AUTO-GC: Automatic Translation of Data Mining Applications to GPU Clusters

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

***Abstract*—Because of the very favorable price to performance ratio of the GPUs, a popular parallel programming configuration today is a cluster of GPUs. However, extracting performance on such a configuration would typically require programming in both MPI and CUDA, thus requiring a high degree of expertise and effort. It is clearly desirable to be able to support higher-level programming of this emerging high-performance computing platform.**

This paper reports on a code generation system that can translate data mining applications on a GPU cluster. Our work is driven by the observation that a common processing structure, that of generalized reductions, fits a large number of popular data mining algorithms. In our solution, the programmers simply need to specify the sequential reduction loop(s) with some additional information about the parameters. We use program analysis and code generation to automatically map the applications to the API of FREERIDE, which is a middleware for parallel data mining. We also automatically generate CUDA code for using the GPU on each node of the cluster.

We have evaluated our system using two popular data mining applications, k-means clustering and Principal Component Analysis (PCA). We observed good scalability over the number of computing nodes, and the automatically generated version did not have any noticeable overheads compared to hand written codes. The speedup obtained by using GPU over using only the CPU on each node of a cluster is between 3 and 21.

***Keywords*-GPGPU, CUDA, Data Mining, cluster**

I. Introduction

As uniprocessor speeds have not been increasing, a popular parallel processing configuration today is a cluster of machines, with an accelerator like GPU on each node. Modern GPUs offer a very favorable price to performance ratio. Thus, even a small cluster of nodes with GPUs can have a very high peak performance. GPU clusters have received a significant attention lately in the parallel computing community [15], [8], [28].

However, extracting performance from such clusters involves a very difficult programmability challenge. Clusters have traditionally been programmed using MPI, whereas GPUs are programmed using CUDA or OpenCL. While both MPI and CUDA have been popular, they both require low-level and explicitly parallel programming. Thus, developing a highly tuned application for a cluster with GPUs requires a lot of programming effort, besides requiring expertise in both. It will clearly be desirable to have compilation and/or runtime systems that can enable higher-level programming of such clusters.

This paper presents such a solution, targeting a particular class of applications. The class of applications we consider are the *data-intensive* applications, including the popular data mining applications. It is very common for a cluster used for data-intensive computing to have visualization capabilities, which means that each node has a powerful graphics card. Thus, we see a good match between data-intensive applications and clusters with GPUs.

In this paper, we offer a runtime and compilation system (AUTO-GC) that is driven by the observation that a common processing structure fits a large number of popular data-intensive applications. The common processing structure is of *generalized reductions*. For applications that follow this structure, parallelization on a cluster can be done by dividing the data instances (or records or transactions) among the nodes. The computation on each node involves reading the data instances in an arbitrary order, processing each data instance, and performing a *local reduction*. The reduction involves only commutative and associative operations, which means the result is independent of the order in which the data instances are processed. After the local reduction on each node, a *global reduction* is performed. A similar method can be used for parallelizing these applications on a GPU.

In our approach, the programmers simply need to specify the sequential reduction loop(s) with some additional information about the parameters. We use program analysis and code generation to map the applications to a distributed memory cluster, and further accelerate the processing by using the GPU. For the former, our code generation system generates API code for a middleware system, FREERIDE, which we had developed in our previous work [18], [17].

We have evaluated our system using two popular data mining applications, k-means clustering and Principal Component Analysis (PCA). The main observations from our experiments are as follows. The automatically generated middleware version did not have any noticeable overheads compared to hand written codes, and has good scalability over the number of computing nodes. The usage of GPU gives a speedup of between 3 and 21, over the parallel code executing just on CPUs.

The rest of the paper is organized as follows. In Section II, we give background on parallel data mining, our middleware

978-1-4244-6534-7/10/$26.00 ©2010 IEEE

FREERIDE, and GPU computing. Details of the code generation in our system are presented in Section III. The results from our experiments are presented in Section IV. We compare our work with related research efforts in Section V and conclude in Section VI.

II. Background

This section gives an overview of the issues in parallelizing datamining applications on clusters and GPUs. We also describe a middleware system, FRamework for Rapid Implementation of Datamining Engines (FREERIDE), developed in our earlier work [18], [17]. For enabling parallelization on clusters, our system automatically generates the code for FREERIDE API.

A. Parallel Datamining and FREERIDE

FREERIDE is based on the observation that parallel versions of several well-known data mining, OLAP, and scientific data processing algorithms share a similar structure, which is that of *generalized reductions*. This observation has some similarities with the *map-reduce* paradigm that Google has developed [7]. There are also some differences in the generalized reductions that FREERIDE supports and the *map-reduce* style of computations. Particularly, the FREERIDE API alleviates the need for expensive sorting of reduction elements, and thus can help achieve better performance on data mining applications.

```
/ * 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();
}
```

Fig. 1. *Generalized Reduction* Processing Structure of Common Datamining Algorithms

The common structure that FREERIDE exploits is summarized in Figure 1. 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 *irregular*, in the sense that which elements of the reduction objects are updated depends upon the results of the processing of an element. For example, in k-means clustering, each iteration involves processing every point in the dataset. For each point, we determine the closest *center* to this point, and compute how this *center* should be updated.

For algorithms following such generalized reduction structure, parallelization can be done by dividing the data instances (or records or transactions) among the processing threads. 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*.

The following functions need to be written by the application developer using FREERIDE.

Reduction: A reduction function specifies how, after processing one data instance, a *reduction object* (initially declared by the programmer), is updated.

Finalize: After final results from multiple nodes are combined into a single reduction object, the application programmer can read and perform a final manipulation on the reduction object to summarize the results specific to an application.

B. GPU Computing for Datamining Applications

GPUs support SIMD shared memory programming. For such a system, 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 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.
- **Writing update:** Copy the reduction objects back to host memory, and do a global combination.

III. System Design and Implementation

This section describes our code generation system, AUTO-GC. Initially, we give an overview. This is followed by the system API, and details of program analysis and code generation for cluster and GPUs.

A. System components

The overall configuration we consider is as follows. The data files to process are distributed among the computing nodes. On each node, when processing the data blocks, the main computing task, which is implemented as the reduction function, can be executed by the GPU. To enable this, our tool, AUTO-GC, generates both the FREERIDE API code and the CUDA code.

The system design is shown in Figure 2. There are three components in the user input: variable information, reduction function(s), and additional optional functions. AUTO-GC comprises to two components, a program analyzer and a code generator. The program analyzer includes the variable analyzer and the code analyzer. The code analyzer obtains variable access patterns and extracts the reduction objects, with a

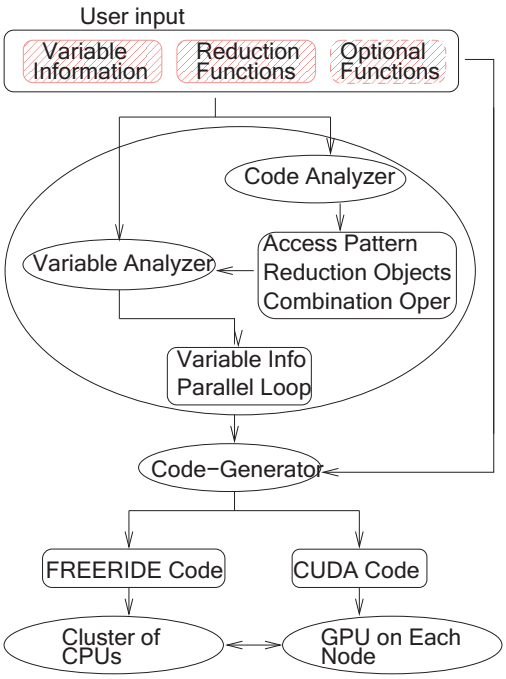

Fig. 2. Overall System Design (User Input is Shown as Shaded Boxes)

combination operation. The variable analyzer extracts variable information which is an input to the code-generator, based on the user input and the code analysis. We used LLVM as the framework for code analysis [21]. We particularly benefited from the clear structure of its Intermediate Representation (IR). After getting the variable information and reduction objects, the code-generator generates the code for the FREERIDE API, and CUDA code for the computation on GPUs. The CUDA functions are invoked by the FREERIDE code for executing the reduction function.

B. System API

Before discussing the program analysis and code generation, we describe the API of the system, i.e., the input that needs to be provided by an application developer.

Using the generalized reduction structure of our target class of applications, we provide a convenient API for the user. The format of input for a reduction function is shown in Figure 3. If there are multiple reduction functions, a user can specify them by including `labels` for each. For each function, the following information is needed.

Variables for Computing: As shown in Figure 3, 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*`. If it is a pointer, `size` is the size of the array it points to, which can be the multiplication of a list of numbers and/or integer variables; otherwise, this field denotes a default `value`. We require all pointers to be one-dimensional, which means the user should marshal the multi-dimensional arrays and structures into 1-D arrays.

label
Variable information:
variable $declare_1$
.....
variable $declare_n$
functions (reduction and some optional functions)

variable declare:
name; type; length[value]

Fig. 3. Format of the User Input

kmeans
step int 0
endcondition int 0
MSE float 999999e+20
k int 10
n int 4096
data float* n 3
update float* 5 k
cluster float* 3 k

Fig. 4. Variable List in the User Input for K-means

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 as shown in Figure 3, 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.

Optional Initialization and Combination Functions from the User: Normally, the initialization and combination for the reduction objects and other variables is done by the code generator component of the system. However, if the user is familiar with CUDA programming, they can provide their own combination and initialization functions, potentially improving the performance.

C. 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 read-only. 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, `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. We obtain such information by analyzing the Intermediate Representation (IR) for the sequential reduction function using LLVM and using Anderson's point-to analysis [2]. More details can be found in our previous paper [26].

```
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 */
  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;
  }
}
```

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

data	*input*
update	*output*
k	*input*
n	*input*
cluster	*input*

Fig. 6. Classification of Variables for K-means Reduction Function

As an example, let us consider the user input for k-means. The two parts used for determining variable access features, variable list and reduction function, are shown in Figure 4 and Figure 5. Figure 6 shows the main part of output obtained by analyzing the IR generated by LLVM for the reduction function.

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 FREERIDE. Because we are focusing on reduction functions where `output` variables are updated with associative and commutative functions only (see Figure 1), the output variables updated by each computing node (and different threads in GPU) can be correctly combined in the end. The operators used are identified by analyzing the IR from LLVM in the similar way as we used in our previous work [26].

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 1. 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.

D. Code Generation for FREERIDE

The issues in generating code for FREERIDE API are as follows. The base class for any application is a template FREERIDE_TECH. For a particular application, we derive its corresponding class from FREERIDE_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. The reduction objects that are to be computed with CUDA needs one copy for each thread.

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, which is done within the ADR framework. Thus, in the Finalize() function, the user can copy the reduction objects to local variables and provide further operations.

To show how the code generation is done, let us take k-means as an example. The user input are shown in Figures 4 and 5. Figure 4 is the variable description, where `step`, `endcondition` and `MSE` are used in testing for termination of the execution, `k` is the number of clusters, `n` is the number of points in the data block, `data` is the input data array, and

```
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]);
  }
}
```

Fig. 7. System Generated Reduction Function of K-means

`update` stores the updates to each cluster, including count, distance, and accumulated point coordinates.

After code analysis, we find that `update` is an `output` variable, so it is determined as the reduction object. In the system generated code, *reductionobject* is updated with the value of `update`, as shown in Figure 7.

E. 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 generate GPU memory allocation and copy functions according to the variable information. 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, in which the device main loop function device_reduc() is called. After device_reduc(), 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 extracted by the code and variable analyzer. 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. 3) Optimizing the use of shared memory. We sort 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].

Other optimizations: Besides the usage of shared memory, we also provide some directives for the user to specify, which can further reduce memory copy between devices. Also, as mentioned in Section III-B, the user can provide their own initialization and combination functions. For example, in PCA, users can provide their own combination function, which reduces unnecessary work [26].

IV. Experimental Results

This section presents an evaluation study with our code generation system, using two popular data mining algorithms. Specifically, we had the following three goals in our experiments:

- Evaluating the overall performance and scalability of the system generated programs, including evaluating gains from using GPUs on each node of a cluster, over the performance on just a cluster of CPUs.
- Comparison of our automatically generated code with a hand-written or manual version, to quantify the overheads of our approach.
- Comparison of the impact of using different computing devices (CPUs and GPUs) for different dataset sizes.

Our experiments were conducted on a 8 node cluster with AMD Opteron 8350 machines, each of which 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.

A. K-means Clustering

k-means [16] clustering is one of the most popular data mining algorithms. In this problem, 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 code generation for k-means was explained with several code examples in Section III-D.

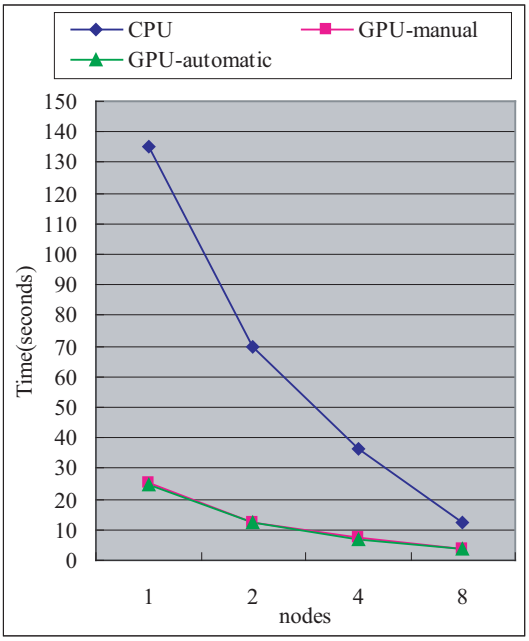

Fig. 8. Execution Time of Different Versions: k-means, 1.5GB dataset

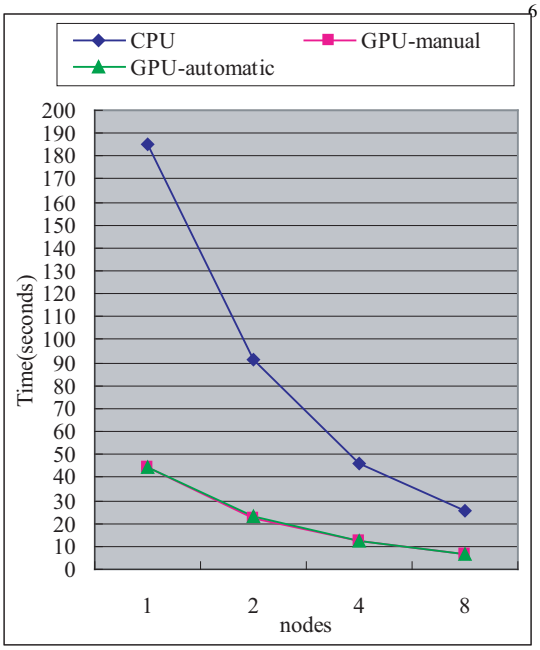

Fig. 9. Execution time of Different Versions: k-means, 3GB dataset

We conducted our tests with three versions: a manually written FREERIDE program that could only use the CPU on each node of the cluster, a manually written FREERIDE application with manually written CUDA code, and finally, a system generated version, where both the FREERIDE API and CUDA codes are automatically generated. These three versions are denoted as `CPU`, `GPU-manual` and `GPU-automatic`.

We experimented with two different datasets, which were 1.5 GB and 3 GB, respectively. The number of dimensions is 3 and the parameter k, the number of clusters to be obtained, is set at 10. In the figures, we only show the performance with the best configuration for all the CUDA version, which is 16 thread blocks and 256 threads per block.

The first set of results are from a dataset with 1.5 GB, and are shown in Figure 8. We can see that each of the three versions has a good scalability, as the number of nodes in the cluster is increased. The automatically generated CUDA code has almost the same performance with the manual CUDA code, which shows that we can generate very efficient code for programs that fall in the structure we specified in Figure 1. The GPU versions have a speedup of more than 5, over the CPU version, when we are using 4 nodes or less. The relative speedup, over the CPU version using the same number of nodes, reduces to 3 when 8 nodes are used. This is because the amount of data to be processed on each node becomes smaller as the same dataset is processed on more nodes. Thus, the execution time gets more dominated by the overheads of initiating the processing on the GPUs, and the global combination time. In Figure 9, the results are shown for the 3 GB dataset. The relative performance of different versions follows a similar trend.

B. Principal Component Analysis

Principal Components Analysis is a popular dimensionality reduction method. This method was developed by Pearson in 1901. Our experiments are conducted on a modified version of F. Murtagh's code [1]. There are four passes on the data set. 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. Finally, the projection of the row points and column on the first m components are written to the output files.

We did not provide a manual CUDA version for PCA, since the functions to be converted to CUDA are relatively simple, and the automatic generated CUDA will be very similar to hand written versions. Thus, the two versions we compare with PCA are as follows. One is the FREERIDE-based CPU version, without using GPUs, denoted as `CPU`, which is written manually. The other is the system generated CUDA version, denoted as `GPU-automatic`. In `GPU-automatic`, we generated FREERIDE APIs, and CUDA code for the computing of `mean`, `standard deviation` and the entire correlation matrix. For all the GPU executions, we used the best configuration, which is 128 threads per block, and 16 blocks in total for one GPU.

Figure 10 shows the performance of the two versions on a data set with 64 M rows and 3 principal components. The CPU-based FREERIDE code has good scalability as the number of nodes is increased. The CUDA version also scaled well, but the performance is worse than the version without GPU. This is because of the domination of I/O with these parameters. Without much computation, the use of GPU does not help improve performance.

In Figure 11, we use a dataset with 2M rows and 64 principal components, and projection is done on the first 6 components. With these parameters, PCA is extremely compute-intensive. Thus, the benefits of using GPU are very significant. The performance of `GPU-automatic` on 1 node has a relative speedup of about 21 over the CPU versions on

[1] http://www.mirrorservice.org/sites/lib.stat.cmu.edu/multi/pca.c

1, 2, 4, and 8 nodes. Because of the compute-intensive nature of this application with these parameters, the benefits of using the GPU are much higher.

From the experiments, we can see that the benefits of using the GPU can vary widely depending upon the nature of the application and the parameters. In the future, we will like to develop cost models that can predict whether or not moving an application to a GPU will be beneficial.

V. Related Work

We now compare our work with related efforts on application development on GPU Clusters, automatic generation or optimization of CUDA, and compiler support for reductions.

There has been a significant interest in exploiting the computing power of GPU-based clusters. At Stony Brook, parallel LBM computation was implemented on GPU clusters, obtaining a relative speedup of nearly 7 by using GPU on 1 node, and about 5 where there were more than 4 nodes [8]. Göddeke *et. al* implemented a multigrid solver on a GPU cluster [9]. To the best of our knowledge, our work is the first to automatically generate code for a cluster of GPUs.

Within the last 2 years, there has been considerable amount of work on automatic generation and/or optimization of CUDA. At UIUC, CUDA-lite [3] is being developed with the goal being to alleviate the need for explicit GPU memory hierarchy management by the programmers. The user input to our system is at a higher-level, in the sense that the programmers do not need to write parallel code. In addition, we are able to target cluster of GPUs. However, our system is limited to a specific class of applications. The same research group is also developing optimizations on CUDA programs [30]. Baskaran *et al.* [5] use the polyhedral model for converting C code into CUDA automatically. Their system is limited to affine loops, and cannot handle irregular reductions we focus on. 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 [19]. Some recent work has also made progress in translating OpenMP into CUDA [22]. The reported results are from simple stencil computations, and there is no support for handling complex reductions. Another group has made an effort in scheduling and separating operators according to the input data set size [31]. In compare to these efforts, our focus is on cluster of GPUs, but our work is restricted to a limited class of applications. Work based on HMPP+TAU provides compiling support for heterogeneous systems, including code generation for CUDA and HMPP application [27]. This system deals with annotated parallel loops, which is simpler than the applications we focused on, but their event driven model and kernel pipelining will be considered in our future work.

Analysis and code generation for reduction operations has been studied by a number of distributed memory compilation projects [1], [4], [10], [14], [20], [33] as well as shared memory parallelization projects [6], [11], [12], [24], [25], [29], [32]. More recently, reductions on emerging multi-cores have also been studied [23]. Our work has many similarities, but has leveraged the features of GPUs. Map-Reduce is a popular framework developed by Google [7], which can be used for data mining applications we target. A GPU version of Map-Reduce, called Mars [13], is also available now. Our approach is based on automatic code generation, and the programmer input is at a higher-level. In addition, we also support a cluster of GPUs.

The work presented here is an extension of our previous work on generating code for a single GPU [26], and the earlier work on the FREERIDE system [18], [17].

VI. Conclusions

This paper has introduced a system to generate code for a cluster of GPUs for a restricted class of applications. We have evaluated our system using two popular data mining applications, k-means clustering and Principal Component Analysis (PCA). We obtained a good scalability with the increasing number of computing nodes. The relative speedup from using GPUs in a cluster was between 3 and 21, as compared to just using CPUs on the same cluster. The code automatically generated by our system did not have any noticeable overheads compared to hand written codes.

References

[1] Vikram Adve and John 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] Lars Ole Andersen. Program analysis and specialization for the c programming language. Technical report, 1994.

[3] Sara Baghsorkhi, Melvin Lathara, and Wen mei Hwu. CUDA-lite: Reducing GPU Programming Complexity. In *LCPC 2008*, 2008.

[4] Prithviraj Banerjee, John A. Chandy, Manish Gupta, Eugene W. Hodges IV, John G. Holm, Antonio Lain, Daniel J. Palermo, Shankar Ramaswamy, and Ernesto Su. The Paradigm Compiler for Distributed-Memory Multicomputers. *IEEE Computer*, 28(10):37–47, October 1995.

[5] Muthu Manikandan Baskaran, Uday Bondhugula, Sriram Krishnamoorthy, J. Ramanujam, Atanas Rountev, and P. Sadayappan. A Compiler Framework for Optimization of Affine Loop Nests for GPGPUs. In *International Conference on Supercomputing*, pages 225–234, 2008.

[6] 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, December 1996.

[7] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: Simplified data processing on large clusters. In *OSDI*, pages 137–150, 2004.

[8] Zhe Fan, Feng Qiu, Arie Kaufman, and Suzanne Yoakum-Stover. GPU Cluster for High Prformance Computing. In *SC '04: Proceedings of the 2004 ACM/IEEE conference on Supercomputing*, page 47, Washington, DC, USA, 2004. IEEE Computer Society.

[9] Dominik Göddeke, Robert Strzodka, Jamaludin Mohd-Yusof, Patrick McCormick, Hilmar Wobker, Christian Becker, and Stefan Turek. Using gpus to improve multigrid solver performance on a cluster. *Int. J. Comput. Sci. Eng.*, 4(1):36–55, 2008.

[10] Manish Gupta and Edith 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, January 1996.

[11] M. Hall, S. Amarsinghe, B. Murphy, S. Liao, and M. Lam. Maximizing Multiprocessor Performance with the SUIF Compiler. *IEEE Computer*, (12), December 1996.

[12] H. Han and Chau-Wen Tseng. Improving Compiler and Runtime Support for Irregular Reductions. In *Proceedings of the 11th Workshop on Languages and Compilers for Parallel Computing*, August 1998.

[13] Bingsheng He, Wenbin Fang, Qiong Luo, Naga K. Govindaraju, and Tuyong Wang. Mars: A MapReduce Framework on Graphics Processors. In *PACT08: IEEE International Conference on Parallel Architecture and Compilation Techniques 2008*, 2008.

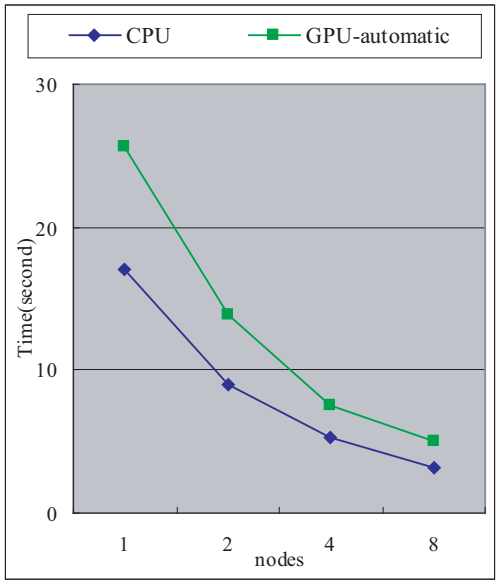

Fig. 10. Execution Time of Different Versions, PCA, 64M rows, 3 principal components

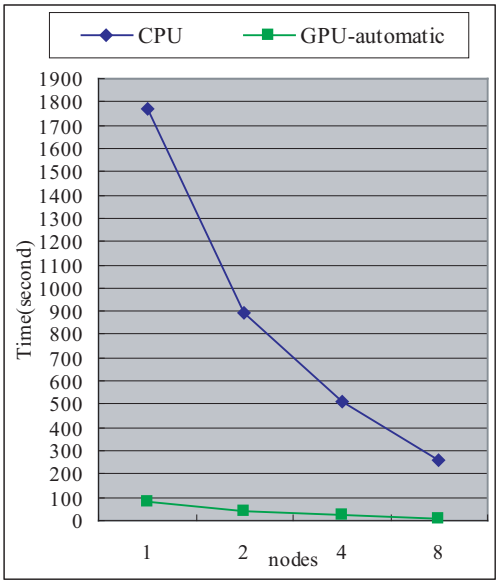

Fig. 11. Execution Time of Different Versions, PCA (log(time)), 2M rows, 64 principal components

[14] Seema Hiranandani, Ken Kennedy, and Chau-Wen Tseng. Compiling Fortran D for MIMD distributed-memory machines. *Communications of the ACM*, 35(8):66–80, August 1992.

[15] Mike Houston. Gpu computation on clusters. 2006.

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

[17] R. Jin and G. Agrawal. Shared memory parallelization of data mining algorithms: Techniques. citeseer.ist.psu.edu/article/jin02shared.html, 2002.

[18] Ruoming Jin and Gagan Agrawal. A Middleware for Developing Parallel Data Mining Implementations. In *Proceedings of the first SIAM conference on Data Mining*, April 2001.

[19] Andreas Klockner. PyCuda, 2008.

[20] 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, October 1991.

[21] Chris Lattner and Vikram 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.

[22] Seyong Lee, Seung-Jai Min, and Rudolf Eigenmann. OpenMP to GPGPU: A Compiler Framework for Automatic Translation and Optimization. In *PPoPP'09*, 2009.

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

[24] Yuan Lin and David 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] Bo Lu and John Mellor-Crummey. Compiler Optimization of Implicit Reductions for Distributed Memory Multiprocessors. In *Proceedings of the 12th International Parallel Processing Symposium (IPPS)*, April 1998.

[26] Wenjing Ma and Gagan 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] Allen D. MALONY, Shangkar MAYANGLAMBAM, Laurent MORIN, Matthew J. SOTTILE, Stephane BIHAN, Sameer S. SHENDE, and Francois BODIN. Performance tool integration in a gpu programming environment: Experiences with tau and hmpp. September 2009.

[28] Fritz McCall and Brad Erdman. A prototype cpu-gpu cluster for research in high performance computing and visualization of large scale applications.

[29] William 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.

[30] Shane Ryoo, Christopher Rodrigues, Sam Stone, Sara Baghsorkhi, Sain-Zee Ueng, John Stratton, and Wen 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] Narayanan Sundaram, Anand Raghunathan, and Srimat Chakradhar. A framework for efficient and scalable execution of domain-specific templates on gpus. In *IPDPS*, 2009.

[32] Hao Yu and Lawrence Rauchwerger. Adaptive Reduction Parallelization Techniques. In *Proceedings of the 2000 International Conference on Supercomputing*, pages 66–75. ACM Press, May 2000.

[33] Hans P. Zima and Barbara Mary Chapman. Compiling for Distributed-Memory Systems. *Proceedings of the IEEE*, 81(2):264–287, February 1993. In Special Section on Languages and Compilers for Parallel Machines.