GPU-Accelerated Large Scale Analytics

Ren Wu, Bin Zhang, Meichun Hsu

HP Laboratories
HPL- 2009-38

Keyword(s):
Data-mining, Clustering, Parallel, Algorithm, GPU, GPGPU, K-Means, Multi-core, Many-core

Abstract:
In this paper, we report our research on using GPUs as accelerators for Business Intelligence(BI) analytics. We are particularly interested in analytics on very large data sets, which are common in today's real world BI 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 the GPUs can be useful accelerators for BI analytics 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. For data sets smaller than GPU's onboard memory, the GPU-accelerated version is 6-12x faster than our highly optimized CPU-only version running on an 8-core workstation, or 200-400x faster than the popular benchmark program, MineBench, running on a single core. This is also 2-4x faster than the best reported work.

For large data sets which cannot fit in GPU's memory, we further show that with a design which allows the computation on both CPU and GPU, as well as data transfers between them, to proceed in parallel, the GPU-accelerated version can still offer a dramatic performance boost. For example, for a data set with 100 million 2-d data points and 2,000 clusters, the GPU-accelerated version took about 6 minutes, while the CPU-only version running on an 8-core workstation took about 58 minutes. Compared to other approaches, GPU-accelerated implementations of analytics potentially provide better raw performance, better cost-performance ratios, and better energy performance ratios.

External Posting Date: March 6, 2009 [Fulltext]
Internal Posting Date: March 6, 2009 [Fulltext]

Approved for External Publication


© Copyright 2009 Hewlett-Packard Development Company, L.P.

GPU-Accelerated Large Scale Analytics

Ren Wu, Bin Zhang, Meichun Hsu

HP Labs

ren.wu, bin.zhang, meichun.hsu@hp.com

Abstract

In this paper, we report our research on using GPUs as accelerators for Business Intelligence(BI) analytics. We are particularly interested in analytics on very large data sets, which are common in today's real world BI 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 the GPUs can be useful accelerators for BI analytics 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. For data sets smaller than GPU's onboard memory, the GPU-accelerated version is 6-12x faster than our highly optimized CPU-only version running on an 8-core workstation, or 200-400x faster than the popular benchmark program, MineBench, running on a single core. This is also 2-4x faster than the best reported work.

For large data sets which cannot fit in GPU's memory, we further show that with a design which allows the computation on both CPU and GPU, as well as data transfers between them, to proceed in parallel, the GPU-accelerated version can still offer a dramatic performance boost. For example, for a data set with 100 million 2-d data points and 2,000 clusters, the GPU-accelerated version took about 6 minutes, while the CPU-only version running on an 8-core workstation took about 58 minutes.

Compared to other approaches, GPU-accelerated implementations of analytics potentially provide better raw performance, better cost-performance ratios, and better energy performance ratios.

INTRODUCTION

Recent research by Gartner found that CIOs see business intelligence (BI) as the number-one technology priority [GAR07]. Next generation business intelligence will have much more sophisticated algorithms, process petabyte-size data sets, and require real-time or near real-time responses for a broad base of users. While massively parallel data management systems have been used to scale the data management capacity, BI analytics and data integration analytics have increasingly become a bottleneck, and can only get worse. High-performance support for these analytics is both a great challenge and a great opportunity.

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 provide substantial computational power to running general purpose applications at a much lower capital equipment cost and at much higher power efficiency, and therefore contribute to a greener economy while lowering operating costs.

In this paper, we report our explorations on using GPUs as accelerators for BI analytics. We are

especially interested in very large data sets that cannot fit into GPU's onboard memory. Large data sets are very common in today's real world BI scenarios.

Using the well-known clustering algorithm K-Means as an example, our results have been very positive. For smaller clustering problems whose entire data sets can fit inside the GPU's onboard memory, and whose cluster centroids (in K-Means, cluster centroids are computed iteratively) can fit inside GPU's constant memory, we have obtained up to 200-450x speedup over a popular benchmark program, or 50-90x over our own highly optimized CPU version on a single CPU core. For very large problems whose data sets cannot fit inside the GPU's onboard memory, the speedup performance appears to decrease somewhat but the GPU-accelerated version can still outperform CPU version by a wide margin.

In addition to the raw computing power, GPU-accelerated implementations potentially also offer improved cost-performance ratios and enegy-performance ratios for BI analytics

RELATED WORK

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 do 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 at best. A few attempts have been made to create new languages or APIs that abstract away from 3D APIs and provide general purpose interfaces [KHA03].

One of the most important advances in GPU computing is NVidia's CUDA solution, which provides both software support- the CUDA programming language extended from 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 GPU 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 great promise [CUA08].

Clustering Algorithms

There are a few published works which have used GPUs for clustering, and in particular, using the K-Means method. Among them, two pieces of work are closely related to ours [CHE07, CHE08, FAN08].

A team at University of Virginia, led by Professor Skadron, conducted a series of research on using GPU to accerelate various general purpose applications, including K-Means. In their earlier work [CHE07], 8x speed up was observed on a G80 GPU over MineBench running on a single core of 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-thread CPU version on a Pentium 4 running MineBench, and about 35x speedup over four-thread MineBench on a dual-core, hyper-threaded CPU [CHE08].

Another team at HKUST and Microsoft Research Asia has also looked into parallel data mining on GPUs [Fan08]. K-Means was one of the two algorithms used in their research. They reported more than 5x

speedup over the earlier version from University of Virginia even though it was still much slower than the newer version from the University of Virginia team.

K-MEANS ALGORITHM AND ITS IMPLEMENTATION

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 a 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 quickly and stops in a finite number of iterations.

There is still active research on the K-Means algorithm itself [e.g., ZHA00, ART07]. In this paper, we are interested in GPU acceleration rather than the K-Means algorithm itself.

Experiment design and hardware

Since MineBench [PIS05] has been used in a few related previous works [CHE07, CHE08], 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 HP XW8400 workstations, with dual quad core Intel Xeon 5345 running at 2.33GHz, 4GB CPU memory; one running Windows XP professional x64 equipped with a Quadro FX 4600 GPU (NVIDIA G80 series) with 768MB onboard device memory, and the other running Windows XP (32bits), equipped with an Nvidia GeForce GTX 280 GPU with 1GB onboard device memory. The GPU code was developed in CUDA, NVidia's C extension to support general GPU computing, 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.

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.

While focusing on the speedups achievable by GPU over CPU, we observe that it is equally important to realize that the CPU has a lot of performance potential as well [INT08]. Careful algorithm and data structure designs, using verious optimization techniques, and the use of CPU's SSE vector capbilities etc, can usually help create a CPU implementation that outperforms the non-optimized CPU version by a

considerable margin as well. 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. Various optimizations have been put into our CPU code, at both the C language level and assemembler level using SSE, developed in Microsoft Visual C++ 2005 with OpenMP turned on for the multi-core support. 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 in BI analytics. Table 1 is the comparsion between MineBench and our optimized 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.

GPU acceleration

The K-Means algorithm spends majority of its time in Step 1, data partitioning, which is relatively easy to parallelize for both CPU and GPU. For example, both MineBench and our own implementation of K-Means on CPU achieved close to linear speedup on our 8-core workstation [Table 1, PIS05].

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

Table 1 Performance comparison between MineBench and optimized implementation
N: Number of data points
D: Number of dimensions for each data point
K: number of clusters
M: number of iterations before stopping

However, due to the dramatic differences in architecture, implementing an algorithm to take full advantage of GPU requires careful designs. Some strategies and principles we discovered and used are discussed here.

Use of the Share-nothing principle

Share-nothing is a strategy for implementing data parallelism by avoiding the use of shared update variables. This strategy is critical to algorithms designed for distributed systems. In the case where the shared variables are used to hold global sufficient statistics, it's been shown that it is possible to first calculate the local sufficient statistics in parallel without the use of shared variables, and then compute the global sufficient statistics and update those globally shared variables as a final step by aggregating from the locally computed statistics. Even though modern GPUs have the global onboard memory shared among all multi-processors, the cost of synchronization for shared update variables usually outweighs its benefits. We have found that following the share-nothing principle in algorithm design is equally important in shared-memory parallel systems [ZHA00].

Beware of the memory hierarchy

GPU has added its own memory hierarchy, which is very different from the CPU's cache/memory hierarchy. Understanding GPU's memory hierarchy is crucial in algorithm design. Several key principles we found crucial are:

- arranging the data to maximize the utilization of memory bandwidth,
- using constant and texture memory for the data that remains constant during one kernel invocation to utilize the cache mechanism, and
- using shared memory for the data that will be accessed multiple times during one kernel invocation.

GPU offers much better memory bandwidth compared to CPU. But the high bandwidth can only be fully utilized by coalesced memory access. GPU favours that multiple threads work together on continuous

memory addresses, rather than having each thread work on its own, while the latter is a common practice in CPU multi-thread optimization. In other words, it is important to design GPU data structure to be column-based instead of row-based as commonly found in algorithms designed for CPU (In the case of K-Means, each row is a data point, while each column is a dimension).

Take full advantage of the asymmetric architecture

GPU and CPU excel in different areas. It is important to partition the problem so that both CPU and GPU are doing what they are best at. In our K-Means example, we found that in many cases a quad-core CPU can do equally well for Step 2, even with the cost of shipping the intermediate result (which is the membership vector) between CPU and GPU. Some of the communication time can be hidden by using asynchronous memory transfers.

Results with a Smaller Data Set - In-GPU-Memory Data Set

In our K-Means implementation, for smaller problems the data has been kept in both GPU's and CPU's memory. The data on GPU has been rearranged to be column-based to avoid non-coalesced memory access. We have identified that the best way to partition the tasks among CPUs and GPUs is to let the CPU calculate the new centroids in each iteration (i.e., Step 2), and let the GPU calculate the data point's membership to the clusters (i.e., Step 1). This is essentially a CPU-GPU hybrid implementation. At each iteration, CPU sends new centroids down to GPU, and GPU returns the updated membership vector back to CPU, and CPU will then check the stopping rule and starts the next iteration if needed. The data transferred for each iteration includes centroids (with a sizeof *k*D*4* bytes) and membership (with a size of *N*4* bytes), but not the full data set itself, which has a size of *N*D*4* bytes. Our results are shown in Table 2 (for comparison with 1-core CPU) and Table 3 (for comparison with 8-core CPU). The speedup ratio of GPU over CPU increases as the number of dimensions (D) and the number of clusters (*k*) increase. For the set of parameters being experimented, we achieved an average of 190x speedup over the MineBench implementation on a 1-core CPU, and 49x over our own optimized implementation on a 1-core CPU. Our experiments have shown that each of the design principles discussed above played a crucial role in achieving this level of speedup.

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

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

It is interesting to note that, when the number of clusters is relatively small (e.g. *k*<70), a pure GPU-version of the implementation actually outperforms the hybrid version. Since our main focus was to investigate K-Means algorithm on large data sets and large numbers of clusters, we will focus on the CPU-GPU hybrid version of implementation in this paper.

dataset				time (s)			speed ups (x)	
N	D	K	M	MineBench	HPL CPU	HPL GPU	MineBench	HPLC
2000000	2	100	50	19.36	5.06	1.45	13.4	3.5
2000000	2	400	50	70.93	15.52	2.16	32.8	7.2
2000000	8	100	50	39.81	13.30	2.48	16.1	5.4
2000000	8	400	50	152.25	45.34	4.53	33.6	10.0
4000000	2	100	50	38.74	10.44	2.88	13.5	3.6
4000000	2	400	50	141.84	30.14	4.36	32.5	6.9
4000000	8	100	50	79.60	26.73	4.95	16.1	5.4
4000000	8	400	50	304.46	91.06	9.03	33.7	10.1
							24.0	6.5

Table 3 Speedups, compare to CPU versions running on 8 cores

Large Data Sets – Multi-taksed Streaming

Compared to the CPU's main memory, which can go up to 128GB now, GPU's onboard memory is very limited. With the ever increaseing 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 seperately,

- If the algorithm requires going over the data set with multiple iterations, the entire data set has to be copied from CPU's main memory to GPU's memory at every iteration, while the PCIe bus is the only connection between CPU's main memory and GPU's memory.

The first one 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 [ZHA00]. 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 [VOL08]. Heavy data transfer can pose a significant delay.

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

Fig. 1 is a much simplified algorithm we used.

```
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: Our implementation for stream based K-Means (per iteration)

Streaming K-Means Algorithm:

The data set is partitioned into large blocks. At every iteration, we process these blocks in turn, with overlap, until all of them have been processed. Each block has to be transferred from CPU to GPU, transposed to a column-based layout, have cluster membership computed for each data point, and then transferred back to CPU, where CPU does a partial aggregation for each centroid. In addition, CUDA streams have been used to keep track of the progress on each stream. Note that all calls are asynchronous, which allows computation and memory tranfers to proceed concurrently

Another issue has to do with handling of data transposition (from row-based to column-based). When the data size fits into the GPU memory, the data set is transposed once and used for all iterations. When the data does not fit into the GPU memory, either transposition has to be performed per iteration, 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. Avoiding transposition altogether and force GPU to work on row-based data proved to be unaccpetable in GPU performance. We solved this problem by inventing a method for a separate GPU kernel to transpose the data block once it transferred. Our

experiments have shown that this is the best solution to this problem.

A series of experiments have also been run to determine what is the optimal number of streams, as shown in Table 4. It turns out that with the current GPU hardware, running with two streams works the best, offering much better performance than the running with only one stream (no overlapping), while running with more than two streams does not bring in any additional benefit.

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, number of streams

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 5. However, even with this overhead, the use of GPU still offers significant speedup over pure CPU implementation, and it is signficant that GPU can now be applied to bigger data sets.

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

Table 5 performance comparison, vs. non-stream version

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 resonable size and in most cases suffcient for our purposes (i.e., to hold shared read-only data.) However, as the number of clusters and the number of dimensions increase, the constant memory will eventually become too small to hold these data. We therefore explore the use of texture memory, whose limit in current hardware is 128MB.

GTX280

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

Table 6 Cost for keeping centroids in other type of memory
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 6, overally speaking the program is about 2.2x slower using texture memory than using constant memory. (It is interesting to also note that, as shown in Table 6, the global memory can even be faster than texture memory when the centroid array becomes large enough to cause too many cache misses on the texture memory, making it a tossup between texture and global memory.)

It is also interesting to see the same comparsion on an earlier CUDA-enabled hardware, Quadro FX 4600, shown in Table 7. Here the texture memory is about 4.1 times slower than the constant memory, while the global memory is about 18.2 times slower! Furthermore, the use of global memory appears to get even worse in comparison as the centroid array increases its size. So

FX4600

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	1.58	4.2	24.44	2.7	15.5
2000000	2	400	50	3.78	14.16	64.09	3.7	17.0
2000000	8	100	50	3.58	14.86	63.97	4.2	17.9
2000000	8	400	50	9.42	54.86	250.96	5.8	26.6
4000000	2	100	50	3.14	8.31	48.63	2.6	15.5
4000000	2	400	50	7.59	28.28	127.61	3.7	16.8
4000000	8	100	50	7.16	29.72	127.59	4.2	17.8
4000000	8	400	50	18.84	109.47	n/a	5.8	n/a
							4.1	18.2

Table 7 Costs for keeping centroids in other type of memory

while in the current hardware, the distinction between global memory and texture memory has been narrowed, the difference is much bigger on the earlier hardware. This illustrates the importance of understanding the tradeoffs in managing the GPU memory hierarchy in performance-oriented implementation and algorithm design.

Results with Very Large Data sets

The results with very large data sets are shown in Table 8, where the number of data points has been increased to up to 100 million. The results are further explained below:

dataset				time (s)			speed ups (x) vs.	
N	D	K	M	MineBench	HPL CPU	HPL GPU	MineBench	HPL CPU
10000000	2	1000	50	6922.70	1391.95	15.61	443.5	89.2
10000000	2	2000	50	13763.60	2721.08	28.75	478.7	94.6
10000000	8	1000	50	14949.52	4299.67	42.83	349.0	100.4
10000000	8	2000	50	29763.14	8534.25	335.94	88.6	25.4
20000000	2	1000	50	13850.42	2784.00	31.16	444.5	89.3
20000000	2	2000	50	27528.02	5442.58	57.55	478.3	94.6
20000000	4	1000	50	19556.45	4819.36	46.48	420.7	103.7
20000000	4	2000	50	39162.05	9516.32	93.11	420.6	102.2
50000000	2	1000	50	34613.97	6959.94	77.98	443.9	89.3
50000000	2	2000	50	68819.34	13605.90	146.49	469.8	92.9
50000000	4	1000	50		12048.92	158.99		75.8
50000000	4	2000	50		23793.58	263.37		90.3
100000000	2	1000	50		13919.02	201.85		69.0
100000000	2	2000	50		27211.69	341.27		79.7
							403.8	85.5

Table 8 Performance on large data sets

- MineBench runs on a 32-bit system and is no longer applicable when data set size goes beyond 50 million of 2-d data elements. This is why MineBench numbers are missing in the last few rows. "HPL CPU" is based on the optimized CPU implementation on a 64-bit system and is able to go up to 100 million of 2-d data elements.
- At the point where $D \times k$ = 16,000, constant memory can no longer hold the read-only shared centroid data, and the implementation switches over to using texture memory. This is shown in row 4 of Table 8. (This is the only scenario where texture memory is used in this set of experiments.) Observe the drop in speedup performance when this switch happens.
- At the point when N x D = 200 million (which are shown at the last 4 rows in Table 8), the data set can no longer fit in the GPU's memory, and the implementation switches to mutli-tasked streaming. Observe the moderate drop in speedup performance in these scenarios due to overhead of multi-tasked streaming.
- The algorithm benefits from the larger number of clusters, and more than 478 times speedup can be achieved vs. MineBench. The speedup drops by a few times when a switch to texture memory occurs (i.e., shown in row 4), but it is still about 88 times faster. Overall, the GPU accelerated version offers 400x speedup over the baseline benchmark based on MineBench, and 88 times against our highly optimized CPU-only version.
- Compared to MineBench, our baseline benchmark program, the GPU-accelerated version is about 200x-400x faster than the CPU version running on a single core, data setwith higher speedups for larger data sets. It is about 40x-80x faster than MineBench running on a single Quad-core chip, and about 20x-40x faster than MineBench running on an 8-core workstation (see Table 1 for 8-core results). Compared to our own highly optimized CPU version on a single core, the GPU accelerated version is still about 50x-88x faster, or 12x-22x vs. quad core, and 6x-11x vs. 8 cores workstation. These represent significant performance gains. In addition, more than one GPU board can be installed into one machine, which will provide even more performance boost.

Compare to Previous Work on Using GPU for Clustering

We have downloaded the latest version from University of Virginia and have made changes so that we can run same benchmark tests. It has shown a very respectable performance. Table 9 shows the results, both programs was compiled using the same compiler and same compiler flags, running on the same

hardware with GTX280 GPU installed.

Our version is about 2 to 4x faster, with an average of 2.7x. It also shows that the performance gap increases with the data set size, dimensionality, and the number of clusters.

We have also downloaded Gminer from their website. Again, we have used the same compiler and same compiler flags, running on same hardware[1]. The results we obtained have shown that both our solution and UoV's solution are much faster than GMiner, which is also consistant with UoV's reports [FAN08]. The bitmap approach, while elegant in expressing the problem, seems not a good choice for performance consideration. It take more space as well when *k* is large. This approach also requires to use shared memory during centroids calculation, which has a tighter limit (16KB vs. 64KB).

None of the works reported have dealt with the problem of a large data set which can't fit inside GPU's memory.

N	D	K	M	U.V.	HPL	Speedups(x)
2000000	2	100	50	2.84	1.45	2.0
2000000	2	400	50	5.96	2.16	2.8
2000000	8	100	50	6.07	2.48	2.4
2000000	8	400	50	16.32	4.53	3.6
4000000	2	100	50	5.64	2.88	2.0
4000000	2	400	50	11.94	4.36	2.7
4000000	8	100	50	12.85	4.95	2.6
4000000	8	400	50	34.54	9.03	3.8
						2.7

Table 9 Comparison with results reported by U. of Virginia

N	D	K	M	Gminer	HPL	Speedups(x)
2000000	2	100	50	63.46	1.45	43.8
2000000	2	400	50	61.39	2.16	28.4
2000000	8	100	50	192.05	2.48	77.4
2000000	8	400	50	226.79	4.53	50.1
4000000	2	100	50	130.36	2.88	45.3
4000000	2	400	50	126.38	4.36	29.0
4000000	8	100	50	383.41	4.95	77.5
4000000	8	400	50	474.83	9.03	52.6
						50.5

Table 10 Comparison with GMiner

DISCUSSION

There are other ways to accelerate analytics, for example, using a large cluster of machines and specially designed hardware accelerators. However, it will need a very large cluster to offer equivalent performance, or very expensive custom designed hardware. Our research shows that the GPU based approach offers the best cost-performance ratio (About $400 for NVidia GeForce 280). It also offers a much better energy-performance ratio than clusters. (NVidia GeForce 280's peak power consumption is at about 180Ws.)

While there has been considerable interest in using GPU in high performance computing [LUP08], there is relatively little work so far in exploring the use of GPU in business intelligence. Some work has been done to GPU to speed up decompression in data warehouses [LEH08], however the results are at this time inconclusive. It is our thesis that with research insights in design principles, using GPU on the server side to accelerate BI has a great deal of potential.

CONCLUSION

We have reported our research on using GPU to accelerate large scale BI analytics and the design and implementation principles. For problems that can fit inside GPU's memory, our GPU-accelerated version can be 200-400x faster than MineBench, our baseline benchmark, running on a single CPU core, and about 50x-88x faster than our highly optimized CPU version running on a sinlge core. These results are better than previous works.

For larger problems whose data sets do not fit in the GPU's memory, we further showed that GPUs can still offer significant performance gains. For example, a very large data set with one hundred million 2-dimensional data points, and two thousand clusters, can be processed by GPU in less than 6 minutes, compared to 450 minutes with our highly optimized CPU version on a single core, representing an 80x speedup.

[1] As of today, Dec 15, 2008, the downloaded source contained a few bugs. The results shown was after our own bug-fixing without contacts to the original authors. However, the performance reported (time per iteration) should not change because of this.

In addition to K-Means, we are exploring implementation of other BI algorithms, such as EM, K-Harmonic Menans, Support Vector Machines, and Combinatorial optimization, as well as data management algorithms such as index search and database layout and compression. Our hypothesis is that a sweet spot exists for exploiting the GPGPU technology in Business Intelligence, and this work will be furthered to derive the general principles in algorithm and implementation design as well as in infrastructure software support to enable GPGPU technology to be utilized in BI.

References

[ART07] K-Means++ The Advantages of Careful Seeding. D. Arthur, S. Vassilvitskii, 2007 Symposium on Discrete Algorithms, 2007.

[CHE07] A performance Study of General Purpose Application on Graphics Processors. S. Che et al, Workshop on GPGPU, Boston, 2007

[CHE08] A Performance Study of General-Purpose Application on Graphics Processors Using CUDA. S. Che et al, J. Parallel-Distrib. Comput. 2008.

[CUA08] CUDA Zone. http://www.nvidia.com/object/cuda_home.html

[FAN08] Parallel Data Mining on Graphics Processors. W. Fang et al. Technical Report HKUST-CS08-07, Oct 2008.

[GAR07] http://www.gartner.com/it/page.jsp?id=501189 2007

[INT08] Intel® 64 and IA-32 Architectures Software Developer's Manuals. http://www.intel.com/products/processor/manuals/ 2008.

[KHA03] Exploring VLSI Scalability of Stream Processors", D. Khailany et al, Stanford and Rice University, 2003.

[LEH08] Exploiting Graphic Card Processor Technology to Accelerate Data Mining Queries in SAP NetWeaver BIA", W. Lehner et al. Workshop on High Performance Data Mining (HPDM), 2008.

[LUP08] Acclerating HPC Using GPU's. G. Lupton and D. Thulin, http://www.hp.com/techservers/hpccn/hpccollaboration/ADCatalyst/downloads/accelerating_HPC_Using_GPU's.pdf, June 2008.

[PIS05] NU-MineBench 2.0, Tech. Rep. CUCIS-2005-08-01, J. Pisharath, et al, Northwestern University, 2005.

[VOL08] LU, QR and Cholesky Factorizations using Vector Capabilities of GPUs. V. Volkov and J. Demmel, Technical Report No. UCB/EECS-2008-49, 2008.

[ZHA00] 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.