Compiler and Runtime Support for Enabling Generalized Reduction Computations on Heterogeneous Parallel Configurations

Vignesh T. Ravi   Wenjing Ma   Gagan Agrawal
Department of Computer Science and Engineering
The Ohio State University Columbus OH 43210
{raviv,mawe,agrawal}@cse.ohio-state.edu

Abstract

In the last 3-4 years, there has been very high research interest in effectively utilizing multi-cores for various domains of applications. At the same time, researchers have attempted to unleash the power of GPU for general-purpose computations. Nowadays, it is very common for a desktop or a notebook to be shipped with a multi-core CPU and GPU, either on-chip or connected to a PCI-Express. Today, typically when an application is ported to a multi-core architecture or a GPU, the other resource, with its high processing capabilities, remains idle.

This paper describes a compiler and runtime framework that can map a class of applications to a cluster of nodes, with a multi-core CPU and GPU on each node. The class of applications we support are the ones involving generalized reductions. Starting with C functions with added annotations, we automatically generate a cluster middleware API code, as well as the CUDA code for exploiting the GPU. The runtime system dynamically partitions the work between CPU cores and the GPU. Our experimental results from two applications, whose processing structure follows the generalized reduction structure, shows that by using a heterogeneous architecture, we can achieve significantly improved performance, as compared to the performance using only the GPU or the multi-core CPU. Moreover, dynamic partitioning of the work achieves much better performance than the two static schemes we have considered.

1. Introduction

Starting within the last 3-4 years, it is no longer possible to improve the processor performance by simply increasing clock frequencies. As a result, multi and many-core architectures and/or accelerators have become the cost-effective means for scaling performance. So far, a large effort has been made to study and understand the difficulties involved with multi-core programming. Particularly, library support and programming models are being developed for efficient programming on multi-core platforms [9, 29].

[Copyright notice will appear here once 'preprint' option is removed.]

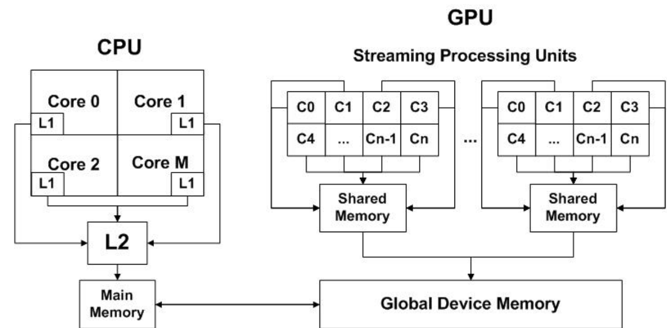

Figure 1. A Popular Heterogeneous Computing Platform

At the same time, many research efforts have focused on using the high computing power of graphics hardware (GPUs) for general-purpose computations [4, 30, 13, 33]. While a variety of application classes have been very successfully mapped to GPUs, programmability remains a challenge. Nvidia's CUDA, the most widely used programming language for GPUs to date [27], requires relatively low-level programming and manual memory management. Furthermore, obtaining high performance on a GPU clearly requires a very detailed understanding of the GPU architecture, and memory hierarchy optimizations.

One trend that is emerging, but has received very limited attention, is that computing platforms are becoming increasingly heterogeneous. Nowadays, it is very common for a desktop or a notebook to be shipped with a multi-core CPU and GPU(s), either on-chip or connected to the PCI-Express card. Figure 1 shows this architecture. Recently, the Larrabee architecture proposed by Intel [32] has indicated that many-core CPU/GPU can be a promising platform for optimized data-parallel processing. Furthermore, there is also a trend towards clusters of multi-core machines with a GPU on each node. One example is clusters used for visualization of very large datasets, where a GPU on each node is needed for supporting graphics operations. Such a cluster can have excellent performance/price ratio for general purpose processing as well.

It has been shown that for most applications, and with some programming effort, we can effectively utilize either the multi-core CPU, or the GPU, separately. But, in almost all studies so far, while doing so, the other resource, with its very high peak performance, remains idle.

Effectively exploiting a heterogeneous architecture involves a number of challenges. Particularly, three important issues are, *programmability*, *work distribution*, and *performance*.

Programmability: Today, an effort which wants to exploit the power of a multi-core and a GPU will require programming in a combination of OpenMP or Pthreads with CUDA. Besides high programming effort, this requires programmer(s) with expertise in both. Programming a cluster of heterogeneous cores is even more challenging. OpenCL, initially developed by Apple, and now being standardized by Khronos group, is still in its very early stages [19]. Furthermore, it still requires explicit parallel programming. Recently, Saha *et. al* [31] have proposed a programming model for heterogeneous parallel computing, but it still requires low-level programming and memory management. Clearly, it will be desirable if a heterogeneous platform can be used starting from a high-level language or API.

Work Distribution: Dividing work between processing units in a heterogeneous setting is challenging. Furthermore, today's CPU core and GPU core differ considerably in processing capabilities, memory availability, and communication latencies. Developing schemes that can work across many different applications is clearly desirable.

Performance: Given a lack of efforts in this area, it is not clear what kind of performance gain can be obtained from such heterogeneous configurations, and for what kind of applications. Venkatasubramanian *et. al* [34] have implemented stencil kernels on a hybrid CPU/GPU system. Though they were not able to get much performance gain out of the hybrid system, they believe that performance gains are possible with a different hardware configuration, based on their prediction model. Understanding the performance potential of such platforms requires detailed experimentation with more applications.

This paper addresses the above three issues, focusing on a particular class of applications. The applications we consider are the ones where the communication pattern is limited to *generalized reductions* [14]. In recent years, they have also been referred to as the *map-reduce* class of applications. Applications with these patterns have been considered as one of the *dwarfs* in the Berkeley view on parallel processing [1].

In our work, we have developed compiler and run-time support for heterogeneous configurations. Applications following the generalized reduction structure can be programmed using a simple C interface, where some additional annotations are needed. Our code generation system automatically generates the CUDA code, and the required API for an existing runtime system, which allows execution on cluster of multi-core nodes. The framework internally handles the mapping of CUDA code to GPU and multi-core code to CPUs. The memory management involved with the GPU is also handled by our code generation. Dynamic load balancing support is extended by our runtime system for effective distribution of workload between the CPU and the GPU.

Our evaluation has been performed using two popular data mining algorithms. We show that the heterogeneous configuration can improve the performance significantly over the CPU-only and GPU-only results. We have also studied the factors that are critical to the performance of the heterogeneous version. Our dynamic load balancing scheme performs better than the two static policies we have considered, and achieves a near-perfect work distribution between the two computing resources, i.e. the CPU and the GPU.

The rest of the paper is organized as follows. In Section 2 we describe our approach on handling reduction based computations for heterogeneous configurations. In Section 3, we elaborate on the language and compiler support we provide. In Section 4, we present our experimental results using two applications. In Section 5, we discuss the related work and conclude in Section 6.

[1] Please see http://view.eecs.berkeley.edu/wiki/Dwarf_Mine

2. Approach and System Design

This section gives a description of the class of generalized reduction applications we focus on, a cluster middleware we had developed in the past, and our approach for supporting this class of applications on a heterogeneous platform.

2.1 Generalized Reductions and a Cluster Middleware

```
/ * Outer Sequential Loop * /
While () {
    / * Reduction Loop * /
    Foreach (element e) {
        (i,val) = process(e);
        Reduc(i) = Reduc(i) op val;
    }
 / * operation on the combined Reduc * /
 Finalize();
}
```

Figure 2. *Generalized Reduction* Processing Structure

The processing structure we focus on is summarized in Figure 2. The function *op* is an associative and commutative function. Thus, the iterations of the *foreach* loop can be performed in any order. The data-structure *Reduc* is referred to as the reduction object. The reduction performed is, however, *irregular*, in the sense that which elements of the reduction objects are updated depends upon the results of the processing of an element.

In our earlier work, we had made the observation that parallel versions of several well-known data mining, OLAP, and scientific data processing algorithms share this generalized reduction structure [17, 18]. This observation has some similarities with the motivation for the *map-reduce* paradigm that Google has developed [9].

For algorithms following such generalized reduction structure, parallelization can be done on both shared memory and distributed memory platforms by dividing the data instances among the processing threads and/or nodes. A copy of the reduction object is created for each thread. The computation performed by each thread will be iterative and will involve reading the data instances in an arbitrary order, processing each data instance, and performing a *local reduction*. After such local processing, a global combination is performed. In distributed memory settings, such global combination requires interprocessor communication.

In our earlier work, we had developed a middleware focusing on these applications, targeting cluster of multi-core machines [17, 18]. The middleware API exploits the common processing structure, and the middleware enables parallelization on both shared memory and distributed memory configurations. In the middleware API, the following functions need to be written:

Reduction: The data instances owned by a processor and belonging to the subset specified are read. A reduction function specifies how, after processing one data instance, a *reduction object* (initially declared by the programmer), is updated. The result of this processing must be independent of the order in which data instances are processed on each processor. The order in which data instances are read from the disks is determined by the runtime system.

ProcessNextIteration: This is an optional function that can be implemented by an application programmer. Typically, this function should include a program logic specific to an application that would control the number of iterations for which the application should run.

Finalize: After final results from multiple nodes are combined into a single reduction object, the application programmer can read and

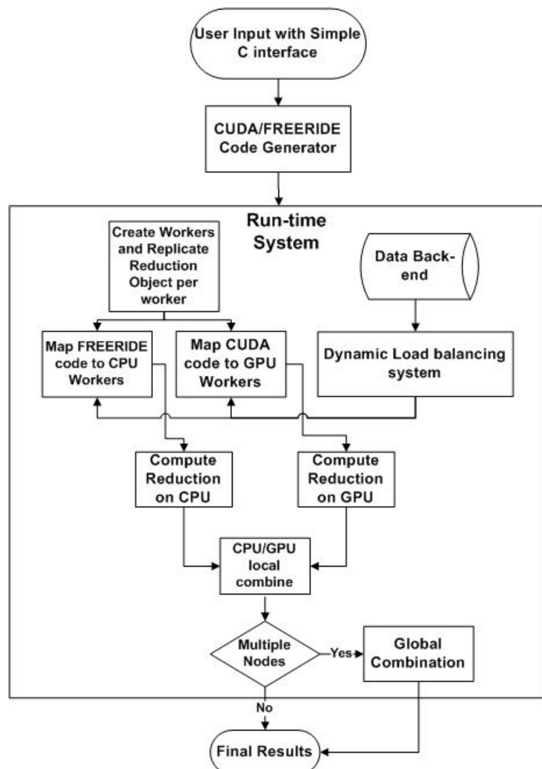

Figure 3. High-level Architecture of the System

perform a final manipulation on the reduction object to summarize the results, in a manner that is specific to the application.

2.2 GPU Computing for Generalized Reductions

GPUs support SIMD shared memory programming. For shared memory systems, one simple approach for avoiding race conditions is that each thread keeps its own replica of the reduction object on the device memory, and the work is done separately by each thread. At the end of each iteration, a global combination is done either by a single thread, or using a tree structure and involving a large number of threads. Then, the finalized reduction objects are copied to host memory.

Three steps are then involved in the local reduction phase: read a data block, compute a reduction object update based on the data instance, and write the reduction object update. A more detailed approach of what has to be performed on the GPU is as follows:

Data read: The data to be processed is copied from host to device memory, followed by allocation of reduction objects and other data structures to be used during the course of computation.

Computing update: Multi-threaded reduction operation executed on the device. The data block is divided into small blocks such that each thread only processes 1 data transaction. Since each thread has its own replica of the reduction object, race conditions while accessing the same object are avoided.

Writing update: Copy the reduction objects back to host memory, and do a global combination if necessary.

2.3 Approach for a Heterogeneous Platform

Based on the approaches for parallelizing generalized reductions on a cluster (using our middleware) and on a GPU, as we described above, we have developed a framework for mapping these applica-

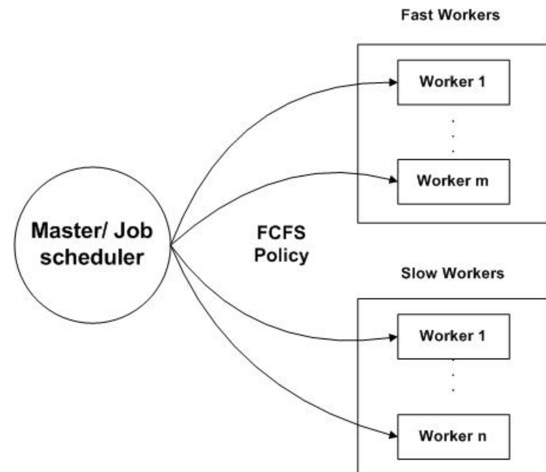

Figure 4. Dynamic Load Balancing Scheme

tions to a heterogeneous platform. The approach and the design of the system for enabling it are shown in Figure 3.

Our system takes generalized reductions written in C with some added annotations. A code generation module automatically generates the API code for our cluster middleware, and CUDA code. The language and compiler approach is elaborated in Section 3.

On each node of the cluster, our runtime system creates one thread for each CPU core to be used for processing, and another thread for managing the GPU. A separate copy of the reduction object is created for each thread. A key component of the runtime system is the dynamic load balancing module, which divides the work between different threads, including the thread that manages the GPU. This module is further elaborated on in Subsection 2.4.

After the processing is finished by different CPU threads, a local combination is performed to obtain the final results on this node. If the processing is being performed on a cluster, this is followed by a global combination of the reduction object.

2.4 Dynamic Load balancing Scheme

One of the most important challenges is the efficient work load balancing. This is a difficult problem, because CPU and GPU cores have different capabilities, and their relative performance can depend upon the application.

Clearly, there are two alternatives for load balancing, i.e. a static scheme or a dynamic scheme. We believe that a static scheme cannot be effective, and thus, a dynamic scheme is needed. Therefore, we have taken a simple and light-weight dynamic load balancing approach. Later in this paper, we will show that this scheme clearly outperforms two static options we have considered.

Our dynamic load balancing scheme resembles a classical *master-slave* model, implemented with a *pull* strategy. The dataset is divided into *chunks*. The scheduling decision is made for each chunk, thus, the chunk size or the *split size* is an important parameter that can impact the performance of the application. The master acts as a job scheduler. The scheduling policy used by the master is a simple First Come First Served (FCFS) policy. Instead of master flooding the workers with jobs, the workers pull jobs from master whenever they are finished with the assigned work. The master assigns the jobs in the order of request. The rationale behind choosing such a simple policy is that, a faster worker ends up requesting more data to process when compared to the slower worker. This approach makes sure that a faster worker gets a fair share of work for its superior speed, while a reasonable amount of work is also

completed by a slower worker. By keeping the policy simple, the overhead of load balancing remains insignificant.

Each worker in the system has a private work queue. When the worker polls the scheduler for more work, the scheduler fills the queue of the worker. The worker retrieves the job from the queue for further processing. The choice of separate queue for each worker is justified, since this will avoid any contention among the workers for the work. A simple description of this load balancing scheme is shown in the Figure 4.

3. Language Support and Code Generation

This section describes how the code for our cluster middleware and CUDA is automatically generated. Initially, we describe the input that we expect from application developers.

3.1 System API

Using the generalized reduction structure of our target class of applications, we provide a convenient API for the user. There are multiple reduction functions, and a user can specify them by including `labels` for each. For each function, the following information is needed.

Variables for Computing: The declaration of each variable follows the following format:

`name, type, size[value]`

`name` is the name of the variable, `type` can be either a numeric type like `int` or pointer type like `int*` , which indicates an array. If this is a pointer, `size` is the size of the array, which can be a list of numbers and/or integer variables, and the size of the array is the multiplication of these terms; otherwise, this field denotes a default `value`.

Sequential Reduction Function: The user can write the sequential code for the main loop of the reduction operation in C. Any variable declared inside the reduction function should also appear in the variable list, and memory allocation for these variables is not needed.

User Defined Finalize Function: After the reduction objects are combined at the end of each iteration, there might be some extra work to do with the reduction objects. This work can be done by providing a finalize function.

3.2 Program Analysis

There are two main components in the program analyzer, the code analyzer and the variable analyzer. The code analyzer accomplishes two tasks: obtaining the access pattern and extracting the reduction objects with their combination operation.

These two tasks are performed in the following way:

Obtaining Variable Access Features: We classify each variable as one of `input`, `output` and `temporary`. An `input` variable is input to the reduction function, which is not updated in the function, and does not need to be returned. An `output` variable is updated and to be returned in the reduction function. A `temporary` variable is declared inside the reduction function for temporary storage. Thus, an `input` variable is *read-only*, and `output` and `temporary` variables are *read-write*. Variables with different access patterns are treated differently in declaration, memory allocation strategies, and result combination, as described in the rest of this section.

Such information can usually be obtained from simple inspection of a function. However, since we are supporting C language, complications can arise because of the use of pointers and aliasing. In our implementation, we first generate an Intermediate Representation (IR) for the sequential reduction function using LLVM. Second, we used Anderson's point-to analysis [3] to obtain the `point-to` set for each variable in the function's argument list. Finally, we trace the entire function. When a `store` operation is found, if the destination of the store belongs to a `point-to` set of any variable in the function's argument list, and the source is not in the same set, we conclude that it is an `output` variable. All the other variables in the argument list are denoted as `input` variables, and all the variables that do not appear in the argument list are considered `temporary` variables.

```
void kmeans_count(float* data, float* cluster, float* update,
int k, int n)
{
   for(int i=0;i<5*k;i++)update[i]=0; /* initialize the output variable */
   for(int i=0;i<n;i++)
   {
       float min=65536*65, dis;
       float* mydata=data+i*DIM;
       int min_index=0;
       for (int i=0;i<k;i++) {
            float x1,x2,x3;
            x1 = cluster[i*DIM];
            x2 = cluster[i*DIM+1];
            x3 = cluster[i*DIM+2];
            dis = sqrt( (mydata[0]-x1)* (mydata[0]-x1)+
            (mydata[1]-x2)* (mydata[1]-x2)+
            (mydata[2]-x3)* (mydata[2]-x3) );
            if (dis<min)
            {
               min=dis;
               min_index=i;
         /* find the cluster with minimum distance */
            }
       }
       /* update the output variable */
       update[5*min_index] += mydata[0];
       update[5*min_index+1] += mydata[1];
       update[5*min_index+2] += mydata[2];
       update[5*min_index+3] += 1;
       update[5*min_index+4] += min;
   }
}
```

Figure 5. User-defined Reduction Function for K-means

Extracting Reduction Objects and Combination Operations: The `output` variables are identified as the reduction objects. At the end of each iteration, the reduction objects on each node are combined into a single one, by using the MPI calls automatically invoked by our cluster middleware. Because we are focusing on reduction functions where `output` variables are updated with associative and commutative functions only (see Figure 2), the output variables updated by each computing node (and different threads in GPU) can be correctly combined in the end. However, we need to identify the particular operator that is being used. Earlier, we have generated the `point-to` sets for each parameter of the reduction function. We now conduct a new scan on the IR to find the reduction operator for each reduction object. In the combination function, the values for a reduction object from each thread is combined using this function.

After the above information has been extracted, the variable analyzer will proceed to summarize the variable information and extract the parallel loops.

Analysis for Parallelization: We map the structure of the loop being analyzed to the canonical reduction loop we had shown earlier in Figure 2. We focus on the main outer loop and extract the *loop variable*. We also identify (symbolically) the number of iterations in the loop, and denote it as *num_iter*. If there are nested loops, for simplicity, we only parallelize the outer loop.

The variable analyzer focuses on the variables accessed in the loop. If a variable is only accessed with an affine subscript of the loop variable, it is denoted as a *loop* variable. Note that this variable could be an input, output, or temporary variable. The significance of denoting it is that when run on GPU, a *loop* variable can be distributed among the threads, while all the other variables need to be replicated, if they are written in the loop.

3.3 Code Generation for Cluster Middleware

The issues in generating code for the API of our cluster middleware, which we had described earlier in Section 2.1, are as follows. The base class for any application is a template MIDDLEWARE_TECH. For a particular application, we derive its corresponding class from MIDDLEWARE_TECH, with the variables in each reduction function declared as class members. There are three main functions in the class. We discuss the code generation for each of them as below.

Initialization: After variable analysis, we already know which variables form the reduction object. In the Initialization() function, these variables are declared and initialized with the default values given by the user. Arrays are allocated with the given size. One thing that needs attention for a heterogeneous version is the reduction objects that are to be computed with CUDA. Since each thread needs its own copy, the size of the variable is the declared size multiplied with a *block number* and a *thread number* within the block. The *block* here refers to thread block in CUDA.

Reduction: The Reduction() function is the main processing function for the data blocks. The computation in the sequential reduction function given by the user is included in this function. At the end of the function, the reduction objects are updated with the output of the local reduction. For each reduction function, the user can denote whether to use GPU or not in the input file. If GPU is chosen, a CUDA version for the reduction function is generated, as described in the next subsection.

Finalize: As described previously, after one iteration, every data block has been processed, and the reduction objects have been combined with MPI message passing at the back end. Thus, they are copied to the corresponding local variables, and the user provided functions are added after that, if any.

```
void reduc_class::kmeans(void *block)
{
   float* data=(float*)block;
   kmeans_func(step,endcondition,k,n,
   MSE,data,update,cluster);
   for (int RO_i=0;RO_i<1;RO_i++)
   {
        for (int RO_j=0;RO_j<1*5*k;RO_j++)
              reductionobject->reduction(RO_i,RO_j,
              update[RO_j]);
   }
}
```

Figure 6. System Generated Reduction Function of K-means

To show how the code generation is done, let us take k-means as an example. A part of the user input was shown earlier in Figure 5. After code analysis, we find that `update` is an *output* variable, so it is determined as the reduction object for this reduction function. Then, reduction object is allocated according to its size. In the system generated code, *reductionobject* is updated with the value of *update*, as shown in Figure 6.

3.4 Code Generation for CUDA

Using the user input and the information extracted by the variable and code analyzer, the system next generates corresponding CUDA code and the host functions invoking CUDA-based parallel reductions.

Grid Configuration and Kernel Invocation: The host reduction function `host_reduc()` which invokes the kernel on device has 3 parts:

`Declare and Copy`: We allocate device memory for variables to be used by the computing function on the GPU. We copy the ones that are needed to be read from host memory to device memory. Currently, we allocate memory for all variables except the `temporary` variables that are going to use shared memory. As we described earlier, *loop* variables are distributed across threads, depending upon how they are accessed across iterations. The read-write variables not denoted as *loop* might be updated simultaneously by multiple threads, so we create a copy for each thread. Again, because of the nature of the loops we are focusing on, we can assume that a combination function can produce the correct final value of these variables.

`Compute`: We configure the thread grid on the device, and invoke the kernel function. Different thread grid configurations can be used for different reduction functions in one application. Currently, we configure the thread grid manually. In our future work, we hope to develop cost models that allow us to configure thread grids automatically.

`Copy updates`: We copy the variables needed by the host function. We perform the global combination for `output` variables which are not *loop* variables.

Generating Kernel Code: This task includes generating global function reduc() and device function device_reduc(), as well as device functions init() and combine(), if necessary. reduc() is the global function to be invoked by the host reduction function. It performs the initialization for the variables involved. The device main loop function device_reduc() is then invoked. Finally, one thread will execute combine() which performs the global combination. Between invocation of each function and at the end of reduc(), a __syncthreads() is inserted.

Generating Local Reduction Function: device_reduc() is the main loop to be executed on the GPU. This function is generated by rewriting the original sequential code in the user input, according to the information generated by the code and variable analyzer phases. The modifications include: 1) Dividing the loop to be parallelized by the number of blocks and number of threads in each block. 2) Rewriting the index of the array which are distributed. For example, we have an access to `data[i]`, it is changed to `data[i+index_n]`, where `index_n` is the offset for each thread in the entire grid. 3) Optimizing the use of shared memory. We sorted the variables according to their sizes, and allocate shared memory for variables in the increasing order, until no variable can fit in. The details of the shared memory layout strategy can be found in our previous work [26].

4. Experimental Results

In this section, we report on the results from a number of experiments we conducted for evaluating the performance scalability for two applications that involve generalized reductions. The main goals of our experiments are as follows.

- Examine the performance benefits of using the heterogeneous platform, compared to parallel but CPU-only and GPU-only versions.
- Study how varying input data *chunk* or *split* size impacts the performance on heterogeneous platform.
- Show how the dynamic load balancing scheme compares against two different static schemes, and how the distribution

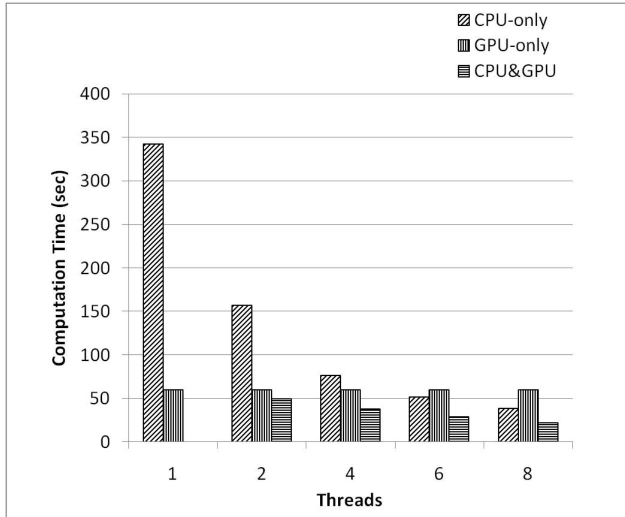

Figure 7. Scalability of k-means - Base Version

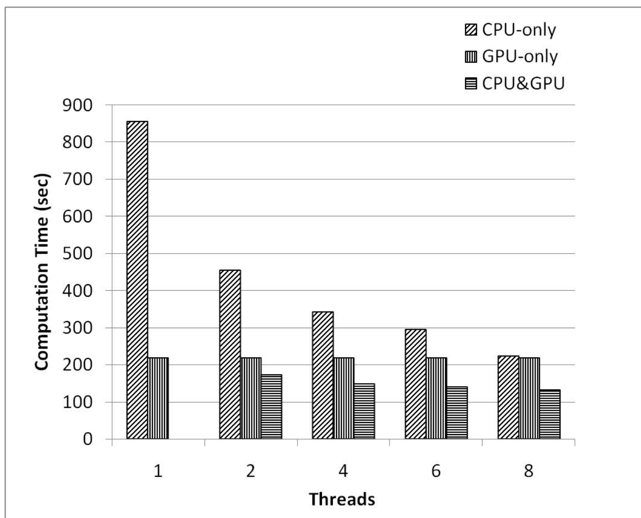

Figure 8. Scalability of PCA - Base Version

of the work between CPU and GPU is impacted by the choice of the chunk size.

- Finally, we evaluate the scalability of these two applications on a cluster of 8 nodes, with each node having a multi-core CPU and a GPU.

Our experiments were conducted on a 8 node cluster with AMD Opteron 8350 machines, each with 8 cores. Each of the nodes is equipped with a GeForce 9800 GX2 graphic card. The amount of memory on each node is 16 GB, and the interconnect network in the cluster is Infiniband.

The two applications we considered are as follows. The first application is k-means clustering. Clustering is one of the key data mining problems and k-means [16] is one of the most popular algorithms. The clustering problem is as follows. We consider transactions or data instances as representing points in a high-dimensional space. Proximity within this space is used as the criterion for classifying the points into clusters. Four steps in the sequential version of k-means clustering algorithm are as follows: 1) start with `k` given centers for clusters; 2) scan the data instances, for each data instance (point), find the center closest to it and assign this point to the corresponding cluster, 3) determine the `k` centroids from the points assigned to the corresponding center, and 4) repeat this process until the assignment of points to cluster does not change.

The second application is principal component analysis or (PCA), which is a popular dimensionality reduction method. This method was developed by Pearson in 1901. There are three passes on the dataset. First, the `mean` value of the column vectors are determined. Next, the `standard deviation` of column vectors are calculated. In the third pass, the correlation matrix is computed, and then, triangular decomposition is done, and the eigenvalues are computed.

4.1 Scalability of Applications on the Heterogeneous Platform

In this experiment, we compare three different versions. The first is the `CPU-only` version, the second is `GPU-only` version, and the third is the `hetero` or heterogeneous version, where a combination of one GPU and certain number of CPU cores are used. When not stated otherwise explicitly, the `hetero` version would refer to the use of 8 CPU threads and the GPU.

For all results with GPU, we report results that are obtained only from the best thread block configuration. In the `hetero` version, if K CPU cores are used, only $K - 1$ cores are used for actual computations, while the last core is used for coordination with the GPU thread.

Results from K-means: For experimenting with k-means, we used a dataset of size 6.4 GB. The dataset contains 3-dimensional points. The number of clusters, k, was set to 125. For this set of experiments, the dataset was divided into 512 chunks. The results are presented in the Figure 7. The GPU-only version is identical irrespective of the number of threads. With 1 thread, the `hetero` version cannot be executed. Thus, results for the `hetero` version are shown 2 thread onwards only.

Our results show that, with the CPU-only version, there is a very good scalability with the increasing number of cores. The CPU-only version speedup with 8 threads is about 8.8 times. This super-linear speedup is because of the very small size of the reduction object in k-means, that leads to a very effective utilization of caches. The GPU-only version also has a very good speedup when compared to the single thread CPU version, i.e. 5.71 times, including all data movement times between the host and device and vice-versa.

The `hetero` version is able to get a significant performance improvement when compared to the CPU-only and GPU-only versions. The speedup achieved with 8 CPU cores and 1 GPU, when compared to a single thread CPU version and GPU-only version are 15.6 and 2.72, respectively. Moreover, as compared to the performance of the 8-thread CPU-only version, there is more than a factor of 2 performance improvement. The speedups are very good because, both CPU-only and GPU-only versions have very good scalability. Hence, one resource does not turnout to be a bottleneck for the other in this application. Moreover, since the size of reduction object is very small, the overhead of local combination phase between CPU and GPU is also quite low.

Results from Principle Component Analysis (PCA): For experiments with PCA, we used a dataset of size 2.1 GB. The entire dataset was divided into 512 chunks. The number of columns, m, was set to 64. The observed results from PCA are shown in Figure 8. The results from PCA are quite different from the results with k-means. First, the scalability achieved with CPU-only version is not linear. The speedup achieved with 8 CPU threads, when compared to a single thread version, is about 3.8 only. This is because, the algorithm runs for three passes. The first and the second passes have very little computation when compared to the third

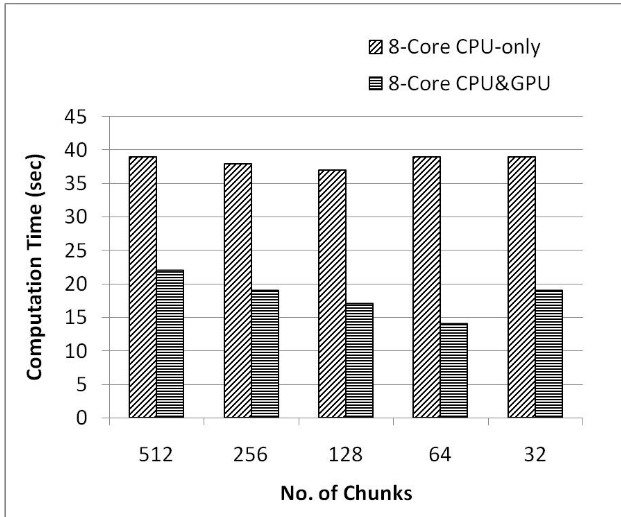

Figure 9. k-means - Impact of Varying No. of Chunks

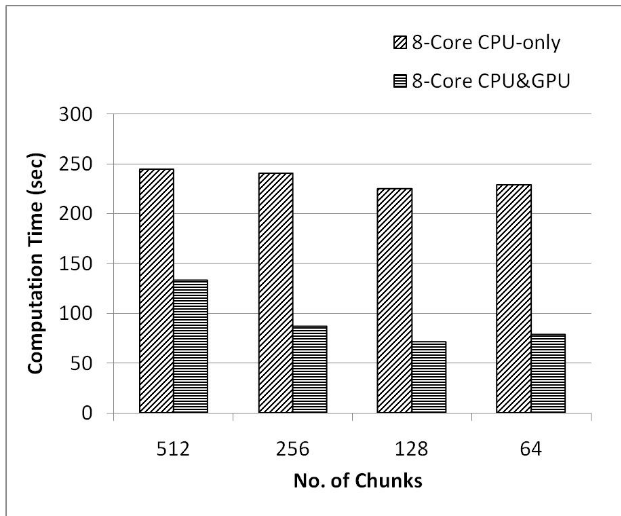

Figure 10. PCA - Impact of Varying No. of Chunks

pass. Moreover, local combination of reduction objects has to be performed at the end of each pass. This leads to a sub-linear scalability.

The speedup of the GPU-only version is quite good, about 3.9 times when compared to a single thread CPU version. For the `hetero` version, with 8 CPU cores and a GPU, we were able to achieve a speedup of 6.42 over the 1-thread CPU version. When compared to a 8-thread CPU-only version and the GPU-only versions, the `hetero` version (with 8 CPU threads) has a performance improvement of 69% and 65%, respectively. Though, these performance improvements are significant, they are not as good as for k-means. This is consistent with the sub-linear scalability of the CPU-only version.

We consider the above two sets of results as the base-version for the rest of this section.

4.2 Impact of Number of Chunks

In this subsection, we study the impact of chunk-size, or alternatively, the number of chunks a dataset is divided into, on the performance of the heterogeneous platform. We are comparing the performance results of 8-thread CPU-only, and the `hetero` (with 8 CPU threads) versions for varying chunk numbers. We used the same dataset size for both the applications as in the previous subsection.

Results from k-means: For k-means, the results in the previous subsection involved 512 chunks from the 6.4 GB dataset. This corresponded to 12.8 MB data chunk. We increased and decreased the number of chunks for the same dataset size. With increasing chunk numbers, we were not able to achieve any gain in the performance of the `hetero` version. When we decreased the number of the chunks, we observed a performance gain. The results are shown in Figure 9. With decreasing number of chunks, or increasing chunk size, the performance of the CPU-only version does not change. This is because the work is still evenly divided among the CPU cores, and communication latency is also not a factor. For the `hetero` version, fewer chunks imply fewer function calls to the GPU device, resulting in lesser overhead and better overall performance. But, as we decrease the chunk numbers, we were able to improve performance only up to a certain point. The performance of the `hetero` version started to decrease at 32 chunks. With k-means, for a dataset of 6.4 GB, the optimal number of chunks was found to be 64. With 64 chunks, the speedup of the `hetero` version, when compared to a 1-thread CPU version, is about a factor of 24.5. Overall, as compared to the case with 512 chunks, we increased the performance by about a factor of 1.5.

Results from PCA: Again for PCA, we considered 512 chunks to be the base version. This corresponds to a data chunk size of 4 MB. Similar to what we observed with k-means, increasing the number of chunks did not result in any performance improvements, thus we show results only from 512 and fewer chunks. The results are shown in Figure 10. Again, the performance of CPU-only version did not change. The `hetero` version saw significant performance improvements as the number of chunks was reduced. For PCA, with a dataset size of about 2.1 GB, 128 chunks, or a chunk size of 16 MB, resulted in the best performance. Further reducing the number of chunks resulted in a decreased performance. This is because, with an increase in the chunk size, I/O time for reading a data chunk increases, and hence, the worker threads spend more time waiting for the work. For PCA, with 128 chunks, the speedup of `hetero` version when compared to a 1-thread CPU version is about 12. Also, we had about a 87% increase in performance by decreasing the number of chunks from 512 to 128.

4.3 Evaluation of Load Balancing Schemes

In this subsection, we evaluate the performance of our dynamic load balancing scheme, comparing it against two static load balancing schemes, and further examine, the role the number of chunks play in the performance of the dynamic scheme. The two static load balancing schemes are `Naive` and `Smart`. In the naive static scheme, the data is partitioned equally between the group of CPU cores and the GPU. Smart static scheme is a *computational-power-aware scheme*. Here, the data is partitioned based on the processing power ratio between a CPU core and the GPU. This ratio is obtained by actual execution of the application on one core of the CPU and the GPU. If GPU is X times faster than the 1-thread CPU version, then, for the `hetero` version, if each CPU thread takes n chunks, GPU thread takes $n * X$ chunks.

For each of the load balancing schemes, we show the two components that constitute the total computation time, the *work-time*, and the *idle-time*. The former refers to the time when both CPU and GPU are simultaneously busy, whereas, the latter includes any time windows when either of the resources is idle.

The comparison of different load balancing schemes for k-means and PCA are shown in Figures 11 and 12. All results are from the `hetero` version, with 8 CPU threads. The important ob-

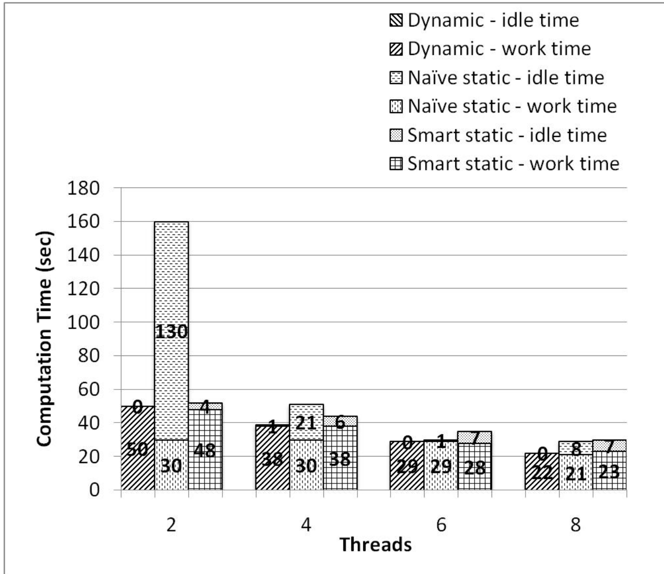

Figure 11. K-means: Comparison of Dynamic and Static Load Balancing Schemes

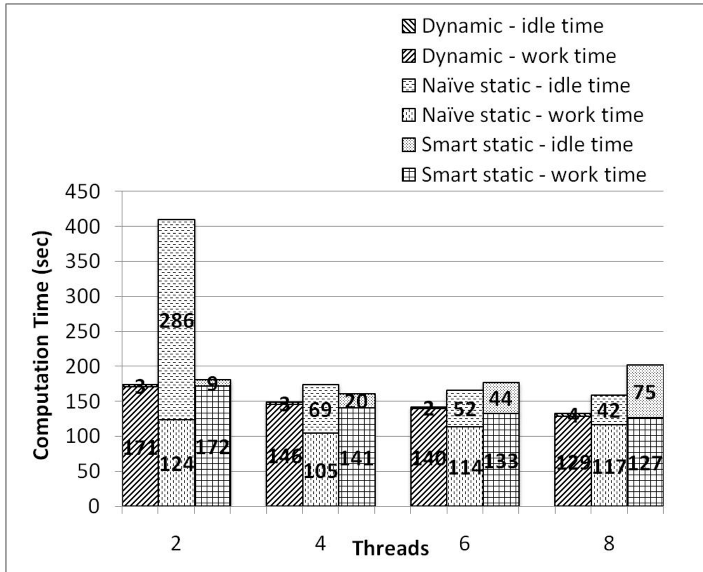

Figure 12. PCA: Comparison of Dynamic and Static Load Balancing Schemes

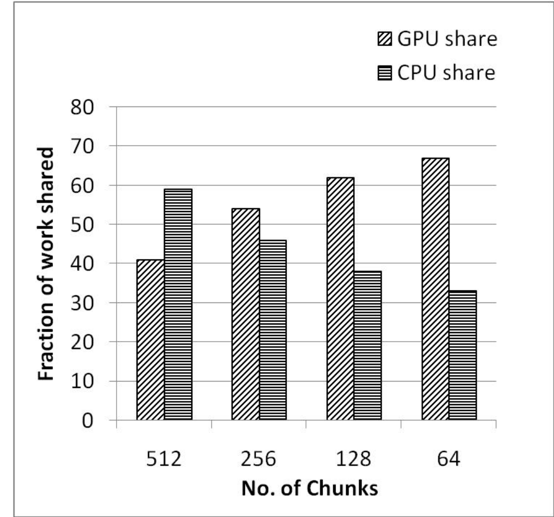

Figure 13. K-means - Distributed of Work

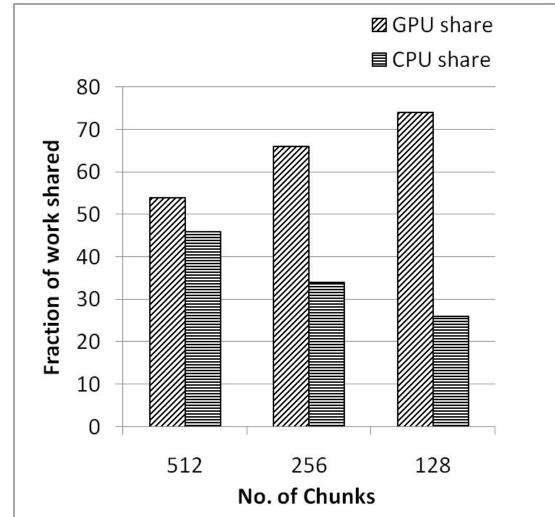

Figure 14. PCA - Distributed of Work

servations are as follows. For all the thread configurations, the performance of the dynamic scheme is better than both the static versions. For some of the thread configurations, smart static scheme performs better than naive scheme, while naive scheme outperforms smart scheme at other times. This shows that, it is hard to design a static scheme that performs well for all configurations. Both the static schemes suffer from large idle time for most of the thread configurations. Therefore, a dynamic load balancing system is required to find a near-perfect work share between the CPU threads and the GPU thread.

In the previous subsection, we had seen the impact of the number of chunks on the overall performance. To further understand this in the context of the dynamic scheme, we show the variation in the fraction of work performed by the group of CPU threads and the GPU thread over different number of chunks. The results are shown in Figure 13 and 14. For k-means, with the base-version, CPU threads perform more work than the GPU thread. This is because, with a higher number of chunks, GPU spends more time copying the data back and forth between the host and the device

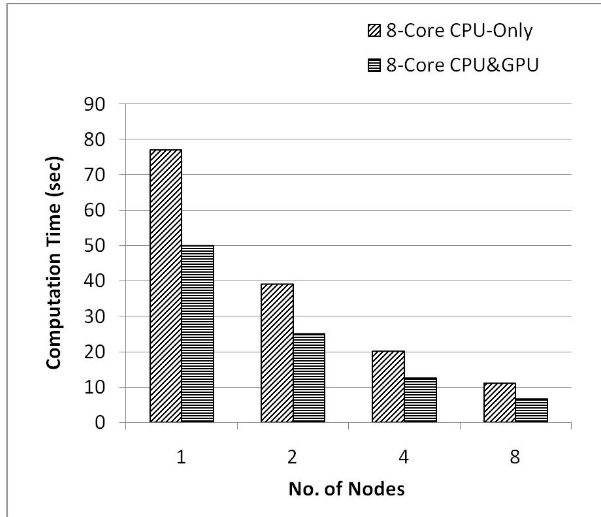

Figure 15. K-means: Scalability on 8 Nodes

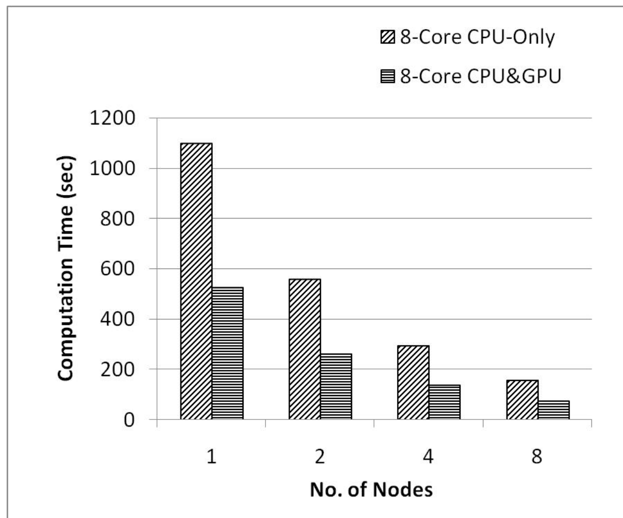

Figure 16. PCA: Scalability on 8 Nodes

memory. But, with decreasing number of chunks, work share of GPU thread increases. Beyond 64 chunks, the threads start to spend more time waiting for the chunks to arrive. So, for k-means, for this particular dataset size, the fraction of work share resulting in best performance was, 33% for CPU and 67% for GPU. For PCA, for the dataset size we considered, the best performance was obtained with 128 chunks, when the fraction of work performed by the CPU was 26%, and the remaining 74% of the work was performed by the GPU.

4.4 Scalability On a Cluster

We next demonstrate that the `hetero` version we generate could be scaled on a cluster of nodes with multi-core CPU and a GPU. We used a cluster of 8 nodes, with each node containing 8-CPU cores and a GPU. For k-means, we used a dataset of size 12.8 GB. For PCA, we used a dataset of size 8.5 GB. The number of chunks in the dataset was chosen in such a way that, when the dataset is partitioned between 8 nodes, each node gets a number of chunks that results in best possible performance, as chosen by the previous experiments.

The results for the experiments on the cluster are shown in Figures 15 and 16. We consider two versions, a CPU-only and a `hetero` version, on 1, 2, 4, and 8 nodes of the cluster. Both versions have a total of 8 CPU threads on each node. For k-means, using 8 nodes, with 8 CPU threads on each node, but no GPU, we get a speedup of about 56, over a single threaded CPU version. This shows a high parallel efficiency (about 88%). On the same 8 nodes, using 8 CPU cores and the GPU, we obtain a speedup of about 95 over a sequential version. This again shows that the `hetero` version can be scaled on a cluster, and use of GPUs can enhance performance even on a cluster.

For PCA, the CPU only version, with 8 nodes, and 8 cores on each node, achieves a speedup of 22.1 over a sequential single threaded version. The main reason for the limited speedup is that a single node, 8 thread version has only a speedup of 3.6. On 8 nodes, the `hetero` version achieves a speedup of 45.8. Again, this shows that we can continue to scale the `hetero` version on a cluster, and the use of GPU gives a substantial performance benefit.

5. Related Work

We now compare our work with related work on language support and application development on heterogeneous architectures, compiler support for GPU programming, and runtime and compiler support for reductions.

Recently, many research efforts have focused on exploiting the combined power of both CPU and GPU. Open Computing Language (OpenCL) [19] is a programming language based on C and C++ for heterogeneous programming with CPU, GPU, and other computing resources. This was initially developed by Apple and is currently being standardized by Khronos Group. Chamberlain *et. al* [8] have discussed their vision for application development for hybrid computing systems. But, there are no concrete design or implementation yet. Exochi [35] is a programming environment that utilizes a heterogeneous platform for media kernels, showing performance improvements. Recently, after Larrabee model was introduced, Intel has developed a programming model for heterogeneous computing of the x86 platform [31]. Here, they also describe a new memory model, but programming API is still relatively low-level. Venkatasubramanian *et. al* [34] have studied a stencil kernel for a hybrid CPU/GPU platform. They have evaluated their hybrid performance on two different hardware configurations.

Our effort is distinct in supporting a very high-level programming API, and showing significant performance gains from the use of a heterogeneous platform.

In the last 2 years or so, many research efforts have focused on easing programming on the GPUs. At UIUC, CUDA-lite [4] is being developed with the goal of alleviating the need for explicit GPU memory hierarchy management by the programmers. The same group also investigated optimization on CUDA programming [30]. Baskaran *et al.* use the polyhedral model for converting C code into CUDA automatically [6]. Their system is limited to affine loops. A version of Python with support of CUDA, Pycuda, has also been developed, by wrapping the CUDA functions and operations into classes that are easy to use [20]. The work at Purdue is focusing on translating OpenMP to CUDA [22]. Tarditi *et al.* have developed an approach for easing the mapping of data-parallel operations on GPUs [33]. Rapidmind offers somewhat similar approach targeting both GPUs and multi-cores[2]. Also, AMD has come up with their own streaming SDK [2], with higher level API, for programming their GPUs. The key distinct aspect of our work is that we focus on exploiting the computing power of both multi-cores and GPUs.

[2] http://www.rapidmind.net/

Analysis and code generation for reduction operations has been studied by a number of distributed memory compilation projects [1, 5, 10, 15, 21, 37] as well as shared memory parallelization projects[7, 11, 12, 24, 25, 28, 36]. More recently, reductions on emerging multi-cores have also been studied [23]. Our automatic code generation work has many similarities, but is distinct in considering a different computing platform. Map-reduce is a widely used parallel computing runtime system developed by Google [9]. Phoenix [29] is an implementation of map-reduce for shared-memory systems that includes a programming API for multi-core CPUs. As we stated earlier, there is already a GPU version of map-reduce, called Mars [13]. Our work targets the same class of applications, but considers the computing power of both GPUs and multi-core CPUs in a cluster. Furthermore, we offer a higher-level API (almost sequential).

6. Conclusions

Because of the growing popularity of both multi-cores and accelerators, common computing platforms today have heterogeneous processing components. Exploiting the processing power of such configurations is a growing challenge. In this paper, we have developed compiler and runtime support targeting a particular class of applications for such a heterogeneous configuration.

The key aspects of our approach and results are as follows. We have shown that targeting a limited class of applications eases the code generation challenge for both multi-core clusters and GPUs. Overall, we have shown that performance using a heterogeneous platform can be significantly better than performance using only a multi-core CPU or a GPU. Moreover, we have shown that dynamic work distribution clearly outperforms static schemes. Furthermore, the granularity in performing such dynamic distribution, i.e., the chunk or split size, is a critical factor for achieving high performance.

In the future, we will like to develop models for predicting the optimal chunk size. We will also like to expand our work to consider applications with other communication patterns.

References

[1] V. Adve and J. Mellor-Crummy. Using integer sets for data-parallel program analysis and optimization. In *Proceedings of the SIGPLAN '98 Conference on Programming Language Design and Implementation*, June 1998.

[2] AMD. AMD Stream SDK. ati.amd.com/technology/streamcomputing.

[3] P. Anderson, D. Binkley, G. Rosay, and T. Teitelbaum. Flow Insensitive Points-To Sets. *scam*, 00:0081, 2001.

[4] S. Baghsorkhi, M. Lathara, and W. mei Hwu. CUDA-lite: Reducing GPU Programming Complexity. In *LCPC 2008*, 2008.

[5] P. Banerjee, J. A. Chandy, M. Gupta, E. W. H. IV, J. G. Holm, A. Lain, D. J. Palermo, S. Ramaswamy, and E. Su. The Paradigm Compiler for Distributed-Memory Multicomputers. *IEEE Computer*, 28(10):37–47, Oct. 1995.

[6] M. M. Baskaran, U. Bondhugula, S. Krishnamoorthy, J. Ramanujam, A. Rountev, and P. Sadayappan. A Compiler Framework for Optimization of Affine Loop Nests for GPGPUs. In *International Conference on Supercomputing*, pages 225–234, 2008.

[7] W. Blume, R. Doallo, R. Eigenman, J. Grout, J. Hoelflinger, T. Lawrence, J. Lee, D. Padua, Y. Paek, B. Pottenger, L. Rauchwerger, and P. Tu. Parallel programming with Polaris. *IEEE Computer*, 29(12):78–82, Dec. 1996.

[8] R. D. Chamberlain, J. M. Lancaster, and R. K. Cytron. Visions for application development on hybrid computing systems. *Parallel Comput.*, 34(4-5):201–216, 2008.

[9] J. Dean and S. Ghemawat. Mapreduce: Simplified data processing on large clusters. In *OSDI*, pages 137–150, 2004.

[10] M. Gupta and E. Schonberg. Static analysis to reduce synchronization costs in data-parallel programs. In *Conference Record of the 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 322–332. ACM Press, Jan. 1996.

[11] M. Hall, S. Amarsinghe, B. Murphy, S. Liao, and M. Lam. Maximizing multiprocessor performance with the SUIF compiler. *IEEE Computer*, (12), Dec. 1996.

[12] H. Han and C.-W. Tseng. Improving compiler and runtime support for irregular reductions. In *Proceedings of the 11th Workshop on Languages and Compilers for Parallel Computing*, Aug. 1998.

[13] B. He, W. Fang, Q. Luo, N. K. Govindaraju, and T. Wang. Mars: A MapReduce Framework on Graphics Processors. In *PACT08: IEEE International Conference on Parallel Architecture and Compilation Techniques 2008*, 2008.

[14] High Performance Fortran Forum. High Performance Fortran language specification. *Scientific Programming*, 2(1–2):1–170, 1993.

[15] S. Hiranandani, K. Kennedy, and C.-W. Tseng. Compiling Fortran D for MIMD distributed-memory machines. *Commun. ACM*, 35(8):66–80, Aug. 1992.

[16] A. K. Jain and R. C. Dubes. *Algorithms for Clustering Data*. Prentice Hall, 1988.

[17] R. Jin and G. Agrawal. A middleware for developing parallel data mining implementations. In *Proceedings of the first SIAM conference on Data Mining*, Apr. 2001.

[18] R. Jin and G. Agrawal. Shared Memory Parallelization of Data Mining Algorithms: Techniques, Programming Interface, and Performance. In *Proceedings of the second SIAM conference on Data Mining*, Apr. 2002.

[19] Khronos. OpenCL 1.0. http://www.khronos.org/opencl/.

[20] A. Klockner. PyCuda. http://mathema.tician.de/software/pycuda, 2008.

[21] C. Koelbel and P. Mehrotra. Compiling global name-space parallel loops for distributed execution. *IEEE Transactions on Parallel and Distributed Systems*, 2(4):440–451, Oct. 1991.

[22] S. Lee, S.-J. Min, and R. Eigenmann. 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*, pages 101–110, New York, NY, USA, 2008. ACM.

[23] S.-W. Liao. Parallelizing user-defined and implicit reductions globally on multiprocessors. In C. R. Jesshope and C. Egan, editors, *Asia-Pacific Computer Systems Architecture Conference*, volume 4186 of *Lecture Notes in Computer Science*, pages 189–202. Springer, 2006.

[24] Y. Lin and D. Padua. On the automatic parallelization of sparse and irregular Fortran programs. In *Proceedings of the Workshop on Languages, Compilers, and Runtime Systems for Scalable Computers (LCR - 98)*, May 1998.

[25] B. Lu and J. Mellor-Crummey. Compiler optimization of implicit reductions for distributed memory multiprocessors. In *Proceedings of the 12th International Parallel Processing Symposium (IPPS)*, Apr. 1998.

[26] W. Ma and G. Agrawal. A translation system for enabling data mining applications on gpus. In *ICS '09: Proceedings of the 23rd international conference on Conference on Supercomputing*, pages 400–409, New York, NY, USA, 2009. ACM.

[27] NVidia. NVIDIA CUDA Compute Unified Device Architecture Programming Guide. version 2.0. http://developer.download.nvidia.com/compute/cuda/2.0-Beta2/docs/Programming_Guide_2.0beta2.pdf, June 7 2008.

[28] W. M. Pottenger. The Role of Associativity and Commutativity in the Detection and Transformation of Loop-Level Parallelism. In *Conference Proceedings of the 1998 International Conference on Supercomputing (ICS)*, pages 188–195. ACM Press, July 1998.

[29] C. Ranger, R. Raghuraman, A. Penmetsa, G. Bradski, and C. Kozyrakis. Evaluating mapreduce for multi-core and multiproces-

sor systems. In *Proceedings of International Symposium on High Performance Computer Architecture, 2007*, pages 13–24, 2007.

[30] S. Ryoo, C. Rodrigues, S. Stone, S. Baghsorkhi, S.-Z. Ueng, J. Stratton, and W. mei Hwu. Program Optimization Space Pruning for a Multithreaded GPU. In *Proceedings of the 2008 International Symposium on Code Generation and Optimization, April 2008*, pages 195–204. ACM, April 2008.

[31] B. Saha, X. Zhou, H. Chen, Y. Gao, S. Yan, M. Rajagopalan, J. Fang, P. Zhang, R. Ronen, and A. Mendelson. Programming model for a heterogeneous x86 platform. *SIGPLAN Not.*, 44(6):431–440, 2009.

[32] L. Seiler, D. Carmean, E. Sprangle, T. Forsyth, M. Abrash, P. Dubey, S. Junkins, A. Lake, J. Sugerman, R. Cavin, R. Espasa, E. Grochowski, T. Juan, and P. Hanrahan. Larrabee: a many-core x86 architecture for visual computing. *ACM Trans. Graph.*, 27(3):1–15, August 2008.

[33] D. Tarditi, S. Puri, and J. Oglesby. Accelerator: using data parallelism to program gpus for general-purpose uses. In J. P. Shen and M. Martonosi, editors, *ASPLOS*, pages 325–335. ACM, 2006.

[34] S. Venkatasubramanian and R. W. Vuduc. Tuned and wildly asynchronous stencil kernels for hybrid cpu/gpu systems. In *ICS '09: Proceedings of the 23rd international conference on Supercomputing*, pages 244–255, New York, NY, USA, 2009. ACM.

[35] P. H. Wang, J. D. Collins, G. N. Chinya, H. Jiang, X. Tian, M. Girkar, N. Y. Yang, G.-Y. Lueh, and H. Wang. Exochi: architecture and programming environment for a heterogeneous multi-core multithreaded system. In *PLDI '07: Proceedings of the 2007 ACM SIGPLAN conference on Programming language design and implementation*, pages 156–166. ACM Press, 2007.

[36] H. Yu and L. Rauchwerger. Adaptive reduction parallelization techniques. In *Proceedings of the 2000 International Conference on Supercomputing*, pages 66–75. ACM Press, May 2000.

[37] H. P. Zima and B. M. Chapman. Compiling for distributed-memory systems. *Proceedings of the IEEE*, 81(2):264–287, Feb. 1993. In Special Section on Languages and Compilers for Parallel Machines.