A New Method for GPU based Irregular Reductions and Its Application to K-Means Clustering

Balaji Dhanasekaran*
Advanced Micro Devices, Inc.
balaji.dhanasekaran@amd.com

Norman Rubin
Advanced Micro Devices, Inc.
norman.rubin@amd.com

ABSTRACT

A frequently used method of clustering is a technique called k-means clustering. The k-means algorithm consists of two steps: A map step, which is simple to execute on a GPU, and a reduce step, which is more problematic. Previous researchers have used a hybrid approach in which the map step is computed on the GPU and the reduce step is performed on the CPU. In this work, we present a new algorithm for irregular reductions and apply it to k-means such that the GPU executes both the map and reduce steps. We provide experimental comparisons using OpenCL. Our results show that our scheme is 3.2 times faster than the hybrid scheme for k = 10, an average 1.5 times faster when the number of clusters, k = 100 and on average equal for k = 400, on an ATI Radeon® HD 5870 (best speedup was 3.5 times) compared to the hybrid approach. In addition, we compare the GPU code with the standard OpenMP benchmark, MineBench. In that implementation, both the map and reduce steps are computed on the CPU. For large data sizes, the new GPU scheme shows great promise, with performance up to 35 times faster than MineBench on a four core Intel i7 CPU.

Categories and Subject Descriptors

I.3.1 [**COMPUTER GRAPHICS**]: Hardware Architecture-Graphics processors, Parallel processing; D.1.3 [**PROGRAMMING TECHNIQUES**]: Concurrent Programming-Parallel programming

General Terms

Algorithm, Performance

Keywords

Clustering, K-means, Data-mining, GPGPU, Heterogeneous Computing

1. INTRODUCTION

A common method used to explore large data sets is clustering analysis. Clustering problems arise in many different applications, including data mining and knowledge discovery [7], data compression and vector quantization [8], pattern recognition and pattern classification [4], and gene clustering [1].

A frequently used method of clustering is a technique called k-means clustering [10]. A k-means clustering problem starts with a set of n data points in a real d-dimensional space, $\Re^d$, where d is the number of attributes. It then determines a set of k points in $\Re^d$, called centroids such that the mean squared distance from each data point to its nearest centroid is minimized (a centroid of a cluster is the average of all the points belonging to that cluster). K-means is widely used because of its simplicity and quick convergence. However,the run-time performance of k-means is a concern with large data sets. The main reason for this concern is that we often can find the correct parameter of k only by performing several runs of K-means with different numbers of clusters and different starting points.

On many existing GPU accelerated implementations of k-means, the map step of k-means is performed on the GPU and the reduce step is computed on the CPU. As a result, the entire output of the map step has to be transferred from the GPU to the CPU through the PCIe bus. This approach may work for smaller data sets. But, when we have a larger data set or when we want to use multiple GPUs instead of a single GPU, this approach of transferring huge amounts of data back and forth between the CPU and the GPU is not scalable. This scalability issue is the main motivation for our work on k-means. We propose a new irregular reduction algorithm in which both the map and reduce steps of k-means can be computed on the GPU with minimal data transfer between the CPU and the GPU.

The contributions of this paper include:

- A new algorithm for floating-point irregular reductions and histograms on the GPU. This approach is fast and easily implementable on modern GPUs.
- Experimental comparisons of implementations of k-means using both the new scheme and a traditional hybrid implementation using a combination of Intel TBB and OpenCL. Our results show a 66% speedup on an ATI Radeon HD 5870 [2] (average speedup is 16%).
- Additional experimental comparisons with CPU implementations of k-means written using TBB [14] and OpenMP. For large data sizes, the GPU version shows great promise, with performance up to 30 times faster than a four-core Intel i7 CPU.

We used MineBench [12] as the baseline for all our performance comparisons. This benchmark is often used for GPU experiments on k-means. This OpenMP, multi-core CPU bench-

*Work was done while the author was an intern at AMD

GPGPU-4, Mar 05-05 2011, Newport Beach, CA, USA.

mark has been used in several other related previous works [3, 17].

The rest of the paper is organized as follows. Section 2 introduces k-means clustering. Sections 3 and 4 provide the background related to the architecture of ATI Radeon HD 5870 and OpenCL. Section 5 deals with the various types of reduction problems and existing solutions. Section 6 describes our proposed irregular reduction algorithm. Section 7 deals with the implementation of k-means clustering using the proposed irregular reduction algorithm, and section 8 discusses the experimental results. Sections 9 and 10 deal with the related work and conclusions.

2. K-MEANS CLUSTERING

Initially, the k-means algorithm randomly chooses k data points as the centroids for each cluster. In each iteration, k-means executes a map step, which associates each data point with its nearest centroid according to a distance metric. The result of the map step is a $membership$ vector indicating the new cluster for each data point. After the map step, k-means executes a reduce step. This step computes new centroids by taking the mean of all the data points in each cluster. K-means repeats these two steps until no data point changes its cluster.

The map step has to compute the distance from every data element (n elements each with d attributes) to the centroid of every cluster (k clusters). Mapping requires $O(knd)$ operations. This step is data-parallel and maps well on to the GPU.

The reduce step first needs to add $O(dn)$ floating-point values, forming $k * d$ sums. Next it has to count the number of elements in each cluster (n adds). Finally, the reduce needs to form new centroids by dividing each sum by the corresponding count.

The summations in the reduce are a special kind of reduction called an irregular reduction or a histogram, in which the data points that need to be added are randomly arranged. These reductions are sometimes difficult to parallelize effectively on a GPU.

One possible map-reduce implementation could use an intermediate sort step to collect all objects with the same new centroid assignment. This approach is examined in [9]. After the data is sorted, the reduction would have a regular layout. While attractive for its simplicity, we believe that the overhead required by a sort makes this approach impractical.

3. THE GPU ARCHITECTURE

We describe an ATI Radeon HD 5870 as an example of a GPU architecture. The HD 5870 contains 20 compute units (CUs), each containing 16 processing elements. The processing elements are five-way VLIW processors, resulting in a total of $20 * 16 * 5 = 1,600$ stream processors. Each compute unit is a single-instruction multiple-data (SIMD) device. In each time step, each processing element in a single compute unit executes the same instruction, but on different data. In the same time step, each compute unit can execute different instructions.

A processing element has access to five different types of memory:

- **Registers** are fast, read/write memory, but are limited in number and are not shared. Each compute unit has 256 sets of vector registers. Each set contains 64 vector registers. Each vector register contains four 32-bit values.
- **Local memory** is a second read/write memory, also limited in size (32 KB). Local memory is shared among the processing elements on a compute unit, but is not shared across compute units.
- **Constant memory** is a larger read-only memory that can be accessed through a special high-speed cache.
- **Image memory** can be either read-only or write-only, and is also accessed through a special high-speed cache.
- **Global memory** is read/write memory shared across all threads. Compared to the other memory types, global memory has a relatively long access latency of 100-600 cycles. Global memory is larger (ATI Radeon HD 5870 has 1 GB) compared to local memory, but may be small compared to the CPU memory (often 4-8 GB).

GPU memory is connected to the CPU via a slow connection called the PCIe bus. Common transfer rates are 10 GB/sec for CPU to main memory, 153 GB/sec for GPU to global memory, and less than 6 GB/sec for PCIe traffic.

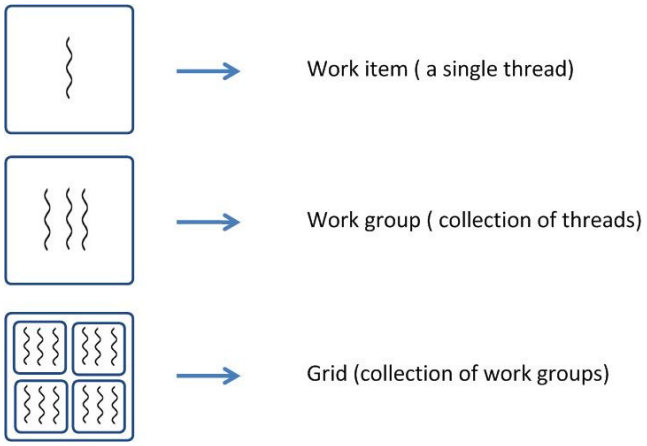

Figure 1: OpenCL thread hierarchy

4. OPENCL

Graphics processors (GPUs) were originally designed strictly to accelerate the graphics pipeline. Recognizing the potential for huge performance gains from GPUs, many efforts have been made to use them to perform general-purpose computing by mapping applications onto graphics APIs. This has been known as the General Purpose GPU (GPGPU) approach. However, expressing a general problem in existing graphics APIs proved to be very difficult. One of the most important advances in GPU computing has been the development of high-level C-like languages like CUDA and OpenCL for data-parallel programming on the GPU.

CUDA is a proprietary programming language that has allowed many researchers to use GPUs to accelerate their applications, and many of resultant papers have reported respectable speedup compared to CPU-only implementations. Recently, a new industry standard-language called OpenCL has appeared. Since OpenCL is supported by multiple vendors, it allows researchers to accelerate applications on a wide range of hardware platforms. To a developer, OpenCL is an extended ANSI C programming language and an associated library. Programs consist of a host section, which runs on the CPU, and a kernel section, which runs on the GPU. The host code takes care of transferring data to and from the GPU, and also takes care of initiating the kernel code. In OpenCL, the GPU is regarded

as a co-processor capable of executing a large number of work items in parallel. A single source program includes the host code that runs on the CPU and the kernel code that runs on the GPU.

Threads executing in OpenCL are organized into a three-level hierarchy. At the bottom, a single parallel execution of a kernel (a thread) is called a work item. Each work item executes on a processing element. The work item is assigned to the same compute unit for the duration of its execution. The middle level of the hierarchy is the work group. Logically, work items execute in work groups. The work items within a group can communicate and synchronize. An entire work group executes on a single compute unit. On the ATI Radeon HD 5870, work groups can contain up to 256 work items. At the top level, multiple work groups execute in a grid or index space of computation. Work items are scheduled by hardware onto the CU's in groups, called wavefronts. Each wavefront contains at most 64 work items. Figure 1 illustrates the thread hierarchy in OpenCL.

Listing 1: Pseudo-code for regular reduction kernel

```
/*Let values[] be the shared array that needs to
  be reduced
Let reduce_array_size be the number of elements
  that need to be reduced
Let work_item_count be the number of work items
Let work_item_id be the ID of the current work
  item*/

for(int i = work_item_count/2; i>0; i = i/2)
{
 if(work_item_id < (reduce_array_size/2))
 {
        reduce_array_size = ceil(reduce_array_size
          /2);
        val[work_item_id] += val[work_item_id +
          reduce_array_size];
 }
 barrier(CLK_LOCAL_MEM_FENCE);

}
```

5. REDUCTIONS

Given a set of points, reduction combines them into a single point or a smaller set of points. We have assumed that the results of the reductions are independent of the order of floating-point operations. Reductions can be classified into two types: regular reductions and irregular reductions.

5.1 Regular Reductions

A regular reduction is a reduction in which all the data items that need to be reduced are located consecutive to each other. As a result, regular reductions can be easily computed on a GPU. Regular reductions can be implemented on the GPU by assigning one work item per value. Work item i can combine elements at i and $i+s$ where s is a power of two, s=1,2,4,8 etc. and the reduction can easily be done in $\log_2(n)$ time. Listing 1 shows the pseudo-code for a regular reduction kernel on the GPU. For this pseudo code we assume that the size of the work-group is a power of two. Figure 2 depicts a step-wise regular reduction.

In the first iteration in Figure 2, work item 0 combines *values*$[0]$ and *values*$[0+4]$ (i.e *values*$[0+2^2]$), work item 1 combines *values*$[1]$ with *values*$[1+4]$, and so on. In the second iteration, *values*$[0]$ is combined with *values*$[0+2]$, and *values*$[1]$ is combined with *values*$[1+2]$. In the third iteration, *values*$[0]$ is combined with *values*$[1]$, and we have the final result.

5.2 Irregular Reductions

In the case of irregular reductions, the elements that need to be reduced are not located in consecutive order; instead, they are arranged in random order. Irregular reduction (sometimes called histogram) has been difficult and inefficient on the GPU. This often means that GPU-based implementations of algorithms that require histogram calculation transfer large blocks of data between the GPU and the CPU. This can be a significant bottleneck. We therefore sought an efficient way to calculate the reduction completely on the GPU.

6. GPU IRREGULAR REDUCTION ALGORITHM

One way of performing the irregular reduction on the GPU would be to assign one work item per value, and then sort the work items based on their cluster IDs and compute the $log_2(n)$ reduction. However, $log_2(n)$ reduction does not require a complete sort. The only requirement is that all the work items belonging to the same cluster must be consecutive to each other. Within a cluster, the work items can be arranged in any order, and the clusters themselves can also be arranged in any order. So, instead of performing a complete sort, our proposed algorithm imposes a partial order on the work items such that all the work items belonging to the same cluster have consecutive IDs.

To use a $log_2(n)$ reduction, our method assigns each work item a new ID, *new_workitem_ID*, and establishes an index array, *index*, such that a work item with a new id, *i*, can combine elements at indexes, *index*$[i]$ and *index*$[i+s]$ where *s* is a power of 2 ($s = 1, 2, 4$, etc). The index array contains a list of work items ordered such that all work items assigned to the same cluster are consecutive to each other.

The first step in our algorithm is to initialize the index array to zero. Then, each work item atomically increments *index*$[c]$ by 1, where *c* is the corresponding cluster ID of that work item. The atomic add returns the previous value of *index*$[c]$ to the work item. Thus, by doing an atomic add of 1 to *index*$[c]$, each work item allocates a unique position for itself within its cluster. This position is called *id_within_cluster*. After all work items have performed this step, *index*$[c]$ contains the total number of work items belonging to cluster *c*. Thus, all the work items have received a unique id within their cluster, *id_within_cluster*, and they also know the number of work items belonging to their cluster, *reduce_array_size*.

To assign global unique IDs to each of these work items, we need to find an offset for each of the clusters based on the number of work items belonging to that cluster. We use a global variable *global_offset*, for this purpose.

Work item 0 initializes *global_offset* to zero. Each work item whose *id_within_cluster* is 0 atomically adds its *reduce_array_size* to global_offset, thus allocating a unique offset for its clusters. All of the work items have their ID within cluster and their cluster offset and, hence, their unique global ID, $new\_workitem\_ID$, can be computed by adding their $id\_within\_cluster$ with the $offset$. Thus, we have assigned new IDs to each of the work items such that work items belonging to the same cluster have consecutive IDs.

Now, we update the index array, $index$ with old work item

Listing 2: Pseudo-code for irregular reduction kernel

```
//Let index[] be the array used to compute unique
  consecutive IDs for work items

//All elements in array index are initialized to
  zero

//Let c be the cluster the work item belongs to

//Let id_within_cluster be the ID of the work item
   within its cluster

id_within_cluster = atomic add(1, index[c])

/*After all the work items execute the previous
  statement, index[c] will contain the total
  number of work items belonging to the cluster c
  which is the number of elements to be reduced in
   that cluster*/

reduce_array_size = index[c]

//Let global_offset be the shared variable used to
   find the offset of the clusters
//global_offset is initialized to zero.

if(work_item_id == 0) global_offset = 0

if(id_within_cluster == 0)
{
 index[c] = atomic add(reduce_array_size,
   global_offset)
}

//Let offset be the offset of cluster c

offset = index[c]

//Let new_workitem_ID be the new ID

new_workitem_ID = offset + id_within_cluster

//Let us update index[] with work_item_ids such
  that consecutive work_items in the index array
  belong to the same cluster

index[new_workitem_ID] = work_item_id

//actual reduction

for(int i = work_item_count/2; i >0; i = i/2)
{
 if(id_within_cluster < (reduce_array_size/2))
 {
        reduce_array_size = ceil(reduce_array_size
          /2);
        values[index[new_workitem_ID]]+=
        values[index[new_workitem_ID+
          reduce_array_size]];
 }
 barrier(CLK_LOCAL_MEM_FENCE);
}
```

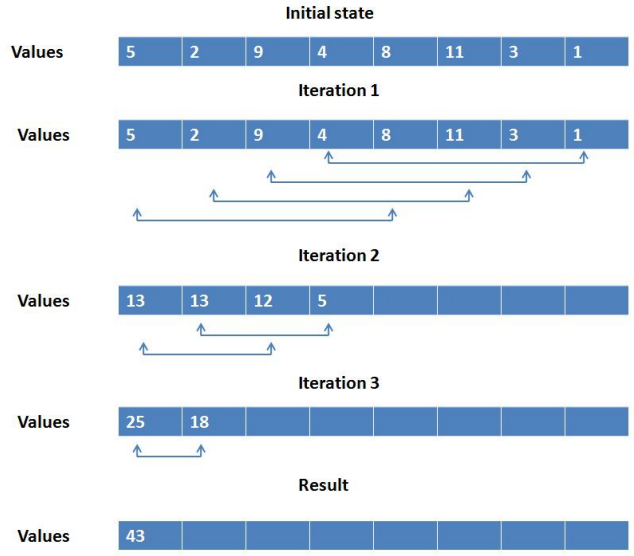

Figure 2: Stepwise illustration of regular reduction

ID such that:

$index[new\_workitem\_ID] = work\_item\_ID$.

The next step is to perform the irregular reduction, similar to the regular reduction, with one slight modification. Instead of accessing the *values*[] array directly, we index it using the *index*[] array so the correct elements get reduced. Listing 2 shows the pseudo-code for irregular reduce, and Figures 3 and 4 depict a step-wise irregular reduction.

In Figures 3 and 4, we considered an example with a work group of size 8 (the actual work groups used in this study was 256 work items). In Figure 3, array *values*[] represent the values that need to be reduced, and cluster c represents the corresponding cluster IDs. Let's focus on work items $0, 1$, and 3. The first step is to determine the *id_within_cluster*. Each work item atomically adds 1 to *index*$[c]$. Work item 0 receives an *id_within_cluster* of 0 (since there is only one work item belonging to cluster 0). Work items 1 and 3 receive *id_within_cluster* of 0 and 1. The next step is to find the *offset*. Work item 0 adds the number of work items in cluster 0 (which is 1) to *global_offset*. Work item 0 receives an *offset* of 0. So, the new ID of work item 0,*new_workitem_ID* is 0 (the sum of *id_within_cluster* and *offset*). Work item 1 adds the number of work items in cluster 2 (which is 2) to *global_offset*. It receives an offset of 1 (since work item 0 added a 1 previously). Thus, the new IDs of work items 1 and 3 are 1 and 2. Thus, we have assigned new IDs such that work items belonging to the same cluster have consecutive IDs.

After assigning new IDs, we update the index array, *index* with the old IDs. So, *index*$[0]$ is updated with 0, *index*$[1]$ with 1, and *index*$[2]$ with 3 (because work item 3 has been assigned a new ID of 2). Using the index array, we perform the irregular reduction. Work item 1 adds the value at its location with the value at the index, *index*$[1 + 2^0]$ (i.e., *index*$[2]$, which is nothing but work item 3). Thus, though work items 1 and 3 are not consecutive to each other, we have reduced them using the index array. All the other reductions in Figure 4 follow the same pattern.

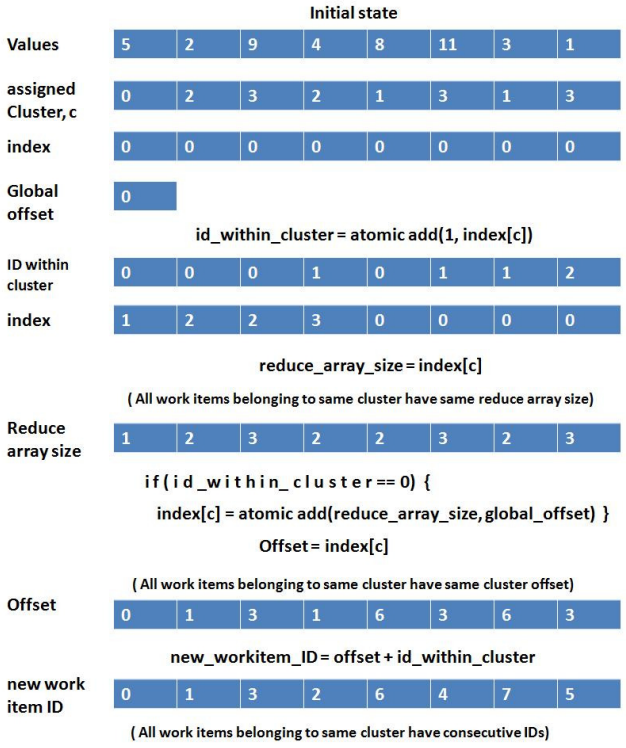

Figure 3: Step-wise illustration of the irregular reduction algorithm. This figure illustrates the assignment of new IDs to the work items such that all the work items belonging to the same cluster have consecutive IDs.

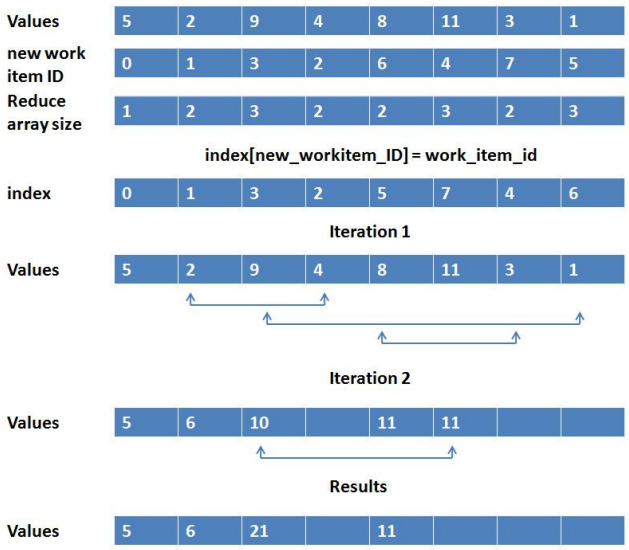

Figure 4: Step-wise illustration of the irregular reduction algorithm (Continued). This figure illustrates step-wise irregular reduction after new IDs have been assigned to the work items.

The proposed algorithm uses atomic operations in two statements, one while finding the *id_within_cluster* and the other while finding the *offset*. If a collision occurs in any of these two statements, the GPU hardware will take care of resolving the collisions. The order in which the collisions would be resolved is non-deterministic. The problem with these collisions is that they serialize the execution. But, interestingly, the cost of collisions in our algorithm is relatively low. If every work item is assigned to a different cluster, then there are no collisions in first statement and $|wavefront|$ collisions at the second one. On the other hand, if every item is assigned to the same cluster there are $|wavefront|$ collisions at the first statement and none at the second one. So, the worst case for the number of collisions is $(|wavefront| + 1)$ per reduction. If there are p clusters in a wavefront, the number of collisions is given by the following equation:

$$NumberOfCollisions = WavefrontSize/p + p \qquad (1)$$

Each such collision results in a serialized execution.

7. IMPLEMENTATION

We have implemented the k-means kernels using OpenCL. One of the most powerful features of OpenCL is dynamic compilation. Users of OpenCL are provided with the capability to compile the kernels at run time. We have used this feature to specialize the k-means kernels for the current instance of the problem. We have performed loop unrolling based on the number of clusters and attributes, and have used dynamic compilation to perform loop unrolling specialized to each instance of the problem.

The first step in our implementation is to copy the data from the CPU to the GPU. In the case of the CPU, a cache-friendly data layout is row-based (i.e., all the attributes for a given data point are kept together). However, on the GPU, a memory-friendly arrangement would be column-based, (i.e., all values of a given attribute are stored next to each other). Our implementation transfers data from the CPU to the GPU and then uses a kernel to transpose the data. The data set is transposed once and used for all iterations.

We placed the current centroid data into the GPU constant memory. Farivar [6] reported an order of magnitude speedup using the hybrid approach when he moved the centroid data from global memory to constant memory.

The next step in our implementation is to find the cluster centers and repeat the process. In a real system, we would iterate until data points do not change their clusters. However, in order to compare measurements, we forced the number of iterations to 50. We validated the code by making sure that all the versions of k-means produced the same answers. A given iteration consists of:

1. Initialize the part of global memory that stores the partial sums.
2. (a) Perform the map. We used kernels with work groups of size 256.
   (b) Perform a partial reduce. We have used $n/256$ work groups, where n is the number of points. Each of these work groups generate $(d+1)*k$ partial sums, where d is the number of attributes and k is the number of clusters. The additional attribute is used to store the number of points belonging to that cluster in the partial sum. This kernel also generates the

number of points that have changed their cluster during the current iteration in this partial sum.

3. A third kernel adds all the partial sums. In this kernel, we have used work groups of size 256. We have used one work group per cluster. Output of this kernel is the new cluster centroids and the sum of the numbers of data points that have changed their clusters.

4. At the end of the iteration, depending on the number of points that have changed their clusters, the CPU decides on whether to continue with the next iteration.

Since the kernel in step 2 is writing at most $k*(a+1)$ values, we preallocate all the space required for all work-groups. For the hybrid scheme we remove steps 2b and 3, and modified step 2a to write a membership vector, one integer per element back to the CPU.

8. EXPERIMENTAL RESULTS

Experiments were performed on a four-core Intel i7 CPU system with two graphics cards running 32-bit Microsoft® Windows®Vista. The system consists of a four-core Intel Core i7 920 CPU with a clock speed of 2.67 GHz and 8 GB of RAM. We used an ATI Radeon HD 5870 Graphics card. The software versions used were ATI-Stream-v2.2 (302) and ATI Catalyst ™10.7.

We used randomly generated data sets for running our experiments. The actual values were uniformly distributed and generated by the Mersenne Twister method [11].

Since we had fixed the seed of our random number generator, we were able to repeat the experiments. The number of k-means iterations is forced to 50 for all the experiments. The timing reported is the total wall clock time for all iterations, including the time for calculation and communication between the CPU and the GPU and excluding the time taken for initializing the data sets.

8.1 MineBench performance

We begin with some preliminary measurements. First, figure 5 shows that the execution time is linear in the data size (the product of the number of attributes and the number of objects). The x-axis is the data set size, from 1 to 32 million floating-point words of data, the y-axis is the total execution time in seconds. Each line corresponds to a different number of attributes. Each panel provides data for a different value of k. To make the linear slope clear we used different y-axis scales.

After some experimentation, we determined that the performance of both MineBench and TBB is quite sensitive to specific Microsoft Visual Studio compiler optimization settings.

The best setting we found was:

Option	Meaning
/0x	maximum optimization
/Ob2	maximum inline
/Oi	Generate intrinsic functions
/Ot	favor fast code
/GL	Whole Program optimization
/Oy	no frame pointers
/EHsc	assume no extern c function will throw an exception
/MT	multi-threaded executable

The numbers presented are based on the above settings.

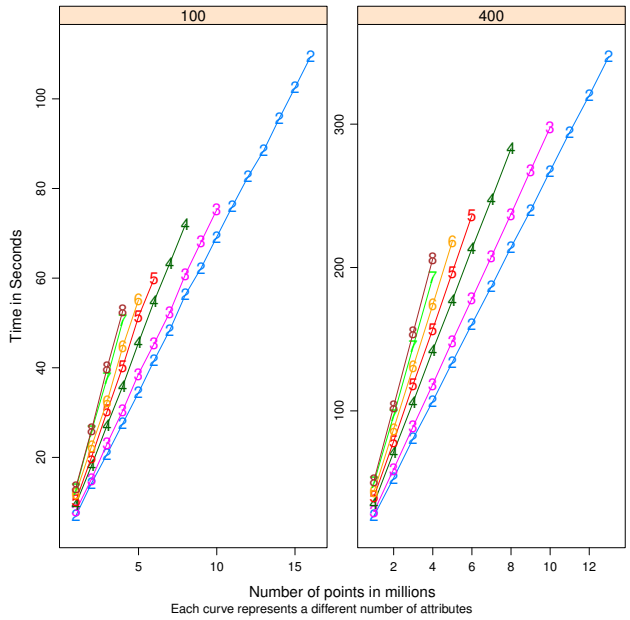

Figure 5: Linear scaling of MineBench performance. X-axis represents the number of points and Y-axis represents the time taken in seconds. The curves represent different number of attributes and the two sub graphs represent clusters of size 100 and 400.

n	d	k	MineBench	TBB
1000000	4	400	35.84	35.99
1000000	6	400	43.92	43.92
4000000	2	100	26.96	29.31
1000000	8	400	51.86	52.08
4000000	8	200	102.83	106.40

Figure 6: Comparisons of our TBB implementation of k-means with MineBench's k-means. The MineBench and TBB values are the time taken in seconds.

8.2 Comparison of our TBB implementation of k-means with The OpenMP MineBench's k-means

To compare the performance of the GPU only and hybrid k-means implementations, we needed to ensure the TBB code used as the CPU side of the hybrid k-means is implemented efficiently.

Figure 6 shows the performance in seconds for MineBench k-means compared to our TBB version. The TBB numbers appear to be almost the same as hand-tuned MineBench code. We did find that, for small values of k, TBB appears to be faster than MineBench since it has cache-aware memory allocators. Figure 6 shows that our CPU implementation of k-means is as efficient as our baseline, MineBench's k-means. Since our work is focused on the GPU reduction, we did not explore the TBB/MineBench code in detail.

8.3 GPU k-means compared to hybrid k-means

We have implemented both a hybrid k-means and a GPU-

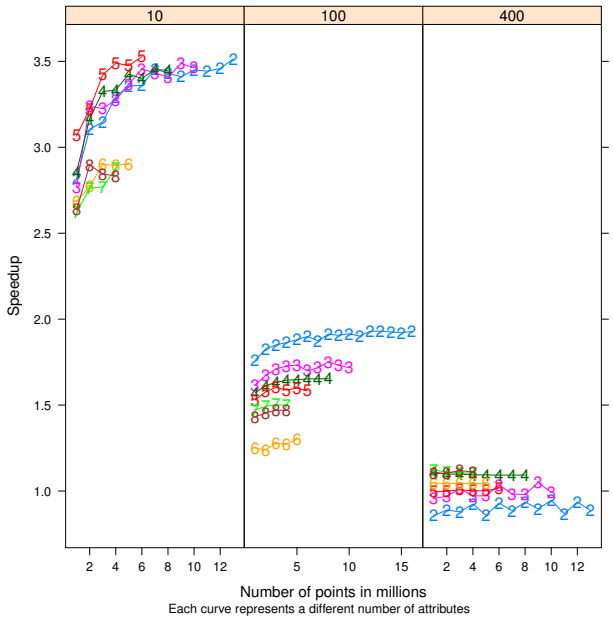

Figure 7: Speedup of GPU-only k-means compared to hybrid k-means.

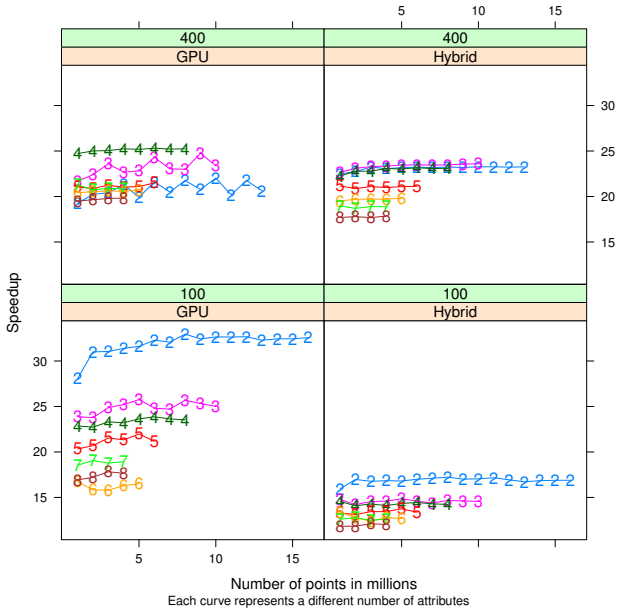

Figure 8: Speedup of GPU and hybrid k-means compared to CPU k-means.

only k-means. In the hybrid k-means, the map is computed on the GPU and the reduce is done on the CPU. In the GPU-only implementation, both the map and reduce steps are computed entirely on the GPU. The CPU's only task is controlling the number of iterations.

In the hybrid k-means, there is significant data transfer between the CPU main memory and the GPU memory. After the map step, the entire membership vector (4 bytes per data point) is transferred from the GPU to the CPU, and after the reduce step, the new clusters ($(4 * k * d)$ bytes) are transferred from the CPU to the GPU. In the case of GPU-only k-means, the only data that needs to be transferred from the GPU to the CPU is a single 4-byte value representing the number of points that have changed their clusters in the current iteration.

Figure 7 shows the speedup of the GPU-only k-means compared to the hybrid k-means.

The range of values in the figure are:

number of clusters	minimum	mean	maximum
10	2.6	3.2	3.5
100	1.2	1.7	1.9
400	.9	1.0	1.1

As the number of clusters increases, the performance of the algorithm declines. There are two reasons for this. First, as k approaches the work group size, fewer and fewer elements are added within the work group. Most work items will simply execute a series of tests and then decide not to execute any useful work. Second, with larger values of k, almost all work items will atomically update the global count, causing collisions.

In this figure, we ordered the data by number of integer words transferred from GPU to CPU per iteration. Since our algorithm does not have the overhead of data transfer, performance should be better towards the right.

8.4 GPU only k-means compared with MineBench k-means

Figure 8 shows the speedups of GPU-only and hybrid k-means compared to MineBench on a four-core Intel i7 CPU. From the figure we see the algorithm is more than 35 times faster than the four-core CPU for k=100.

9. RELATED WORK

There have been several publications that explored the use of GPUs for clustering using the k-means method. Among them, three pieces of work are closely related to ours [[3], [17], [5]].

CHE et al. [3] conducted a series of research experiments using GPUs to accelerate various general-purpose applications including k-means. They focused on one of the most important problems in implementing k-means, which is determining the efficient layout of k-means data in the GPU memory hierarchy. They chose to place the data points in the GPU's image memory. They used constant memory to store the k centroids. Each work item is responsible for finding the nearest centroids of one data point. Each work group has 256 items. Each work item calculates the distance between one corresponding data point and every centroid, then finds the minimum distance and associates the data point with the nearest centroid. Next, each work group calculates a partial sum based on the data points in the work group. A single work item is used to calculate each dimension of the partial sum. Finally, the partial sums are copied back to the CPU, which then serially calculates the new centroid by adding the partial centroid sets. This work achieved quite good speedups. An Nvidia GTX 260 showed 72x speedup compared to a single CPU core and 35 times speedup compared to MineBench on a dual-core CPU.

A team at HKUST and Microsoft Research Asia [5] has also looked into parallel data mining on GPUs. The focus of that research was an elegant bit-field algorithm that tracks the elements assigned to each cluster. In this research, each work

group contains 128 work items. Each work item calculates the distance from one data point to every centroid, and then updates a bit in a bit array that stores the nearest centroid for each data point. In a second pass, each work item is responsible for one centroid; it finds all the corresponding data points from the bit array and takes the mean of those data points as a new centroid.

Finally, a third team at HP labs [17] focused on computing k-means for very large data sets. In particular, they looked at ways to stream data so they could process data sets that would not fit into GPU memory. They reported speedups of around 100x compared to a single-core CPU and about 15x compared to an eight-core CPU. The HP team also used a hybrid approach. That is, they execute the map on the GPU, transfer the resultant membership vector back to the CPU, and compute the new centroids on the CPU.

In all the three research efforts, the map part of the computation proved easy to parallelize on the GPU compared to the reduce part. In our work, we concentrated on the reduce problem and found a new promising algorithm to handle irregular reductions on the GPU.

A complementary technique for hybrid k-means was proposed by [13]. Rather than assign all map operations to the GPU and all reduce operations to the CPU, this work splits the input data set into sections. They then assign each section to a processor, either a GPU or a CPU. Each processor computes both the map-reduce on its assigned section, with the CPU combining final results.

For previous generation GPU's, which do not have atomic operations, computing histograms has proved difficult. Several approaches [15] considered using the traditional graphics pipeline, while [16] describes a way to simulate mutex locks by tagging memory locations.

10. CONCLUSIONS AND FUTURE WORK

The performance results for OpenCL are likely to improve in the future. For this study, we used early releases of OpenCL tools (AMD 2.1). We expect later releases will show improved performances.

We have developed a general GPU-based irregular reduction algorithm. It may be possible to use the same technique for many other histogram-related problems. We have developed a very high speed-implementation of k-means. Given that the CPU is doing very little when both the map and reduce executes on the GPU, one possible future research direction would be to use the CPU to process part of the computation. Another possible area of future work would be generic map reduce. In the case of k-means, the number of keys (i.e., the number of clusters, k) is fixed and predefined, but this is not the case with generic map reduce. It would be interesting to adapt our irregular reduce algorithm for generic map reduce. Another potential area of research is implementing k-means on multiple GPUs. Since we have eliminated most of the memory copy overhead, we should get almost linear performance improvement as we increase the number of GPUs.

11. ACKNOWLEDGMENTS

We would like to thank Prof. Kim Hazelwood for her valuable feedback.

12. REFERENCES

[1] A. Bagirov and K. Mardaneh. Modified global k-means algorithm for clustering in gene expression data sets. In *Proceedings of the 2006 Workshop on Intelligent Systems for Bioinformatics (WISB 2006), Volume 73*, page 28, Hobart, Australia.

[2] *ATI* Radeon HD 5870 GPU feature summary, `http://www.amd.com/us/products/desktop/graphics/ati-radeon-hd-5000/Pages/ati-radeon-hd-5000.aspx`, 2009.

[3] S. Che, M. Boyer, J. Meng, D. Tarjan, J. Sheaffer, and K. Skadron. A Performance Study of General-Purpose Applications on Graphics Processors using CUDA. In *Journal of Parallel and Distributed Computing*, Volume 68, pages 1370–1380, 2008.

[4] R. O. Duda, P. E. Hart, and D. G. Stork. *Pattern Classification and Scene Analysis, Second ed.* Wiley and Sons, New York, 2000.

[5] W. Fang, K. K. Lau, M. Lu, X. Xiao, C. K. Lam, P. Y. Yang, B. He, Q. Luo, P. V. Sander, and K. Yang. Parallel Data Mining on Graphics Processors. Technical Report HKUST-CS08-07, The Hong Kong University of Science and Technology, Oct, 2008.

[6] R. Farivar, D. Rebolledo, E. Chan, and R. Campbell. A Parallel Implementation of K-Means Clustering on GPUs. In *WorldComp 2008*, Las Vegas, Nevada, July 2008.

[7] U. Fayyad, G. Piatetsky-Shapiro, P. Smyth, and R. Uthurusamy. *Advances in Knowledge Discovery and Data Mining*. MIT press, 1996.

[8] A. Gersho and R. Gray. *Vector Quantization and Signal Compression*. Kluwer Academic Pub, 1992.

[9] B. Hong-tao, H. Li-li, O. Dan-tong, L. Zhan-shan, and L. He. K-means on Commodity GPUs with CUDA. In *Computer Science and Information Engineering* World Congress, Los Alamitos, CA, USA, 2009, vol. 3, IEEE Computer Society,page 655.

[10] S. Lloyd. Least Squares Quantization in PCM. *IEEE Transactions on Information Theory*, 28(2):129–137, 1982.

[11] M. Matsumoto and T. Nishimura. Mersenne Twister: A 623-Dimensionally Equidistributed uniform Pseudo-Random Number Generator. *ACM Transactions on Modeling and Computer Simulation (TOMACS)*, Volume 8, Pages 3–30, 1998.

[12] J. Pisharath, Y. Liu, W. keng Liao, G. Memik, A. Choudhary, and P. Dubey. Nu-MineBench: 2.0 Tech Report cucis-2005-08-02. Center for Ultrascale Computing and Information Security. Northwestern University, 2005.

[13] V. T. Ravi, W. Ma, D. Chiu, and G. Agrawal. Compiler and Runtime Support for Enabling Generalized Reduction Computations on Heterogeneous Parallel Configurations. In *ICS '10: Proceedings of the 24th ACM International Conference on Supercomputing*,June 2, 2010, Tsukuba, Ibaraki, Japan, pages 137–146.

[14] J. Reinders. *Intel Threading Building Blocks*. O'Reilly, 2007.

[15] T. Scheuermann and J. Hensley. Efficient Histogram Generation Using Scattering on GPUs. In *Proceedings of the 2007 Symposium on Interactive 3D Graphics and Games*, page 37. ACM, 2007.

[16] R. Shams and R. Kennedy. Efficient Histogram Algorithms for NVIDIA CUDA Compatible Devices. In *International Conference on Signal Processing and Communication Systems, Gold Coast, Australia, Dec. 2007, page 418*.

[17] R. Wu, B. Zhang, and M. Hsu. Clustering billions of data points using gpus. In *UCHPC-MAW '09: Proceedings of the Workshop on UnConventional High Performance Computing/Memory Access*, pages 1–6, New York, NY, USA, 2009. ACM.