Computational Statistics via GPU

Andrew Harp
The University of Texas
aharp@cs.utexas.edu

1. Introduction

Many tasks in data mining and statistics are inherently parallel. While modern commodity desktop processors typically contain two to four cores, graphics processing units (GPUs) can contain hundreds. However, as opposed to as opposed to CPUs which offer more flexibility in program control, GPUs are specialized to perform massively parallel operations synchronously on multiple data. To obtain the maximum speedup from a GPU, one cannot simply write a multithreaded program as for a CPU. Care must be taken to coalesce memory requests between neighboring threads and minimize the branching factor, as threads on the same multiprocessor operate in lockstep.

Many modern applications are increasingly turning to GPUs for specialized computations, such as [3]. Physics can now be processed efficiently on graphics cards, for example. For my project, I propose to adapt a few statistical techniques for running on GPUs. Similar to creating Mex modules for Matlab to reap substantial speedups in key functions, I can in turn embed GPU code in my C++ function.

2. Project Goals

For a first task, I will write a GPU version of IQAgent. Immediately, sampling massive amounts of numbers from the distribution is a a good candidate for optimization. Parallelizing the creation of the distribution could be trickier perhaps impossible–but his would ostensibly be the part called most often over time.

For the second task, I chose to implement a GPU-driven version of Expectation Maximization for Gaussian Mixture Models [2]. EM with Gaussian Mixture Models is a generalized clustering approach that continually reestimates the model in terms of itself and the data until convergence is reached.

I will demonstrate how the graphics card scales to larger data sizes versus the CPU. My expectation will be that because of memory latency, small amounts of data can be processed faster on the CPU, but as the amount of data grows, the GPU will outpace the CPU and continue to enlarge its lead. The rate at which this occurs is a question I investigate.

3. Technical Background

NVidia's CUDA is an SDK toolkit which allows special C functions to run directly on the multiprocessors of their modern graphics cards. Previously, some scientific computing had been done with GPUs, but much of this was done by tricking the GPU into doing the computations required with graphical operations. OpenCL is another competing general purpose GPU computing standard, but it has not yet reached the same level of maturity.

With CUDA, one writes a host function and a kernel (device) function. The host is considered to be the CPU side of the operation. It is responsible for copying the data to the graphics card, calling the kernel function after configuring the execution parameters, and copying the results back.

Kernel functions run as a grid of blocks, which are in turn composed of threads. Every thread in a block runs on the same GPU multiprocessor, and runs in lockstep with the other threads in that block. Blocks do not communicate with other blocks, but threads in a block can share fast local cache memory and use synchronization functions if they need to wait for other threads to finish.

Since the local data cache size is limited, and most data will reside in the shared global GPU memory. When calling a CUDA kernel, parameters are passed by value and placed in the local cache, but array pointers will reference the shared global memory, which has a significant penalty for reading.

When data is required from the global cache, it is best that adjacent threads in the block request adjacent memory locations. Because threads run simultaneously in lockstep, there will be only be a single delay as the individual memory requests will be coalesced into a larger request. It is normal to see kernels written such that threads will request data for other threads to process, in order to rearrange the memory requests.

Data must be placed back into global memory at a preallocated position by the end of the thread's execution, so that the result may be copied back off the card by the host.

3.1. IQagent

So far I have been able to integrate IQagent.h into a C++ program which adds a configurable number of random data points to the population. The program also generates a variable number of random numbers 0 to 1. It then uses two methods of sampling from the population at these randomly generated points.

The first method is a simple for loop, which sequentially calls *agent.report()* for every number in the random sample using standard CPU-based C++. This method is used to provide a baseline performance metric.

The second is a custom function which copies the data of the IQAgent struct to the graphics card via CUDA API calls. It then calls a CUDA kernel, which maps every thread in every block of the execution to computing a single sample from the population at the given values. The actual code that runs on the GPU that actually computes a given value is identical to *IQagent.report*, however it is the ordering of the points picked that make it run efficiently in parallel.

Once both methods have returned, their results are compared to each other as a sanity check to make sure the GPU-based method is getting the correct answer.

3.1.1 Results

The results are in line with what I had expected. For small numbers of samples, the standard CPU based sampling method is faster. However, the GPU based method begins to dominate at around the order of 10,000,000 samples. By 200,000,000 samples, on a Pentium 3.0GHZ machine, the breakdown is thus:

Sampling from random points (CPU)... 23.88s

Sampling from random points (GPU)... 5.282s
Copying to GPU... 25ms
Computing... 900ms
Copying results back... 2449ms

There is some missing time in the breakdown because I did not specifically track the time spent allocating and deallocating memory on the GPU (but it is included in the overall duration of 5.282s). All verifications passed on the resulting sampled population. I could not run larger tests because of memory limitations on my GPU.

It can be seen that the biggest bottleneck is copying the results back from the card. This makes sense because generally data goes from CPU to GPU to display, and not GPU to CPU. When you compare the actual computation times of 23.88s and 0.9s, you see the more than 20x speedup expected. For jobs where a smaller amount of data needs to be transported back to the CPU, the GPU based method would be advantageous much earlier.

So clearly, if you regularly need to sample ridiculous amounts of data from a distribution using IQagent, GPU acceleration can pay off. As this was a trivial problem with dubious real-world payoff potential, I did not investigate further, so that I could turn my attention to EM with GMMs.

3.2. Expectation Maximization with Gaussian Mixture Models

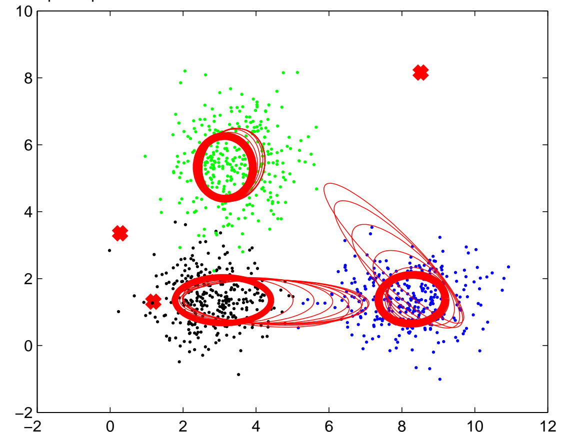

Figure 1. Example of guessed means and sigmas converging on actual parameters over multiple iterations. Red Xs mark initial guesses.

While initially I had planned to construct a particle filter to find the best allocation of Gaussians, on closer analysis I determined that simple EM was already an ideal candidate for running on the GPU.

Expectation Maximization has two parts. The expectation step, where the data is expressed in terms of the model, and the maximization step, where the model is updated to maximize the likelihood of the data.

With Gaussian Mixture Models, every particle is a assigned a likelihood of being generated by one of k clusters, each with its own mean *mu* and covariance *sigma*.

3.2.1 Implementation

My program's interface is a Matlab Mex function derived from an an example in the course lecture notes. This Mex example code provided a means to integrate NR3's gausmixmod.h into Matlab. I added three additional options to the Mex function.

1. 'togpu': Copies data from the CPU-based GMM to the GPU.

2. 'fromgpu': Copies data from the GPU back into the CPU-based GMM.

3. 'gstep': Runs the specified number of iterations with the GPU kernel on the GPU based data.

In this way it is easy to compare the GPU and the CPU implementations side by side. All other external functionality is the same, with the exception that 'gstep' does not return a log-likelihood. This is because log-likelihood was not necessary for my experiment, since I always run a fixed number of EM steps. However, it would not be very difficult to implement, requiring only a memory fetch after the expectation step and an additional parameter.

3.2.2 Kernel Construction

The bulk of my work on this project, apart from familiarizing myself with CUDA and getting the appropriate build environment set up, has gone into converting gausmixmod's expectation and the maximization steps from standard C++ to a kernel capable of efficiently running on a GPU. While the functionality is identical, the resulting code is hardly recognizable when compared to the original source.

First I needed to copy the data from the host (CPU-side) to the device (GPU side). Similar to the IQAgent test, I used loops to extract the internal data from the gauxmixmod's data members into plain *float* arrays. This matrix flattening could potentially be a significant source of overhead. I then copied the data over to the GPU with the bulk transfer function. Because the point data is static, I optimized the transfer functions so that it only ever copied to the GPU once, and never copied back.

I transposed the data in several arrays on the GPU so that the threads in my kernel would access sequential offsets in the resulting one-dimensional array in the same step. This allows for memory coalescing, meaning that the multiprocessor will batch transfer a block of data back and forth.

I used single instead of double precision because GPUs typically take many times longer to perform 64-bit calculations than 32-bit calculations, if they can do them at all. To verify that the results are accurate, I compute the MSE difference of the resulting Mus and Sigmas. Rarely are these MSEs above 1e-10.

The major task in actually writing the kernel logic was unrolling the loops, similar to using *arrayfun* in Matlab. The kernel's threads and blocks needed to be mapped to the loops in the original code so that the integrity of the computation would be maintained, but would be done in as parallel nature as possible.

I decided that each threads would be mapped to an individual data point n, and that each block of threads would be mapped to a different cluster k and dimension m. My reasoning for this allocation was that there are multiple loops which sum across data points, but no computationally intensive loops that sum across clusters or dimensions. Since all the threads on a block can talk to each other, it made sense to map data points to threads. Loops involving clusters or dimensions could be computed separately from each other.

However, there are only 512 threads per block, so if there are more than 512 data points my code must loop *nn/512* times, shifting the mapping offset by 512 each time.

Additionally, various functions are performed in parallel. Summation is performed in a Butterfly pattern [1], where every entry paired with complementary entries in such a way that after logn steps, every entry in the array contains the sum of all entries.

Array copying is performed by having every thread in the block grab one data item, and copy it from the start offset to the destimation offset, plus its thread ID. Array zeroing is performed in a similar manner.

I also embedded the Cholesky decomposition functions into the GMM kernel, allowing me to remove variables and combine some of the remaining loops for additional performance.

There are a few code blocks where I make execution conditional on on the thread or the block id. In some cases, there would be no good in a block processing that segment of code, as the result would be redundant. Thus I make the entire block skip it, and hope this will make the block finish faster so that another block can be scheduled on the multi-processor. In these cases, I could potentially further increase the overall speed by segmenting the kernel function such that every block would always see only code relevent to it.

In other cases, most notably on my threadid == 0 conditions, allowing other threads to enter the code segment could have potentially undesirable results, as I am usually writing a single value to global memory, or computing a single value for the rest of the block to use. Segregating my code in this situation is necessary, even though it means the bulk of the threads must wait around for the first thread to exit the block.

3.2.3 Experimental results

My experimental setup consists of a Matlab function which takes in the number of clusters, the dimensionality of the data, and the number of data points to divide between the clusters. It then randomly generates clusters with random means and identity matrices for their covariances. Since this experiment was designed to test the equivalence of the results to the reference implementation as well as the speedup, I was not concerned with generating extremely interesting data, just lots of it. The run-time of the GMM code does not depend on the quality of the data, just the quantity.

The program then takes the sample data and runs it through the CPU-based reference implementation, and then the GPU-based version, noting the time that each takes. It compares the resulting means and sigmas to make sure the

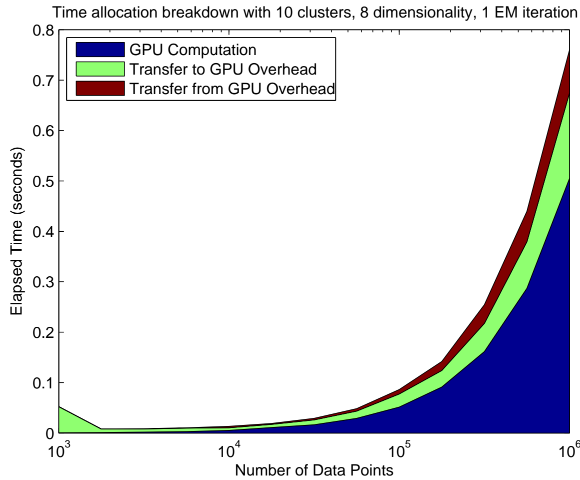

Figure 2. Interestingly, there is a spike in copy time for very small amounts of data. This spike occurred in all trial runs.

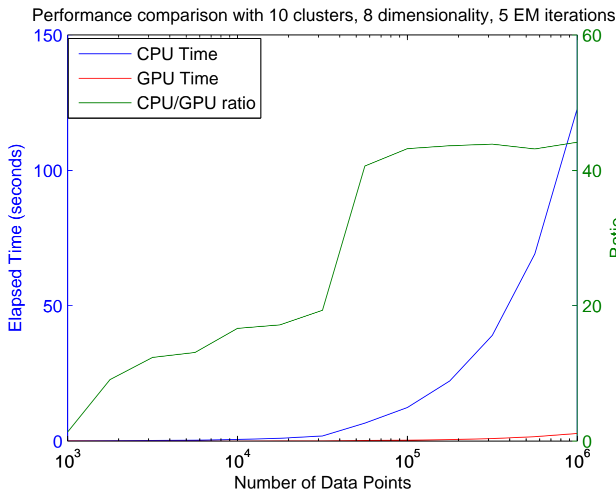

Figure 3. As data size increases, the CPU computation time responds much more drastically than the GPU. The inflection points seen in the ratio are likely do to fullness of utilization of the GPU, which can increase of decrease according to emergent scheduling patterns as the amount of data increases.

results are valid. Time spent copying the data to and from the GPU is included when comparing to CPU performance, as I thought it only fair to include the overhead one would actually see when this technique is used in practice. However, I also ran an experiment illustrating how this overhead is amortized by increasing amounts of data.

Because of the range of parameters that this algorithm can be run with, it is impossible to give a single number for the performance. Without including the overhead, speedups of over 60x were seen. Even with the overhead included and with only a few hundred data points, the GPU method can

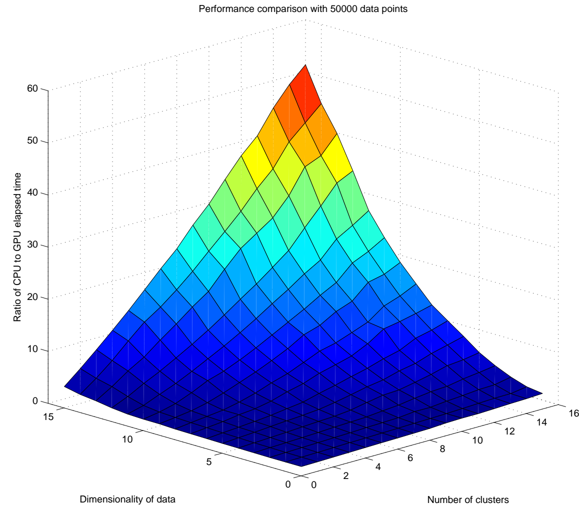

Figure 4. Increasing the number of clusters and dimensionality of the data makes using the GPU much more lucrative, with the current parallelization scheme

still compete favorably, as seen by tracking the ratio line in figure 2. With 2 clusters, 1 dimensions, and 50000 data points, the ratio of performance is already up to 4.18.

3.3. Conclusion

My experimental results are about what I had expected – with proper factorization, parallel computing via GPU is known to provide order of magnitude speedups to certain problems. The factorization of the problem is important. The maximum benefit occurs when the problem can be sliced into as many independant dimensions as possible to allow the problem to be decomposed into as many independant chunks as possible. With GMM EM, this occurs as the dimensionality increases, as the number of clusters increase, and as the amount of data increases.

Additionally, problems which require very little communication compared to computation fare better. IQAgent required the transmittal of a full floating point value per value returned. Scaling the computation meant scaling the amount of data transfer as well, which was a bottleneck. However, with GMMs, the point data only needs to be transfered once, one-way to the GPU. After that, it is not necessary to transfer the relatively insignificant means and covariance matrices back until the computation is finished.

The main benefit of this project is an increased understanding of EM and GMMs for myself, experience with parallel computation, and code that will provide anyone with a modern NVidia graphics card accelerated GMM EM.

References

[1] M. Garland. Parallel computing in cuda, 2008. http://www.gpgpu.org/static/asplos2008/ASPLOS08-3-CUDA-model-and-language.pdf. 3

[2] J. D. M. Rennie. A short tutorial on using expectation-maximization with mixture models. http://people.csail.mit.edu/jrennie/writing, March 2004. 1

[3] V. Simek and R. Asn. GPU Acceleration of 2D-DWT Image Compression in MATLAB with CUDA. In *Computer Modeling and Simulation, 2008. EMS'08. Second UKSIM European Symposium on*, pages 274–277, 2008. 1