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

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

ABSTRACT

A trend that has materialized, and has given rise to much attention, is of the increasingly heterogeneous computing platforms. Presently, it has become very common for a desktop or a notebook computer to come equipped with both a multi-core CPU and a GPU. Capitalizing on the maximum computational power of such architectures (i.e., by simultaneously exploiting both the multi-core CPU and the GPU) starting from a high-level API is a critical challenge. We believe that it would be highly desirable to support a simple way for programmers to realize the full potential of today's heterogeneous machines.

This paper describes a compiler and runtime framework that can map a class of applications, namely those characterized by *generalized reductions*, to a system with a multi-core CPU and GPU. Starting with simple C functions with added annotations, we automatically generate the middleware API code for the multi-core, as well as CUDA code to exploit the GPU simultaneously. The runtime system provides efficient schemes for dynamically partitioning the work between CPU cores and the GPU. Our experimental results from two applications, e.g., k-means clustering and Principal Component Analysis (PCA), show that, through effectively harnessing the heterogeneous architecture, we can achieve significantly higher performance compared to using only the GPU or the multi-core CPU. In k-means, the heterogeneous version with 8 CPU cores and a GPU achieved a speedup of about 32.09x relative to 1-thread CPU. When compared to the faster of CPU-only and GPU-only executions, we were able to achieve a performance gain of about 60%. In PCA, the heterogeneous version attained a speedup of 10.4x relative to the 1-thread CPU version. When compared to the faster of CPU-only and GPU-only versions, we achieved a performance gain of about 63.8%.

Categories and Subject Descriptors

C.1.3 [**Other Architecture Styles**]: Heterogeneous (hybrid) systems

General Terms

Heterogeneous Systems

Keywords

Generalized Reductions, Dynamic Work Distribution, Multi-cores, GPGPU

1. INTRODUCTION

The traditional method to improve processor performance, i.e., by increasing clock frequencies, has become physically infeasible. To help offset this limitation, multi-core CPU and many-core GPU architectures have emerged as a cost-effective means for scaling performance. This, however, has created a programmability challenge. On the one hand, a large body of research has been focused on the effective utilization of multi-core CPUs. For instance, library support and programming models are currently being developed for efficient programming on multi-core platforms [8, 25]. And on the other hand, scientists have also been seeking ways to unleash the power of the GPU for general-purpose computing [2, 26, 10, 29]. While a variety of applications have been successfully mapped to GPUs, programming them remains a challenging task. For example, Nvidia's CUDA [23], the most widely used computing architecture for GPUs to date, requires low-level programming and manual memory management.

An emerging trend that is now beginning to receive attention [20, 30] is of increasingly heterogeneous computing platforms. It has become common for today's desktops and notebooks to have both multi-core CPUs and GPU(s). Figure 1 shows such an architecture. Increasingly, there is also evidence that such a trend is likely to continue. For example, the future products announced in 2009 by AMD involve *fusion* of CPU and GPU computing[1].

In most studies scaling applications on modern multi-core CPUs, the possibility of using the GPU to further accelerate the performance has not been considered. Similarly, in studies involving GPGPUs, the multi-core, with a very high peak performance, remains idle. Effectively exploiting the aggregate computing power of a heterogeneous architecture comprising the multi-core CPU and a GPU is one of the most critical challenges facing high-performance computing. Particularly, three important issues that must be addressed are: programmability, performance, and work distribution. We offer a brief overview of each issue below.

Programmability: With current technologies, exploiting the power of a multi-core CPU and a GPU simultaneously would require programming a combination of OpenMP or Pthreads with CUDA. Programming under just one of these paradigms requires much effort,

[1]http://sites.amd.com/us/fusion/Pages/index.aspx

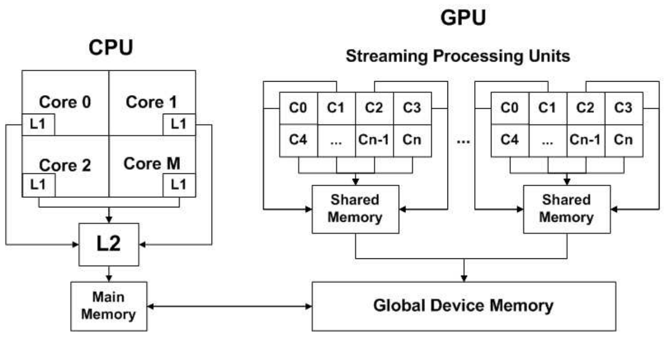

Figure 1: A Popular Heterogeneous Computing Platform

let alone having expertise in both. Programming support for such heterogeneous systems is garnering increasing attention from the community. For instance, Open Computing Language (OpenCL) [14] is a C-based programming framework with promising heterogeneous processing support, but it is still in early developmental phases. In any case, OpenCL still requires explicit parallel programming, which is a complexity we hope to avoid. In another effort, Saha *et. al* [27] have recently proposed a programming model for heterogeneous parallel computing, but again, it still requires low-level programming and memory management. Clearly, invoking the full power of a heterogeneous architecture has so far been restricted to a small class of experts. It is thus highly desirable to program on a heterogeneous platform starting from a high-level language or a highly accessible API.

Performance: Given a lack of efforts in this area, it is not clear what type of performance gain can be obtained from such heterogeneous configurations, and for what kind of applications. A recent study implemented compute-intensive algorithms, e.g., stencil kernels, over a hybrid CPU/GPU system [30]. The experimental results in this study showed that performance gains were only marginal with their system configurations. We, however, believe that a deeper understanding of the performance potential of heterogeneous platforms may require more detailed experimentation with disparate applications.

Work Distribution: Dividing the work among heterogeneous multiprocessors is an important challenge that directly impacts performance. Furthermore, CPU cores and GPU cores differ considerably in their processing capabilities, memory availability, and communication latencies. Because of this reason, despite their simplicity in design and implementation, static work distribution schemes are typically unsuitable within heterogeneous settings. A dynamic distribution scheme that can vary workloads according to the compute resource's capabilities could utilize the heterogeneity more effectively. Recently, Luk *et al.* had proposed Qilin [20], an adaptable workload mapping scheme based on offline training. It is desirable, however, if work distribution can be performed with dynamic schemes that do not require offline training.

This paper addresses the above three challenges, focusing on the particular class of applications where the communication pattern is based on *generalized reductions*, more recently known as Map-Reduce [8] class of applications. This application class has been considered as one of the *dwarfs* in the Berkeley view on parallel processing [2]. We have developed a compiler and runtime support for heterogeneous multiprocessors. To support ease of programming, applications which follow the generalized reduction structure can be programmed using a sequential C interface, with some additional annotations. Our code generation system automatically generates a parallel API code for multi-core CPU, and the CUDA code for the GPU. Then, a middleware runtime automatically maps the CUDA code to GPU and multi-core code to CPU. The memory management involved with the GPU is also handled automatically by our code generation. Dynamic work distribution support is provided by our runtime system for effective distribution of workload between the CPU and the GPU.

An extensive evaluation has been performed using two popular data mining algorithms that follow the generalized reduction pattern. We show that exploiting heterogeneous configurations simultaneously leads to significant gains in speedup over the CPU-only and GPU-only results. Our key results are as follows. For the k-means clustering application, the heterogeneous version with 8 CPU cores and a GPU achieved a speedup of about $32.09\times$ relative to 1-thread CPU. When compared to the faster run of either CPU-only and GPU-only versions, we achieved a performance gain near 60%. For the Principle Component Analysis (PCA) application, a well known dimensionality reduction method, the heterogeneous version achieved a speedup of about $10.4\times$ relative to 1-thread CPU. When compared to the faster of CPU-only and GPU-only versions, a speedup of around 63.8% was achieved.

The remaining sections of the paper is organized as follows. In Section 2 we describe our approach on handling generalized reduction-based computations for heterogeneous architectures. In Section 3, we elaborate on the language and compiler support that is provided. We present our experimental results in Section 4. In Section 5, we discuss related efforts and finally conclude in Section 6.

2. APPROACH AND SYSTEM DESIGN

This section offers a description of the class of generalized reduction applications on which we focus. We explain an approach for mapping generalized reduction algorithms on a multi-core CPU and a GPU independently, followed by an approach for enabling hybrid execution.

2.1 Generalized Reduction Structure

```
/ * Outer Sequential Loop * /
While (unfinished) {
    / * Reduction Loop * /
    Foreach (element e) {
        (i,val) = process(e);
        Reduc(i) = Reduc(i) op val;
    }
}
```

Figure 2: *Generalized Reduction* Processing Structure

The processing structure on which we focus is summarized in Figure 2. The outer *while* loop is a sequential loop and controls the number of times the inner *foreach* loop is executed, typically until *unfinished* (end condition) is false. The success or failure of *unfinished* is dependent on the results obtained at the end of each iteration of the *foreach* loop. Due to this dependency, the outer *while* loop cannot be parallelized. However, we note that the *op* function is an associative and commutative function, which permits the iterations of the *foreach* loop to be performed in any order. Therefore the entire *foreach* loop is data parallel. Inside the *foreach* loop, each data instance, e, is processed to produce an (i, val) pair, where i is a key and val is the value that is to be accumulated. The data structure, *Reduc*, is referred to as the *reduction object*, and

[2]See http://view.eecs.berkeley.edu/wiki/Dwarf_Mine

the val obtained after each processing iteration is accumulated into this reduction object based on the corresponding *key*. The reduction performed is, however, *irregular*, in the sense that which elements of the reduction object are updated depends upon the results of the processing of data instance.

In our earlier work, we had made an observation that parallel versions of many well-known data mining, OLAP, and scientific data processing algorithms share the aforementioned generalized reduction structure [12, 13]. This observation has some similarities with the motivation for the *map-reduce* paradigm that Google has developed [8].

2.2 Parallelization Approach for Multi-cores

Applications following such generalized reduction structure can be parallelized on shared memory platforms by dividing the data instances among the processing threads. A key challenge with such parallelization, however, is the potential data races with updates to the reduction object. The reduction object, shown as $Reduc$ in Figure 2, is shared among all the processing threads and is used to accumulate the results obtained after processing each data instance. Furthermore, in the application we target, it is not possible to statically partition the reduction object such that each processing thread updates disjoint parts of the reduction object due to processing dependencies. A simple strategy to avoid such data race is to *privatize* the reduction object. Specifically, a copy of reduction object is created and initialized for each processing thread. Each thread only updates its own copy of the reduction object. The resulting reduction objects are merged or combined at the end to obtain the final results.

In our previous work, we have developed a middleware system [12, 13] to support the parallelization of this class of applications. Operations including reading data instances from the disk, initial thread creation, replicating reduction object, and merging reduction objects are automatically performed by this middleware. This, in turn, simplifies our code generation task; our system needs only to generate code for the middleware API. Such code generation is quite straight-forward, and is not discussed in detail in this paper.

2.3 GPU Computing for Generalized Reductions

GPUs support SIMD shared memory programming. Specifically, a host (driver) program is required to launch and communicate with a GPU process to execute the *device* function in parallel. Therefore, GPU computations involve the following three steps: (1) Copying the data block from the host (main) memory to GPU device memory, (2) launching the GPU kernel function from host and executing it, and (3) copying the computed results back from device memory to host memory.

Now, consider parallelizing *generalized reduction* applications on GPUs. The reduction objects are shared by all the GPU threads. Again there are potential data races, which can be avoided by replicating the reduction object for each thread. The data block is divided into small blocks such that each thread only processes one data block at a time. The results from each thread are then merged to form the final result. Finally, the merged reduction object is copied back to the host memory. Thus, three steps are required for implementing a *reduction* computation: (1) Read a data block, (2) compute the reduction object updates based on the data instance, and (3) write back the reduction object update.

2.4 System Design for a Heterogeneous Platform

Based on the approaches for parallelizing generalized reductions on a multi-core CPU and on a GPU, as we described above, we have developed a framework for mapping these applications to a heterogeneous platform. We only give a brief description of our approach in this subsection, while key components of our system are elaborated in the later sections. A high-level design of the system is shown in Figure 3.

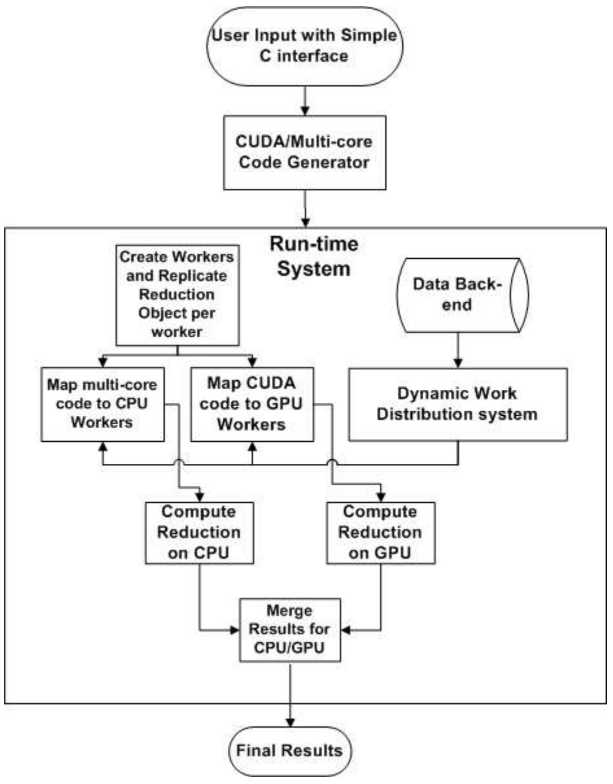
User Input with Simple C interface
CUDA/Multi-core Code Generator
Run-time System
Create Workers and Replicate Reduction Object per worker
Data Back-end
Map multi-core code to CPU Workers
Map CUDA code to GPU Workers
Dynamic Work Distribution system
Compute Reduction on CPU
Compute Reduction on GPU
Merge Results for CPU/GPU
Final Results

Figure 3: High-level Architecture of the System

Our system takes generalized reduction applications written in C with some supplementary metadata for annotating variable information (described later). A code generation module automatically generates the middleware API code for the multi-core CPU, and CUDA code for the GPU. The language and compiler approach is elaborated in Section 3, particularly focusing on the details of automatic code generation for CUDA. Our runtime system creates the worker threads, including one for each CPU core and one for managing the communication with the GPU. Consistent with our mechanism for parallelizing these applications on either of CPUs or a GPU, a separate copy of the reduction object is created for each of these threads to avoid race conditions while performing the reduction computations. A key component of the runtime system is our *dynamic work distribution* scheme which divides the work between different threads, including the thread that manages the GPU. This module is further elaborated on in the next subsection. After the processing is finished by all the threads, the results from each thread are merged together to form a final result.

2.5 Dynamic Work Distribution

In a heterogeneous setting with multi-core CPU and GPU, cores have disparate processing capabilities. While their relative peak performance can be known in advance, our experiences have shown that relative performance of CPU and GPU can vary significantly across applications. This renders work distribution to be more challenging. Particularly, a *static* distribution scheme is not likely to be effective. Thus, our work focuses on developing dynamic distribution schemes. Within the general class of dynamic schemes, there could be two possible approaches, which are *work sharing* and *work stealing*.

Work Sharing: This is one of the classical techniques, based on a *centralized* approach. In general, the scheduler enqueues the work in a *globally shared* work list, and an idle processor consumes work from this work list [32].

Work Stealing: In work stealing, a private work list is maintained with each processor [4, 5]. When a processor becomes idle by finishing the work in its private queue, it searches the other busy processors for work they could *steal*. Work stealing is shown to be a useful technique for balancing the workload with irregular applications [7, 9] especially on very large systems where a centralized scheme may be prohibitive.

In designing our system, we have taken a work sharing approach based on several reasons. In a heterogeneous computing environment, the potential load imbalance is due to the disparity among the processing cores. In our work, we focus on data parallel applications, where the amount of computation is proportional to the amount of input data assigned to each processor. Furthermore, communication with the GPU involves high latency, and the maximum workload that can be distributed to a GPU at a time is limited by the capacity of its device memory. Because of these factors, a work stealing approach would be impractical. Considering our class of target applications and the limitations of the GPU device memory, we have designed two dynamic work distribution schemes based on work sharing.

Uniform-chunk Distribution Scheme: This scheme resembles a classical *master-slave* model, and is shown in Figure 4. The entire data set is divided into *chunks* of *uniform* size. A master thread acts as a job scheduler, and uses the First Come First Served (FCFS) scheduling policy. There exists a global work list shared by all processors, into which the master enqueues work whenever there is empty space. When a process becomes idle, it simply requests work from this list. The rationale behind the FCFS policy is that, a faster worker ends up requesting more work to process when compared to the slower worker. This approach ensures that a faster worker is rewarded with more work, while a reasonable amount of work is also completed by a slower worker. By keeping the policy simple, the overhead of work distribution is amortized.

Clearly, a critical factor with the above scheme is the choice of the *chunk size*. Intuitively, and also supported by our experiments, we can see the following trends. In the case of the GPU, because there is a high latency for invoking kernel function and data transfer and that the processing speed is faster, it would favor a larger chunk size. However, each CPU core is slower, but has much lower latency. Thus, a smaller chunk size is preferable for CPU cores. This leads us to the following scheme:

Non-uniform-chunk Distribution Scheme: This distribution scheme is shown in Figure 5. The initial partition of the entire data set is still of a uniform chunk size, and we typically choose a relatively smaller chunk size. When a CPU thread requests for data, the chunk with the original size is provided. When a GPU thread requests for data, a certain number of chunks of the initial size are merged together to form a larger chunk at runtime. This chunk is then forwarded to the GPU for processing. This scheme ensures that the GPU spends most of its time actually processing the data rather than spending on transferring data and the results between host and device memory. At the same time, it ensures proper distribution of work among CPUs. To this end, when only a small fraction of data is left for processing, the scheme does not allow a single CPU core to slow down the entire application.

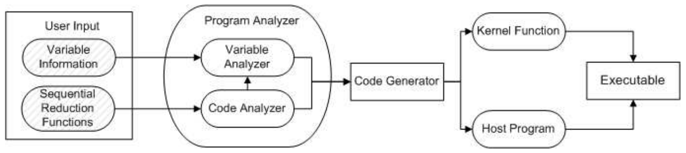

Figure 6: Code Generation Process

3. LANGUAGE SUPPORT AND CODE GENERATION

As we had shown in Figure 3, a key component of our framework is a code generation system. This component includes code generation for our multi-core runtime system and GPUs. The details of code generation for our runtime system are specific to the API of this system, and are not presented here. In this section, we focus on our approach for automatically generating the CUDA code for the GPU. For clarity, this process is depicted in Figure 6. We initially describe the input that is expected from application developers.

3.1 User Input

Our overall goal is to enable a simple framework, which can be incorporated with various parallel languages and their compilation systems in the future. The application developer first identifies the generalized reduction structure, then an application can be expressed using one or more `reduction` functions. In case of multiple reduction functions, the user should identify each of these reduction functions using an unique `label`. For each of these functions, the following information is required from the user.

Variable List for Computing: The declaration of each variable follows the format: `name, type, size[value]` where `name` is the name of the variable, `type` can be either a numeric type such as `int` or pointer type like `int*`, which would indicate an array. If `type` is a pointer, then `size` is used to specify the size of the referenced array, which in turn is a list of numbers and/or integer variables. The size of the array is the product of the elements in the list. For single, numeric variables, the size of the variable defaults to its type.

For instance, `K int`, denotes the declaration of a variable `K` of type `int`, whose size defaults to the size of an integer. Similarly, `update_centers float* 5 K`, indicates the declaration of a variable `update_centers` which is of type `float*` and that it requires an allocation of size `5*K*sizeof(float)`.

Sequential reduction function: Recall the reduction loop discussed earlier in Figure 2. Such a reduction loop can be conveniently expressed as a function which reads a data block, processes the data block, and finally updates the result into the reduction object (shown as `Reduc` in Figure 2). The user can write sequential code for the outer `foreach` loop of the reduction operation in C. Any variable declared inside the reduction function should also appear in the aforementioned variable list, but the memory allocation for these variables will be automatically generated later.

The input of the above form can also be extracted by one of the popular parallel compilation systems. In our current implementation, however, it is explicitly input by the users. Such user specifications are then forwarded to the `program analyzer`, which is described in the next subsection.

3.2 Program Analysis

There are two main components in the program analyzer: the *code analyzer* and the *variable analyzer*, but before we describe the *code analyzer*, the program variables' access features must first be obtained.

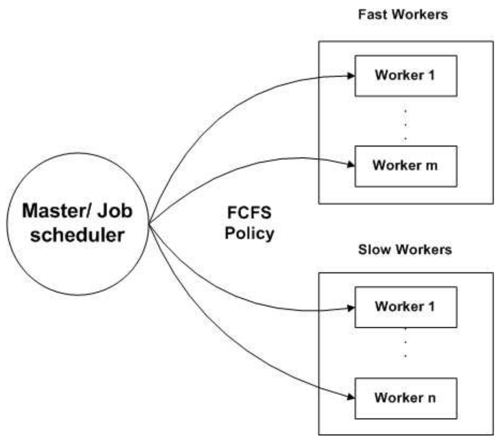

Figure 4: Uniform Chunk-Size Distribution Scheme

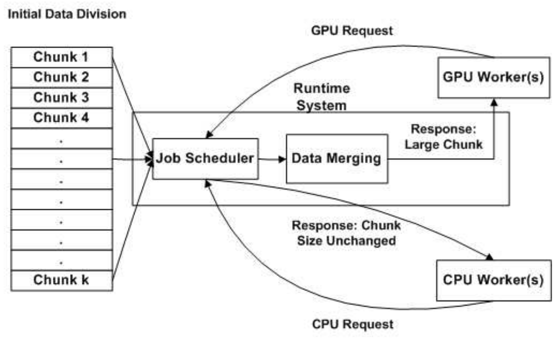

Figure 5: Non-Uniform Chunk-Size Distribution Scheme

Variable Analysis and Classification: Based on the access features, we require that each variable be classified as one of the following: `input`, `output`, or `temporary` variable. An `input` variable is one that is only read inside the reduction function and is neither updated inside the function, nor is returned by the function. In contrast, an `output` variable is modified in the function and is returned from the reduction function. Finally, a `temporary` variable is declared inside the reduction function for temporary storage. Thus, an `input` variable is *read-only*, while the `output` and `temporary` variables are *read/write*. Variables with different access patterns are treated differently in declaration, memory allocation strategies, and combination operation, which will be described later in this section.

In order to classify the program variables into above three categories, information can usually be obtained from a simple inspection of a function. But, because we support the C language, complications can arise due to the use of pointers and aliasing. To handle such complexity, we employ LLVM [18], which generates an intermediate representation (IR) of the user-specified sequential reduction function. Next, we apply Andersen's points-to analysis [1] on the IR to obtain the `points-to` set, i.e., the set of pointers that refer to the same object, for each variable in the function's argument list. Finally, the entire function in its IR form is traced, and when a `store` operation is encountered, we classify the variable as one of the three things: (1) If the destination of the store belongs to a `points-to` set of any variable in the function's argument list and the source is not in the same set, then we conclude that it is an `output` variable. (2) Otherwise, all the other variables in the argument list are denoted as `input` variables, and (3) all variables that do not appear in the argument list are classified as `temporary` variables.

Code Analysis: We now explain the functionality of the *code analyzer*, which accomplishes two main tasks: (1) Extracting the reduction objects with their combination operation, and (2) Extracting the parallel loops. The reduction objects and their combination operations must first be extracted. Simply based on previous variable classification, the `output` variables are identified as the reduction objects. Again, we are focusing on reduction functions where `output` variables are updated with only associative and commutative operations (see Figure 2), which allows variables updated by each thread to be combined at the end to produce *correct* results. However, we do need to identify the particular operator that is being used for updating reduction object variables. To do this, recall that we earlier generated the `points-to` sets for each parameter of the reduction function. Therefore, a new scan is performed on IR to find the reduction operator for each reduction object. In the `combine` function, the values in a reduction object from each thread are merged using this operator.

After the above information has been extracted, we can extract the parallel loops. The structure of the loop being analyzed is mapped to the canonical reduction loop we had shown earlier in Figure 2. We focus on the outer `foreach` loop and extract its *loop variable*. We also identify (symbolically) the number of iterations in the loop and denote it as *num_iter*. If there are nested `foreach` loops, for simplicity, we only parallelize the outer loop. If a variable is accessed only with an affine subscript of the loop index, it is denoted as a *loop* variable. For instance, in an element `lv[2*i+j]`, `2*i+j` is the affine subscript and `lv` is the loop variable. Note that this variable could be an `input`, `output`, or `temporary` variable. The significance of a loop variable is that when executed on GPU, disjoint portions of a *loop* variable can be distributed among the threads, while all the other variables need to be replicated if they are updated inside the loop.

3.3 Code Generation for CUDA

Using the user input and the information extracted by the variable and code analyzers, our system then generates the corresponding CUDA code and the host functions for invoking the CUDA-based parallel reductions. The CUDA code generated consists of two halves. First is the host (driver) code which initiates, invokes the device function, and manages communication with the GPU, and the second half is the device function which executes on the GPU.

Generating Host function: The host function, `host_func()`, which invokes the kernel function on device consists of following parts:

`Declare and Copy:` In this step, we declare and allocate device memory for variables that are to be used by the kernel function on the GPU. These variables are then copied from the host to the device memory. Memory is allocated for all variables, except the `temporary` variables, as they can utilize shared memory. As we described earlier, disjoint portions of the *loop* variables are distributed across threads. The *read/write* variables that are not *loop* variables might be updated by multiple threads simultaneously, and these variables are replicated for each thread. Also, we can assume that a combine function will produce correct results due to the associative and commutative operations.

`Compute:` In this step, the execution configuration (thread/

block configuration) for the GPU is defined. Then, the kernel function is invoked. While different thread/block configurations are possible, we empirically identify the best configuration for each application.

`Copy Updates:` After the kernel computation is complete, results needed by the host function should be copied over from the GPU, back to host. Then the `output` variables from each block (inter-block) are combined to produce the final result.

Generating Kernel Code: The kernel code that is to be executed on the GPU is then generated. This task involves producing the device function, `device_reduc()`. If there are multiple reduction functions specified by the user, more than one `device_reduc_label()` functions will be generated. Each of these functions will be identified by the unique label that was assigned by the user. Overall, the device function is generated by rewriting the original sequential code in the user input based on the information produced by the code and variable analysis phases. The modifications include: (1) Dividing the loop that is to be parallelized by the number of thread blocks and number of threads within each block. (2) Regroup the index of the array that is distributed. For example, consider an access to `data[i]`. This is changed to `data[i+index_n]`, where `index_n` is the offset for each thread in the entire thread grid. (3) Optimize the use of the GPU's shared memory. In this phase, the variables are sorted according to their size and shared memory is allocated for variables in the increasing order, until no variable can fit in shared memory. In our experience, this has been shown to be a simple, but effective, heuristic for capitalizing on the fast accesses of the GPU's shared memory. Finally, `combine()` will be invoked to perform the merging of results among all threads within a block (intra-block). `__syncthreads()` are inserted appropriately to provide synchronization.

4. EXPERIMENTAL RESULTS

In this section, we report results from a number of experiments that were performed. Our experiments were conducted on an AMD Opteron 8350 machine (running Redhat Linux) with 8 CPU cores and 16 GB of main memory. This machine is also equipped with a GeForce 9800 GTX graphics card, which contains 512MB of device memory. As such, we evaluate the scalability of two applications involving *generalized reductions*.

The first application we consider is k-means. Clustering is one of the key data mining problems and k-means [11] is one of the most ubiquitous approaches. The clustering problem is as follows: Given a set of points in high-dimensional space, we want to classify these points into K groups, i.e., clusters. The proximity within this space is used as the criterion for classifying the points into these clusters. In general, k-means consists of four steps: (1) Begin with K random centroids, (2) for each data instance (point), find the centroid closest to it and assign this point to the corresponding cluster. (3) With these assignments, recalculate the K centroids to be the average among all the points assigned to it. (4) Repeat this entire process until the centroids converge. In our experiments, the number of clusters, K, was fixed at 125, and the data used for k-means, a 6.4GB file, contains a set of nearly 100 million 3-Dimensional points.

The second application is Principal Component Analysis (PCA) [24], a widely used dimensionality reduction method. This algorithm performs three passes on the data set, which represents a matrix. (1) First, the `mean` value of the column vectors are determined, followed by (2) the calculation of the `standard deviation` of column vectors. (3) In the third pass, the correlation matrix is computed, and then, triangular decomposition is performed, and finally the eigenvalues are also computed. The experiments on PCA were conducted with a data set of size 8.5 GB. The number of columns of our resulting covariance matrix, m, was set to 64.

To exploit the parallelism that is afforded by the generalized reduction structure, the data used in our experiments are split into smaller, independently processed "chunks." While uniform chunk sizes are common in practice due to the simplification of data distribution, we argue that distributing non-uniform chunk sizes to corresponding computing resources could be beneficial in a heterogeneous environment. With these two applications (k-means and PCA), we wish to demonstrate the main goals with our experiments: 1) To evaluate the performance from a multi-core CPU and the GPU independently, and to study how the *chunk-size* impacts this performance.

2) To study performance gain that can be achieved while simultaneously exploiting both multi-core CPU and GPU, (ie., the heterogeneous version). We also provide an elaborate evaluation with two dynamic distribution schemes that were discussed in earlier section.

4.1 Scalability of CPU-only and GPU-only versions

In our initial experiment, we focus on the `CPU-only` version and the `GPU-only` version separately. For all experiments with the GPU, we report results that are obtained only from the best *thread block configuration* (after having experimented with multiple configurations). The results for k-means and PCA are presented in Figure 7 and Figure 8 respectively. For the `CPU-only` execution (top x-axis), the chunk size has been fixed at 12.5MB for k-means and 16MB for PCA, and we report the execution times when varying the number of worker threads from 1 to 8. In the `GPU-only` execution (bottom x-axis), we vary the chunk sizes distributed to the GPU. Thus, both the figures use two x-axes, the bottom x-axis represents varying GPU chunk size, while the top x-axis represents the number of CPU threads. As our preliminary experiments suggested that the chunk size did not impact the performance of the `CPU-only` version, we did not show results varying them.

Results from K-Means: In the `CPU-only` version, the performance with 8 threads resulted in a speedup close to 9. This superlinear performance is due to low communication overheads required by k-means, and mostly from the benefits of the additional aggregate cache memory that is available with a larger number of threads. The `GPU-only` version performs better with increasing chunk size. This is due to the fact that, with larger chunk sizes, fewer costly device function calls are required. Indeed, the latency associated with invoking a device function and initiating a data transfer are quite significant in GPUs. The best performance from a GPU version is observed when the data size of each chunk is 200 MB, which is about a factor of 20× speedup relative to the 1-thread CPU version. We could not experiment with a chunk size of more than 200 MB as the GPU device memory is only 512 MB, and significant space is required for storing program data structures. Our results also show that the `GPU-only` version is faster than the 8-core `CPU-only` version by a factor of 2.2.

Results from Principle Component Analysis (PCA): With 8 threads in the `CPU-only` version, we are able to achieve a speedup of around 6.35. Unlike k-means, this application has substantial parallelization overheads. For the `GPU-only` version, we increase the data chunk size from 16 MB to 256 MB, with the latter being the upper limit on the chunk size we can use, again due to the limited GPU memory. The best performance from the `GPU-only` execution is obtained when the data chunk size is 256 MB, which is about a factor of 4 faster relative to the 1-thread CPU version. Interestingly, contrary to what we observed for k-means, here, the 8-thread `CPU-only` version is faster than the `GPU-only` version

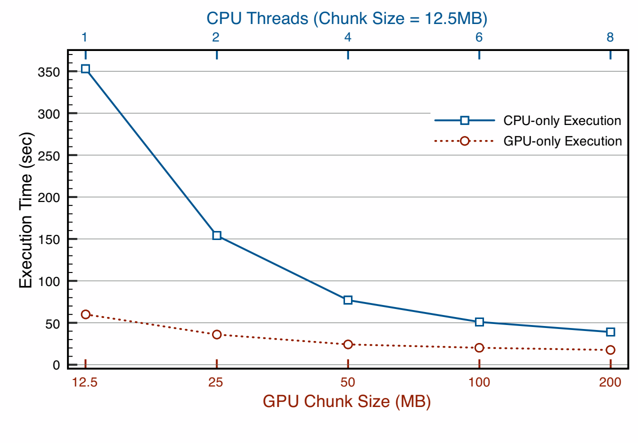

Figure 7: Scalability of K-Means with CPU-only and GPU-only

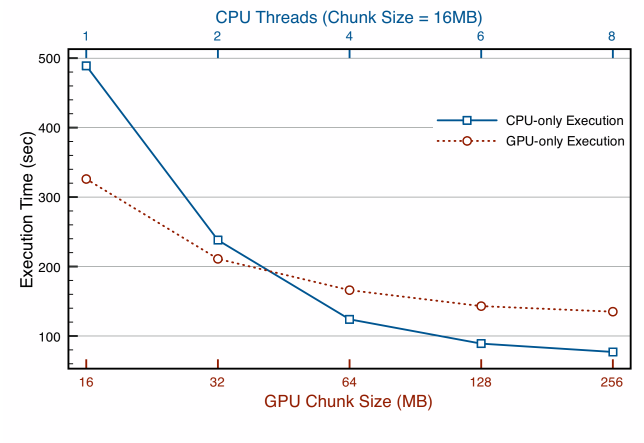

Figure 8: Scalability of PCA with CPU-only and GPU-only

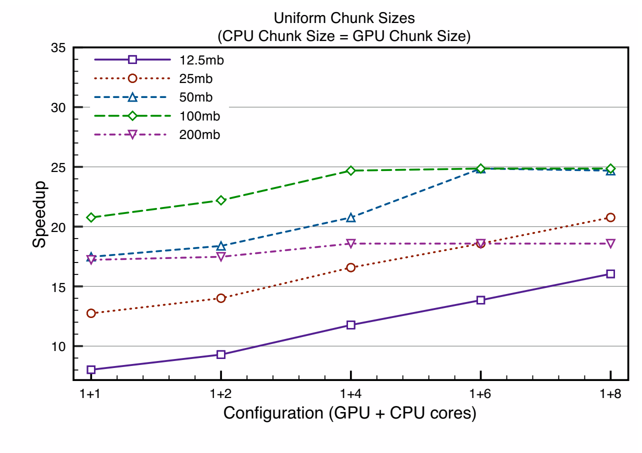

Figure 9: K-Means Using Heterogeneous Version with Uniform Chunk Size

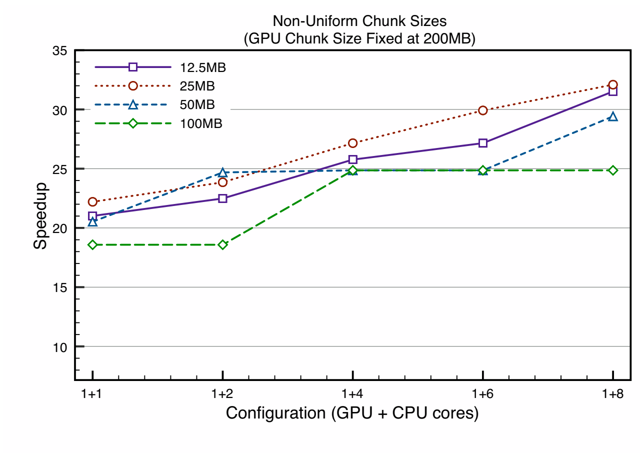

Figure 10: K-Means Using Heterogeneous Version with Non-Uniform Chunk Size

by a factor of 1.75.

One observation from our experiments is that the relative performance of the `CPU-only` and `GPU-only` versions varies significantly between the two applications we have considered. We believe that this is because k-means is highly compute-intensive relative to the number of memory accesses, while in PCA, the reverse is true. Moreover, in k-means, we can exploit the shared memory to achieve high speedups on the GPU.

4.2 Performance of Heterogeneous Configurations

We now focus on execution with heterogeneous configurations, and particularly, we study the impact of chunk size and our dynamic work distribution schemes.

Recall that we had described two dynamic work allocation schemes earlier in this paper. In the first scheme, which we refer as *uniform chunk size* (UCS), each data chunk processed by the CPU and GPU workers is of the same size. In the second case, the data chunks processed by CPU and GPU workers vary in sizes, and this scheme is referred to as *non-uniform chunk size* (NUCS).

Results from K-Means: We first discuss UCS and NUCS results for k-means, which have been juxtaposed as Figure 9 and Figure 10. The x-axis denotes the heterogeneous configuration. For instance, *1+4* indicates the simultaneous usage of 1-GPU and 4-CPU cores. Let us focus first on Figure 9. To illustrate the impact of chunk size, we have varied it from 12.5 MB to 200 MB. For the configuration of *1+8*, we observe that the best performance is attained when chunk size is 100 MB. As we increase the chunk size, the performance of heterogeneous version increases, similar to what we observed previously with the `GPU-only` case. However, as it can be seen in this heterogeneous version, the growth in performance halts at a certain point, and then actually degrades (as in the case when chunk size = 200 MB). This is in contrast to our previous results in Figure 7, which showed that a chunk size of 200 MB was optimal in that independent-GPU version. Another interesting observation is that little, if any, speedup can be observed between the *1+4* and *1+8* configurations for the best case, i.e., when chunk size = 100 MB. Overall, with UCS, the best speedup achieved with the heterogeneous version is nearly 25 times relative to the 1-thread CPU version. Also, compared to the faster of the `CPU-only` and `GPU-only` versions, we are able to achieve a performance improvement of around 23.9%. Also, dynamic scheme does not require any prior information about the computational power ratio of different processors.

Next, we consider the use of the NUCS strategy. In Figure 10, there are four versions, with each corresponding to a particular chunk size assigned to the CPU threads. For the GPU, we

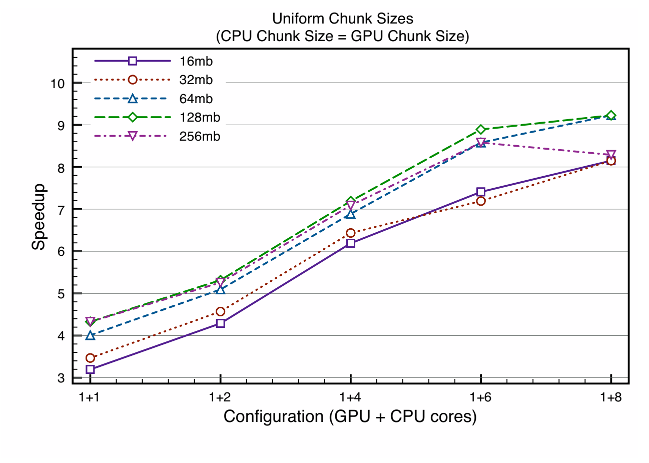

Figure 11: PCA Using Heterogeneous Version with Uniform Chunk Size

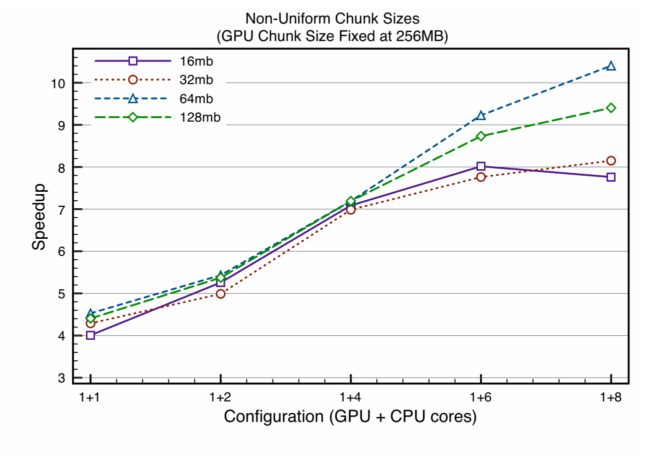

Figure 12: PCA Using Heterogeneous Version with Non-Uniform Chunk Size

K-Means		PCA	
Chunk Size (MB)	Idle %	Chunk Size (MB)	Idle %
100	35.2	128	11.3
200	45	256	16.9

Table 1: % Idle with Uniform Chunk Size

have fixed the chunk size as 200 MB (recall that Figure 7 showed us that a chunk size of 200 MB was optimal for the GPU). The larger GPU chunk size is created dynamically by merging multiple smaller chunks, irrespective of the CPU chunk size. It can be observed from the results that the version with the CPU chunk size of 25 MB tends to perform the best. In fact, the best speedup that we achieved using *1+8* configuration with NUCS is nearly 32 times over the 1-thread CPU version. Also, when compared to the faster of the `CPU-only` and `GPU-only` versions, we have improved the performance by 60%. In comparing the best results from UCS, we have improved the performance by about 36% – a solid indication of the benefits afforded by the NUCS approach for this application. It should also be noted that this 36% performance gain is mostly obtained by *almost* eliminating the time spent by fast processors waiting for slower processors to complete (*idle time*), which we will elaborate later.

Results from PCA: The result from UCS in PCA is shown in the Figure 11. The best performance on the *1+8* configuration is achieved using both 64 MB and 128 MB chunk size. Again, tantamount to the effects noted previously for k-means, PCA's performance increases with growth in chunk size upto a point and then decreases. Overall, a speedup of nearly 9 using the heterogeneous version is obtained when compared to the 1-thread CPU version, and when compared to the faster of `CPU-only` and `GPU-only` version, we are able to gain 45.2% in relative performance.

Next, the results for PCA with NUCS are shown in the Figure 12. Similar to k-means, we again compare the four CPU chunk size variations (between 16 MB and 128 MB). Akin to the earlier experiment, a chunk of size 256 MB is used for GPU processing in all versions (since it was determined in Figure 8 to be optimal for `GPU-only` in PCA). The version with the initial chunk size of 64 MB offers us the best performance. The best speedup that we achieved using *1+8* configuration with non-uniform chunk size is about 10.4x relative to the 1-thread CPU version. Also, it should be recounted that the base version results for PCA suggested that 8-core CPU version was faster than the GPU. And when compared to this version, we have improved the performance by 63.8%. We have furthermore improved the performance of the best results from UCS by 18.6%.

We make the following key observations behind the results from the experiments comparing UCS and NUCS. First, GPUs are faster than a single CPU core, but have a much higher latency in terms of initiating data movement. Thus, a larger chunk size helps lower the latency for GPUs. In comparison, a lower chunk size for CPUs helps improve load balance and reduce the likelihood of any CPUs becoming idle. To further validate this observation, we measured the *idle time*, by which we mean that either the CPU or the GPU is idle, for increasing chunk sizes. These results are shown in Table 1. We show only the results from the *1+8* thread configuration with larger chunk size, since contention between CPU threads is highest here. From the table we can see that for k-means, the faster resource spends about 35% and 45% of the entire execution time waiting for the slower resource to finish at the end of the execution. Similarly for PCA, the faster resource waits for about 11% and 17% of the execution time for the slower one to finish. It should be noted that, for both the applications, the performance improvement in NUCS version over UCS version is achieved by `almost` eliminating this *idle time*.

4.3 Work Distribution

Up till now, our experiments and results mainly focused on the performance that we could achieve using the heterogeneous configuration compared to the best of the `CPU-only` and `GPU-only` versions. It is also interesting, however, to assess the fraction of the work done by each of these resources. We show this analysis in Figure 13 and 14 for k-means and PCA respectively. In both these applications, we are showing results from the UCS and NUCS schemes, with focus on the *1+8* configuration.

Results from K-Means: First consider the version with UCS (on the left half of Figure 13). For smaller chunk sizes, the CPU does majority of the work compared to the GPU. But, on the contrary, we observed in the results from base version that the GPU is much faster than the 8-core `CPU-only` version. This is, however, expected via our previous argument which suggested that smaller chunk sizes cause GPUs to more frequently invoke high-latency device function calls and data transfer between host and device memory. This argument is validated as we observe that an increase in

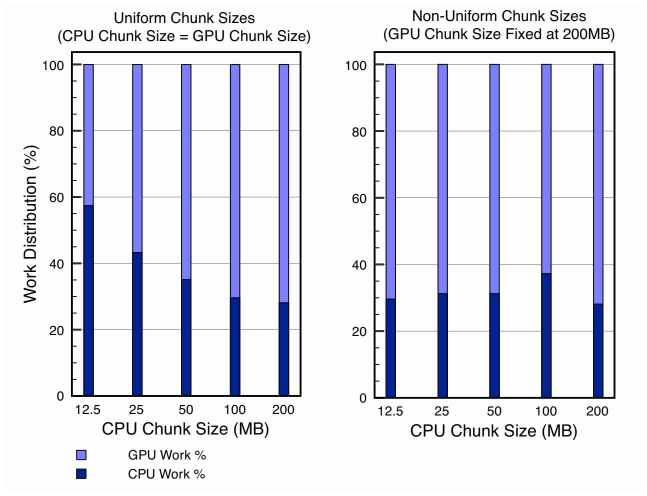

Figure 13: Work Distribution (K-Means)

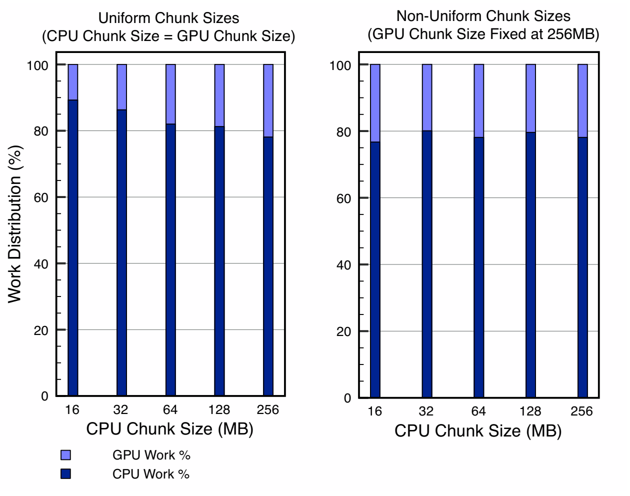

Figure 14: Work Distribution (PCA)

chunk size leads to a corresponding increase in the GPU's proportion of work. We note that the best performance with UCS was obtained when the distribution of work is 70.4% and 29.6% respectively for GPU and CPU threads.

Let us now consider the NUCS version (on the right half of Figure 13). Here we have five versions which indicate the corresponding CPU chunk size. All the five versions achieve a work proportion close to the optimally observed distribution of 70.4% (GPU) and 29.6% (CPU). The GPU transfers a larger chunk (200 MB), effectively reducing data movement overheads. As expected from previous results, the best performance was achieved from the 25 MB chunk size for the CPU. The work proportion corresponding to this configuration was 68.75% and 31.25% for GPU and CPU, respectively.

Results from PCA: We saw that the base version results from PCA showed that, contrary to k-means, 8-core `CPU-only` version is faster than the `GPU-only` version. Consider, first, the UCS results (on the left half of Figure 14). Varying the uniform chunk size for both CPU and GPU only causes the GPU to take on slightly more work. Again, this is due to the heavy data movement nature of PCA. The best performance was obtained in this case when the distribution of work is 18% and 82% for GPU and CPU respectively. Next, we consider NUCS (on the right half of Figure 14). Tantamount to our observation in k-means, all five versions share very similar distribution of work. The best performance is achieved from a configuration with an initial chunk size of 64 MB and the work proportion corresponding to this configuration was 22% and 78% for GPU and CPU respectively.

5. RELATED WORK

We now compare our work against related efforts in language support and application development on heterogeneous architectures, compiler support for GPU programming, and runtime and compiler support for reductions. The technological trend toward systems with varying computing elements has thus invoked much interest in building support for harnessing a common system's full computing power [6, 16]. In one recent example, the Open Computing Language (OpenCL) [14] is a C-based framework for heterogeneous programming, and it has begun to garner implementation support from multiple vendors. One distinct difference between OpenCL and CUDA is that OpenCL is *device agnostic* and enables data parallel programming on CPU, GPU, and any other device. OpenCL, while promising, is currently in the early phases of its realization. Our system differs in its ability to map sequential code written for the class of generalized reduction applications directly to data parallel code, and our runtime is capable of scheduling fairly among the heterogeneous devices.

Other related efforts include Exochi [31], which is a programming environment that utilizes a heterogeneous platform for media kernels. Venkatasubramanian *et. al* [30] have studied a stencil kernel for a hybrid CPU/GPU platform. They have evaluated their hybrid performance on two disparate hardware configurations. Kunzman *et. al* [17] have provided a framework for abstracting operations on cell processors. Helios[22] provides an operating system for tuning applications on heterogeneous platforms. However, it is restricted to satellite kernels. Our effort is distinct from the above in supporting a very high-level programming API and achieving performance through a dynamic work distribution scheme. Qilin [20], a system for a heterogeneous computing environment similar to the one which we describe, is based on an adaptable scheme. The Qilin system trains a model for data distribution, but the amount of training that is required before a model is effectively fitted is unclear. Our work is based on dynamic work distribution schemes, without the requirement for any offline training.

In the last few years, many research efforts have focused on alleviating the difficulties of programming on GPUs. For instance, a group at UIUC is developing CUDA-lite [2] with the goal of reducing the need for explicit GPU memory hierarchy management by programmers. The same group also investigated optimization techniques in CUDA programming [26]. Baskaran *et al.* use the polyhedral model for converting C code into CUDA automatically [3]. A version of Python with support of CUDA, Pycuda, has also been developed, by wrapping CUDA functions and operations into classes that are easy to use [15]. Efforts have also been initiated on translating OpenMP to CUDA [19]. Other research groups have also considered automatic CUDA code generation, though considering restricted domains [21, 28]. Tarditi *et al.* have developed an approach for easing the mapping of data-parallel operations on GPUs [29].

6. CONCLUSIONS

We have developed compiler and runtime support targeting a particular class of applications for such a heterogeneous configuration. Our compilation system generates middleware API for multi-core

CPU, as well as CUDA code for GPU, starting from a sequential C code with some additional annotations. Our runtime system automatically maps the computations to the heterogeneous processing elements. We also discuss two dynamic work distribution schemes that can effectively realize the power of such platforms.

Our experimental results show that significant performance gain can be achieved on heterogeneous configurations using effective work distribution. Our key results are as follows. For the k-means application, the heterogeneous version with 8 CPU cores and 1 GPU, achieved a speedup of about 32.09x relative to 1-thread CPU. When compared to the faster of the `CPU-only` and `GPU-only` versions, a performance gain of about 60% can be observed. For PCA, the heterogeneous version managed a speedup of about 10.4 relative to 1-thread CPU, and when compared to the best of CPU-only and GPU-only versions, we achieved a performance gain of about 63.8%.

Acknowledgements:

This research was supported by NSF grants CCF-0833101 and IIS-0916196.

7. REFERENCES

[1] L. O. Andersen. *Program Analysis and Specialization for the C Programming Language*. PhD thesis, DIKU, University of Copenhagen, 1994.

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

[3] 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.

[4] P. Berenbrink, T. Friedetzky, and L. A. Goldberg. The natural work-stealing algorithm is stable. *SIAM J. Comput.*, 32(5):1260–1279, 2003.

[5] R. D. Blumofe and C. E. Leiserson. Scheduling multithreaded computations by work stealing. In *SFCS '94: Proceedings of the 35th Annual Symposium on Foundations of Computer Science*, pages 356–368, Washington, DC, USA, 1994. IEEE Computer Society.

[6] 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.

[7] G. Cong, S. Kodali, S. Krishnamoorthy, D. Lea, V. Saraswat, and T. Wen. Solving large, irregular graph problems using adaptive work-stealing. In *ICPP '08: Proceedings of the 2008 37th International Conference on Parallel Processing*, pages 536–545, Washington, DC, USA, 2008. IEEE Computer Society.

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

[9] J. Dinan, D. B. Larkins, P. Sadayappan, S. Krishnamoorthy, and J. Nieplocha. Scalable work stealing. In *SC '09: Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*, pages 1–11, New York, NY, USA, 2009. ACM.

[10] 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.

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

[12] 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.

[13] 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.

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

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

[16] R. Kumar, D. M. Tullsen, N. P. Jouppi, and P. Ranganathan. Heterogeneous chip multiprocessors. *Computer*, 38(11):32–38, 2005.

[17] D. M. Kunzman and L. V. Kalé. Towards a framework for abstracting accelerators in parallel applications: experience with cell. In *SC '09: Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*, pages 1–12, New York, NY, USA, 2009. ACM.

[18] C. Lattner and V. Adve. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *Proceedings of the 2004 International Symposium on Code Generation and Optimization (CGO'04)*, Palo Alto, California, Mar 2004.

[19] 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.

[20] C.-K. Luk, S. Hong, and H. Kim. Qilin: exploiting parallelism on heterogeneous multiprocessors with adaptive mapping. In *Micro-42: Proceedings of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture*, pages 45–55, New York, NY, USA, 2009. ACM.

[21] 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.

[22] E. B. Nightingale, O. Hodson, R. McIlroy, C. Hawblitzel, and G. Hunt. Helios: heterogeneous multiprocessing with satellite kernels. In *SOSP '09: Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*, pages 221–234, New York, NY, USA, 2009. ACM.

[23] 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.

[24] K. Pearson. On lines and planes of closest fit to systems of points in space. *Philosophical Magazine*, 2(6):559–572, 1901.

[25] C. Ranger, R. Raghuraman, A. Penmetsa, G. Bradski, and C. Kozyrakis. Evaluating mapreduce for multi-core and multiprocessor systems. In *Proceedings of International Symposium on High Performance Computer Architecture, 2007*, pages 13–24, 2007.

[26] 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.

[27] 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.

[28] N. Sundaram, A. Raghunathan, and S. Chakradhar. A framework for efficient and scalable execution of domain-specific templates on GPUs. In *IPDPS*, 2009.

[29] 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.

[30] 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.

[31] 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.

[32] W. Zhao and J. A. Stankovic. Performance analysis of fcfs and improved fcfs scheduling algorithms for dynamic real-time computer systems. In *IEEE Real-Time Systems Symposium*, pages 156–165, 1989.