.

CUDA Accelerated Entropy Constrained Vector Quantization and Multiple K-Means

John Ashley
NVIDIA UK Ltd.
London, UK
jashley@nvidia.com

Amy J. Braverman
Jet Propulsion Laboratory
Pasadena, CA, USA
amy.j.braverman@jpl.nasa.gov

***Abstract*—Multi-trial sampled K-means performance and scalability is studied as a stepping stone towards a Graphical Processing Unit implementation of Entropy Constrained Vector Quantization for interactive data compression. Basic parallelization strategies and data layout impacts are explored with K-means. The K-means implementation is extended to Entropy Constrained Vector Quantization, and additional tuning specific to the anticipated use case is performed. The results obtained are sufficiently promising that this will in the next phase be applied to the interactive exploration and visualization of very large satellite datasets.**

Keywords- K-Mean; Entropy Constrained Vector Quantization; Graphical Processing Unit; CUDA.

I. INTRODUCTION

This is a work in progress, and is currently in an early stage. The ultimate goal is to apply k-means or Entropy Constrained Vector Quantization (ECVQ) to interactive data compression as a component in a data exploration and visualization solution. Initially, this would support exploration of the Multi-angle Imaging SpectroRadiometer (MISR) Level 3 aerosol data sets. This phase of the work is intended to create an understanding of the scalability and performance of simple Graphical Processing Unit (GPU) accelerated versions of ECVQ and its algorithmically similar but simpler cousin, k-means clustering. K-means and ECVQ offer the potential to provide information preserving data compression on large datasets. Further research will then apply data visualization and exploration techniques on the reduced datasets.

In this paper, we first discuss k-means and ECVQ, and then proceed to an overview of GPUs and CUDA programming. The structure of the research codes, including the primary parallelization and tuning options explored, are presented. A review of some of the prior art is followed by relative performance results. Conclusions are drawn and the next steps in the work are briefly discussed.

II. K-MEANS AND ECVQ

The k-means algorithm is familiar in a wide variety of contexts. In this paper, a relatively simple version of the algorithm is adopted using sampling and multiple trials to extend the scalability to large datasets. Interested readers can consult a variety of texts and surveys of the field [1].

ECVQ can be viewed as a compression or clustering technique similar to k-means but with information-theoretic penalties applied to the distance function used to assign vectors to cluster centroids [7]. Specifically, it adds a multiple of the absolute value of the log of the fraction of measurements in a cluster to the L2 distance measure, which adds computational complexity but minimal additional data movement to the k-means algorithm. It is useful in a variety of contexts; this work was motivated by the use of ECVQ in the production of certain compressed NASA datasets [3].

Ignoring initialization and iteration limits, the basic algorithm to implement k-means with a number of training sets, each of a certain sample size, and a testing set is shown in Fig. 1.

```
For each training-set
    Repeat until labels are stable
        For each sample assign to nearest k
        For each k, calculate new centroid
For each testing set's final centroids
    For each test case assign to nearest k
    For each k calculate a quality measure
    Aggregate quality measures for the test set
Select best set of k
```

Figure 1: Algorithm Outline.

In Fig. 1, "nearest" is the L2 norm in the case of k-means or the L2 norm augmented by the information theoretic penalty function in the case of ECVQ.

There are several levels of parallelism available in the algorithm, as evidenced at the highest level by the "for each" lines of the algorithm. There are also serial data dependencies between the steps in the repeat loop and before the selection of the best k at the end.

III. GPUS AND CUDA

The General Purpose Graphics Processing Unit (GPGPU or usually just GPU) is the evolution of massively parallel hardware graphics accelerators whose base data types are the pixel, triangle, and image and whose basic functions were rasterization and shading. The programming model for these accelerators and their performance has evolved significantly, to the point where NVIDIA's CUDA extensions to C++ and Fortran have been used in supercomputers and industrial

 ISBN: 978-1-61208-365-0

.

systems since 2006. There are numerous resources for learning more about GPU programming and software that is already GPU accelerated [4].

A. Logical Programming Model-- CUDA

A function to be executed on the GPU is called a *kernel*. These kernels are logically composed of many lightweight *threads* (frequently thousands to millions) which are launched as a *grid*. These threads are grouped together into *blocks*. The same kernel code is executed by every thread, but threads have unique thread and block indices which means each thread can operate on different data and follow different paths through the code. Threads within a block can synchronize with each other and can use a block-wide very fast pool of *shared memory* for coordination.

All threads in all blocks have access to fast global device memory on the accelerator. There are also ways to access the normal (relatively slow) memory of the host CPU system that are not important in the context of this problem.

As a programmer, the CUDA logical model guarantees that all the threads within a block will be assigned to an active hardware Streaming Multiprocessor with multiple cores at the same time. It does not guarantee which blocks will execute in what order.

B. Physical Execution Model

Physically, the GPU uses a hardware block and thread scheduler to assign blocks to hardware units called *Streaming Multiprocessors* (SMs). Depending on the hardware generation and model of the chip, the GPU may have one or more (14-16 for high end compute cards on the Fermi and recent Kepler generation chips) SMs each. In the latest Kepler generation GPUs, each of these SMs has 192 *cores* that perform the actual computing.

Because of the very large number of cores, and the way they are designed, GPUs are also sometimes characterized as being "throughput oriented" as opposed to traditional serial CPU cores which are "latency oriented". A throughput oriented GPU's thousands of cores are designed to enable very large numbers of concurrent threads to execute (or be ready to execute) simultaneously, thus hiding any latency a single set of threads might experience with productive work for other threads. A CPU's dozen or so cores, in contrast, use much larger caches, branch prediction, out of order execution, and other optimizations to prevent latency from causing a very costly context switch of a very small number of threads.

GPU cores within an SM are grouped into sets of 32 cores. Threads from a block are assigned to cores in groups of 32 called *warps*. Threads within a warp effectively share a program counter and can follow different branches of code using hardware managed masking. Note that when a warp has threads which need to follow multiple paths of a branch construct, the warp will effectively serialize execution of each portion of the branch – this is commonly referred to as branch divergence; the GPU manages all this in hardware whereas on the CPU, the vector units face the same issue but have to manage it in software using explicit mask registers.

Blocks which have more than 32 threads require more than one warp of resources in the SM to execute. SMs have varying capacities in regards to resources available but generally are most efficient when running many blocks and warps on a SM simultaneously; this lets the SM hide the latency caused by various memory accesses and other activities by always having one or more warps ready and waiting to execute.

The memory hierarchy on the GPU, in order of increasing size and decreasing bandwidth, consists of thread private *registers*, block private *shared memory*, shared *L2 cache*, and *global device memory*. Peak bandwidth for global device memory on recent high end compute GPUs is 288 GB/second, significantly higher than traditional CPU memory.

The performance numbers in this research were developed on an NVIDIA K40 GPU with factory (base) settings, using the CUDA 5.5 toolkit on Linux. The conclusions should be broadly applicable to other hardware when scaled appropriately for compute and bandwidth.

IV. CODE STRUCTURE

In the particular implementation strategy we use, which is modeled after the processing used to create ECVQ compressed datasets for the MISR and AIRS satellite data, some number of trials are performed, each using a set of training vectors sampled from the overall population. These are clustered to produce candidate cluster centroids. Candidate cluster centroids are evaluated against a larger testing set and the final set of cluster centroids are selected.

The focus of this investigation is on the performance of the three most important and time consuming components of the solution – steps that will be referred to as *Label*, *Calculate*, and *Score*.

A. Parallelization Strategy

Label assigns training vectors to cluster centroids, and determines if the solution has converged. Each trial is performed by a single thread block, and each thread within that block will work on one training vector, evaluating the distance from each centroid, and performing the assignment. In cases where there are more training vectors than there are threads per block, the threads will stride over the set of vectors until each vector has been updated.

Calculate updates the cluster centroids based on the training vectors assigned to that cluster centroid. Each trial is performed by a single thread block, and each thread within that block will work on one centroid, which it will update. In cases where there are more centroids than there are threads per block, the threads will stride over the set of centroids until each centroid has been updated.

Score will evaluate every testing vector against the set of centroids from each trial, and produced the per cluster dispersion. Each thread block will evaluate the set of testing vectors against the centroids from a single trial. The threads within the block will each stride over the testing vectors until all vectors have been evaluated.

 ISBN: 978-1-61208-365-0

.

B. K-Means Codes

A basic, "naïve" version of k-means along the lines of the code described above was created. This version is referred to as *K1*. This code attempts to be as simple as possible, and as straightforward, to reflect code that could be delivered to, maintained by, and possibly extended by a typical scientific programmer, if such a person actually exists.

One common performance issue for GPU codes is related to memory access patterns – memory requests from within a warp can be combined if they are for adjacent memory locations. This can drastically increase the efficiency of the memory subsystem and have substantial performance impacts on the code. *K1* uses the first layout in Fig. 2, below. A second version of the code, using the second layout below, called *K2*, was also tested. This code differs only in the data layout used, and while perhaps less "natural" to a scientific programmer, would not be difficult to maintain or extend.

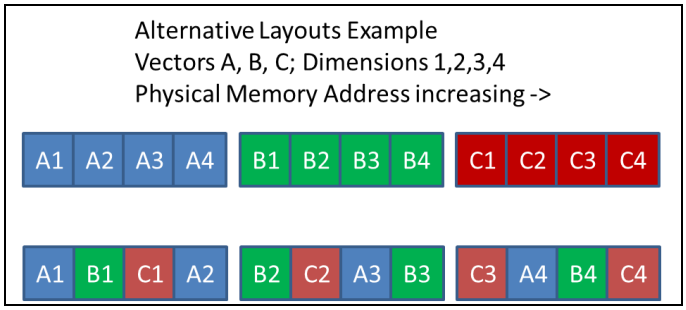

Figure 2: Memory Layouts.

C. ECVQ Code

One of the versions of the k-means code (*K2*) was converted into an ECVQ code via the addition of the information theoretic penalty functions and associated bookkeeping. This version is called *E2*.

A second version of the ECVQ code was also developed that begins to explore the performance impacts of tuning for a specific final problem size. This version is called *E3*, and the very minor tuning specializations it incorporates are discussed along with the performance results.

V. PRIOR ART

The motivation for the current work is based firmly in a selection of the prior art. A technique for displaying large numbers of weighted clusters of high dimensional data vectors applied to the AIRS cluster compressed data appears promising for application to the MISR dataset as well [6]. Current processing of the MISR dataset using ECVQ faces the same operational limits as the AIRS data, and can take no more than 2 hours per 5-days of input data per 5^o x 5^o earth grid cell but provides excellent compression and information retention [2,7]. While this performance is acceptable for offline data processing, allowing interactive use in visualization will require higher performance.

K-means is closely related to ECVQ [2] and is well studied on the GPU [5,9,10]. Most prior work has focused on the relative acceleration with respect to various CPU only codes. In this work, we explore only the actual performance of the GPU version of the code, although absolute performance numbers for earlier generations of GPUs are available for some problems [10].

The use of ECVQ in data compression is well established [2]. There is a small body of work on the use of ECVQ in satellite data applications [3,7,8]. To date there appears to be little if any published work on using GPUs to accelerate computation of ECVQ.

VI. PERFORMANCE RESULTS

This research is currently using parameterized random data generation for test purposes. This allows exploration of the parameter space for performance characterization and allows the code to be self-contained. The first actual deployment target of this research will be a MISR aerosol dataset with 8 data dimensions plus spatial and time metadata. Current NASA processing on this dataset uses 200 trials each with 200 training vectors, with a maximum K value of 100. Our basic test case is deliberately chosen to be similar to these values. Other envisioned uses would be to apply the technique to NASA AIRS datasets with 32 data dimensions plus meta-data; this defines the size of one of the larger test cases.

A. Test Cases

We define a set of parameters which will be varied to examine the performance of the code over a small range of values as defined in Table 1.

TABLE 1. TEST SCALE PARAMETERS.

Variable	Description
N	Data dimensions
K	Maximum number of clusters
T	Trials
S	Training Vectors (Sample Size)
V	Testing Vectors

We use these parameters to both define test cases and set expectations on the performance of each problem phase (Label, Calculate, Score) for each test. The phases are sensitive to the Test Scale Parameters as shown in Table 2.

TABLE 2. PERFORMANCE SENSITIVITY EXPECTATIONS.

Phase/Value	N	K	T	S	V
Label	X	X	X	X	
Calculate	X	X	X	X	
Score	X	X	X		X

 ISBN: 978-1-61208-365-0

.

Given that there is going to be some overhead in each phase, we expect good code to have better than linear response to an increase in the appropriate parameters, unless we exceed the capacity of some resource (such as cache) that was not exhausted at the lower scale. More specifically, if we increase N by a factor of 4, we would expect all phases to show execution time increases of something somewhat less than four.

Test cases were defined with Test Case 1 as a "base" case, with each of cases 2-6 testing the scaling of a different parameter. Test Case 7 shows the scaling on a larger problem. Details of each test case are in Table 3.

TABLE 3. TEST CASES.

Test Case	N	K	T	S	V
1	8	128	256	256	100,000
2	8	256	256	256	100,000
3	8	128	512	256	100,000
4	8	128	256	512	100,000
5	8	128	256	256	200,000
6	32	128	256	256	100,000
7	8	128	1024	1024	1,000,000

B. *Scaling with K1 Across Cases*

The first round of testing verified the expected scaling behavior of the K1 "naïve" version of the code across the various test cases. The actual timing results for 100 iterations of Label and Calculate, and one final round of Scoring, on Test Case 1, are in Table 4.

TABLE 4. MEAN EXECUTION TIMES.

Execution	100xLabel	100xCalculate	Score
Time	104.1ms	129.3ms	2.58s

The Score phase is far longer than the other phases, as would be expected, as V >> S. All performance charts following this express performance relative to the values in Table 4. For the K1 code, performance is broadly in line with expectations, as can be seen from Fig. 3.

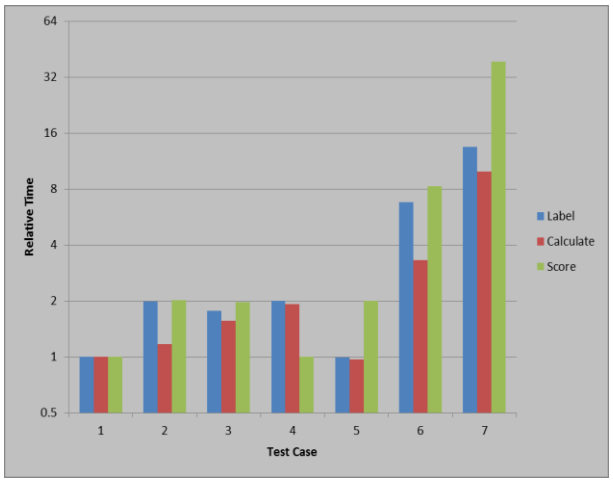

Figure 3. Relative performance of K1 across test cases.

We see roughly expected scaling except that Calculate doesn't see as much impact in Test Case 2 as might be expected, since it should have runtime proportional to the Label step. The actual code in K1 is quite naïve, and launches more threads per trial than are needed for K=128; specifically, it launches 256 threads. In cases with less than 256 clusters, some threads start and exit, doing no effective work. In Test Case 2, these threads work and then exit, and so have minimal impact on the runtime. The Label and Score steps, in Test Case 6, seem to be running longer than expected. This appears to be because more registers spill to cache with D=32 than with D=8, causing some additional impact.

C. *Relative Performance of Codes in Test 7*

In Test Case 7, we would expect to see any impact from improvements in code or increases in the complexity of calculations to be magnified by the scale of the problem. To that end, codes K2 and E2 were tested against Test Case 7, with the results shown in Fig. 4.

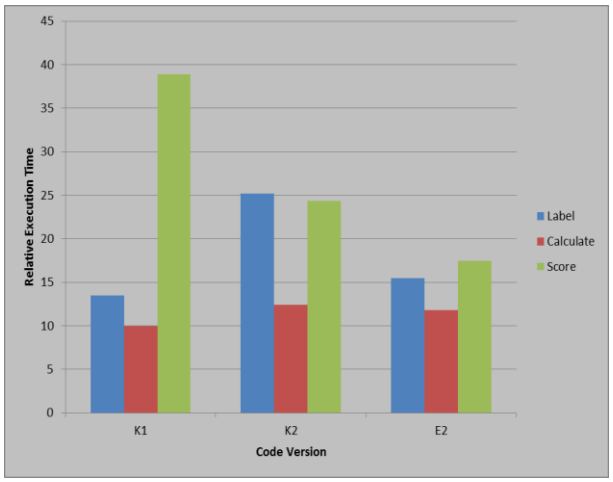

Figure 4. Relative Performance of Code Versions.

 ISBN: 978-1-61208-365-0

.

As we discussed earlier in sections IV.B and IV.C, the *K2* version of the K-means code uses a more GPU performance friendly data layout than *K1*. This code is actually slower than the naïve code for Label and Calculate, which appears to be due to cache and compiler generated code optimization interactions, but the additional efficiency of the memory layout pays significant dividends on the much longer Score phase. Because the overall runtime is significantly reduced, the code for the first ECVQ version (*E2*) was based on the *K2* code and data layout.

D. Initial Exploration of Problem Specialization

The code in E2 is already almost fast enough for interactive use on real problems, but it would be better if it were even faster. Earlier we admitted that unlike typical "finished" work, this code was relatively general and made relatively few concessions to what is sometimes called "ninja tuning". We did, however, explore a very simple specialization to the problem using a very simple performance enhancer, namely, a "#pragma unroll" directive. This directive, embedded in a comment before a loop, instructs the compiler to rewrite the loop to have more copies of the code inside it and make correspondingly less trips through the loop. Our code *E3* is optimized for values of D that are a multiple of 8, with one inner loop in each phase unrolled by a factor of 8. The performance impact on Test Case 1, the test case closest to our envisaged real world workload, is significant, as can be seen in Fig. 5.

The Score phase, in particular, benefited greatly from this very minor optimization.

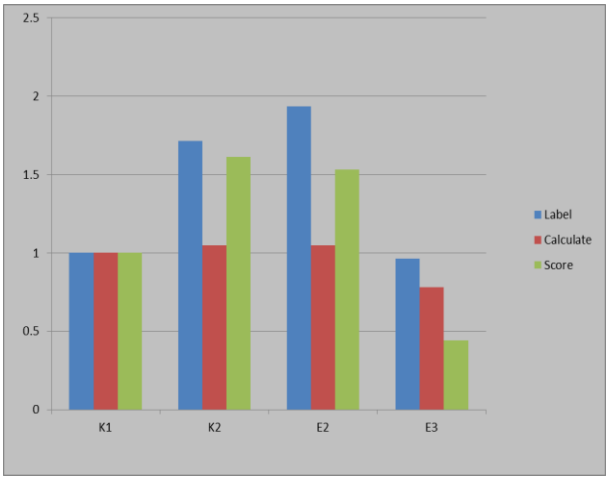

Figure 5. Relative performance including tuned code.

Additional, possibly more code invasive tuning should allow for further improvements.

VII. Conclusions and Future Work

We set out to explore the suitability of K-means or ECVQ for interactive data compression in support of data exploration of a specific NASA dataset. Scalability testing on synthetic data showed that the code has relatively predictable and linearly bounded performance. Conversion of the compression from K-means to ECVQ was shown to have minimal impact to overall performance.

Finally, with even very simple performance tuning that specializes the solution to the anticipated problem, significant performance gains were observed, with overall execution times approach one second, which would be sufficient for many interactive uses.

Next steps for this research include exploring some additional tuning to reflect the anticipated problem sizes and then to do performance and quality testing with real data. These next steps will then be incorporated into a data exploration environment for the MISR data.

Acknowledgment

John Ashley would like to thank Dr. Daniel Carr at George Mason University, for his support and insights during this work. John Ashley would also like to thank my employer, NVIDIA, for their support.

References

[1] R. Xu and D. Wunsch, "Clustering," IEEE Press Series on Computational Intelligence, 2009.

[2] P. A. Chou, T. Lookabaugh, and R. M. Gray, "Entropy-constrained vector quantization," IEEE Trans. on Acoustics, Speech, and Signal Processing, Jan 1989, Vol 37, Issue 1, 1989, pp.31- 42.

[3] A. J. Braverman, E. J. Fetzer, B. H. Kahn, E. M. Manning, R. B. Oliphant, and J. P. Teixeira, "Massive dataset analysis for NASA's Atmospheric Infrared Sounder," Technometrics, 2012 [b], 54:1, pp. 1-15.

[4] NVIDIA Corporation, "http://docs.nvidia.com/cuda," 2014.03.12, unpublished.

[5] B. Catanzaro, "https://github.com/BryanCatanzaro/kmeans," 2014.03.12, unpublished.

[6] J. Ashley, "Techniques for exploring cluster compressed geospatial-temporal satellite datasets," Ph.D. Dissertation, George Mason University, 2013.

[7] A. Braverman, ''Compressing massive geophysical datasets using vector quantization," Journal of Computational and Graphical Statistics, 2002, 11, pp. 44–62.

[8] A. Braverman, E. Fetzer, A. Eldering, S. Nittel, and K. Leung, ''Semi-streaming quantization for remote sensing data," Journal of Computational and Graphical Statistics, 2003, 12 (4), pp. 759–780.

[9] S. A. Arul Shalom, M. Dash, and M. Tue, "Efficient k-means clustering using accelerated graphics processors," Data Warehousing and Knowledge Discovery Lecture Notes in Computer Science Volume 5182, 2008, pp. 166-175.

[10] K. J. Kohlhoff, V. S. Pande, and R. B. Altman, "K-means for parallel architectures using all-prefix-sum sorting and updating steps," IEEE Transactions on Parallel and Distributed Systems, Aug 2013, Vol 24, No. 8, pp. 1602-1612.

 ISBN: 978-1-61208-365-0