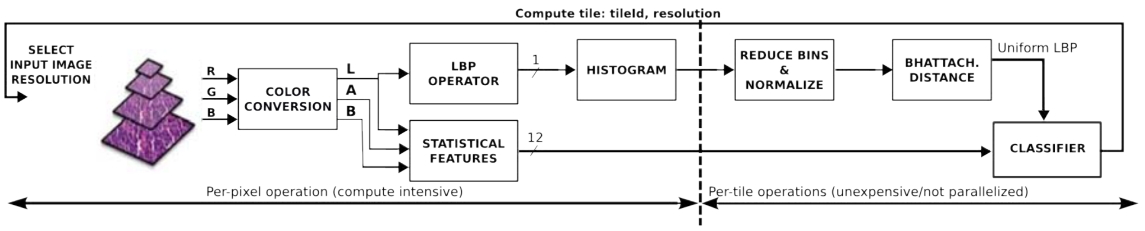

Compute tile: tileId, resolution
SELECT INPUT IMAGE RESOLUTION
R
G
B
COLOR CONVERSION
L
A
B
LBP OPERATOR
1
HISTOGRAM
STATISTICAL FEATURES
12
REDUCE BINS & NORMALIZE
BHATTACH. DISTANCE
Uniform LBP
CLASSIFIER
Per-pixel operation (compute intensive)
Per-tile operations (unexpensive/not parallelized)

Figure 1: NBIA Flow Chart

Although an important active research topic, generating code for the GPU is beyond the scope of this paper. We assume that the necessary code to run the application on both the CPU and the GPU are provided by the programmer and we focus on the efficient coordination of the execution on heterogeneous environments, as most of the related work relegates this problem to the programmer and either focuses only on the accelerator performance or assumes the speedup of the device is constant for all tasks. For GPU programming we refer to the CUDA [23] toolkit and mention other GPU programming research based on compiler techniques [18, 26], specialized libraries [7, 12, 21], or applications [34]. Also, though we focus on GPUs and CPUs, the techniques are easily adapted for multiple different devices.

The rest of the paper is organized as follows: in Section 2 we present our use case and motivating application. Section 3 presents the Anthill, the framework used for evaluation of the proposed techniques, and how it has been extended to target multiple devices. Section 4 presents our performance estimation approach and Section 5 describes the run-time optimizations. An extensive experimental evaluation is presented in Section 6 and the conclusions are summarized in Section 7.

2. MOTIVATING APPLICATION

In this section we describe the Neuroblastoma Image Analysis System (NBIA), a real-world application developed by Sertel et al. [31] that is used as both a motivating example as well as an evaluation platform for the techniques discussed in this paper. NBIA assists the determination of prognosis for neuroblastoma, which is a cancer of the sympathetic nervous system that mostly affects children. The prognosis of the disease is currently determined by expert pathologists based on visual examination under a microscope of tissue slides. The slides can be classified into different prognostic groups conditioned to the differentiation grade of the neuroblasts, among other issues.

Manual examination by pathologists is an error-prone and very time consuming process. Therefore, the goal of NBIA is to assist in the determination of the prognosis of the disease by classifying the digitized tissue samples into different subtypes that have prognostic significance. The focus of the application is on the classification of stromal development as either stroma-rich or stroma-poor, which is one of the morphological criteria in the disease's prognosis that contributes to the categorization of the histology as favorable and unfavorable [32].

Since the slide images can be very high resolution (over 100K x 100K pixels of 24-bit color), the whole-slide NBIA first decomposes the image into smaller image tiles that can be processed independently. The analysis is done using a multi-resolution strategy that constructs a pyramid representation [28], with multiple copies of the image tiles from the decomposition step, each one with a different resolution. As an example, a three-layered, multi-resolution analysis could be constructed with (32×32), (128×128), and (512×512) tiles sizes. The analysis for each of the tiles proceeds starting at the lowest resolution, and stops at the resolution level where the classification satisfies some pre-determined criterion.

Tile classification is achieved based on statistical features that characterize the texture of tissue structure. Therefore, NBIA first applies a color space conversion to the La*b* color space, where color and intensity are separated and the difference between two pixel values is perceptually more uniform, enabling the use of Euclidean distance for feature calculation. The texture information is extracted using co-occurrence statistics and local binary patterns (LBPs), which help characterize the color and intensity variations in the tissue structure. Finally, the classification confidence at a particular resolution is computed using hypothesis testing; either the classification decision is accepted or the analysis resumes with a higher resolution, if one exists. The result of the image analysis is a classification label assigned to each image tile indicating the underlying tissue subtype, e.g., stroma-rich, stroma-poor, or background. Figure 1 shows the flowchart for the classification of stromal development. More details on the NBIA can be found in [31].

The NBIA application was originally implemented in the filter-stream programming model as a set of five components: (i) Image reader, which reads tiles from the disk; (ii) Color conversion, which converts image tiles from the RGB color space to the La*b* color space; (iii) Statistical features, which is responsible for computing a feature vector of co-occurrence statistics and LBPs; (iv) Classifier, which calculates the per-tile operations and conducts a hypothesis test to decide whether or not the classification is satisfactory; and (v) Start/Output, which controls the processing flow. The computation starts by reading tiles at the lowest resolution, and only restarts computations for tiles at a higher resolution if the classification is insufficient, according to the Classifier filter. This loop continues until all tiles have satisfactory classifications, or they are computed at the highest resolution. Since the Color conversion and Statistical feature filters are the most computationally intensive, they were implemented for GPUs. For our experiments, we started from the original implementations of the neuroblastoma applications for both GPU and CPU except that we utilize both resources simultaneously at run-time. There is no significant difference in the application code, our changes only affect the runtime support libraries.

NBIA was chosen as our motivating application because it is an important real-world and complex classification system that we believe to have a skeleton similar to a broad class of applications. Its approach to divide the image into several tiles that can be independently processed through a pipeline of operations is found, for instance, in several image processing applications. Furthermore, the multi-resolution processing strategy which concurrently generates tasks with different granularities during the execution time is also present in various research areas as computer vision, image processing, medicine, and signal processing [13, 15, 22, 28].

In our experience with the filter-stream programming model, most applications are bottleneck free, and the number of active internal tasks are higher than the available processors [8, 11, 17, 35, 41]. Thus, the proposed approach to exploit heterogeneous resources consists of allocating multiple tasks concurrently to processors where they will perform the best, as detailed in Section 3. Also, the computation time of applications' tasks dominates the overall application execution time, so the cost of the communication latency is offset by the speedups on the computation. Although these premises are valid for a broad range of applications, there is also interesting work on linear algebra computation for multicore processors [33], for instance, including heterogeneous computing with GPUs [2], where the amount of application's concurrent tasks are smaller than the number of processors. The authors improved application performance by using scheduling algorithms that take into account the task dependencies, in order to increase the number of concurrent active tasks, hence allowing them to take advantage of multicore systems.

3. ANTHILL

Anthill is a runtime system based on the dataflow model and, as such, applications are decomposed into processing stages, called *filters*, which communicate with each other using unidirectional *streams*. The application is then described as a multi-graph representing the logical interconnection of the filters [4]. At run time, Anthill spawns instances of each filter on multiple nodes of the cluster, which are called transparent copies. Anthill automatically handles run-time communication and state partitioning among transparent copies [36].

The filter programming abstraction provided by Anthill is event-oriented. The programmer provides functions that match input buffers from the multiple streams to a set of dependencies, creating event queues. Anthill run-time controls the non-blocking I/O issues that are necessary. This approach derives heavily from the message-oriented programming model [24, 6, 40].

The programmer also provides handling functions for each of the events on a filter which are automatically invoked by the runtime. These events, in the dataflow model, amount to asynchronous and independent tasks and as the filters are multithreaded, multiple tasks can be spawned provided there are pending events and compute resources. This feature is essential in exploiting the full capability of current multicore architectures, and in heterogeneous platforms it is also used to spawn tasks on multiple devices. To accomplish that, Anthill allows the user to provide for the same event multiple handlers for each specific device.

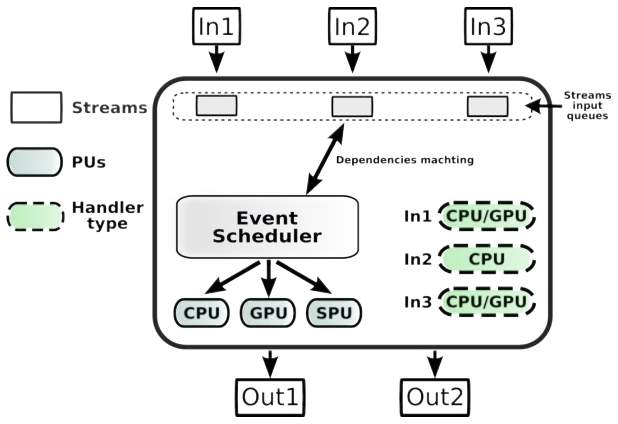


Figure 2: Filters architecture.

Figure 2 illustrates the architecture of a typical filter. It receives data from multiple input streams (*In1, In2, and In3*), each generating its own event queue, and there are handler functions associated with each of them. As shown, these functions are implemented targeting different types of processors for each event queue. The *Event Scheduler*, depicted in the picture is responsible for consuming events from the queues invoking appropriate handlers according to the availability of compute resources. As events are consumed, eventually some data is generated on the filter that needs to be forwarded to the next filter. This is done by the run-time system though it is not depicted in the figure.

The assignment of events to processors is demand-driven. Thus, when events are queued, they are not immediately assigned to a processor. Rather, this occurs on-demand as devices become idle and new events are queued. In the current implementation, the demand-driven, first-come, first-served (DDFCFS) task assignment policy is used as default strategy of the Event Scheduler.

The first decision for the DDFCFS policy is to select from which queue to execute events; this decision is made in a round-robin fashion provided there is a handling function for the available processor. Otherwise, the next queue is selected. Whenever a queue is selected, the oldest event on that queue is then dispatched. This simple approach garantees assignment to different devices according to their relative performance in a transparent way.

4. PERFORMANCE ESTIMATOR

As highlighted earlier, at the core of the techniques proposed in this paper is the fact that the relative performance of GPUs is data dependent. With that in mind, the decision about where to assign each task has to be delayed until run-time and determining its relative performance is central to our whole approach.

Despite the fact that modeling the performance of applications has been an open challenge for decades [9, 16, 39], we believe the use of relative fitness to measure performance of the same algorithm or workload running on different devices is accurate enough, and is far easier to predict than execution times. However, this task should not be left to the application programmer, but rather, should be part of the system and so we propose the *Performance Estimator*.

The proposed solution, depicted in Figure 3, uses a two-phase strategy. In the first phase, when a new application is implemented, it is benchmarked for a representative workload and the execution times are stored. The profile generated in this phase consists of the application input parameters, targeted devices and execution times, and construes a training dataset that is used during the actual performance prediction.

The second phase implements a model learning algorithm, and can employ different strategies to estimate the targeting relative performance. However, it is important to notice that modeling the behavior of applications based on their inputs is beyond any basic regression models. Also, it is beyond the scope of this paper to study this specific problem and propose a final solution. Rather, we propose an algorithm which we have validated experimentally and which was shown to yield sufficient accuracy for our decision making.

Our algorithm uses kNN [10] as the model learning algorithm. When a new application task is created, the k nearest executions in our profile are retrieved based on a distance metric on the input parameters and their execution times are averaged and used to computed the relative speedup of the task on the different processors. The employed distance metric for the numeric parameters first normalizes them, dividing each value by the highest value of each dimension, and then uses an Euclidian distance. The non-numeric attributes, on the other hand, gives a distance of 0 to attributes with

Benchmark	Speedup avg. error	CPU Time avg. error	Description	App. source
Black-Scholes	2.53196	70.5005	European option price	CUDA SDK [23]
N-body	7.34924	11.5763	Simulate bodies iterations	CUDA SDK [23]
Heart Simulation	13.7902	41.9759	Simulate electrical heart activity	[27]
kNN	8.76774	21.1858	Find k-nearest neighbors	Anthill [37]
Eclat	11.3244	102.6155	Calculate frequent itemsets	Anthill
NBIA-component	7.3812	30.3574	Neuroblastoma (Section 2)	[11, 29]

Table 1: Evaluating the performance estimator prediction.

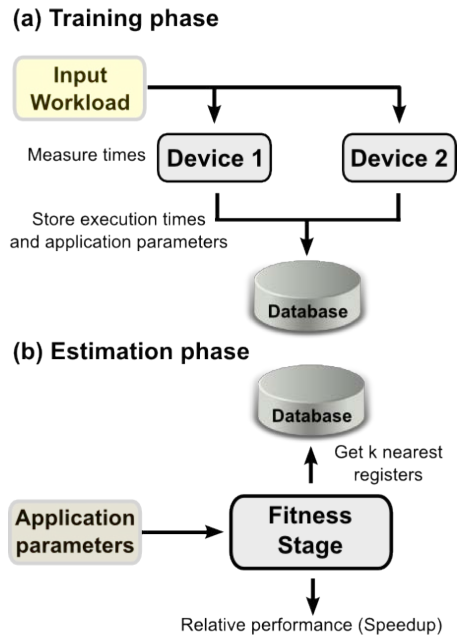


Figure 3: Anthill relative performance estimator.

a complete matching, otherwise it is 1.

For the purpose of evaluating the effectiveness of the Performance Estimator, we evaluated six representative applications (described in Table 1 with the technique described above. The evaluation has two main purposes: (i) to understand if the proposed technique performs an acceptable estimation; and, (ii) to discuss our insights that relative performance (speedup) is easier to predict accurately than execution times. The results shown were obtained by performing a first-phase benchmark using a workload of 30 jobs, which are executed on both the CPU and the GPU. The estimator errors are calculated using a 10-fold cross-validation, and $k = 2$ was utilized as it achieved near-best estimations for all configurations.

The average speedup error for each application is shown in Table 1. First of all, our methodology's accuracy was high, since the worst-case error is not higher than 14%, while the average error among all the applications is only 8.52%. In our use case application, whose accuracy was about the average, this error level does not impact the performance because it is sufficient to maintain the optimal order of the tasks. We also used the same approach to estimate task execution times for the CPU, using the same workload as before. During this second evaluation, we simply computed the predicted execution times as the average of the k nearest samples' execution times. The CPU execution time error is also shown in the same table; those errors are much higher than the speedup errors for all applications, although the same prediction methodology is employed.

This empirical evaluation is interesting for different reasons: (i) as our task assignment relies on relative performance estimation, it should perform better for this requirement than for time-based strategies; (ii) the speedup can also be used to predict execution times of an application in different runtime environments. For instance, if the execution time in one device is available, the time in a second processor could be calculated utilizing the relative performance between them. The advantage of this approach is that the estimated execution time error would be equal to the error of the predicted speedup.

We believe that relative performance is easier to predict, because it abstracts effects like conditional statements or loop breaks that highly affect the execution time modeling. Moreover, the relative performance does no try to model the application itself, but the differences between devices when running the same program.

5. PERFORMANCE OPTIMIZATIONS

In this section we discuss several run-time techniques for improving the performance of replicated dataflow computations, such as filter-stream applications, on heterogeneous clusters of CPU- and GPU-equipped machines. First, we present our approach to reduce the impact of data transfers between the CPU and the GPU by using CUDA's asynchronous copy mechanism. Next, in Section 5.2, we present a technique to better coordinate CPU and GPU utilization. Lastly, in Section 5.3, we propose a novel stream communication policy that improves the performance of heterogeneous clusters, where computing nodes have different processors, by coordinating the task assignment.

5.1 Improving CPU/GPU data transfers

The limited bandwidth between the CPU and the GPU is a critical barrier for efficient execution of GPU kernels, where for several applications the cost of data transfer operations is comparable to the computation time [34]. Moreover, this limitation has strongly influenced application design, increasing the programming challenges, and reducing collaboration opportunities between CPUs and GPUs.

An approach to reduce GPU idle time during data transfers is to overlap the data transfer with useful computation. Similar solutions have been used in other scenarios, where techniques such as double buffering are used to keep processors busy while data is transferred among memory hierarchies of multicore processors [30]. The approach employed in this work is also based on the overlapping of communication and computation, but on NVidia GPUs double buffering may not be the most appropriate technique because these devices allow multiple concurrent transfers among CPU and GPU. Thus, the problem of providing efficient data transfer becomes challenging, as the performance can be improved up to a saturation point by increasing the number of concurrent transfers; unfortunately, the optimal number of concurrent transfers varies according to the computation/communication rates of the tasks been processed and the size of the transferred data. We show this empirically in Section 6.2.

The solution of overlapping communication with computation to reduce processor idle time consists of assigning multiple concurrent processing events to the GPU, overlapping the events' data transfers with computation, and defining the number of events that maximizes the application performance. We assume that the data to be copied from the CPU to GPU are the data buffers received through the filter input streams. For the cases where the GPU kernel's input data is not self-contained in the received data buffers, the copy/format function can be rewritten according to each filter's requirements.

Algorithm 1 Algorithm to control the CPU/GPU data transfers

```
concurrentEvents = 2; streamStepSize = 2;
stopExponetialGrowth = 0;
while not $EndOfWork$ do
    for i := 0, eventId = 0; i $<$ numConcurrentEvents; i++ do
        if event $\leftarrow$ tryToGetNewEvent() then
            asyncCopy(event.data, event.GPUData, ..., event.cuStream)
            activeEvents.insert(eventId++, event)
    for i := 0; i $<$ activeEvents.size; i++ do
        $proc(event[i])$
    for i := 0; i $<$ activeEvents.size; i++ do
        event $\leftarrow$ activeEvents.getEvent(i)
        waitStream(event.cuStream)
        asyncCopy(event.outGPUData, event.outData, ..., event.cuStream)
    for i := 0; i $<$ activeEvents.size; i++ do
        event $\leftarrow$ activeEvents.getEvent(i)
        waitStream(event.cuStream)
        send(event)
        activeEvents.remove[i]
    curThroughput $\leftarrow$ calcThroughput()
    if currentThroughput $>$ lastThroughput then
        concurrentEvents += streamStepSize;
        if stopExponetialGrowth $\neq$ 1 then
            streamStepSize *= 2;
    if curThroughput $<$ lastThroughput and concurrentEvents $>$ 2 then
        concurrentEvents -= streamStepSize;
        streamStepSize /=2 ; stopExponetialGrowth = 1;
```

After copying the GPU kernel's input data to the GPU, and after running the kernel, it is typically necessary to copy the output results back to the CPU and send them downstream. During this stage, instead of copying the result itself, the programmer can use the Anthill API to send the output data to the next filter, passing the address of the data on the GPU to the runtime environment. With the pointer to this data, Anthill can transparently start an asynchronous copy of the data back to the CPU before sending it to the next filter. It is also important to highlight that our asynchronous copy mechanism uses the Stream API of CUDA SDK [23], and that each Anthill event is associated with a single CUDA stream.

Because application performance can be dramatically influenced by GPU idle time, we propose an automated approach to dynamically configure the number of concurrent, asynchronous data copies at runtime, according to the GPU tasks' performance characteristics. Our solution is based on an approach that changes the number of concurrent events assigned to the GPU according to the throughput of the application. Our algorithm (Algorithm 1) starts with two concurrent events and increases the number until the GPU's throughput begins to decrease. The previous configuration is then saved, and with the next set of events, the algorithm continues searching for a better number of concurrent data copies by starting from the saved configuration. In order to quickly find the saturation point, the algorithm increases the number of concurrent data copies exponentially until the GPU throughput begins to decrease. Thereafter, the algorithm simply makes changes by one concurrent data copy at a time.

Since current NVidia GPUs only allow concurrent transfers in one direction, our algorithm has been designed with this in mind. To maximize performance given this limitation, it dispatches multiple transfers from the CPU to the GPU, executes the event processing, and finally all transfers of the data back to the CPU, sequentially. If data transfers in each direction are not grouped, the asynchronous, concurrent data copy mechanism is not used, and the GPU driver defaults to the slower synchronous copy version. Although not depicted in the algorithm, we guarantee that the number of $concurrentEvents$ is never smaller than 1, and its maximum size is bounded by the available memory.

5.2 Intra-filter task assignment

The problem of assigning tasks in heterogeneous environments has been the target of research for a long time [1, 2, 3, 5, 14, 15, 19]. Recently, with the increasing ubiquity of GPUs in mainstream computers, the scientific community has examined the use of nodes with CPUs and GPUs in increasing detail. In Mars [12], an implementation of the MapReduce programming model for CPU- and GPU-equipped nodes, the authors evaluate the collaborative use of CPUs and GPU where the Map and Reduce tasks are divided among them, when using a fixed relative performance between the devices. The Qilin system [21] argues that the processing rates of system processors dependent on the data size. By generating a model of the processing rates for each of the processors in the system in a training phase, Qilin determines how best to split the work among the processors for successive executions of the application. However, for certain classes of applications, such as our image analysis application, the processing rates of the various processors are data-dependent, meaning that such a static partitioning will not be optimal for these cases.

Indeed, in dataflow applications, there are many internal tasks which can exhibit these data-dependent performance variations, and we experimentally show that taking these variations into account can significantly improve application performance. Heterogeneous processing has been previously studied [15], but in this work the authors target methods to map and schedule tasks onto heterogeneous, parallel resources where the task execution times do not vary according to the data. Here, we show that this data-dependent processing rate variability can be leveraged to give applications extra performance.

In order to exploit this intra-filter task heterogeneity, we then proposed and implemented a task assignment policy, called demand-driven dynamic weighted round-robin (DDWRR) [38] in the Anthill *Event Scheduler* module, previously shown in Section 3. As in DDFCFS, the assignment of events to devices is demand-driven: the ready-to-execute tasks are shared among the processors inside a single node and are only assigned when a processor becomes idle, and the first step of selecting from which stream to process events is done in round-round fashion.

The main difference between DDWRR and DDFCFS is in the second phase, when an event is chosen from the selected stream. In this phase, DDWRR chooses events according to a per-processor weight that may vary during the execution time. This value is the computed estimation of the event's performance when processed by each device. For example, this value could be the event's likely execution time speedup for this device when compared to a baseline processor (e.g., the slowest processor in the system). During the execution, this weight is then used to order ready-to-execute events for each device, and when a certain processor is available, the highest weighted event in the queue is chosen to be processed. Therefore, DDWRR assigns events in an out-of-order fashion, but instead of using it for speculative execution or to reduce the negative impact of data dependencies [25], it is used to sort the events according to its suitability for each device.

It is also important to highlight that DDWRR does not require an exact speedup value for each task because it is only necessary to have a relative ordering of events according to their performance. The estimator described in Section 4 has sufficient accuracy for our purposes.

5.3 Inter-filter optimizations: on-demand dynamic selective stream

On distributed systems, performance is heavily dependent upon the load balance as the overall execution time is that of the slowest node. Our previous techniques deals only with the events received in a single instance of a filter. To optimize globally, however, when there are multiple instances of a filter, we need to consider which of those instances should receive and process the messages we send.

We present a novel stream communication policy to optimize filter-stream computations on distributed, heterogeneous, multi-core, multi-accelerator computing environments. To fully utilize and achieve maximum performance on these systems, filter-stream applications have to satisfy two premises that motivate the proposed policy: (i) the number of data buffers at the input of each filter should be enough to keep all the processors busy, making it possible to exploit all of the available resources, but not so high as to create a load imbalance among filter instances; (ii) the data buffers sent to a filter should maximize the performance of the processors allocated to that filter instance.

Based on these premises, we propose an on-demand dynamic selective stream (ODDS) policy. This stream policy implements an $n \times m$ on-demand directed communication channel from n instances of a producer filter F_i to m instances of a consumer filter F_j. As such, ODDS implements a policy where each instance of the receiver filter F_j can consume data at different rates according to its processing power.

Because instances of F_j can consume data at different rates, it is important to determine the number of data buffers needed by each instance to keep all processors fully utilized. Moreover, as discussed previously, the number of buffers kept in the queue should be as short as possible to avoid load imbalance across computing nodes. These two requirements are, obviously, contradictory, which poses an interesting challenge. Additionally, the ideal number of data buffers in the filter's input queue may be different for each filter instance and can change as the application execution progresses as the data stream changes. Not only do the data buffers' characteristics change over time, but the communication times can vary due to the load of the sender filter instances, for example. ODDS is comprised of two components: Dynamic Queue Adaptation Algorithm (DQAA) and Data Buffer Selection Algorithm (DBSA). DQAA is responsible for premise (i), where as DBSA is responsible for premise (ii). In the next two subsections we describe these two algorithms in more detail.

5.3.1 *Dynamic Queue Adaptation Algorithm (DQAA)*

Our solution to control the queue size on the receiver side derives from concepts developed by Brakmo et al. for TCP Vegas [20], a transport protocol which controls flow and congestion in networks by continuously measuring network response (packet round trip times) and adjusting the transmission window (number of packets in transit). For our purposes, we continuously measure both the time it takes for a request message to be answered by the upstream filter instance and the time it takes for a processor to process each data buffer, as detailed in Figure 4. Based on the ratio of the request response time to the data buffer processing time, we decide whether the length of the $StreamRequestSize$ (the number of data buffers assigned to a filter instance, which includes data buffers been transferred plus received and queued), must be increased, decreased or left unaltered. The alterations are the responsibility of the thread

Algorithm 2 ThreadWorker ($proctype$, tid)

```
for all $proctype$, $targetrequestsize(tid) = 1$, $requestsize(tid) = 0$ do
  while not $EndOfWork$ do
    if $|StreamOutQueue(proctype)| > 0$ then
      $d \leftarrow$ GETDATABUFFER($StreamOutQueue(proctype)$)
      $requestsize(tid) --$
      $timetoprocess \leftarrow$ PROCESSDATABUFFER($d$)
      $targetlength \leftarrow \frac{requestlatency}{timetoprocess}$
      if $targetlength > |targetrequestsize(tid)|$ then
        $targetrequestsize(tid) ++$
      if $targetlength < |targetrequestsize(tid))|$ then
        $targetrequestsize(proctype) --$
```

Algorithm 3 ThreadRequester ($proctype$,tid)

```
while not $EndOfWork$ do
  while $|requestsize(tid)| < targetrequestsize(tid)$ do
    $p \leftarrow$ CHOOSESENDER ($proctype$)
    $sendtime \leftarrow$ TIMENOW ( )
    SENDMESSAGE(REQUESTMSG($proctype$),$p$)
    $m \leftarrow$ RECEIVEMESSAGE($p$)
    $recvtime \leftarrow$ TIMENOW ( )
    if $m \neq \emptyset$ then
      $d \leftarrow m.data$
      INSERT($StreamOutQueue$, $d$)
      $requestlatency \leftarrow recvtime - sendtime$
      $requestsize(tid) ++$
```

Figure 4: Receiver Threads

ThreadWorker, that computes its target request size after finishing the processing of each of its data buffers and updates the current request size if necessary.

In parallel, the **ThreadRequester** thread observes the changes in the $requestsize$ and the target stream request size for each **ThreadWorker**. Whenever the $requestSize$ falls below the target value, instances of the upstream filter are contacted to request more data buffers, which are received and stored in the filter $StreamOutQueue$. While these requests occur for each **ThreadWorker**, the $StreamOutQueue$ creates a single queue with the received data buffers. Once all of the buffers residing in the shared $StreamOutQueue$ have been received, the queue also maintains a queue of data buffer pointers for each processor type, sorted by the data buffers' speedup for that processor.

5.3.2 *Data Buffer Selection Algorithm (DBSA)*

Our approach for selecting a data buffer to send downstream is based on the expected speedup value when a given data buffer is processed by a certain type of processor. This algorithm is similar to the one described earlier to select a task for a given device, and it also relies on the Performance Estimator to accomplish that.

Whenever an instance of filter F_j demands more data from its input stream, the request includes information about the processor type which caused the request to be issued (because, according to Figure 4, the **ThreadRequester** will generate specific request messages for each event handler thread). Upon receipt of the data request, the upstream filter instance will select, from among the queued data buffers, the best suited for that processor type.

The algorithm we propose, which runs on the sender side of the stream, maintains a queue of data buffers that is kept sorted by the speedup for each type of processor versus the baseline processor. When the instance of F_i which received a request chooses and sends the data buffer with the highest speedup to the requesting processor, it removes the same buffer from all other sorted queues. On

Algorithm 4 ThreadBufferQueuer

```
while not $EndOfWork$ do
    if $StreamInQueue \neq \emptyset$ then
        $d \leftarrow$ GETDATABUFFER($StreamInQueue$)
        INSERTSORTED($d$,$SendQueue$)
```

Algorithm 5 ThreadBufferSender

```
while not $EndOfWork$ do
    if $\exists requestmsg$ then
        $proctype \leftarrow requestmsg.proctype$;
        $requestor \leftarrow requestmsg.sender$;
        $d \leftarrow$ DATABUFFERSELECTIONALGORITHM($SendQueue$,$proctype$);
        SENDMESSAGE(DATAMSG($d$),$requestor$);
```

Figure 5: Sender Threads

the receiver side, as stated above, a shared queue is used to minimize the load imbalance between the processors running in the system.

In Figure 5, we show the operation of DBSA. The thread **ThreadBufferQueuer** is activated each time the filter F_i's instance sends a data buffer through the stream. It inserts the buffer in the *SendQueue* of that node with the computed speedups for each type of processor. Whenever a processor running an instance of F_j requests a data buffer from that filter instance F_i, the thread **ThreadBufferSender** processes the request message, executes DBSA and sends the selected data buffer to the requesting filter F_j.

6. EXPERIMENTAL RESULTS

We have carried out our experiments on a 14-node PC cluster. Each node is equipped with one 2.13 GHz Intel Core 2 Duo CPU, 2 GB main memory, and a single NVIDIA GeForce 8800GT GPU. The cluster is interconnected using switched Gigabit Ethernet. In experiments where the GPU is used, one CPU core of the node is assigned to manage it, and is not available to run other tasks. We have run each experiment multiple times such that the maximum standard deviation is less than 3.2%, and presented the average results here. The speedups shown in this section are calculated based on the single CPU-core version of the application. We also fused the GPU NBIA filters to avoid extra overhead due to unnecessary GPU/CPU data transfers and network communication; thus, our optimizations are evaluated using an already optimized version of the application.

6.1 Effect of tile size on performance

In the first set of experiments, we analyzed the performance of the CPU and GPU versions of NBIA as a function of the input image resolution. During this evaluation, we generated different workloads using a fixed number of 26,742 tiles, while varying the tile size. We also assumed that all tiles are successfully classified at the first resolution level, so that NBIA will only compute tiles of a single resolution.

The speedup of the GPU versus one CPU core is shown, for various tile sizes, in Figure 6; these results are labeled "Synchronous copy." The results show high variations in relative performance between the CPU and the GPU: while their performance is similar for 32x32 pixel tiles, the GPU is almost 33 times faster for 512x512 pixel tiles. For small tasks, the overhead of using the GPU is proportional to the analysis execution time, making its use inefficient.

Figure 6 also shows that the performance of NBIA is strongly affected by the input tile size. Moreover, the speedup variation shows that different types of processing units have different performance capabilities. This performance heterogeneity could be observed in

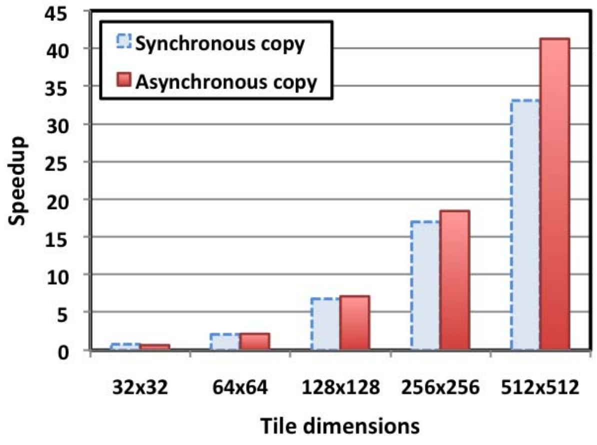


Figure 6: NBIA: synchronous vs. asynchronous copy.

other applications too, such as linear algebra kernels, data mining applications like kNN and Eclat, FFTs, and applications that employ pyramid multi-resolution representations [13, 22, 28].

In large-scale parallel executions of NBIA, multiple processing tasks will process different tile resolutions concurrently, making the performance of the devices vary according to the tasks they process. This heterogeneity creates the demand for techniques to efficiently use the CPU and GPU collaboratively, as discussed in Section 6.3.

6.2 Effect of async. CPU/GPU data transfers

The results of our approach to improve CPU/GPU data transfers are presented in this section. For the sake of our evaluation, we used two applications: NBIA, and a vector incrementer (VI), that divides a vector into small chunks which are copied to the GPU and incremented, iterating over each value six times (resulting in a computation to communication ratio of 7:3).

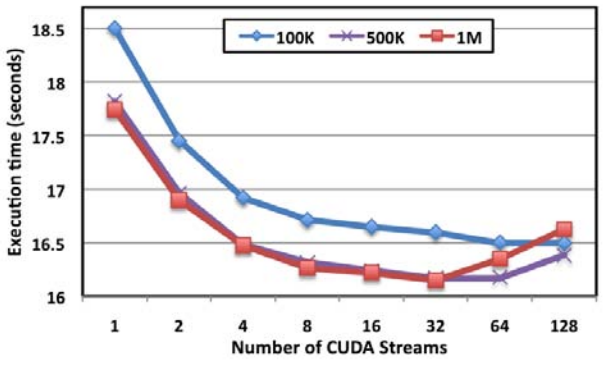


Figure 7: VI: Number of streams vs. input data chunk size.

Shown in Figure 7 are the VI execution times as we vary the number of concurrent events/CUDA streams for three data chunk sizes: 100K, 500K, and 1M, using an input vector of 360M integers. These results show that a large number of streams are necessary to attain maximum performance. Please also note that the number of CUDA streams required for optimum execution times varies with the size of data being transferred. Additionally, while we did not have a chance to run our experiments on more than one type of GPU, the memory and network bandwidth of the GPU are an important factor influencing the optimal number of CUDA streams for minimal application execution time. Therefore, on an environment with mixed GPU types, an optimal single value might not exists, even if the application parameters as were static.

	Input chunk size		
CUDA stream size	100K	500K	1M
Best static stream size (secs.)	16.50	16.16	16.15
Proposed dynamic algorithm (secs.)	16.53	16.23	16.16

Table 2: VI: Static vs. dynamic number of CUDA Streams.

We believe the results in Figure 7 strongly motivate the need for an adaptive method to control the number of concurrent CPU/GPU transfers, such as we have proposed in Section 5.1. The performance of the proposed algorithm is presented in Table 2, where it is compared to the best performance among all number of fixed number of CUDA streams. The results achieved by the dynamic algorithm are very close to the best static performance, and are within one standard deviation (near to 1%) of the average static performance numbers.

The improvement in NBIA execution time due to the use of asynchronous data copies is presented in Figure 6, where the tile size is varied (but tiles are classified at the first resolution level). The results show that NBIA performance improves for all tile sizes. For tiles of 512×512 the data transfer overhead was reduced by 83%, resulting in a gain of roughly 20% application performance.

6.3 Effect of intra-filter task assignment

This section shows how different intra-filter task assignment policies affect the performance of NBIA during large-scale execution of the application, with multiple resolution tiles concurrently active in the processing pipeline. These experiments were run using a fixed number of 26,742 tiles and two resolution levels: (32×32) and (512×512). We varied the tile recalculation rate (the percent of tiles recalculated at higher resolution) to show how it affects the performance of NBIA and our optimizations.

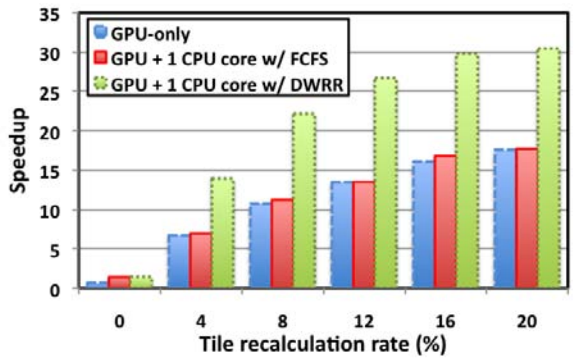


Figure 8: Intra-filter assignment policies.

Figure 8 presents the speedup of NBIA using various system configurations and intra-filter task assignment policies: GPU-only, and GPU+CPU with DDFCFS and DDWRR policies. For these experiments, DDFCFS significantly improves the application performance only for a tile recalculation rate of 0%. When no tiles are recalculated, both the CPU and the GPU process tiles of the smallest size, for which they have the same performance. Therefore, for this recalculation rate, the speedup is roughly 2 when adding a second device of any type. For reference, we also include the execution time of the CPU-only version of NBIA as the recalculation rate is varied in Table 3.

Recalc. rate (%)	0	4	8	12	16	20
Exec. time (sec)	30	350	665	974	1287	1532

Table 3: Execution time of the CPU-only version of NBIA as the recalculation rate is varied.

	Tile size	
	32×32	512×512
DDFCFS	1.52%	14.70%
DDWRR	84.63%	0.16%

Table 4: Number of tiles processed by CPU using 16% tile recalculation.

When increasing the recalculation rate however, the DDWRR policy still almost doubles the speedup of the pure GPU, while DDFCFS achieves no or only little improvement. For instance, with 16% tile recalculation, the GPU-only version of the application is 16.06 times faster than CPU version, while using the CPU and GPU together achieved speedups of: 16.78 and 29.79, for DDFCFS and DDWRR, respectively. The profile of the tasks processed by the CPU for each policy, shown in Table 4, explains the performance gap. When using DDFCFS, the CPU processed some tiles of both resolutions, while DDWRR schedules the majority of low resolution tiles to the CPU, leaving the GPU to focus on the high resolution tiles, for which it is far faster than the CPU. The overhead due to the task assignment policy, including our on-line performance estimation, was negligible.

6.4 Effect of inter-filter optimizations

In this section, we evaluate the proposed on-demand dynamic selective stream task assignment policy - ODDS. Our evaluation was conducted using two cluster configurations to understand both the impact of assigning tasks at the sender side and the capacity of ODDS to dynamically adapt the **streamRequestsSize** (the number of target data buffers necessary to keep processors busy with a minimum load imbalance). The cluster configurations are: (i) a homogeneous cluster of 14 machines equipped with one CPU and one GPU, as described in the beginning of Section 6; and (ii) a heterogeneous cluster with the same 14 machines, but turning off 7 GPUs. Thus, we have a cluster with heterogeneity among machines, where 7 nodes are CPU- and GPU-equipped machines, and 7 nodes are dual-core CPU-only machines.

In Table 5 we present three demand-driven policies (where consumer filters only get as much data as they request) used in our evaluation. All these scheduling policies maintain some minimal queue at the receiver side, such that processor idle time is avoided. Simpler policies like round-robin or random do not fit into the demand-driven paradigm, as they simply push data buffers down to the consumer filters without any knowledge of whether the data buffers are being processed efficiently. As such, we do not consider these to be good scheduling methods, and we exclude them from our evaluation.

The First-Come, First-Served (DDFCFS) policy simply maintains FIFO queues of data buffers on both ends of the stream, and a filter instance requesting data will get whatever data buffer is next out of the queue. The DDWRR uses this same technique as DDFCFS on the sender side, but sorts its receiver-side queue of data buffers by the relative speedup to give the highest-performing data buffers to each processor. Both, DDFCFS and DDWRR, have a static value for requests for data buffers during execution, which is chosen by the programmer. For ODDS, discussed in Section 5.3, the sender and receiver queues are sorted by speedup and the receiver's number of requests for data buffers is dynamically calculated at runtime.

Demand-driven Scheduling Policy	Area of effect	Queue Policy		Size of request for data buffers
		Sender	Receiver	
DDFCFS	Intra-filter	Unsorted	Unsorted	Static
DDWRR	Intra-filter	Unsorted	Sorted by speedup	Static
ODDS	Inter-filter	Sorted by speedup	Sorted by speedup	Dynamic

Table 5: Different demand-driven scheduling policies used in Section 6.

6.4.1 Homogeneous cluster base case

This section presents the results of experiments run in the homogeneous cluster base case, which consists of a single CPU/GPU-equipped machine. In these experiments, we compared ODDS to DDWRR. DDWRR is the only one used for comparison because it achieved the best performance among the intra-filter task assignment policies (see Section 6.3). These experiments used NBIA with asynchronous copy, and 26,742 image tiles with two resolution levels, as in Section 6.3, and the tile recalculation rate is varied.

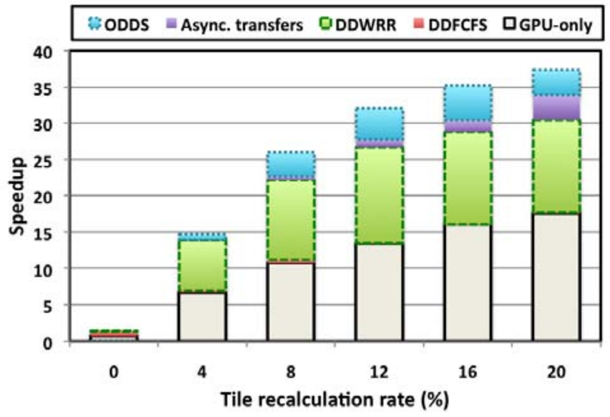


Figure 9: Homogeneous base case evaluation.

The results, presented in Figure 9, surprisingly show that even for one processing node ODDS could surpass the performance allowed by DDWRR. The gains due of Async. transfers and ODDS to DDWRR using 20% of tile recalculation, for instance, is around 23%. The improvements obtained by ODDS are directly related to the ability to better select data buffers that maximize the performance of the target processing units. It occurs even for one processing machine because the data buffers are queued at the sender side for both policies, but ODDS selects the data buffers that maximize the performance of all processors of the receiver, improving the ability of the receiver filter to better assign tasks locally.

6.4.2 Heterogeneous cluster base case

The demand-driven stream task assignment policies are again evaluated in this section, where the base case consists of two computing nodes: the first equipped with one CPU and one GPU, and the second being a dual-core CPU-only machine. Figure 10 presents the speedups for each stream policy as the tile recalculation rate is varied.

When comparing the results for the homogeneous cluster base case vs. the heterogeneous cluster base case, shown in Figures 9 and 10, respectively, notice that DDFCFS achieves slightly better performance with the additional dual-core CPU of the second computing node, so does DDWRR. However, the performance of ODDS increased significantly. For instance, at 8% recalculation rate, DDWRR and ODDS achieve, respectively, 23x and 25x the performance of a single CPU-core on the homogeneous base case

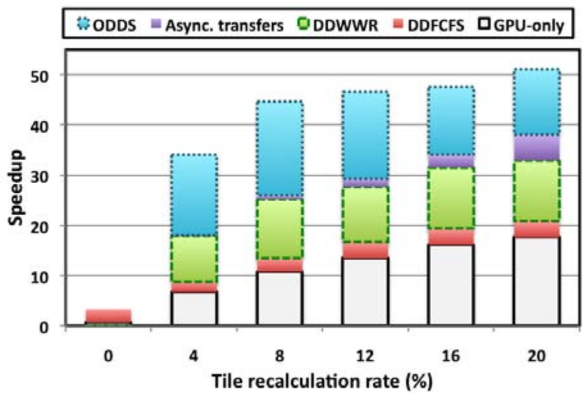


Figure 10: Heterogeneous base case evaluation.

cluster; on two heterogeneous nodes, DDWRR's speedup slightly increases to 25, while the performance of ODDS increases to a speedup of 44.

To understand how the computation is distributed in these experiments, we next present in Table 6 the profile of the data buffers processed by the GPU when using each stream, at an 8% tile recalculation rate. For the homogeneous base case experiments, it is notable that the performance difference between DDFCFS and DDWRR and ODDS occurred because in the DDFCFS scheme the CPU did not significantly collaborate in the execution. That is, 92-98% of both the low resolution and the high resolution tiles are processed by the GPU, leaving nothing much for the CPU to do. However, the DDWRR and ODDS algorithms show a preference to give the GPU the vast majority of the high resolution buffers, and save the majority of the low resolution buffers for the CPU.

The profiles' comparison with one and two nodes is also useful to understand the impact of adding an extra CPU-only node to the performance. The DDFCFS performance gains are simply because in the configuration with two nodes the CPU was able to process a slightly higher proportion of tiles at both resolutions. The DDWRR scheduling scheme, on the other hand, could not efficiently utilize the second node. As shown in Table 6, under DDWRR, the GPU processed almost the same number of low resolution tiles and few more high resolution tiles than when using only one machine. When the ODDS approach is utilized, since the decision about where each buffer should be sent is made initially at the sender, ODDS was able to intelligently utilize the second additional CPU for the processing of the remaining low resolution tiles as well as a few high resolution tiles.

An important factor in the preceding heterogeneous base case experiments is the choice of the number of the data buffer requests that maximizes performance. In Figure 11, we show the number of requests that gives the best execution time for each stream policy and tile recalculation rate. These values were determined via exhaustive search. For policies with static numbers of requests, the programmer is responsible for determining this parameter.

The DDWRR stream approach achieved better performance for

Config.	Homogeneous base case			Heterogeneous base case		
Scheduling	DDFCFS	DDWRR	ODDS	DDFCFS	DDWRR	ODDS
Low res.(%)	98.16	17.07	6.98	84.85	16.72	0
High res.(%)	92.42	96.34	97.89	85.67	92.92	97.62

Table 6: Percent of tiles processed by the GPU at each resolution/stream policy.

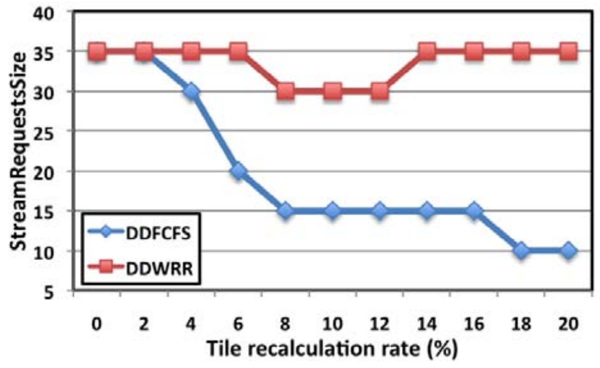


Figure 11: Best streamRequestsSize: number of data buffer requests + received by a filter.

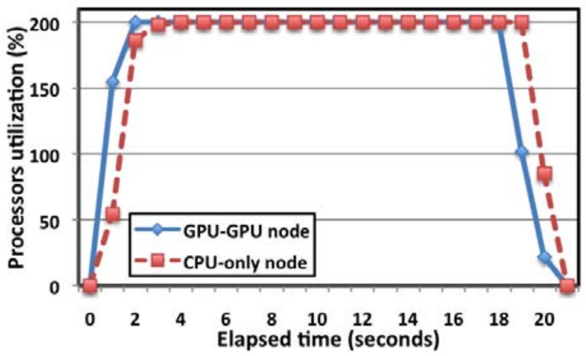
a higher number of requests as it is important for this stream policy to have a large number of data buffers on the input queue the CPU/GPU machines to create opportunities for intra-filter scheduling. DDFCFS, on the other hand, had better performance with a smaller streamRequestsSize because it results in less load imbalance among the computing nodes. For both DDFCFS and DDWRR, the best performance was achieved in a configuration where processors utilization is not maximum during the whole execution. For these policies, leaving processors idle may be better than requesting a high number of data buffers and generating load imbalance among filter instances at the end of the application's execution.

In contrast to both DDFCFS and DDWRR, ODDS can adapt the number of data requests according to the processing rate of each receiver filter instance, providing the ability to better utilize the processors during the whole execution (see Figure 12(a)). To show this, we show in Figure 12(b) how ODDS changes the streamRequestsSize dynamically in one execution of the NBIA application. This experiment used a 10% reconfiguration rate. As expected, the streamRequestsSize varies as the execution proceeds, adapting to the slack in each queue. It is especially notable that at the end of the execution where there is a build-up of higher-resolution data buffers to be computed. It is this build-up of the higher-resolution data buffers (and their longer processing times on the CPU-only machine) which causes DQAA to reduce the number of requests of the CPU-only machine, reducing the load imbalance among the computing nodes at the tail end of the application's execution.

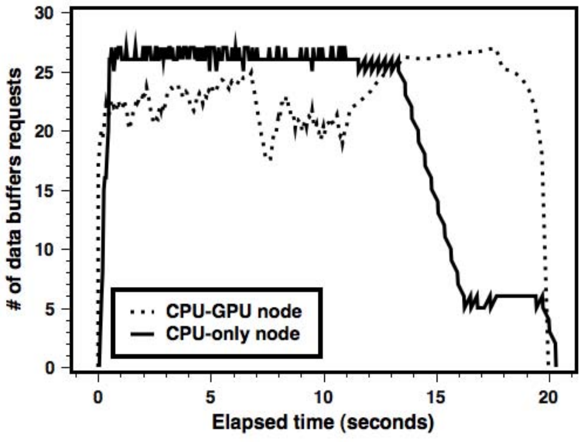
6.4.3 Scaling homogeneous and heterogeneous cases

Finally, in this section, we show NBIA performance as the number of machines is increased for each cluster type. The scaling of the homogeneous cluster was done by simply increasing the number of nodes, while for the heterogeneous cluster 50% of the computing nodes are equipped with both a CPU and a GPU, and the other 50% are without the GPUs. The results have been performed using 267,420 image tiles with two resolution levels, as before. We used an 8% tile recalculation rate, and the speedups are calculated according to a single CPU-core version of the application.

In Figure 13, we first present the NBIA speedups on the homogeneous cluster for four configurations of the application: GPU-only, and GPU and CPU collaboratively using three stream policies, presented in Table 5: DDFCFS, DDWRR, and ODDS. The



(a) CPU utilization for ODDS



(b) ODDS data requests size

Figure 12: ODDS execution detailed.

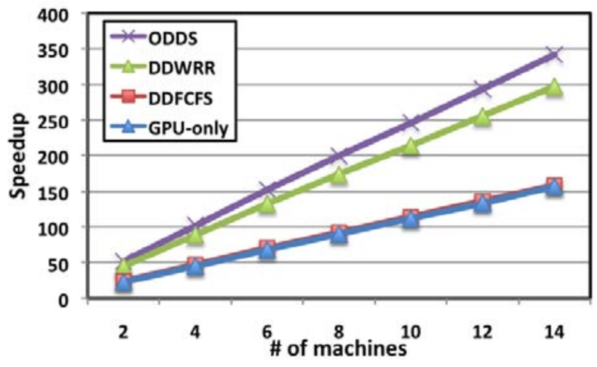


Figure 13: Scaling homogeneous base case.

results show that the DDFCFS could not improve the application performance much over the GPU-only version of the application. DDWRR, on the other hand, doubled the performance of the GPU-only configuration. Further, the ODDS was 15% faster than DDWRR even in the homogeneous environment as it can better choose which data buffers to send to requesting processors, due to its knowledge of downstream processors' performance characteristics.

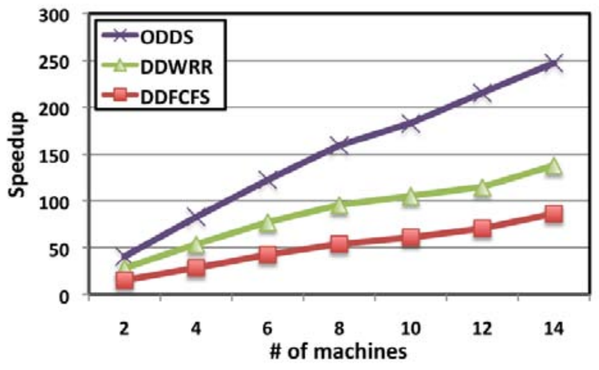


Figure 14: Scaling heterogeneous base case.

The experimental results of increasing the number of machines in the heterogeneous environment are presented in Figure 14. The ODDS again showed the best performance among the stream policies, but it now almost doubled the performance achieved by DDWRR. Once again, the DDWRR and DDFCFS results for each number of machines are the best among the different of the number of buffers requests, while ODDS automatically adapted it.

The speedup achieved by ODDS when using 14 heterogeneous nodes is also four times higher than seven GPU-only machines, showing that significant gains can be achieved by mixing heterogeneous computing nodes. The ODDS task assignment policy's better scalability is due to its ability to adapt the streamRequestsSize to the available hardware, reducing inefficiency due to load imbalance and processor under-utilization. Also, by targeting data buffers that maximize the performance of the downstream consumers, ODDS can dynamically maximize the processor performance over the whole range of data buffers in the application.

Indeed, by using a GPU cluster equipped with 14 nodes and by using ODDS to coordinate the execution of the application, the execution time of NBIA for an 8% tile recalculation rate is reduced from over 11 minutes to just over 2 seconds. Certainly, much of the speedup is due to the GPUs, but the GPU-only version of the application takes nearly 5 seconds to achieve the same analysis results. In real-world scenarios, many of these biomedical images will need to be analyzed together, and gains in performance such as those offered by ODDS can bring tangible benefits.

7. CONCLUSIONS

In this paper, we presented and evaluated several run-time optimizations for filter-stream applications on heterogeneous environments. These optimizations have the capacity to improve performance in such systems by reducing the overhead due to data transfers between CPUs and GPUs, and by coordinating appropriate and collaborative task execution on CPUs and GPUs.

The experimental results for a complex, real-world biomedical image analysis application, which exhibits data-dependent processing speedups, show that our optimizations reduce the total overhead due to data transfers between the CPU and the GPU up to 83%. Also, the appropriate coordination between CPUs and GPUs doubles the performance of the GPU-only version of the application by simply adding a single CPU-core. Moreover, our proposed ODDS stream policy provides the developer with a straightforward way to make efficient use of both homogeneous and heterogeneous clusters, arguing for the use of heterogeneous processing nodes. The proposed performance estimator was evaluated for several applications, showing good relative performance prediction, while estimating execution times directly gave poor results.

The task assignment policy proposed in this paper allocates the entire GPU for each filter internal task it has to process. As future work, we intend to consider the concurrent execution of multiple tasks on the same GPU to exploit filters' intrinsic data parallelism. This may not be only a source for performance optimization, but could also ease the development of GPU kernels, since partitioning the task among the GPU's execution units would be obviated. The performance estimator is another focus for future work, where we plan to evaluate more sophisticated model learning algorithms and to use the current prediction model in other contexts.

Acknowledgments

The authors would like to thank the five anonymous reviewers for their comments, which were very valuable. This work was partially supported by CNPq, CAPES, Fapemig, and INWeb; by the DOE grant DE-FC02-06ER2775; by AFRL/DAGSI Ohio Student-Faculty Research Fellowship RY6-OSU-08-3; by the NSF grants CNS-0643969, OCI-0904809, OCI-0904802 and CNS-0403342; and computing time from the Ohio Supercomputer Center.

8. REFERENCES

[1] Arpaci-Dusseau, R.H., Anderson, E., Treuhaft, N., Culler, D.E., Hellerstein, J.M., Patterson, D., Yelick, K.: Cluster I/O with River: Making the fast case common. In: IOPADS '99: Input/Output for Parallel and Distributed Systems (1999)

[2] Augonnet, C., Thibault, S., Namyst, R., Wacrenier, P.A.: Starpu: A unified platform for task scheduling on heterogeneous multicore architectures. In: Euro-Par '09: Proceedings of the 15th International Euro-Par Conference on Parallel Processing. pp. 863–874 (2009)

[3] Berman, F.D., Wolski, R., Figueira, S., Schopf, J., Shao, G.: Application-level scheduling on distributed heterogeneous networks. In: Supercomputing '96: Proceedings of the 1996 ACM/IEEE conference on Supercomputing. p. 39 (1996)

[4] Beynon, M., Ferreira, R., Kurc, T.M., Sussman, A., Saltz, J.H.: DataCutter: Middleware for filtering very large scientific datasets on archival storage systems. In: IEEE Symposium on Mass Storage Systems. pp. 119–134 (2000)

[5] Beynon, M.D., Kurc, T., Catalyurek, U., Chang, C., Sussman, A., Saltz, J.: Distributed processing of very large datasets with DataCutter. Parallel Comput. 27(11), 1457–1478 (2001)

[6] Bhatti, N.T., Hiltunen, M.A., Schlichting, R.D., Chiu, W.: Coyote: a system for constructing fine-grain configurable communication services. ACM Trans. Comput. Syst. 16(4), 321–366 (1998)

[7] Buck, I., Foley, T., Horn, D., Sugerman, J., Fatahalian, K., Houston, M., Hanrahan, P.: Brook for gpus: stream computing on graphics hardware. ACM Trans. Graph. 23(3), 777–786 (2004)

[8] Catalyurek, U., Beynon, M.D., Chang, C., Kurc, T., Sussman, A., Saltz, J.: The virtual microscope. IEEE Transactions on Information Technology in Biomedicine 7(4), 230–248 (2003)

[9] Fahringer, T., Zima, H.P.: A static parameter based performance prediction tool for parallel programs. In: ICS '93: Proceedings of the 7th international conference on Supercomputing. pp. 207–219 (1993)

[10] Fix, E., Hodges, J.: Discriminatory analysis, nonparametric discrimination, consistency properties. Computer science technical report, School of Aviation Medicine, Randolph Field, Texas (1951)

[11] Hartley, T.D., Catalyurek, U.V., Ruiz, A., Ujaldon, M., Igual, F., Mayo, R.: Biomedical image analysis on a cooperative cluster of gpus and multicores. In: 22nd ACM Intl. Conference on Supercomputing (Dec 2008)

[12] He, B., Fang, W., Luo, Q., Govindaraju, N.K., Wang, T.: Mars: A mapreduce framework on graphics processors. In: Parallel Architectures and Compilation Techniques (2008)

[13] Hoppe, H.: View-dependent refinement of progressive meshes. In: SIGGRAPH 97 Proc. pp. 189–198 (Aug 1997), `http://research.microsoft.com/ hoppe/`

[14] Hsu, C.H., Chen, T.L., Li, K.C.: Performance effective pre-scheduling strategy for heterogeneous grid systems in the master slave paradigm. Future Gener. Comput. Syst. (2007)

[15] Iverson, M., Ozguner, F., Follen, G.: Parallelizing existing applications in a distributed heterogeneous environment. In: 4th Heterogeneous Computing Workshop (HCW'95) (1995)

[16] Kerbyson, D.J., Alme, H.J., Hoisie, A., Petrini, F., Wasserman, H.J., Gittings, M.: Predictive performance and scalability modeling of a large-scale application. In: Supercomputing '01: Proceedings of the 2001 ACM/IEEE conference on Supercomputing (CDROM). pp. 37–37 (2001)

[17] Kurc, T., Lee, F., Agrawal, G., Catalyurek, U., Ferreira, R., Saltz, J.: Optimizing reduction computations in a distributed environment. In: SC '03: Proceedings of the 2003 ACM/IEEE conference on Supercomputing. p. 9 (2003)

[18] Lee, S., Min, S.J., Eigenmann, R.: OpenMP to GPGPU: a compiler framework for automatic translation and optimization. In: PPoPP '09: Proceedings of the 14th ACM SIGPLAN symposium on Principles and practice of parallel programming. pp. 101–110 (2009)

[19] Linderman, M.D., Collins, J.D., Wang, H., Meng, T.H.: Merge: a programming model for heterogeneous multi-core systems. SIGPLAN Not. 43(3), 287–296 (2008)

[20] Low, S., Peterson, L., Wang, L.: Understanding tcp vegas: A duality model. In: In Proceedings of ACM Sigmetrics (2001)

[21] Luk, C.K., Hong, S., Kim, H.: Qilin: Exploiting parallelism on heterogeneous multiprocessors with adaptive mapping. In: 42nd International Symposium on Microarchitecture (MICRO) (2009)

[22] Maes, F., Vandermeulen, D., Suetens, P.: Comparative evaluation of multiresolution optimization strategies for multimodality image registration by maximization of mutual information. Medical Image Analysis 3(4), 373 – 386 (1999)

[23] NVIDIA: NVIDIA CUDA SDK (2007), `http://nvidia.com/cuda`

[24] O'Malley, S.W., Peterson, L.L.: A dynamic network architecture. ACM Trans. Comput. Syst. 10(2) (1992)

[25] Patkar, N., Katsuno, A., Li, S., Maruyama, T., Savkar, S., Simone, M., Shen, G., Swami, R., Tovey, D.: Microarchitecture of hal's cpu. IEEE International Computer Conference 0, 259 (1995)

[26] Ramanujam, J.: Toward automatic parallelization and auto-tuning of affine kernels for gpus. In: Workshop on Automatic Tuning for Petascale Systems (July 2008)

[27] Rocha, B.M., Campos, F.O., Plank, G., dos Santos, R.W., Liebmann4, M., Haase, G.: Simulations of the electrical activity in the heart with graphic processing units. Accepted for publication in Eighth International Conference on Parallel Processing and Applied Mathematics (2009)

[28] Rosenfeld, A. (ed.): Multiresolution Image Processing and Analysis. Springer, Berlin (1984)

[29] Ruiz, A., Sertel, O., Ujaldon, M., Catalyurek, U., Saltz, J., Gurcan, M.: Pathological image analysis using the gpu: Stroma classification for neuroblastoma. In: Proc. of IEEE Int. Conf. on Bioinformatics and Biomedicine (2007)

[30] Sancho, J.C., Kerbyson, D.J.: Analysis of Double Buffering on two Different Multicore Architectures: Quad-core Opteron and the Cell-BE. In: International Parallel and Distributed Processing Symposium (IPDPS) (2008)

[31] Sertel, O., Kong, J., Shimada, H., Catalyurek, U.V., Saltz, J.H., Gurcan, M.N.: Computer-aided prognosis of neuroblastoma on whole-slide images: Classification of stromal development. Pattern Recognition, Special Issue on Digital Image Processing and Pattern Recognition Techniques for the Detection of Cancer 42(6) (2009)

[32] Shimada, H., Ambros, I.M., Dehner, L.P., ichi Hata, J., Joshi, V.V., Roald, B.: Terminology and morphologic criteria of neuroblastic tumors: recommendation by the international neuroblastoma pathology committee. Cancer 86(2) (1999)

[33] Song, F., YarKhan, A., Dongarra, J.: Dynamic task scheduling for linear algebra algorithms on distributed-memory multicore systems. In: SC '09: Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis (2009)

[34] Sundaram, N., Raghunathan, A., Chakradhar, S.T.: A framework for efficient and scalable execution of domain-specific templates on gpus. In: IPDPS '09: Proceedings of the 2009 IEEE International Symposium on Parallel and Distributed Processing. pp. 1–12 (2009)

[35] Tavares, T., Teodoro, G., Kurc, T., Ferreira, R., Guedes, D., Meira, W.J., Catalyurek, U., Hastings, S., Oster, S., Langella, S., Saltz, J.: An efficient and reliable scientific workflow system. IEEE International Symposium on Cluster Computing and the Grid 0, 445–452 (2007)

[36] Teodoro, G., Fireman, D., Guedes, D., Jr., W.M., Ferreira, R.: Achieving multi-level parallelism in filter-labeled stream programming model. In: The 37th International Conference on Parallel Processing (ICPP) (2008)

[37] Teodoro, G., Sachetto, R., Fireman, D., Guedes, D., Ferreira, R.: Exploiting computational resources in distributed heterogeneous platforms. In: 21st International Symposium on Computer Architecture and High Performance Computing. pp. 83–90 (2009)

[38] Teodoro, G., Sachetto, R., Sertel, O., Gurcan, M., Jr., W.M., Catalyurek, U., Ferreira, R.: Coordinating the use of GPU and CPU for improving performance of compute intensive applications. In: IEEE Cluster (2009)

[39] Vrsalovic, D.F., Siewiorek, D.P., Segall, Z.Z., Gehringer, E.F.: Performance prediction and calibration for a class of multiprocessors. IEEE Trans. Comput. 37(11) (1988)

[40] Welsh, M., Culler, D., Brewer, E.: Seda: an architecture for well-conditioned, scalable internet services. SIGOPS Oper. Syst. Rev. 35(5), 230–243 (2001)

[41] Woods, B., Clymer, B., Saltz, J., Kurc, T.: A parallel implementation of 4-dimensional haralick texture analysis for disk-resident image datasets. In: SC '04: Proceedings of the 2004 ACM/IEEE conference on Supercomputing (2004)