A Translation System for Enabling Data Mining Applications on GPUs

Wenjing Ma
Department of Computer Science and Engineering
The Ohio State University
Columbus, OH
mawe@cse.ohio-state.edu

Gagan Agrawal
Department of Computer Science and Engineering
The Ohio State University
Columbus, OH
agrawal@cse.ohio-state.edu

ABSTRACT

Modern GPUs offer much computing power at a very modest cost. Even though CUDA and other related recent developments are accelerating the use of GPUs for general purpose applications, several challenges still remain in programming the GPUs. Thus, it is clearly desirable to be able to program GPUs using a higher-level interface.

In this paper, we offer a solution that targets a specific class of applications, which are the data mining and scientific data analysis applications. 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 map the applications to a GPU. Several additional optimizations are also performed by the system.

We have evaluated our system using three popular data mining applications, k-means clustering, EM clustering, and Principal Component Analysis (PCA). The main observations from our experiments are as follows. The speedup that each of these applications achieve over a sequential CPU version ranges between 20 and 50. The automatically generated version did not have any noticeable overheads compared to hand written codes. Finally, the optimizations performed in the system resulted in significant performance improvements.

Categories and Subject Descriptors

D.1.3 [**Concurrent Programming**]: Parallel Programming

General Terms

Design

Keywords

GPGPU, CUDA, Data Mining

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
ICS'09, June 8–12, 2009, Yorktown Heights, New York, USA.
Copyright 2009 ACM 978-1-60558-498-0/09/06 ...$5.00.

1. INTRODUCTION

The availability of large datasets and increasing importance of data analysis for scientific discovery is creating a new class of high-end applications. Recently, the term *Data-Intensive SuperComputing* (DISC) has been gaining popularity [8], and includes applications that perform large-scale computations over massive datasets. This class of applications includes data mining and scientific data analysis. Developing new data mining algorithms for scientific data processing has been an active topic for at least the past decade.

With increasing dataset sizes, need for interactive response from analysis tools, and recent trends in computer architecture, we believe that this area is facing a significant challenge with respect to achieving acceptable response times. Starting within the last 3-4 years, it is no longer possible to improve processor performance by simply increasing clock frequencies. As a result, multi-core architectures and accelerators like Field Programmable Gate Arrays (FPGAs) and Graphics Processing Units (GPUs) have become cost-effective means for scaling performance.

Modern GPUs offer an excellent performance to price ratio for scaling applications. Furthermore, the GPU computing capabilities and programmability continue to improve rapidly. A very significant recent development had been the release of CUDA (Compute Unified Device Architecture) by NVIDIA. CUDA allows GPU programming with C language-like features, thus easing the development of non-graphics applications on a GPU. More recently, OpenCL seems to be emerging as an open and cross-vendor standard for exploiting computing power of both CPUs and GPUs.

Even prior to these developments, there had been a growing interest in the use of GPUs for non-graphics applications [9, 11, 14, 17, 19, 21, 43], as also documented in the GPGPU (General Purpose computing with GPUs) web-site[1]. There are several reasons why it is desirable to exploit GPU computing power for data mining applications. Users with a single desktop usually have a powerful GPU to support their graphics applications. Such users can speedup their data mining implementations with this GPU. In other scenarios, a cluster may be available for supporting large-scale data processing. Such clusters often need to have visualization capabilities, which means that each node has a powerful graphics card.

Even though CUDA (and now OpenCL) are accelerating the use of GPUs for general purpose applications, several challenges still remain in programming the GPUs. Both CUDA and OpenCL involve explicit parallel programming, and explicit management of its complex memory hierarchy. In addition, allocating device memory, data movement between CPU and device memory, data movement between memory hierarchies, and specification of thread grid configurations is explicit. This implies a significant learning curve

[1] www.gpgpu.org

for programmers who want to improve the performance of their applications using the GPUs. Thus, it will clearly be desirable to be able to program GPUs using a higher-level interface. Furthermore, as we will show in this paper, application performance on GPUs can be optimized through methods that are not very obvious or intuitive. Such optimizations can be easily and automatically performed through an automatic code generation system.

In this paper, we offer a solution that is driven by the observation that a common processing structure fits a large number of popular data mining applications. We had earlier made the observation that parallel versions of several well-known data mining techniques share a relatively similar structure [29, 28]. We carefully studied parallel versions of apriori association mining [2], Bayesian network for classification [13], k-means clustering [27], k-nearest neighbor classifier [24], artificial neural networks [24], and decision tree classifiers [37]. In each of these methods, parallelization can be done by dividing the data instances (or records or transactions) among the nodes or threads. 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. Thus, we can expect similarities in how they can be ported on GPUs.

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 map the applications to a GPU. Several additional optimizations are also performed by the middleware. In addition, we allow the programmers to provide other functions and annotation, which can help achieve better performance. Overall, our work shows that very simple program analysis and code generation techniques can allow us to support a class of applications on GPUs with a higher-level interface than CUDA and OpenCL.

We have evaluated our system using three popular data mining applications, k-means clustering, EM clustering, and Principal Component Analysis (PCA). The main observations from our experiments are as follows. The speedup that each of these applications achieve over a sequential CPU version ranges between 20 and 50. The automatically generated middleware version did not have any noticeable overheads compared to hand written codes. Finally, the optimizations performed in the system resulted in significant performance improvements.

The rest of the paper is organized as follows. In Section 2, we give background on GPUs and GPGPU. In Section 3, we discuss parallel data mining algorithms and give an overview of our system. Details of implementations of our system are presented in Section 4. The results from our experiments are presented in Section 5. We compare our work with related research efforts in Section 6 and conclude in Section 7.

2. GPU AND GPGPU

Our work has used GeForce 8800 GTX and 9800 GX2 graphics cards. In this section, we give a brief description of the architecture and programming model of 8800 GTX card, which is also common to many other newer cards.

This particular device has 16 multiprocessors, each with a 575 MHz core clock and 16 KB of shared memory. The device memory totals 768 MB, with memory bandwidth of 86.4 GB/sec and 384-bit memory interface. Starting with the 8 Series GeForce, NVIDIA has started supporting high-level programming of the GPUs through CUDA, which is a C-like parallel language. The computation to be performed by the device can be written as in normal C, with some predefined parameters and functions. Critical parameters of the computation, such as the configuration of the thread grid and size of shared memory to be used, have to be supplied by the developer explicitly.

The kernel function is executed by the GPU in a SIMD manner, with threads executing on the device organized as a grid of *thread blocks*. Threads in one block have access to the same *shared memory*, which is a small piece of memory with high access speed. A mechanism for thread synchronization within one block is provided [38]. Each thread block is executed by one multiprocessor, and the threads within a block are launched in *warps*. Warps of threads are picked by the multiprocessor for execution, the exact order is undefined. The number of threads in a warp is fixed for a particular architecture. In the GeForce 8800 GTX model that we used, 32 threads are launched in every warp. The number of thread blocks, however, can be varied by the developer based on requirements of computation or other preferences, with the maximum number being 65536 in one grid.

```
#define BLOCK 8
#define THREADS 256
void compute(int* A, int* v, int n)
{
   int* A_d, *v_d;
   CUDA_SAFE_CALL(cudaMalloc((void**) &A_d,
   n* sizeof(int) ));
   CUDA_SAFE_CALL(cudaMemcpy(A_d, A, n * sizeof(int),
     cudaMemcpyHostToDevice ));
   CUDA_SAFE_CALL(cudaMalloc((void**) &v_d,
   n* sizeof(int) ));
   CUDA_SAFE_CALL(cudaMemcpy(v_d, v, n * sizeof(int),
     cudaMemcpyHostToDevice ));
   dim3 grid(BLOCK, 1, 1);
   dim3 threads(THREADS, 1, 1);
   add_device<<< grid, threads, 0 >>>(A_d, v_d, n);
   CUDA_SAFE_CALL(cudaMemcpy(v, v_d, n * sizeof(int),
     cudaMemcpyDeviceToHost ));
   CUDA_SAFE_CALL(cudaFree(A_d));
   CUDA_SAFE_CALL(cudaFree(v_d));
}
__global__ void add_device(int* A_d, int* v_d, int n)
{
   const unsigned int bid=blockIdx.x;
   const unsigned int tid=threadIdx.x;
   __syncthreads();
   for(int i=0;i < n;i+=THREADS*BLOCK)
        v_d[i+bid*THREADS+tid]+=A_d[i+bid*THREADS+tid];
  __syncthreads();
}
```

Figure 1: Sample CUDA program

To illustrate how GPUs are programmed with CUDA, let us consider the example in Figure 1. In this simple code, we add the values of each element in array A[] to v[]. A[] and v[] are arrays of n integers. `compute()` is the function that invokes the kernel on the device. `add_device()` is the kernel function. The directive `__global__` implies that this function is called by host and executed on device. First, A[] and v[] are copied to device memory, then the kernel function is configured and invoked. After the kernel function returns, values of v[] are copied back to host memory. In this example, shared memory is not used.

OpenCL, which is the emerging open and cross-vendor standard, offers similar programming abstractions. An example code

can be found from the Wiki entry for OpenCL (Please refer to http://en.wikipedia.org/wiki/OpenCL).

3. SYSTEM DESIGN

Though CUDA and OpenCL are accelerating the use of GPUs for non-graphics applications, it still requires explicit parallel programming. Moreover, the programmers are also responsible for managing the memory hierarchy and for specifying data movement. As we can see from the example in Figure 1, knowledge of CUDA functions for invoking procedures, allocating memory, and data movement is also needed.

Our system is designed to ease GPU programming for a specific class of applications. Besides a C program to be executed on CPUs, the only required input from programmers is explicit recognition of reduction functions to be parallelized on GPUs, with additional information about the variables. Given such user input, the system can generate CUDA functions that execute these reduction functions in parallel, and the host functions invoking them. While the current implementation targets CUDA, we believe that the system can be easily extended to generate OpenCL code as well.

The architecture of the system is shown in Figure 2. There are four components in the user input. The first three are analyzed by the system, they are: variable information, reduction function(s), and additional optional functions. The fourth component is the host program. The system itself has three components: code analyzer, which obtains variable access patterns and combination operations, variable analyzer, and the code generator. By analyzing the variables and the sequential reduction function(s), the system generates the kernel functions, grid configuration, and other necessary code. By compiling these functions with the the user-specified host program, an executable file is generated.

We used LLVM as the framework for program analysis [32]. We particularly benefited from the clear structure of its Intermediate Representation (IR).

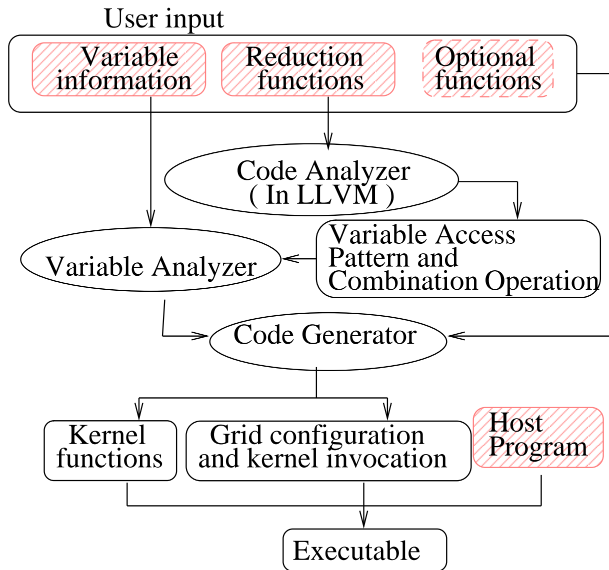

Figure 2: Overall System Design: User Input is Shown as Shaded Boxes

3.1 Parallel Data Mining

Our system exploits a common structure underlying most data-intensive and data mining algorithms. In our previous work [29, 28], we have made the observation that parallel versions of several well-known data mining techniques share a similar structure. We have carefully studied parallel versions of apriori association mining [2], Bayesian network for classification [13], k-means clustering [27], k-nearest neighbor classifier [24], artificial neural networks [24], and decision tree classifiers [37].

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

Figure 3: *Generalized Reduction* Processing Structure of Common Datamining Algorithms

The common structure behind these algorithms is summarized in Figure 3. The function *op* is an associative and commutative function. Thus, the iterations of the *foreach* loop can be performed in any order. The data-structure *Reduc* is referred to as the reduction object. The reduction performed is, however, *irregular*, in the sense that which elements of the reduction objects are updated depends upon the results of the processing of an element. For example, in k-means clustering, each iteration involves processing each point in the dataset. For each point, we determine the closest *center* to this point, and compute how this *center* should be updated.

The generalized reduction structure we have identified from data mining algorithms has some similarities with the *map-reduce* paradigm that Google has developed [15]. It should be noted that our first work on generalized reduction observation with regard to parallel data mining algorithms was published in 2001 [29], prior to the *map-reduce* paper by Dean and Ghemawat in 2004. There are also some differences in the generalized reductions that we focus on and the *map-reduce* style of computations.

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*.

Our system targets GPU-based parallelization of only the functions that follow this structure. By targeting a limited class of functions, we can simplify program analysis and automatic generation of GPGPU programs, while still offering a simple and high-level interface for the programmers.

3.2 System API

Using the common generalized reduction structure of our target applications, we provide a convenient API for a programmer. The format of input for a reduction function is shown in Figure 4. If there are multiple reduction functions, for example, the E phase and M phase in EM clustering, the user can define more than one section by specifying `labels` for each one. A host program, not shown in Figure 4, invokes these reduction functions. Besides the label and the host program, the other components are as follows.

Variables for Computing: As shown in Figure 4, the declaration of each variable follows the following format:

`name, type, length[value]`

`name` is the name of the variable, `type` can be either a numeric type like `int` or pointer type like `int*` , which indicates an array. If this is a pointer, `length` is the size of the array, which can be

```
label
Variable information:
  variable_declare_1
  variable_declare_2
  ......
  variable_declare_n
.
functions // reduction and some optional functions
.

variable_declare:
  name, type, length[value]
```

Figure 4: Format of the User Input

a list of numbers and/or integer variables, and the size of the array is the multiplication of these terms. Otherwise, this field denotes a default `value`. We require all pointers to be one-dimensional, which means the user should marshal the multi-dimensional arrays and structures into 1-D arrays.

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 4, and memory allocation for these variables is not needed.

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.

An example of user input for the k-means clustering algorithm is shown in Figure 5. The first line is the number of reduction functions, which is 1 here. The second line is the label `kmeans`. The following 5 lines are variable descriptions. Then, a sequential reduction function is provided.

```
1
kmeans
k int
n int
data float* n 3
update float* 5 k
cluster float* 3 k

void device_reduc(float* data, float* cluster, float* update,
int k, int n)
{
  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; }
      }
      update[5*min_index] += mydata[0];
      update[5*min_index+1] += mydata[1];
      update[5*min_index+2] += mydata[2];
      update[5*min_index+3] += 1;
      update[5*min_index+4] += min;
  }
}
.
```

Figure 5: User Input for k-means

4. SYSTEM IMPLEMENTATION

This section describes the implementation of our system.

4.1 Code and Variable Analysis

The program analysis part comprises of three components. The first of these components is obtaining variable access information from a reduction function.

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

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

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

Figure 6: Classification of Variables for K-means Reduction Function

As an example, let us consider the user input for k-means that we had shown earlier in Figure 5. The output obtained by analyzing the IR generated by LLVM for the reduction function in shown in Figure 6.

Variable Analysis and Parallelization: The variable analysis phase focuses on identifying how the reduction loop should be parallelized and if variables should be distributed or replicated.

We proceed by mapping the structure of the loop being analyzed to the canonical reduction loop we had shown earlier in Figure 3. 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.

Next, we focus 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 a loop variable is that it can be distributed among the threads. All other variables need to be replicated, if they are written in the loop.

Extracting the Combination Operations: After local reduction is done by each thread, we need to combine their `output` variables, which are then copied to the host memory. Because we are focusing on reduction functions where `output` variables are updated with associative and commutative functions only (see Figure 3), the output variables updated by different threads can be correctly combined in the end. However, we need to identify the particular associative and commutative operator that is being used.

Earlier, we had generated the `point-to` sets for each parameter of the reduction function. We now conduct a new scan on the IR to find the reduction operator for each `output` variable. In the combination function, the values for a particular output parameter from each thread is combined using this function.

4.2 Mapping to GPGPU

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

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

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

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

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

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

Generating Local Reduction Function: device_reduc() is the main loop to be executed on the GPU. This function is generated by rewriting the original sequential code in the user input, according to the information generated by the code and variable analyzer phases. The modifications include: 1) Dividing the loop to be parallelized by the number of blocks and number of threads in each block. 2) Rewriting the index of the array which are distributed. For example, we have an access to `data[i]`, it is changed to `data[i+index_n]`, where `index_n` is the offset for each thread in the entire grid. 3) Optimizing the use of shared memory, which we will discuss later.

4.3 Optimizations

We now describe two key optimizations that are implemented in our system.

4.3.1 Dealing with Shared Memory

Shared memory is a fast but very small read-write memory on the GPU. By making effective use of this memory, the performance of GPU applications can be improved dramatically. In various application studies that have been reported on GPUs, users have obtained significantly better performance with effective use of shared memory. However, because of its very small size, deciding which variables to put into shared memory is quite challenging.

Now we will describe the mechanisms we have developed in our system, to make the use of shared memory transparent to the programmers. First, the amount of shared memory that each array needs is calculated with the following expression:

$Size = length$ * sizeof($type$) * $thread_info$

Here, `length` is the length of this variable, `type` is one of `char, int, float`. The last factor `thread_info` is 1 if *input* or *loop* is true, and `n_threads` otherwise. It implies that if an array is read-write and not distributed over all threads, we need `n_threads` copies of it.

To keep our system simple, we have focused on techniques that do not require advanced program analysis. The three schemes we have developed are as follows:

No sorting: In this intuitive approach, the variable declarations are examined one by one. We simply allocate variables to shared memory as long as the memory requirements of all variables allocated do not exceed the total size of the shared memory.

Greedy-sorting: Thus, in this approach, all the arrays are sorted with increasing order of their `size`. We select the variables to allocate onto shared memory from the beginning of this sorted array list, until the size of data on shared memory exceeds its limit.

Write-first sorting: We found a non-intuitive optimization for the shared memory. By allocating variables that are updated in the reduction function at the lowest addresses in the shared memory, we can further improve performance. Thus, our write-first sorting is a variant of the greedy-sort strategy, where we insert variables that are written at the beginning of the sorted list.

4.3.2 Reducing Memory Allocation and Copy Overheads

Memory allocation and data movement overheads can be significant on GPUs. To enable optimization of these costs, we allow users to specify additional directives. Particularly, in applications where a reduction function is invoked repeatedly, or where multiple reduction functions are invoked, user directives can help reduce memory allocation and data movement overheads.

As part of the input file, a user can use two directives, `common` and `extern`, to indicate the features of certain variables. When a variable is denoted as `common`, we allocate memory for this variable only in the first invocation of the reduction function, and not in subsequent iterations. Similarly, when a variable is declared as

extern, it implies that the variable neither needs to be allocated in memory nor to be copied from host memory for this particular reduction function. This means that we expect a valid copy of this variables from the invocation of an earlier reduction function. For example, in our experiments with EM clustering, some of the variables can be declared as common for the E phase, and extern for the M phase. This is because an allocation and copy is needed only for the first invocation of the E phase reduction function, and not for M phase reduction functions, or subsequent invocation of E phase reduction functions. In the future, we will like to use inter-procedural analysis between the host function and various reduction functions to automate the identification of extern and common variables.

5. APPLICATIONS AND EXPERIMENTAL RESULTS

This section reports on three data mining applications we ported on GPUs with our system. We also present a detailed evaluation study. Specifically, we had the following three goals in our experiments:

- Evaluating the overall performance of the system generated programs, measured as their speedup over a single threaded program executed on a CPU.
- Comparison of our system or middleware approach with a manual version, to understand performance advantages or disadvantages of our approach.
- Evaluation of the benefits from a number of optimizations we have implemented in our system.

The sequential baseline executions were obtained on a Dell Dimension 9200 PC. It is equipped with Intel(tm) $Core^{TM}$ 2 E6420 Duo Processor with 2.13 GHz clock rate, 1GB Dual Channel DDR2 SDRAM memory at 667 MHz, a 4MB L2 cache and a 1066 MHz front side bus. The GPU versions used the same CPU, and a 768MB NVIDIA GeForce 8800 GTX, with 16 multiprocessors and 16KB shared memory on each multiprocessor. Some of our experiments were also performed using the GeForce 9800 GX2 card.

5.1 K-means Clustering

Clustering is one of the key data mining problems and k-means [27] is one of the most popular algorithms. The clustering problem is as follows. We consider transactions or data instances as representing points in a high-dimensional space. Proximity within this space is used as the criterion for classifying the points into clusters. Four steps in the sequential version of k-means clustering algorithm are as follows: 1) start with k given centers for clusters; 2) scan the data instances, for each data instance (point), find the center closest to it and assign this point to the corresponding cluster, 3) determine the k centroids from the points assigned to the corresponding center, and 4) repeat this process until the assignment of points to cluster does not change.

The user input was shown earlier in Figure 5. In the variable description, k is the number of clusters, n is the length of the data block, data is the input data, and update stores reduction objects.

The performance of automatically generated programs on a 384 MB dataset is shown in Figure 7. All results are reported as a speedup over a sequential version execution on the 2.13 GHz CPU. Since the execution time does not change over iterations, we only show the execution time of the first 2 iterations. On the X scale, n*m implies executions with m block and n threads per block. The execution time on GPUs had two distinct components: the computation time, and the time spent moving data and results between the CPU and the GPU. We report two different speedup numbers. The computing speedups show the ratio between the execution time on the CPU and the computing time on the GPU. The computing with copy speedups show the ratio between the execution time on the CPU and the total execution time (including data movement time) using the GPU.

We also repeated the same experiment using GeForce 9800GX2. The results are shown in Figure 8. The speedups are somewhat lower than that on GeForce 8800GTX. This is because the memory bandwidth on 1 GPU of GeForce 9800GX2 is lower than that on GeForce 8800. As there was only a small difference in the performance between these two cards, we only report results from 8800GTX card in the rest of this section.

Input: k, # of clusters,
    $Y = \{y_1 \ldots y_n\}$, set of n p-dimensional points,
    ϵ, a tolerance for loglikelihood,
    $maxiterations$, maximum number of iterations.
Output: C, R, W, the matrices containing the updated mixture parameters.
    X, a matrix with cluster membership probabilities.
Initialize: Set initial values for C, R, and W (random or approximate solutions)
WHILE: $\delta(llh) > \epsilon$ and $maxiterations$ has not been reached
  DO E and M steps
  E step
    $C' = R' = W' = llh = 0$
    for i = 1 to n
      $sump_i = 0$
      for j = 1 to k
        $\delta_{ij} = (y_i - C_j)^t R^{-1} (y_i - C_j)$
        $p_{ij} = \frac{w_j}{(2\pi)^{p/2}|R|^{1/2}} exp(-\frac{1}{2}\delta_{ij})$
        $sump_i = sump_i + p_{ij}$
      endfor
      $x_i = p_i / sump_i$, $llh = llh + ln(sump_i)$
      $C' = C' + y_i x_i^t$, $W' = W' + x_i$
    endfor
  M step
    for j = 1 to k
      $C_j = C'_j / W_j$
      for i = 1 to n
        $R' = R' + (y_i - C_j) x_{ij} (y_i - C_j)^t$
      endfor
    endfor
    $R = R'/n$, $W = W'/n$

Figure 9: Sequential code for the Expectation Maximization Algorithm

The best speedups are nearly a factor of 50 over the CPU version. However, when the data movement times are included, the speedup decreases to nearly 20. Another observation is that the execution times of middleware versions are almost identical to the hand-coded version, showing that middleware does not introduce any overheads. In fact, the only observable difference is with 1 block and 64 threads, and in this case, the middleware version is actually faster. This is because with a smaller number of threads, all replicated copies of centroids to be updated (the variable update) fit into the shared memory. The middleware detected this feature and benefited from using shared memory. The manual version was designed to execute on all configurations, and because replicated copies of this variable cannot fit in shared memory with larger num-

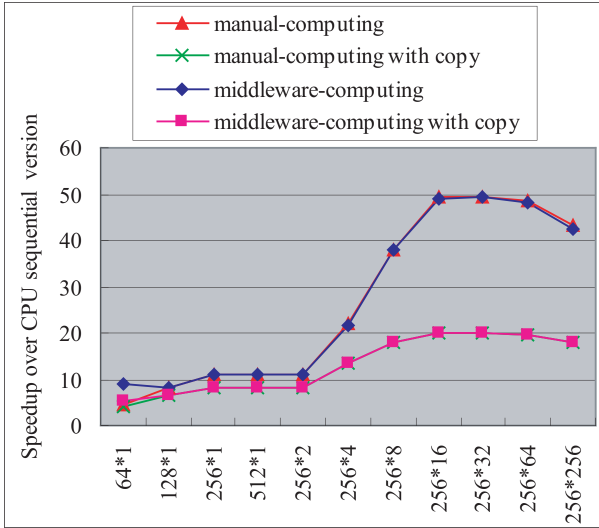

Figure 7: Speedup of k-means on GeForce 8800GTX

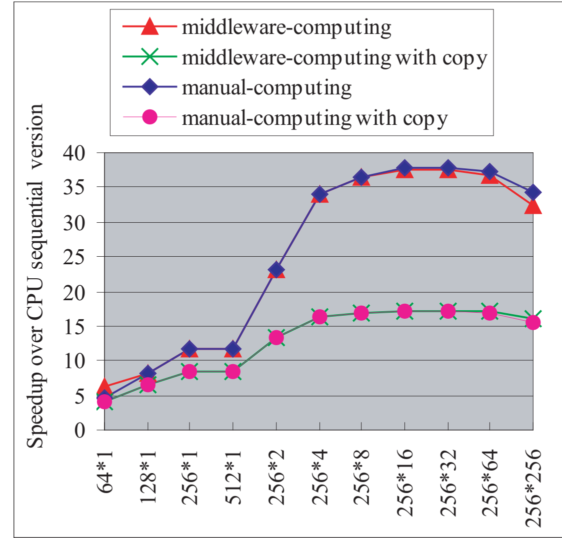

Figure 8: Speedup of k-means on GeForce 9800GX2

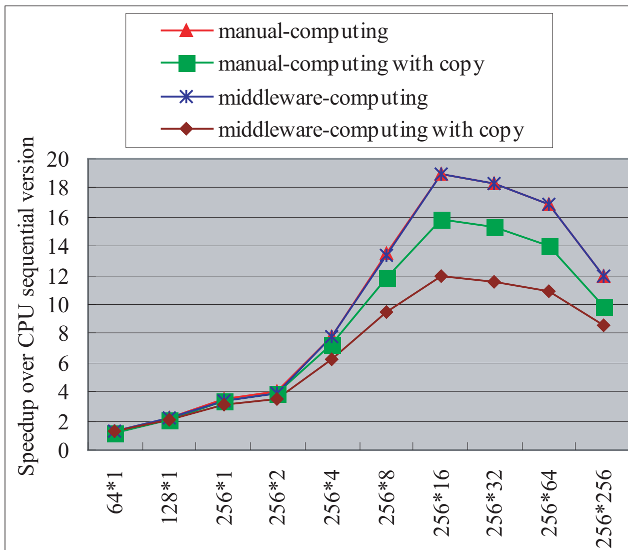

Figure 10: Scalability of EM Application

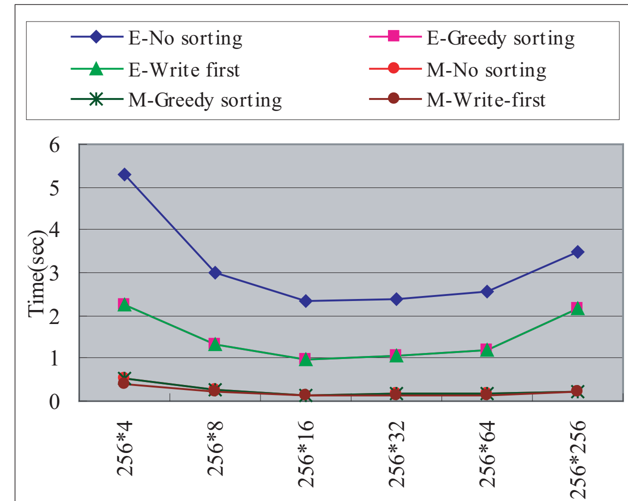

Figure 11: Comparison of E and M Phase computing time among the 3 Shared Memory Layout Strategies

ber of threads, this optimization was not performed. The best performance is obtained with 256 thread per block and 16 or 32 blocks. More threads per block allows more concurrency. The maximum threads we can use in a block is 512, but this configuration does not obtain the best speedup, because of the larger amount of time that is spent on global combination. As there are 16 multiprocessors, best speedups are obtained with 16 or 32 blocks. Using a larger number of blocks only increases contention for resources, and does not allow any more parallelism.

5.2 EM Clustering Algorithm

The second data mining algorithm we have considered is also for clustering. Expectation Maximization (EM) is another popular clustering algorithm. The EM algorithm was first introduced in the seminar paper [16]. EM is a distance-based algorithm that assumes the data set can be modeled as a linear combination of multivariate normal distributions. There are several advantages to using EM for clustering data: it has a strong statistical basis, it is robust to noisy data, it can accept the desired number of clusters as input, it provides a cluster membership probability per point, it can handle high dimensionality and it converges fast given a good initialization [39]. The goal of the EM algorithm is to estimate the means C, the covariances R and the mixture weights W of a Gaussian probability function [39]. The algorithm works by successfully improving the solution found so far. The algorithm stops when the quality of the current solution becomes stable, as measured by a monotonically increasing statistical quantity called *loglikelihood*. The sequential algorithm is shown in Figure 9.

We performed a scalability study, similar to the one we reported earlier for k-means, and the results are shown in Figure 10. We used

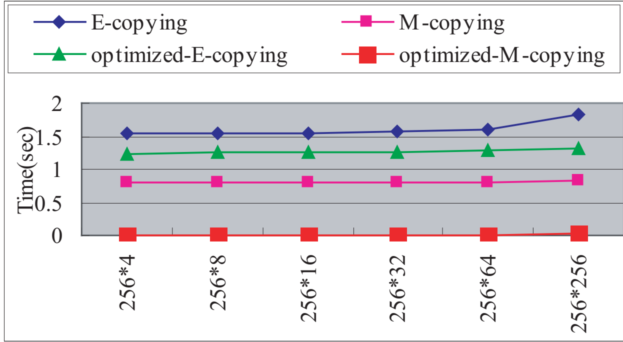

Figure 12: Comparison of E and M Phase memory copy time between Normal and Optimized Memory Allocation Strategies

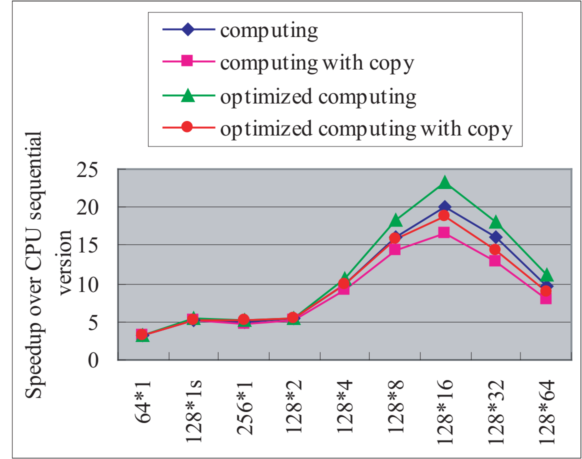

Figure 13: Speedup of PCA: With and Without User-provided Optimized Combination Function

a 12 MB dataset. All execution times are for 11 iterations. The best speedups are nearly 19, though when the data movement costs are included, they reduce to 12. The speedups are lower than what we obtained for k-means, because of a larger number of memory operations, and, relatively, less computation.

Earlier in Section 4.3, we had described several schemes for effectively using the shared memory. The middleware generated version whose performance we have reported is based on the use of scheme that performed the best, which is the `write-first` strategy. This also turns out to be the strategy that the manual version used. Overall, the two versions are almost identical in the compute time, but the manual version is slightly faster in the data copying time.

Next, we focus on examining the impact that different shared memory utilization schemes have on performance. The EM algorithm involves a number of distinct variables that are accessed with different patterns. Thus, unlike k-means and PCA, we notice significant differences from different strategies. The computing time of E and M phases using the 3 strategies are displayed in Figure 11. In the E phase, `no-sorting` is slower than the other two. This is because the other two strategies copied more variables onto shared memory. In the M phase, `no-sorting` again did not do well, but further, `write-first` strategy outperformed the `greedy-sort` strategy. The reason for this seems to be that this particular chip achieves better performance when data to be updated is stored at lower addresses in the shared memory. Overall, the total speedup in the computing time between the version that best uses the shared memory, and a version that does not use shared memory at all, is 40.

We also used the EM application to study the benefits from using optimized copying schemes. The results are shown in Figure 12. We can see that the execution time for both E and M phases is reduced by eliminating unnecessary memory operations. Particularly, the M phase copying costs are reduced to almost zero, as the input data block could be declared as `extern`, and their values can be reused from the values at the end of the E phase reduction function.

5.3 Principal Component Analysis

Principal Components Analysis is a popular dimensionality reduction method. This method was developed by Pearson in 1901. Since it has many steps which are not quite compute-intensive, we only converted the creation of the correlation matrix to CUDA. Though we did not have a manual version for comparison, we did create a version with manually written combination function. This function was more efficient than the version automatically generated by our system.

The speedups on an input matrix of 256K rows, 80 columns are shown in Figure 13. Unlike the previous two applications, the best performance was achieved with 128 threads per block. This, in turn, was because of the increasing overhead of global reductions with a larger number of threads.

The `optimized` versions are the one with user-provided combination function. These versions are faster by nearly 20%, and show a limitation of the current program analysis and code generation performed by our middleware. The best speedups are nearly 24, though including the data movement costs, they reduce to 20. The speedups are higher than what we obtained from EM, but not as high as those from k-means. This is because of a higher fraction of memory accesses, and a relatively larger cost of global combination. This is also the reason that performance decreases rapidly when more than 16 blocks are used.

6. RELATED WORK

Exploring the computing power of GPU has been a topic of much investigation. Before the development of CUDA, Peercy *et al.* [40] proposed a computing model for GPU very similar to CUDA. Data representation and features of operations for GPU computing were also explored by Trancoso *et al.* [46]. *Brook* was a language developed to provide operations for data stream processing on GPUs [9]. Tarditi *et al.* [45] developed techniques to compile C# with an Accelerator, which evaluates the parallel part of the programs on a GPU, and the other parts on the CPU. While their system has many similarities with our work, they do not support complex reductions on parallel collections, which are common in data mining operations. NVIDIA is also making efforts to make CUDA compatible with OpenCL (Open Computing Language), an emerging open and cross-vendor standard for GPU programming [47]. This will be an important issue for future versions of our system.

Analysis and code generation for reduction operations has been studied by a number of distributed memory compilation projects [1,

5, 20, 26, 31, 49] as well as shared memory parallelization projects[7, 22, 23, 35, 36, 41, 48]. More recently, reductions on emerging multi-cores have also been studied [34]. Our work has many similarities, but is specific to the features of GPUs.

At Illinois, CUDA-lite [4] 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 they do not need to write parallel code. However, our system is limited to a specific class of applications. MCUDA [44] is a compiler effort which takes CUDA code as input, and maps it to multi-core machines. Baskaran *et al.* [6] 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 [30]. Some recent work has also made progress in translating OpenMP into CUDA [33]. The reported results are from simple stencil computations, and their is no support for handling complex reductions.

$map\text{-}reduce$ is a widely used parallel computing tool developed by Google, and there is already a CUDA version of $map\text{-}reduce$ called Mars [25]. The $map\text{-}reduce$ API typically results in high overheads for more compute-intensive data mining applications, because of the need for sorting reduction elements. Our system also supports a higher-level (almost sequential) API for these applications.

There have been a large number of application studies with GPUs. We restrict our discussion to only data mining or data-intensive application studies. One of the popular data mining algorithms, k-Nearest Neighbor search, has been studied on GPUs by several groups [10], [42], [18] and [18]. Hall and Hart [21] ported different versions of k-means to GPU using *Cg*. Che *et al.* [12] did an analysis of CUDA computing model and a comparison with other architectures. Particularly, they tested k-means in CUDA, and got a speedup of about 70 over sequential code. This report, published in July 2008, is based on a more advanced GPU (Geforce 260 GTX). Since our system is more general, it is to be expected that we can optimize a single application to the same extent. However, their work does form a basis for additional optimizations we can perform in our system in the future.

7. CONCLUSIONS AND FUTURE WORK

In this paper, we have developed a solution for high-level programming of GPUs. Our solution targets a specific class of applications, which are the data mining and scientific data analysis applications. We exploit the common processing structure, generalized reductions, that fits a large number of popular data mining algorithms. In our approach, the programmers simply need to specify the sequential reduction loop(s) with some additional information about the parameters. Program analysis and code generation is used to map the applications to a GPU. Several additional optimizations (mainly on optimizing memory usage) are also performed to improve the performance.

We have evaluated our system using three popular data mining applications, k-means clustering, EM clustering, and Principal Component Analysis (PCA). The speedup that each of these applications achieve over a sequential CPU version ranges between 20 and 50. The code automatically generated by our system did not have any noticeable overheads compared to hand written codes. Finally, significant performance improvements were obtained with the optimizations we have implemented.

Our work has also indicated additional optimization that can be developed through more advanced compiler analysis techniques. Better code analysis can allow us to optimize the global combination functions without user intervention. Similarly, inter-procedural analysis can enable reduction in memory allocation and copying costs, without requiring specification of *extern* and *common* from the programmer. We can also take into account the variable access frequency for improving shared memory allocation schemes. We will also like to consider *bank conflicts* to further improve the utilization of shared memory.

Acknowledgements

This work was supported by NSF grants 0541058, 0619041, and 0833101. The equipment used for the experiments reported here was purchased under the grant 0403342.

8. 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] R. Agrawal and J. Shafer. Parallel Mining of Association Rules. *IEEE Transactions on Knowledge and Data Engineering*, 8(6):962 – 969, June 1996.

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

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

[5] 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.

[6] 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.

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

[8] Randal E. Bryant. Data-Intensive Supercomputing: The Case for DISC. Technical Report CMU-CS-07-128, School of Computer Science, Carnegie Mellon University, 2007.

[9] I. Buck, T. Foley, D. Horn, J. Sugerman, K. Mike, and H. Pat. Brook for GPUs: Stream Computing on Graphics Hardware, 2004.

[10] Benjamin Bustos, Oliver Deussen, Stefan Hiller, and Daniel Keim. A Graphics Hardware Accelerated Algorithm for Nearest Neighbor Search. In Vassil N. Alexandrov, Geert Dick van Albada, Peter M.A. Sloot, and Jack Dongarra, editors, *Computational Science – ICCS 2006*, volume 3994 of *LNCS*, pages 196–199. Springer, 2006.

[11] Maria Charalambous, Pedro Trancoso, and Alexandros Stamatakis. Initial experiences porting a bioinformatics application to a graphics processor. In *Panhellenic Conference on Informatics*, pages 415–425, 2005.

[12] Shuai Che, Jiayuan Meng, and Jeremy W. Sheaffer. A Performance Study of General Purpose Applications on Graphics Processors.

[13] P. Cheeseman and J. Stutz. Bayesian classification (autoclass): Theory and practice. In *Advanced in Knowledge*

Discovery and Data Mining, pages 61 – 83. AAAI Press / MIT Press, 1996.

[14] Matthias Christen, Olaf Schenk, and Helmar Burkhart. General-Purpose Sparse Matrix Building Blocks using the NVIDIA CUDA Technology Platform. In *First Workshop on General Purpose Processing on Graphics Processing Units*, Oct 2007.

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

[16] Arthur Dempster, Nan Laird, and Donald Rubin. Maximum Likelihood Estimation from Incomplete Data via the EM Algorithm. *Journal of the Royal Statistical Society*, 39(1):1–38, 1977.

[17] 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.

[18] Vincent Garcia, Eric Debreuve, and Michel Barlaud. Fast k Nearest Neighbor Search using GPU, 2008.

[19] Naga Govindaraju, Jim Gray, Ritesh Kumar, and Dinesh Manocha. GPUTeraSort: High Performance Graphics Co-processor Sorting for Large Database Management. In *SIGMOD '06: Proceedings of the 2006 ACM SIGMOD international conference on Management of data*, pages 325–336, New York, NY, USA, 2006. ACM.

[20] 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.

[21] Jesse D. Hall and John C. Hart. GPU Acceleration of Iterative Clustering. Jun 2004.

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

[23] 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.

[24] Jiawei Han and Micheline Kamber. *Data Mining: Concepts and Techniques*. Morgan Kaufmann Publishers, 2000.

[25] 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.

[26] 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.

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

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

[29] 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.

[30] Andreas Klockner. PyCuda, 2008.

[31] 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.

[32] 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.

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

[34] 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.

[35] 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.

[36] 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.

[37] S. K. Murthy. Automatic Construction of Decision Trees from Data: A Multi-disciplinary Survey. *Data Mining and Knowledge Discovery*, 2(4):345–389, 1998.

[38] 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.

[39] C. Ordonez and P. Cereghini. SQLEM: Fast Clustering in SQL Using the EM Algorithm. In *Proceedings of the ACM SIGMOD Conference on Management of Data*, pages 559–570. ACM Press, June 2000.

[40] Mark Peercy, Mark Segal, and Derek Gerstmann. A Performance-oriented Data Parallel Virtual Machine for GPUs. In *SIGGRAPH '06: ACM SIGGRAPH 2006 Sketches*, page 184, New York, NY, USA, 2006. ACM.

[41] 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.

[42] Timothy J. Purcell, Craig Donner, Mike Cammarano, Henrik Wann Jensen, and Pat Hanrahan. Photon Mapping on Programmable Graphics Hardware. In *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS Conference on Graphics Hardware*, pages 41–50. Eurographics Association, 2003.

[43] Erik Sintorn and Ulf Assarsson. Fast Parallel GPU-Sorting Using a Hybrid Algorithm. In *First Workshop on General Purpose Processing on Graphics Processing Units*, Oct 2007.

[44] John Stratton, Sam Stone, and Wen mei Hwu. MCUDA: An Efficient Implementation of CUDA Kernels for Multi-Core CPUs. In *21st Annual Workshop on Languages and Compilers for Parallel Computing (LCPC'2008)*, July 2008.

[45] David Tarditi, Sidd Puri, and Jose Oglesby. Accelerator: Using Data Parallelism to Program GPUs for General-purpose Uses. In *ASPLOS-XII: Proceedings of the 12th international conference on Architectural support for programming languages and operating systems*, pages 325–335, New York, NY, USA, 2006. ACM.

[46] Pedro Trancoso and Maria Charalambous. Exploring Graphics Processor Performance for General Purpose Applications. In *Eighth Euromicro Symposium on Digital Systems Design (DSD 2005)*, pages 306–313, 2005.

[47] Neil Trevett. OpenCL: The Open Standdard for Heterogeneous Parallel Programming, 2008.

[48] 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.

[49] 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.