A Parallel Implementation of K-Means Clustering on GPUs.

Conference Paper · January 2008

Source: DBLP

Reza Farivar
University of Illinois, Urbana-Champaign
29 PUBLICATIONS **667** CITATIONS

SEE PROFILE

Daniel Rebolledo
University of Bío-Bío
2 PUBLICATIONS **94** CITATIONS

SEE PROFILE

Ellick Chan
Stanford University
29 PUBLICATIONS **468** CITATIONS

SEE PROFILE

Roy Campbell
University of Illinois, Urbana-Champaign
610 PUBLICATIONS **14,699** CITATIONS

SEE PROFILE

A Parallel Implementation of K-Means Clustering on GPUs

Reza Farivar, Daniel Rebolledo, Ellick Chan, Roy Campbell
{farivar2, dreboll2, emchan, rhc} @uiuc.edu

Abstract - Graphics Processing Units (GPU) have recently been the subject of attention in research as an efficient coprocessor for implementing many classes of highly parallel applications. The GPUs design is engineered for graphics applications, where many independent SIMD workloads are simultaneously dispatched to processing elements. While parallelism has been explored in the context of traditional CPU threads and SIMD processing elements, the principles involved in dividing the steps of a parallel algorithm for execution on GPU architectures remains a significant challenge.

In this paper, we introduce a first step towards building an efficient GPU-based parallel implementation of a commonly used clustering algorithm called K-Means on an NVIDIA G80 PCI express graphics board using the CUDA processing extensions. Clustering algorithms are important for search, data mining, spam and intrusion detection applications. Modern desktop machines commonly include desktop search software that can be greatly enhanced by these advances, while low-power machines such as laptops can reduce power consumption by utilizing the video chip for these clustering and indexing operations. Our preliminary results show over a 13x performance improvement compared to a baseline 3 GHz Intel Pentium*(R)* based PC running the same algorithm with an average spec G80 graphics card, the NVIDIA 8600GT. The low cost of these video cards (less than $100 market price as of 2008), and the high performance gains suggest that our approach is both practical and economical for common applications.

I. Introduction

Clustering is a common operation that has many applications for data processing [7]. With the rapid growth of digital information collection and analysis, improvements made to this basic workhorse have very significant implications for the operations of large-scale data processing workflows. Internet search engines periodically crawl the web [6], [8], and index the information for fast search and data retrieval. One of the crucial parameters for these services is the liveness of the service, or the time delay before new material is available in the search index. Since the freshness of results is critical to the success of the service, the performance of the underlying data analysis algorithms are essential. K-means clustering is one of the fundamental building blocks driving many of the algorithms used by these services.

While performance is one factor for the adoption of our approach, efficiency is another leading driver. Data centers consume enormous amounts of energy. Rack space for computers, and inter-node communications present significant spatial, performance and backup electrical generation overhead. If we can offload this processing from the power-hungry CPUs to multiple power efficient GPUs on a single node, significant energy savings can be realized. Likewise, on mobile computers such as notebooks, desktop search, spam filtering [9], antivirus and anomaly-based firewall intrusion detection[10], [11], [12] can be greatly accelerated by using the graphics chip to reduce the energy consumed by the main processor.

CUDA technology gives computationally intensive applications access to the tremendous processing power of the latest GPUs through a C-like programming interface. The GeForce 8 series GPUs have up to 128 processing elements running at 1.5 GHz and up to 1.5GB of on-board memory. CUDA is designed from the ground-up for efficient general purpose parallel computation on GPUs. It uses a C-like programming language and does not require remapping algorithms to graphics concepts[14].

In this paper, we propose an algorithm to perform K-means clustering on a NVIDIA GPU using CUDA. When porting an algorithm to be executed in a parallel fashion on a GPU, it should be noted that the principles involved in dividing the steps of a parallel algorithm for execution on GPU architecture could be different from what might be used on a normal shared memory or cluster machine. Moreover, the architectural style of GPUs impose certain limits and coding strategies to gain significant speedups. In Section III, we discuss specific design decisions to accelerate K-means for the CUDA architecture.

The rest of the paper is organized as follows: First, we present a mathematical description of the K-means algorithm in Section II. Section III discusses parallelization strategies, Section IV discusses our CUDA implementation, and Section V evaluates our implementation using NVIDIA G80 graphics card. Finally, Section VI concludes the paper and shows some possible future venues to explore.

II. K-means clustering

A. Algorithm

K-Means is a commonly used clustering algorithm used for data mining. Clustering is a means of arranging n data points into k clusters where each cluster has maximal similarity as defined by an objective function. Each point may only belong to one cluster, and the union of all clusters contains all n points. We describe Lloyd's fast algorithm for computing an approximate solution to the K-means problem. The algorithm assigns each point to the cluster whose center (also called centroid) is nearest. The center is the average of all the points in the cluster that is, its coordinates are the arithmetic mean for each dimension separately over all the points in the cluster. The algorithm steps are [4]:

1) Choose the number of clusters, k.
2) Randomly generate k clusters and determine the cluster centers, or directly generate k random points as cluster centers.
3) Assign each point to the nearest cluster center.
4) Re-compute the new cluster centers.
5) Repeat the two previous steps until some convergence criterion is met (usually that the assignment hasn't changed).

The main disadvantage of this algorithm is that it does not yield the same result with each run, since the resulting clusters depend on the initial random assignments. It minimizes intra-cluster variance, but does not ensure that the result has a global minimum of variance [5]. If, however, the initial cluster assignments are heuristically chosen to be around the final point, one can expect convergence to the correct values. [2] proposes an optimization by choosing initial centroids close to existing clusters. The first and second phases of the algorithm take $\Theta(k)$ time, while the third phase takes $\Theta(nk)$ to complete. Finally, the fourth phase's execution time is in the order of $\Theta(n+k)$. In a typical application $n > k$, therefore the execution time of the algorithm is bound by the third phase. This observation directed us to concentrate our efforts on parallelizing the third phase of the k-means algorithm.

B. Formal Description

To sort data points $\{x_i \,|\, i = 1 \dots n\} \subseteq \mathbb{R}^d$ into clusters, we need a notion of *clustering* and a criterion for the *quality* of the clustering. Our clustering concept is that of *centroidal voronoi diagrams* and the quality metric is the *squared error*, which we define below.

We say that a family of subsets of $\mathbb{R}^d$ $(S_i)_{1\leq i\leq k}$ is a partition of $\{x_i\}$ if $\{x_i\} = \biguplus_i S_i$ and we call $\mu(S_i) \triangleq \frac{1}{|S_i|}\sum_{x\in S_i} x$ the *centroid* of S_i. We say that a partition (S_i) is a *centroidal voronoi diagram* if

$$\forall i,j \in \llbracket 1,k \rrbracket \forall x \in S_i \quad \|x-\mu(S_i)\| \leq \|x-\mu(S_j)\|$$

In other words, points belong to the cluster whose centroid is the nearest. We use the standard euclidean norm $\|.\|$.

We define the *squared error* of a centroidal voronoi diagram $S = (S_i)$ as the sum over all points of their squared distances to the centroids of the clusters they belong to:

$$\sigma(S) = \sum_{i=1}^{k} \sum_{x\in S_i} \|x-\mu(S_i)\|^2$$

Our goal is therefore to partition the n data points x_i into a centroidal voronoi diagram of minimal squared error. Finding an optimal clustering of this kind is NP-hard [1] and so, many approximation algorithms and heuristics have been proposed.

The k-means algorithm is a widely-used heuristic for its simplicity and speed in practice. It consists of an initialization and a series of 2-phase steps. The algorithm starts by selecting initial candidate centroids: there are different ways of selecting them, one of them is randomly. Then, at every step, the algorithm partitions the set of points into k sets corresponding to each centroid: each point is assigned to the set whose centroid is closest. The algorithm then computes the new candidate centroids as the centroids of the reassigned clusters. The first stage requires $\Theta(nk)$ operations while the second requires $\Theta(n+k)$. The algorithm terminates when the centroids stabilize, or equivalently when the partition becomes stable.

Naturally, the overall complexity depends on the number of rounds. Its theoretical complexity is at least $2^{\Omega(\sqrt{n})}$ for some pathological examples [2], but its convergence is much faster in many practical cases. Further, it does not guarantee an optimal solution. The quality of the output is discussed in [3].

In this paper, we discuss a parallel implementation of the one-dimensional ($d = 1$) k-means algorithm on the CUDA framework. Our approach consists of parallelizing the first phase of each step, where we calculate the nearest centroid to each point. Indeed, since k is generally a few orders of magnitude larger than the amount of cores currently available, the algorithm is bound by this phase and not by the second. To see this, observe that with p cores we can accelerate the first phase to $\Theta(nk/p)$, so the ratio between the two is $\Theta(nk/p(n+k)) \simeq \Theta(k/p)$ (for $n \gg k$). The correctness of the algorithm is guaranteed, to the extent that it produces the same output as the original k-means algorithm. Our implementation performs the same steps as the original k-means algorithm in parallel without changing the semantics or logic of the original implementation.

III. Parallelization of K-Means

When parallelizing an application, a designer must be mindful of the underlying characteristics of the architecture. For example, if the target architecture is an 8-core general purpose computer, it is best to distribute the computation to a number of threads proportional to the number of independent processing elements. A suitable work distribution would then be to divide the work evenly among each of the threads. This can be done using traditional OS threading support such as PThreads in POSIX compliant operating systems (Linux, UNIX), or Java threads. In such architectures, it is best not to assign a thread to process each element, as there may be millions of elements ready to be processed, and the context switching overhead will simply become the dominating bottleneck.

The CUDA model works slightly differently in that the frameworks support workloads that consist of many individual tasks (threads) which are scheduled and managed automatically by the GPU hardware. The dispatching of these threads has very negligible overhead, as it takes only one clock cycle to start a new thread. This suggests that the programmer should break the problem into many logical units without regard to the underlying architecture of the parallel system, thus encouraging the use of a high number of threads to increase the chance that the scheduler will find an optimal schedule for the tasks to execute on the highly parallel SIMD architecture. Such a programming model is best utilized by workloads which have a big number of independent threads, the so-called "embarrassingly parallel" applications.

As previously mentioned, the time-dominant phase of the algorithm is in the assignment of data points to each cluster, which takes $\Theta(nk)$ time. In this phase, the algorithm computes the Euclidean distance of each point to the chosen set of centroids and tries to reassign each point to the nearest cluster. To implement this algorithm on CUDA, we can assign the distance calculation of each data point to a single thread. Therefore each thread will loop over all the initial cluster centroids, calculating its assigned data point's distance from all of them, finding the minimum distance from its data point to a centroid, and become a member of the cluster represented by the centroid. When all threads are finished doing this, the membership of each data point is known, and thus the first phase of the algorithm is finished.

If the number of processing elements were equal to the number of data points, this pass could finish in one step. However, in practice the number of processing elements is limited. Therefore, with p cores we can accelerate the first phase to $\Theta(nk/p)$.

In our test application, the data points are distributed in a 1-dimensional array; therefore the distance calculation can be computed with a single scalar subtraction operation. However, if the data resided in a 2-dimensional domain, it would increase the computational requirements per element without adding any pressure on the memory bandwidth. We will see later that in many cases, this parallel algorithm is limited by memory bandwidth rather than processing power. Therefore, multi-dimensional applications of the K-means algorithm are likely to see larger speedup factors, as they are heavily floating point computation-bound. In fact, moving to higher dimensions makes the problem ideal for GPU architectures, which are better suited for computationally-heavy classes of applications.

For the next phase of the algorithm where centroids of clusters are recalculated from the recently assigned member points, the same approach as the serial algorithm is utilized. There are two reasons for this choice:

- The algorithm for this pass requires intensive data sharing, and creates congestion on the memory subsystem. For example, one possible way of parallelizing this pass is to ask all threads whose data point is a member of a certain cluster to add their values to a certain data structure in memory. This would create traffic on the order of $\Theta(n/k)$ for each memory element.
- As mentioned earlier, even with a serial approach, this pass of the algorithm is not the dominant time-consuming factor of the overall solution.

For the second pass of K-means to become at least as time consuming as the first pass, k and n should be chosen in a way that $\Theta(n/k) \leqslant \Theta(n+k)$. As per section II.B, p and k should be in the same range for this condition. In our implementation, there were about 4000 clusters and 32 streaming multiprocessors, therefore it was not worth the effort to parallelize this step.

IV. CUDA IMPLEMENTATION

There are two data structures used in the algorithm:

```
typedef struct {
        unsigned long value;
        unsigned long centroid;
}valpoint;

typedef struct {
        unsigned long value;
        unsigned long numMembers;
}centroid;
```

We implement the CUDA-accelerated k-means algorithm in three distinct stages of operation. The first stage initializes the CUDA hardware, allocates the appropriate host and device memory storage areas, estimates the initial set of centroids and loads the data set into the on-board memory of the graphics card. Note that the usage of CUT_SAFE_CALL and CUDA_SAFE_CALL macros are for error checking purposes.

```
const unsigned long numElements = 1024*1024;
const int numClusters = numElements / 256;
unsigned int timer = 0, mem_size;
valpoint *d_idata, *h_odata;
centroid* d_centroids;

CUT_DEVICE_INIT();
CUT_SAFE_CALL( cutCreateTimer( &timer));
mem_size = numElements * sizeof(valpoint);

// allocate host memory
valpoint* h_idata = (valpoint*) malloc( mem_size);
centroid* h_centroids = (centroid*)
  malloc (numClusters * sizeof(centroid));

// initialize the memory
for( unsigned long i = 0; i < numElements; ++i){
  h_idata[i].value = i + 0xc0000000 ;
  if (i % 128 == 0) {
    //estimate the centroids
    h_centroids[i/256].value = i + 0xc0000000;
    h_centroids[i/256].numMembers = 0;
  }
}

CUT_SAFE_CALL( cutStartTimer( timer));

// allocate device memory for data points
CUDA_SAFE_CALL(
   cudaMalloc( (void**) &d_idata, mem_size));

// copy data points to device
CUDA_SAFE_CALL( cudaMemcpy( d_idata,
    h_idata, mem_size, cudaMemcpyHostToDevice) );

// copy centroids to device
CUDA_SAFE_CALL( cudaMemcpyToSymbol(
    constData, h_centroids,
     sizeof(centroid)* numClusters));
```

Then, the host code will launch all the threads. The threads will be dispatched automatically by the hardware.

```
// setup execution parameters
dim3  grid( numElements / 512, 1); //2048 blocks.

// numElements can be up to 32 Mega samples
dim3  threads( 512, 1, 1); // each block having 512 threads.

printf("Main thread: about to dispatch kernel...\n");

// execute the kernel
testKernel<<< grid, threads >>>(
    d_idata, d_centroids, numClusters);
```

The call to the testKernel function blocks on the host; it will block until all the threads are dispatched and completed. The host code will then check the correct execution of the CUDA kernel, copy the results from the device memory into a host buffer, and calculate the execution time.

```
// check if kernel execution generated and error
CUT_CHECK_ERROR("Kernel execution failed");

//allocate mem for the result on host side
h_odata = (valpoint*) malloc( mem_size);

// copy result from device to host
CUDA_SAFE_CALL( cudaMemcpy( h_odata,
    d_idata, mem_size, cudaMemcpyDeviceToHost) );
CUT_SAFE_CALL( cutStopTimer( timer));
printf( "Time: \%f(ms)\n", cutGetTimerValue( timer));
CUT_SAFE_CALL( cutDeleteTimer( timer));
```

Finally, the host will free up the allocated memory and exit.

```
// cleanup memory
free( h_idata);
free( h_odata);
CUDA_SAFE_CALL(cudaFree(d_idata));
CUDA_SAFE_CALL(cudaFree(d_centroids));
CUT_EXIT(argc, argv);
```

The second part, which is the workhorse of the program, is the kernel running on the GPU device. Each thread will process a single data point, and compute the distance between the point and each centroid. While executing this loop, it will also track the minimum distance to its nearest centroid, and when the loop is completed, the results will be stored in the device's memory.

```
__constant__ centroid constData[4096];
__global__ void
testKernel( valpoint* g_idata, centroid*
      g_centroids, int numClusters) {
  unsigned long valindex = blockIdx.x * 512
      + threadIdx.x ;
  int k, myCentroid;
  unsigned long minDistance;
  minDistance = 0xFFFFFFFF;
  for (k = 0; k<numClusters; k++)
    if (abs((long)(g_idata[valindex].value -
        constData/*g_centroids*/[k].value))
        <minDistance) {
      minDistance = abs((long)(
          g_idata[valindex].value -
              constData[k].value));
      myCentroid = k;
    }

  g_idata[valindex].centroid =
          __constData[myCentroid].value;
  __syncthreads();
}
```

One important aspect of this code is the use of device constant memory for storage of the centroid data. The constant memory is a cacheable part of the device memory. The current G80 cards have 8KB of such memory per thread block. Since each thread loops on all of these centroids, fast access to these variables is essential for the success of this implementation. The effect of this choice was of such importance that without using it (storing the centroids in the device global memory), the performance would have been an order of magnitude slower than the present implementation.

The third part of the program relabels points to the nearest centroid, and computes the next centroid estimation. Since this piece of code executes serially in the host, it is omitted from inclusion in this paper due to space constraints. These steps repeat until the convergence condition is reached. For our algorithm, we are content when the number of reassignments falls under a certain accuracy threshold.

For reference, here is the serial baseline code which we compare our results against:

```
int k, myCentroid;
unsigned long minDistance;

for (long valindex = 0; valindex<1024*1024;
     valindex++){
  minDistance = 0xFFFFFFFF;
  for (k = 0; k<numClusters; k++)
  if (abs((long)(h_idata[valindex].value -
      h_centroids[k].value))<minDistance){
    minDistance = abs((long)(   _
     h_idata[valindex].value-h_centroids[k].value));
    myCentroid = k;
  }

h_idata[valindex].centroid =
    h_centroids[myCentroid].value;
```

It is evident from this piece of code that it takes $\Theta(nk)$ for completion.

V. Experimental Results

We have implemented our parallel CUDA K-means algorithm on an NVIDIA GeForce 8600 PCI express graphics card with 256 MB of dynamic RAM, 32 streaming processors and 8 KB of cache per streaming processor. Our data set consists of a one dimensional array of unsigned long integers, which are 4 bytes in size per point. There are 1 million elements in the array to be clustered, and the number of clusters is chosen to be 4000. Using these parameters, we experienced a 13x speed improvement over a baseline application running on a 2 Ghz host machine. A summary of these results is presented in Table 1.

For an objective comparison, we consider a current top of the line NVIDIA GeForce 8800 GTX Ultra which has 128 streaming processors and runs at a 27% faster clock rate. We project that this card should be able to run the same workload about 5 times faster, or have a 68X performance increase compared to the baseline PC implementation. However, due to our lack of access to one of these cards, these numbers cannot be verified, although we do not see any reason to doubt its projected performance.

TABLE I
RESULTS OF PARALLEL K-MEANS ON CUDA

Platform	Time (s)	Performance Increase
Intel Pentium D, 3 Ghz	9.830	1 X
NVIDIA 8600 GT	0.724	13.57 X
NVIDIA 8800 Ultra GTX	0.144	68 X

VI. Conclusions and Future Work

We have shown how to parallelize an elementary data processing operation used by many applications on a highly parallel graphics processing architecture. Our results suggest that computationally-bound applications have much to gain by using the idle resources offered by the system through the CUDA architecture. As performance and energy consumption become a prime concern for computer architectures, we are bound to see more applications of off-chip parallel computing in every computing domain from large-scale distributed systems to portable computers.

VII. Acknowledgements

We would like to thank the anonymous reviewers for their invaluable constructive feedback. David Luebke, Michael Garland, and Kevin Skadron provided an excellent tutorial at ASPLOS this year, which helped inspire many of the ideas underlying this research. Jason Lowe provided insightful feedback and guidance about programming graphics processors. This research was supported by grants from Motorola and the Siebel foundation. Our experiments were performed on hardware generous donated by NVIDIA.

References

[1] P. Drineas, A. Frieze, R. Kannan, S. Vempala, and V. Vinay, ”Clustering in large graphs and matrices,” in SODA ’99: Proceedings of the tenth annual ACM-SIAM symposium on Discrete algorithms. Philadelphia, PA, USA: Society for Industrial and Applied Mathematics, 1999, pp. 291299.

[2] D. Arthur and S. Vassilvitskii, ”How slow is the k-means method?” in SCG ’06: Proceedings of the twenty-second annual symposium on Computational geometry. New York, NY, USA: ACM, 2006, pp. 144153.

[3] M. Meila, ”The uniqueness of a good optimum for k-means,” in ICML ’06: Proceedings of the 23rd international conference on Machine learning. New York, NY, USA: ACM, 2006, pp. 625632

[4] J. B. MacQueen (1967): ”Some Methods for classification and Analysis of Multivariate Observations”. Proceedings of 5-th Berkeley Symposium on Mathematical Statistics and Probability. Berkeley, University of California Press, 1:281-297

[5] http://en.wikipedia.org/wiki/Cluster_analysis

[6] O. Zamir and O. Etzioni. Grouper: a dynamic clustering interface to Web search results. In Proceedings of the Eighth International World Wide Web Conference, Toronto, Canada, May 1999. Page 8.

[7] Zhexue Huang, ”A Fast Clustering Algorithm to Cluster Very Large Categorical Data Sets in Data Mining.” Workshop on Research Issues on Data Mining and Knowledge Discovery, 1997.

[8] Zeng, H., He, Q., Chen, Z., Ma, W., and Ma, J. 2004. Learning to cluster web search results. In Proceedings of the 27th Annual international ACM SIGIR Conference on Research and Development in information Retrieval (Sheffield, United Kingdom, July 25 - 29, 2004).

[9] Minoru Sasaki, Hiroyuki Shinnou, ”Spam Detection Using Text Clustering” International Conference on Cyberworlds (CW’05), 2005.

[10] Sequeira, K. and Zaki, M. 2002. ADMIT: anomaly-based data mining for intrusions. In Proceedings of the Eighth ACM SIGKDD international Conference on Knowledge Discovery and Data Mining (Edmonton, Alberta, Canada, July 23 - 26, 2002). KDD ’02. ACM, New York, NY, 386-395.

[11] K. Alsabti, S. Ranka, V. Singh. An efficient K-means Clustering Algorithm. In 11th International Parallel Processing Symposium, 1998.

[12] C. Warrender, S. Forrest, B. Pearlmutter. Detecting intrusions using system calls: alternative data models. In IEEE Symposium on Security and Privacy, 1999.

[13] Lu, C.-T.; Boedihardjo, A.P.; Manalwar, P., ”Exploiting efficient data mining techniques to enhance intrusion detection systems,” Information Reuse and Integration, Conf, 2005. IRI -2005 IEEE International Conference on. , vol., no., pp. 512-517, 15-17 Aug. 2005.

[14] Massimiliano Fatica, Won-Ki Jeong,”Accelerating MATLAB with CUDA”. the High Performance Embedded Computing (HPEC) Workshop 2007