Scalable Data Clustering Using Fermi GPUs on FutureGrid (Report Draft, 2012)

ABSTRACT

The applications in science are creating huge amount of data sets. These data sets need to be classified into subsets in order to draw some meaningful conclusions. Data clustering is the statistical analysis process that groups similar objects into relatively homogeneous sets which are called clusters. The computational demands of data clustering grow rapidly. And it is very time consuming for single CPU to processing large data sets. To address this computational demands, we explored several parallel programming models for Fuzz C-means clustering algorithm on NVidia Fermi GPU architecture on the FutureGrid. We implemented C-means with CUDA and scale the program to GPUs cluster through a hybrid usage of MPI and OpenMP. In addition, a MapReduce implementation of C-means is also given and discussed. We evaluated the performance of different implementations of C-means on GPUs and compare their results with that of traditional Intel architecture chips. The results showed that CUDA implementation of C-means on single C2070 Fermi GPU card gave 70x and 10x speedup as compared to CPU implementation on Intel Xeon processor with 1 core and 12 cores respectively.

Keywords

Data Clustering, CUDA, GPU, Programming Models.

1. INTRODUCTION

The applications in science are creating huge amount of data sets. These data sets need to be classified into subsets in order to draw some meaningful conclusions. Data clustering is the statistical analysis process that groups similar objects into relatively homogeneous sets which are called clusters. Data clustering has a wide variety of fields, such as data mining, machine learning, geology, astronomy, and bioinformatics, etc. The nature of the data similarity or distance varies significantly from one application to another. Therefore there has been extensive research and a myriad of clustering techniques developed in the past decades.

Multivariate data clustering techniques were created several decades ago; however the application to the field of flow cytometry only has limited discussion. There has been a recent surge in research activity over the past few years applying multivariate data clustering to flow cytometry data. Multivariate techniques have the potential to use the full multidimensional nature of the data, to find cell populations of interest (that are difficult to isolate with sequential bivariate gating), and to allow analysts to make more sound statistical inferences from the results. Flow cytometry data sets are complex, containing millions of events, dozens of dimensions, and potentially hundreds of natural clusters. The multivariate clustering techniques require intensively computation, and the computational demands grow rapidly as the number of clusters, events, and dimensions increase. This makes it time consuming to analyze a flow cytometry data set thoroughly using a single CPU. Fortunately, many clustering techniques are of parallel processing capability.

The GPUs have become booming parallel systems. GPUs have hundreds of processor cores and thousands of threads running concurrently on these cores, thus because of intensive computing power they are much faster than the CPU. The NVIDIA® Fermi architecture is the next-generation compute architecture for NVIDIA® CUDA™ applications.

The performance of the GPUs applications highly depends on whether the programs can exploit parallelism provided by the underlying multiprocessor architecture. As a result, there is a large need to explore programing models to leverage the computational power of these new GPUs architecture. CUDA technology [5] is a new hardware and software solution for general purpose parallel computing from NVIDIA. CUDA is a parallel programming model and software environment that leverages the parallel computational horsepower of GPU for non-graphics computing in a fraction of the time required on a CPU. The latest version of CUDA can run parallel program on multi-core CPU as well.

The programming paradigm provided by CUDA has allowed developers to utilize the power of these scalable parallel processors with relative ease, enabling them to achieve speedups of several times on a variety of sophisticated applications. Since NVIDIA released CUDA in 2007, a lot of scalable parallel programs were rapidly developed for a wide range of applications, including matrix solvers, sorting, searching, computational chemistry, and physics models. These applications scale transparently to hundreds of processor cores and thousands of concurrent threads.

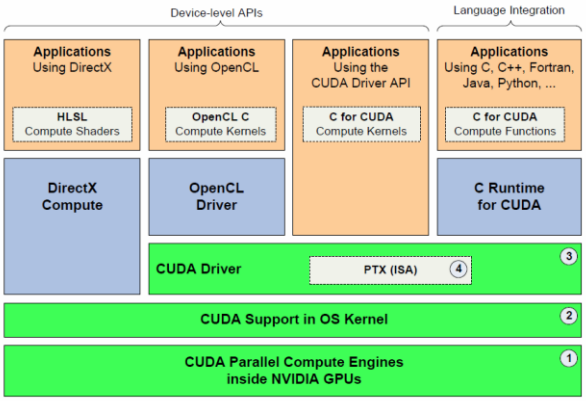

Figure 1: NVIDIA CUDA Framework

2. Parallel Programming Models on GPUs

GPU has shown its incredible power in high performance systems such as Tianhe 1A and Blue water. The programming model is critical to leverage these GPU systems in the respect of performance, programmability, and event power efficiency. In this report, we evaluated four parallel programming models for C-

means on Fermi GPU architecture which include: CUDA, OpenMP, MPI, and MapReduce.

2.1 CUDA C++

Currently, NVIDIA's CUDA toolkit is the most widely used GPU programming toolkit available. It includes a compiler for development of GPU kernels in an extended dialect of C that supports a limited set of features from C++, and eliminates other language features (such as recursive functions) that do not map to GPU hardware capabilities. The CUDA programming model is focused entirely on data parallelism, and provides convenient lightweight programming abstractions that allow programmers to express kernels in terms of a single thread of execution, which is expanded at runtime to a collection of blocks of tens of threads that cooperate with each other and share resources, which expands further into an aggregate of tens of thousands of such threads running on the entire GPU device. Since CUDA uses language extensions, the work of packing and unpacking GPU kernel parameters and specifying various runtime kernel launch parameters is largely taken care of by the CUDA compiler. This makes the host side of CUDA code relatively uncluttered and easy to read.

2.2 Combining OpenMP and CUDA

Since NVIDIA's GPU driver allows only one CPU thread talk to one GPU device at a time, you'll need to use multiple CPU threads to cooperate with multiple GPU devices in a single program. OpenMP (Open Multiprocessing) is an API that supports multi-platform shared memory multiprocessing programming in C, C++. OpenMP uses a portable, scalable model that gives programmers a simple and flexible interface for developing parallel applications for platforms ranging from the standard desktop computer to the supercomputer. Combining OpenMP and CUDA framework can make use of multiple GPUs cards that deployed on single compute node.

2.3 Combining OpenMP, MPI, and CUDA

Many of the HPC applications have been implemented using MPI for parallelizing the application. The simplest way to start building an MPI application that uses GPU-accelerated kernels is to use NVIDIA's nvcc compiler for compiling everything. The nvcc compiler wrapper is somewhat more complex than the typical mpicc compiler wrapper, so it is easier to make MPI code into .cu (since CUDA is a proper superset of C) and compile with nvcc than the other way around. The important point is to resolve the INCLUDE and LIB paths for MPI since by default nvcc only finds the system and CUDA libs and includes.

In one scenario, one could run one MPI thread per GPU, thus ensuring that each MPI thread has access to a unique GPU and does not share it with other threads. On Lincoln this will result in unused CPU cores. In another scenario, one could run one MPI thread per CPU. In this case, on Lincoln multiple MPI threads will end up sharing the same GPUs, potentially oversubscribing the available GPUs. On AC the outcome from both scenarios is the same.

2.4 Combining MapReduce and CUDA

We investigated a MapReduce framework named Mars on graphics processors (GPUs). MapReduce is a distributed programming framework originally proposed by Google for the ease of development of web search applications on a large number of CPUs. Compared with commodity CPUs, GPUs have an order of magnitude higher computation power and memory bandwidth, but are harder to program since their architectures are designed as a special-purpose co-processor and their programming interfaces are typically for graphics applications. Mars hides the programming complexity of the GPU behind the simple and familiar MapReduce interface. It is up to 70 times faster than its CPU-based counterpart for C-means application. We implemented C-means with Mars on an NVIDIA T2070 GPU on FutureGrid, which contains hundreds of processors.

3. Data Clustering Applications

3.1 C-means Application

Fuzzy c-means is an algorithm of clustering which allows one element to belong to two or more clusters with different probability. This method is frequently used in multivariate clustering. This algorithm is based on minimization of the following objective function:

$$J_m = \sum_{i=1}^{N}\sum_{j=1}^{C} u_{ij}^{m}\left\|x_i - c_j\right\|^2$$

M is a real number greater than 1, N is the number of elements. Uij is the value of membership of Xi in cluster Cj. ||Xi-Cj|| is the norm expressing the similarity between the measured and the center. where m is any real number greater than 1, uij is the degree of membership of xi in the cluster j, xi is the ith of d-dimensional measured data, cj is the d-dimension center of the cluster, and ||*|| is any norm expressing the similarity between any measured data and the center. Fuzzy partitioning is performed through an iterative optimization of the objective function shown above. Within each iteration, the algorithm updates the membership uij and the cluster centers cj by:

$$u_{ij} = \frac{1}{\sum_{k=1}^{C}\left(\frac{\|x_i - c_j\|}{\|x_i - c_k\|}\right)^{\frac{2}{m-1}}} \quad (1)$$

$$c_j = \frac{\sum_{i=1}^{N} u_{ij}^{m} \cdot x_i}{\sum_{i=1}^{N} u_{ij}^{m}} \quad (2)$$

This iteration will stop when $\max_{ij}\left\{\left|u_{ij}^{(k+1)} - u_{ij}^{(k)}\right|\right\} < \varepsilon$, where 'e' is a termination criterion between 0 and 1, whereas k are the iteration steps.

Algorithm of C-means with CUDA:

```
 1) Copy data to GPU
 2) DistanceMatrix kernel
 3) MembershipMatrix kernel
 4) UpdateCenters kernel, copy partial centers to
host from GPUs
 5) ClusterSizes kernel, copy cluster sizes to
host from each GPU
 6) Aggregate partial cluster centers and reduce
 10) Compute difference between current cluster
centers and previous iteration.
 11) Compute cluster distance and memberships
using final centers.
```

4. Performance Evaluation

To provide an advanced and uniform evaluation platform, the FutureGrid systems are used. The FutureGrid project provides a capability that makes it possible for researchers to tackle complex research challenges in computer science related to the use and security of grids and clouds. Table1 summarizes characteristic of GPUs cluster named Delta on the FutureGrid.

Table 1. GPUs cluster on FutureGrid

GPU Type	nVIDIA Tesla C2070
GPUs per node	2
RAM	16 GB DDR3 1333 MHz
Memory per node [GB]	192
Total GPUs	32
Cores per GPU	448
CPU type	Intel Xeon 5660
CPU Speed	2.80 GHz
CPUs (cores) per node	2 (12)

As shown in Table 1, Delta is a new 16-node experimental cluster, where each node has 2 NVIDIA Tesla C2075 GPUs with 448 processing cores. The NVIDIA cards were used to evaluate performance of the C-means GPU programs. The corresponding sequential or threaded CPU code was executed as normal on the Intel Xeons under the normal process scheduling mechanisms provided under Red Hat Enterprise Linux 6.

4.1 C-means performance on single GPU

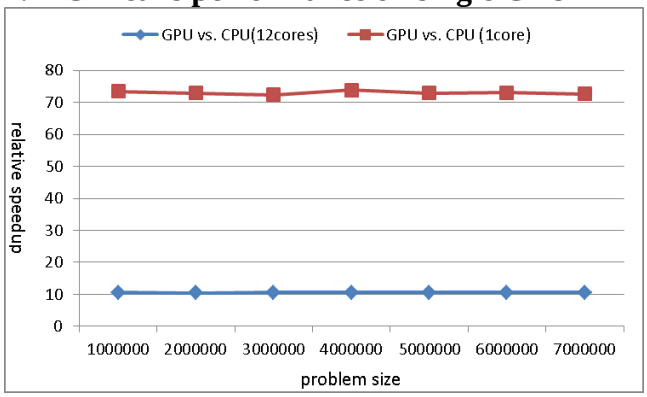

Figure 2: relative speedup of GPU implementation as compared to CPU implementation with 12 cores and 1 core respectively.

We first look at the performance comparision between CPU and GPU implementations of C-means. Figure 2 is the relative speedup of CUDA implementation of C-means as compared to CPU implementations. The job turnaround time of CUDA C-means program included the GPU kernel, CPU sequential, and memcpy between host and device memory. As shown in figure 2, GPU implementation is 10 times and 70 times faster than CPU implmentations with 12 core and 11 cores respectively. One should note there is only minor performance fluctuation for different input data as both CPU and GPU have the very large memory space on each node as shown in table 1.

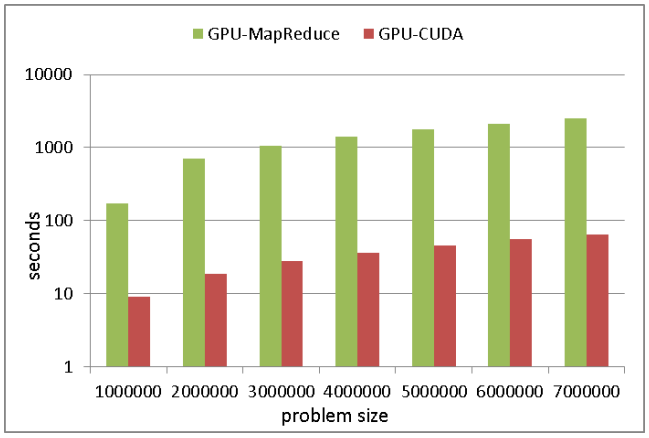

Figure 3: performance of MapReduce implementaiton of C-means

Next, we evalute performance of MapReduce implementation of C-means on single GPU card with input data size range from 1million to 7 million events. The results indicated that the MapReduce implementation has a very slow performance as comapred to pure CUDA implementation. The reason is because the MapReduce framework (Mars) used in this report is designed for previous Tesla GPU architecture, and more importantly there is no local combiner in Mars to perform the parallel reduce computation. As a result, the overhead of reduce stage in MapReduce implementation is similar to that sequntial version run on single CPU. By profiling the MapReduce program, we found that overhead in reduce stage of MapReduce C-means take up to 80% of overall overhead.

4.2 C-means performance on multiple GPUs

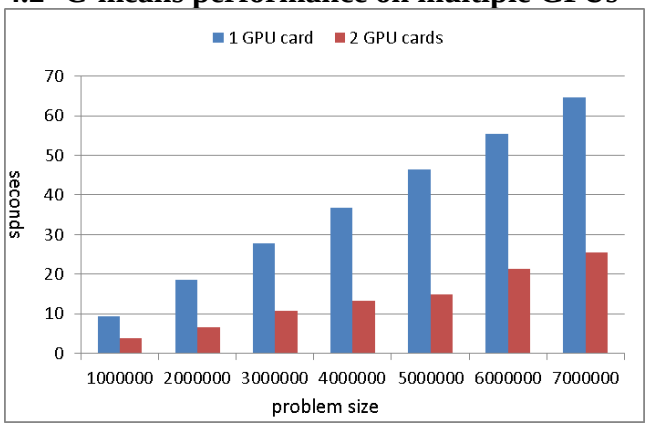

Figure 4: performance CUDA implementation on multiple GPUs

As each compute node in table 1 has two GPU cards, we implemented C-means by combining OpenMP and CUDA to leverage both GPU cards. Results in Figure 4 indicated the super linear speedup of OpenMP implementation with two GPU cards. We profiled overhead components of each OpenMP process, and found the performance gain come from GPU kernel due to reduced size of input data. When running the GPU program with half input data size, there is less memory bandwidth contention among thousands of threads on each GPU card.

4.3 C-means performance on GPU cluster

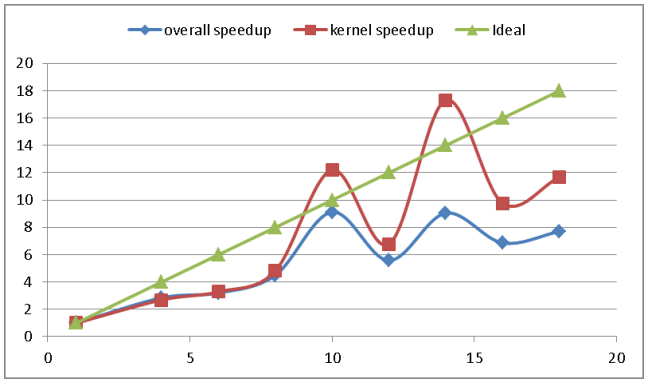

Figure 5: Speedup of MPI/OpenMP implmenetation of C-means on multiple GPUs.

One flaw of current GPUs architecture (including Fermi) is the lack of connection between GPUs nodes in both software and hardware. The message passaing interface, MPI, is the tradition approach to connect distributed program on ditributed memory archticture. And we use a hybrid of MPI, OpenMP, and CUDA to bridge the hardware gap between socket and PCI interface of GPUs nodes. Figure 5 showed the speedup of MPI/OpenMP/CUDA implementation of C-means for 7 million events using up to 18 GPU cards (9nodes with 2 cards each) on GPU cluster. The kernel speedup is cacluated by only measuring the GPU kernel overhead, while overall speedup is caculated by measuring GPU kernel, CPU overhead, and memcpy between device and host memory. As expected, the kernel speedup is higher than overall speedup which contains overhead in sequetnail component. In addition, as showed in Figure 5, there is big performance fluctuation for different number of GPU nodes due to the memory coalesced issue related with input granularity.

5. Conclusion and Future Work

We evaluated four parallel programming models for C-means application on Fermi GPUs on the FutureGrid, which include CUDA, OpenMP, MPI, and MapReduce. The CUDA implementation of C-means gave 70x and 10x speedup as compared to CPU implementation with 1 core and 12 cores respectively. We showed that the traditional parallel programming technical -- OpenMP and MPI can scale the data clustering program to multiple GPU nodes on the FutureGrid with reasonable parallel overhead and at the cost of put more software development burden on developers. The MapReduce/CUDA implementation required less programming effort, but it just gave 2x speedup as compared to the CPU implementation on single core. As a result, more research work should be done to optimize the MapReduce framework on GPUs. For example, a local combiner for threads within the same block, can increase speedup of C-means significantly.

While CUDA gave comparable performance, achieving the maximum throughput and utilization of the GPUs is still difficult task even for the simple parallel applications. The developers need have sophisticated knowledge about the details of warps, the memory hierarchy, and the efficient use of a limited PCIX bus. In addition, if developers want to scale their program to multiple GPU nodes, they have to make a hybrid use of MPI, OpenMP, and CUDA framework, and handle the messaging very carefully. Therefore a user-friendly programming model that aids rather than thwarts efficient implementations is needed. This parallel programming model must seamlessly transition not only between intra-node CPU-GPU computation, but also inter-node CPU-CPU communication. While there is a number of hybrid MPI+CUDA applications that exist, this is not a robust enough programming model to be used at much large scale system. A higher level, uniform programming model that works on HPC Clusters or Cloud (virtual clusters) cores on traditional Intel architecture chip, cores on GPU could be a solution.

References

[1] NVIDIA, “NVIDIA CUDA Programming Guide”. [Online] available: http://developer.nvidia.com/object/cuda_downloads.html

[2] CUDA C/C++ SDK CODE Samples. Technical Report. NVIDIA Corporation. 2012 http://developer.NVIDIA.com/cuda-cc-sdk-code-samples.

[3] David Patterson “The Top10 innovations in the New NVIDIA Fermi Architecture, and the Top3 Next Challenges” Technical Report. 2009.

[4] Bingsheng He, Wenbin Fang, Qiong Luo, Naga K. Govindaraju, and Tuyong Wang. Mars: A MapReduce Framework on Graphics Processors. PACT 2008.

[5] Andrew Pangborn, Gregor von Laszewski “Scalable Data Clustering using GPUs” Paper Draft. 2009.

[6] The Fermi GPU Architecture：http://developer.nvidia.com/cuda-gpus

[7] Jeffrey Dean and Sanjay Ghemawat. MapReduce: simplified data processing on large clusters. OSDI’04. 2004.

[8] The OpenACC Application Programming Interface: http://www.openacc-standard.org/

[9] Judy Qiu, Seung-Hee. Performance of windows multicore systems on threading and MPI. CCGrid’10. 2010.

[10] http://en.wikipedia.org/wiki/Fuzzy_clustering