Clustering Billions of Data Points Using GPUs

Ren Wu
HP Labs
1501 Page Mill Road
Palo Alto, CA 94304
ren.wu@hp.com

Bin Zhang
HP Labs
1501 Page Mill Road
Palo Alto, CA 94304
bin.zhang2@hp.com

Meichun Hsu
HP Labs
1501 Page Mill Road
Palo Alto, CA 94304
meichun.hsu@hp.com

ABSTRACT

In this paper, we report our research on using GPUs to accelerate clustering of very large data sets, which are common in today's real world applications. While many published works have shown that GPUs can be used to accelerate various general purpose applications with respectable performance gains, few attempts have been made to tackle very large problems. Our goal here is to investigate if GPUs can be useful accelerators even with very large data sets that cannot fit into GPU's onboard memory.

Using a popular clustering algorithm, K-Means, as an example, our results have been very positive. On a data set with a billion data points, our GPU-accelerated implementation achieved an order of magnitude performance gain over a highly optimized CPU-only version running on 8 cores, and more than two orders of magnitude gain over a popular benchmark, MineBench, running on a single core.

Categories and Subject Descriptors

I.3.1 [**COMPUTER GRAPHICS**]: Hardware Architecture-Graphics processors, Parallel processing; H.3.3 [**INFORMATION STORAGE AND RETRIEVAL**]: Information Search and Retrieval-Clustering; D.1.3 [**PROGRAMMING TECHNIQUES**]: Concurrent Programming-Parallel programming;

General Terms

Algorithms, Performance, Design, Experimentation

Keywords

Parallel Algorithm, Data-mining, Clustering, Graphics Processor, GPGPU, Accelerator, Multi-core, Many-core, Data parallelism

1. INTRODUCTION

Graphics processors (GPUs) have developed very rapidly in recent years. GPUs have moved beyond their originally-targeted graphics applications and increasingly become a viable choice for general purpose computing. In fact, with many light-weight data-parallel cores, GPUs can often provide substantial computational power to accelerate general purpose applications at a much lower capital equipment cost and much higher energy efficiency, which means much lower operating cost while contributing to a greener economy.

In this paper, we report our work on using GPUs as accelerators for processing very large data sets. Using the well-known clustering algorithm K-Means as an example, our results have been very positive. A very large data set with one billion data points can be clustered at a speed more than 10 times faster than our own highly optimized CPU version running on an 8-core workstation.

2. RELATED WORK

2.1 GPU Computing

Graphics processors (GPUs) are originally designed for a very specific domain - to accelerate graphics pipeline. Recognizing the huge potential performance gains from these GPUs, many efforts have been made to use them to perform general purpose computing, by mapping general purpose applications onto graphics APIs. This has been known as the General Purpose GPU (GPGPU) approach.

However, to express a general problem in existing graphics APIs proved to be cumbersome and counter-intuitive. A few attempts have been made to create new languages or APIs that offer a general purpose interface [7].

One of the most important advances in GPU computing is the Nvidia's CUDA solution, which provides both software support - the CUDA programming language extended from the popular C language, and the CUDA-enabled hardware compute engine - a highly parallel architecture with hundreds of cores and very high memory bandwidth. With the large install base of CUDA-enabled devices and the C-like programming environment, many researchers are now able to use GPUs to accelerate their applications, and many have shown respectable speedup performance compared to CPU-only implementations. Popular commercial applications, such as Adobe's creative suite, Mathematica, etc., have new releases of their CUDA-enabled version; researchers have used CUDA in cloud dynamics, N-Body and molecular dynamics simulations, and database operations, with results that show good promise [4].

2.2 K-Means algorithm

K-Means is a well-known clustering algorithm widely used in both academic research and industrial practices. It shares the properties of a much wider class of statistical algorithms.

Given the number of clusters k, K-Means iteratively finds the k-centres of the data clusters. Each iteration consists of two steps:

- Step 1. Partition the data set into k subsets by assigning each point to the subset whose center is the closest center to the point.
- Step 2. Recalculate the k cluster centers as the geometric centers of the subsets.

The algorithm repeats these two steps until no data point moves from one cluster to another. It has been shown that K-Means converges to a local optimum and stops in a finite number of iterations.

There is still active research on the K-Means algorithm itself [1, 12]. In this paper, we are interested in GPU acceleration rather than the K-Means algorithm itself.

2.3 Experiment design and hardware

Since MineBench [9] has been used in a few related previous works [2, 3], we have used it as our baseline for performance comparison so as to align with previously published results. We also implemented our own CPU version of K-Means algorithm for more accurate comparisons (details in the next section). Two machines used in our experiments are both HP XW8400 workstations with dual quad core Intel Xeon 5345 running at 2.33GHz, each equipped with an Nvidia GeForce GTX 280, with 1GB onboard device memory; one machine has 4GB of memory running Windows XP, and the other has 20GB of memory running Windows XP x64. The GPU code was developed in CUDA on top of Microsoft Visual C++ 2005.

Since our primary interest is on the performance acceleration ratios by GPUs, not the algorithm itself, we used randomly generated data sets. The maximum number of iterations is limited to 50 for all experiments because for our purpose of speedup comparison, it is sufficient to obtain the per-iteration cost. The timing reported is the total wall time for all iterations, including both the time for calculation and communication between the CPU and the GPU, but without the time for initializing the data sets. Both CPU- and GPU-versions give identical new centers under identical initializations of the centers. This confirms the algorithmic correctness of our GPU implementation.

2.4 CPU-only implementation

MineBench is a popular high-performance multi-threaded data-mining package, which includes K-Means as one of its benchmark algorithms. It has been used in a few previous works as the baseline reference.

Table 1. Performance comparison between MineBench and optimized implementation

dataset				MineBench, time (s)			Optimized, time (s)			speed ups (x)		
N	D	K	M	1c	4c	8c	1c	4c	8c	1c	4c	8c
2000000	2	100	50	153.75	38.83	19.36	36.30	9.78	5.06	4.2	4.0	3.8
2000000	2	400	50	563.16	141.67	70.93	118.03	29.92	15.52	4.8	4.7	4.6
2000000	8	100	50	314.17	79.40	39.81	98.73	25.61	13.30	3.2	3.1	3.0
2000000	8	400	50	1214.45	304.16	152.25	354.34	89.47	45.34	3.4	3.4	3.4
4000000	2	100	50	307.60	77.76	38.74	72.61	19.52	10.44	4.2	4.0	3.7
4000000	2	400	50	1127.86	283.26	141.84	236.09	59.76	30.14	4.8	4.7	4.7
4000000	8	100	50	629.28	158.82	79.60	197.44	51.28	26.73	3.2	3.1	3.0
4000000	8	400	50	2428.83	608.62	304.46	708.70	178.73	91.06	3.4	3.4	3.3
										3.9	3.8	3.7

N: Number of data points
D: Number of dimensions for each data point
K: number of clusters
M: number of iterations before stopping

While focusing on exploring potential speedup achievable by GPU over CPU, we also bear in mind that the CPU has a lot of performance potential as well [6]. Careful algorithm and data structure designs, using various optimization techniques, and the use of CPU's SSE vector capabilities etc, can usually help creating a CPU implementation that outperforms the non-optimized version by a considerable margin.

Since we are interested in the performance difference between CPU-only version and GPU-accelerated version, we have developed our own highly optimized K-Means package on CPU as well, trying to push the performance on CPU as much as possible. Our own optimized CPU code for K-Means runs several times faster than MineBench. It provides a better CPU performance benchmark to judge more accurately the value of GPU accelerators. Table 1 is the comparison between MineBench and our optimized CPU version, using 1 core, 4 cores and 8 cores respectively. It is shown that our optimized CPU implementation achieved about 3.8x speedup over MineBench implementation.

2.5 GPU Accelerated K-Means Algorithm

There are a few published works which have used GPUs for clustering, and in particular, using the K-Means method [2, 3, 5]. Among them, a team at University of Virginia, led by Professor Skadron, was one of the best. They did a series of research on using GPU to accelerate various general purpose applications, including K-Means. In their earlier work [2], 8x speed up was observed on a G80 GPU, versus MineBench running on a single core Pentium 4. Subsequently, they fine-tuned their code and achieved much better performance. Their latest version showed about 72x speedup on a GTX 260 GPU over a single threaded CPU version on a Pentium 4 running MineBench, and about 35x speedup over a four-thread CPU version running MineBench on a dual-core, hyper-threaded CPU [3].

Table 2. Speedups, compared to CPU versions running on 1 core

dataset				time (s)			speed ups (x)	
N	D	K	M	MineBench	HPL CPU	HPL GPU	MineBench	HPLC
2000000	2	100	50	153.75	36.30	1.45	106.0	25.0
2000000	2	400	50	563.16	118.03	2.16	260.7	54.6
2000000	8	100	50	314.17	98.73	2.48	126.7	39.8
2000000	8	400	50	1214.45	354.34	4.53	268.1	78.2
4000000	2	100	50	307.60	72.61	2.88	106.8	25.2
4000000	2	400	50	1127.86	236.09	4.36	258.7	54.1
4000000	8	100	50	629.28	197.44	4.95	127.1	39.9
4000000	8	400	50	2428.83	708.70	9.03	269.0	78.5
							190.4	49.4

In our previous paper ([11]), we have reported that our version is about 2-4x faster than that reported in [3]. For data sets smaller than GPU's onboard memory, our results are shown in Table 2. In this table, "HPL CPU" refers to our optimized CPU-only implementation, while "HPL GPU" refers to our GPU-accelerated version. The speedup ratio of GPU over CPU increases as the number of dimensions (D) and the number of clusters (K) increase, and for the set of parameters being experimented, we achieved an average of 190x speedup over MineBench running on single core, and 49x speedup over our own optimized CPU implementation running on single core.

Note that so far none of the published works has tackled the problem with very large data sets that are too large to fit inside GPU's onboard memory.

3. VERY LARGE DATA SETS

3.1 Multi-tasked Streaming

Compared to the CPU's main memory whose size can go up to 128GB in some of high-end machines, GPU's onboard memory is very limited. With the ever increasing problem size, it is often the case that GPU's onboard memory is too small to hold the entire data set. This has posed a few challenges:

- The problem has to be data-partitionable and each partition processed separately.
- If the algorithm requires multiple iterations over the data set, the entire data set has to be copied from CPU's main memory to GPU's memory at every iteration using the PCIe bus which is the only connection between CPU's main memory and GPU's memory.

The first challenge has to be answered by algorithm design, and in our case, K-Means can be data-partitioned in a straightforward manner. The only communication needed between partitions is the local sufficient statistics [12]. However, in the current generation of GPUs, the GPU's memory sub-system is connected to the main memory system via a PCIe bus. The theoretic bandwidth limit is about 4GB/s for PCIe 1.1x16, and the observed limit is about 3.2GB/s [10]. Heavy data transfer can pose a significant delay.

CUDA offers the APIs for asynchronous memory transfer and streaming. With these capabilities it is possible to design the algorithm that allows the computation to proceed on both CPU and GPU, while memory transfer is in progress.

Our stream-based algorithm is illustrated in Fig. 1.

```
Memcpy(dgc, hgc);
while (1)
{
    while (ns = streamAvail() && !done)
    {
        hnb = nextBlock();
        MemcpyAsync(db, hnb, ns);
        DTranspose(db, dtb, ns);
        DAssignCluster(dtb, dc, ns);
        MemcpyAsync(hc, dc, ns);
    }
    while (ns = streamDone())
        aggregateCentroid(hc, hb, ns);

    if (done)
        break;
    else
        yield();
}
calcCentroid(hgc);
```

Figure 1: Implementation for stream based K-Means (per iteration)

3.2 Data Preparation

When the data set is small enough to fit in the GPU memory, we adopted an implementation that transfers the entire data set and an initial set of k-centers to GPU memory upon start; the data set is then transposed in the GPU memory so that the use of GPU memory bandwidth is optimized [8]. Thereafter, for each iteration, GPU computes the "assign center" procedure which assigns a center id to each data point, and transfers the assignment results back to CPU which computes the new set of k-centers. The new set of k-centers is then copied to GPU to start the next iteration. Note that the data set is transferred from CPU to GPU and transposed only once.

In the stream-based algorithm, the data set is partitioned into large blocks. At every iteration, we process these blocks in turn, until all of them have been processed. Processing of each block includes transferring the block from CPU to GPU, transposing it to a column-based layout, computing cluster membership for each data point in the data block, and transferring the assignment results back to CPU. At any given time, more than one block is being processed concurrently. CUDA streams have been used to keep track of the progress on each block. Note that all calls are asynchronous, which give maximum possibilities for overlapping computation and memory transfers.

One important issue we investigated has to do with handling of data transposition (from row-based to column-based). It is not a problem when the data size fits into the GPU memory, in which case the data set is transposed once, kept inside GPU's memory and used for all iterations. However, when the data does not fit into the GPU memory, either transposition has to be performed per iteration at CPU, which proved too high in overhead, or the CPU memory has to keep 2 copies of the data set, one row-based and the other column-based, which is also not practical. Eliminating transposition altogether and forcing GPU to work on row-based data proved to be unacceptable in GPU performance. We solved this problem by inventing a method for a separate GPU kernel to transpose the data block once it is transferred. Our experiments have shown that this is the best solution to this problem.

3.3 Optimal Number of Streams

A series of experiments have also been run to determine what the optimal number of streams is, as shown in Table 3. It turns out that with the current GPU hardware, the choice of two streams works the best, offering much better performance than running with only one stream (no overlapping), while running with more than two streams does not bring any additional benefit.

Table 3. Performance comparison: number of streams

dataset				time (s)			
N	D	K	M	1 stream	2 streams	3 streams	4 streams
2000000	2	100	50	2.50	1.88	1.95	2.08
2000000	2	400	50	3.39	2.48	2.63	2.80
2000000	8	100	50	7.95	5.95	6.16	6.26
2000000	8	400	50	9.98	6.88	7.11	7.36
4000000	2	100	50	5.08	3.80	3.83	4.11
4000000	2	400	50	6.88	4.89	5.09	5.56
4000000	8	100	50	15.95	11.94	12.25	12.58
4000000	8	400	50	19.94	13.73	14.22	14.70

Table 4. Performance comparison: streamed vs. non-streamed version

dataset				time (s)		
N	D	K	M	non-stream	stream (2)	slowdown (x)
2000000	2	100	50	1.45	1.88	1.30
2000000	2	400	50	2.16	2.48	1.15
2000000	8	100	50	2.48	5.95	2.40
2000000	8	400	50	4.53	6.88	1.52
4000000	2	100	50	2.88	3.80	1.32
4000000	2	400	50	4.36	4.89	1.12
4000000	8	100	50	4.95	11.94	2.41
4000000	8	400	50	9.03	13.73	1.52

In enabling stream processing, we have observed that the extra overhead in the form of additional transposition work, kernel launch, and synchronization, has imposed a 1.1-2.5x degradation when compared to the original implementation where the entire data set fits into GPU memory, as shown in Table 4. However, even with this overhead, the use of GPU still offers significant speedup over pure CPU implementation. We believe that this overhead is justifiable since it expands the applicability of GPU computing significantly, enabling it to process much bigger data sets while still delivering significant performance gain.

3.4 Use of Constant vs. Texture Memory – Accommodating Large Number of Clusters

At each iteration, since we postpone the centroids calculation until we finish all the membership assignments, it is best to keep the centroids in constant memory which is cached. In the current generation of Nvidia's GPUs, the size of the constant memory is 64 KB for every multi-processor. This is a reasonable size and in most cases sufficient for our purposes (i.e., to hold shared read-only data.) However, as the number of clusters and number of dimensions increase, the constant memory will eventually become too small. We therefore explore the use of texture memory, whose limit in current hardware is 128MB.

Table 5. Cost for keeping centroids in other type of memory

dataset				time (seconds)			slowdown (x)	
N	D	K	M	C.Mem	T.Mem	G.Mem	T vs.C	G vs. C
2000000	2	100	50	1.45	2.00	3.39	1.4	2.3
2000000	2	400	50	2.16	4.39	5.23	2.0	2.4
2000000	8	100	50	2.48	5.06	5.45	2.0	2.2
2000000	8	400	50	4.53	14.95	9.55	3.3	2.1
4000000	2	100	50	2.88	3.98	6.66	1.4	2.3
4000000	2	400	50	4.36	8.73	10.56	2.0	2.4
4000000	8	100	50	4.95	10.09	10.88	2.0	2.2
4000000	8	400	50	9.03	29.84	19.08	3.3	2.1
							2.2	2.3

C.Mem: Constant Memory
T.Mem: Texture Memory
G.Mem: Global Memory

However, while the texture memory is usually faster than the global memory, it is still much slower than constant memory. In our experiments running on GeForce GTX 280, as shown in Table 5, the program is about 2.2x slower using texture memory than using constant memory on the average. It is interesting to also note that, as shown in Table 5, the global memory can sometimes be even faster than texture memory when the centroids array becomes large enough to cause too many cache misses on the texture memory; this implies that, in general, it is difficult to assert that texture memory is always a better choice than global memory.

3.5 Results with Very Large Data sets

The results of our experiments with very large data sets are shown in Table 6 and 7, where the number of data points has been increased to up to 1 billion with varying number of dimensions and centers.

The results are further explained below:

- The data sets in these experiments are too big to fit in the GPU's memory, and the program has to rely on multi-tasked streaming. GPU-based implementation lost some performance due to the overhead of multi-tasked streaming, but at the same time the larger computation load required to handle larger number of data points and larger number of clusters gave GPU additional edge over CPU. Overall, the program still achieved respectable speedup.

Table 6. Performance on large data sets, large number of data points

dataset				time (s)		speedups
N	D	K	M	CPU (8c)	GPU	
1,000,000,000	2	200	50	4139	508	8.2
1,000,000,000	2	400	50	7470	744	10.0
1,000,000,000	2	600	50	10847	1012	10.7
1,000,000,000	2	800	50	14176	1284	11.0
1,000,000,000	2	1000	50	17515	1558	11.2
						10.2

Table 7. Performance on large data sets, large number of centriods

dataset				time (s)		speedups
N	D	K	M	CPU (8c)	GPU	
100,000,000	2	2000	50	3415	299	11.4
100,000,000	4	2000	50	5969	446	13.4
100,000,000	6	2000	50	8343	2528	3.3
100,000,000	8	2000	50	10711	3354	3.2
						7.8

- The CPU-only program used here is our own highly optimized 64-bit version, running on 8 cores. Still the GPU version offers more than 10x speedup if the centroids array fits inside GPU's constant memory.
- Recall that our optimized CPU-only version running on 8 cores is about 29.3x faster than our baseline – MineBench running on single core (Table 1). The number above translates to about 300x speedup, compared to the baseline program MineBench on a single core. For the data set with one billion 2-d data points and one thousand clusters, the GPU-version took about 26 minutes, our optimized CPU-only version took close to 5 hours on 8 cores, and the baseline program would have taken about 6 days!
- At the point when D and K become too large, constant memory can no longer hold the read-only shared centroid data, and the implementation switches over to using texture memory. The result is shown in last two rows of Table 7. Observe the drop in speedup performance when this switch happens. However, even with this switch, we still achieved a 3x performance gain over 8 cores.

We also note that more than one GPU board can be installed into one machine, each with its own PCIe bus, which will give even more performance boost. It is one of our future goals to study systems with multi-GPU support.

4. CONCLUSION

We have reported our research on using GPU to accelerate a clustering algorithm with very large data sets. For these larger problems, the data set size exceeds GPU's memory. We showed that with implementation of highly efficient kernels and a careful design that maximizes the utilization of memory bandwidth and compute bandwidth on both CPU and GPU, the GPU-accelerated version still offers dramatic performance boost against CPU-only version. In our example, a clustering problem with one billion 2-dimensional data points, and one thousand clusters, can be processed in less than 30 seconds per iteration, compared to about 6 minutes per iteration with our highly optimized CPU version on 8 cores, or more than 11x speed up. The 50 iterations that took our GPU-based implementation 26 minutes to complete would have taken our baseline program MineBench close to 6 days to finish on a single-core CPU.

Note that GPU is still evolving at a fast pace. For example, the latest GPU from Nvidia, GTX 295, has doubled the capacity of the GTX 280 used in this paper.. In addition, more than one GPU board can be installed into the same machine. These can translate into greater performance potential, and potentially enable us to tackle even bigger problems.

5. REFERENCES

[1] K-Means++: The Advantages of Careful Seeding. D. Arthur, S. Vassilvitskii, 2007 Symposium on Discrete Algorithms, 2007.

[2] A performance Study of General Purpose Application on Graphics Processors. S. Che et al, Workshop on GPGPU, Boston, 2007.

[3] A Performance Study of General-Purpose Application on Graphics Processors Using CUDA. S. Che et al, J. Parallel-Distrib. Comput. 2008.

[4] CUDA Zone. http://www.nvidia.com/object/cuda_home.html

[5] Parallel Data Mining on Graphics Processors. W. Fang et al. Technical Report HKUST-CS08-07, Oct 2008.

[6] Intel® 64 and IA-32 Architectures Software Developer's Manuals. http://www.intel.com/products/processor/manuals/ 2008

[7] Exploring VLSI Scalability of Stream Processors. D. Khailany et al, Stanford and Rice University, 2003.

[8] Nvidia CUDA programming Guide, version 2.0. http://developer.download.nvidia.com/compute/cuda/2_0/docs/NVIDIA_CUDA_Programming_Guide_2.0.pdf 2008.

[9] NU-MineBench 2.0. J. Pisharath, et al, Tech. Rep. CUCIS-2005-08-01, Northwestern University, 2005.

[10] LU, QR and Cholesky Factorizations using Vector Capabilities of GPUs. V. Volkov and J. Demmel, Technical Report No. UCB/EECS-2008-49, 2008.

[11] GPU-Accelerated Large Scale Analytics. R. Wu, B. Zhang and M. Hsu. HP Labs Technical Report, HPL-2009-38. http://www.hpl.hp.com/techreports/2009/HPL-2009-38.html

[12] Accurate Recasting of Parameter Estimation Algorithms Using Sufficient Statistics for Efficient Parallel Speed-up: Demonstrated for Center-based Data Clustering Algorithms, Zhang, B, Hsu, M., and Forman, G., PKDD 2000.