K-Means Algorithm Optimization with CUDA

1st Alexandre Flores
PG50165

2nd Pedro Alves
A93272

Abstract

This document describes the process of optimizing a k-means clustering algorithm that exploits parallelization using OpenMP directives by switching to GPU oriented parallel computing.

Index Terms

k-means, clustering, performance, optimization, CUDA, GPU, Occupancy, Reduction

I. Introduction

Previously, when the k-means clustering algorithm was parallelized using OpenMP, scalability issues were encountered due to the threshold of parallelized work exposed by the implementation. This meant that the algorithm's scalability with the total number of samples and clusters was not great.

By using GPU parallelization, it is possible to efficiently utilize a far great number of parallel threads to perform simple tasks (such as that of distributing a single sample into a cluster). As this document details, using CUDA, it is possible to easily achieve far better scaling and overall performance because of this fact.

II. Base Implementation

As a base implementation for this project, the previously implemented algorithm using OpenMP parallelization was used. The base implementation took advantage of parallelizing two computational hotspots:

- The distribution of samples across centroids
- The summation of cluster sample counts and sample coordinates

In order to convert this implementation to CUDA, a plan was outlined. Firstly, the parallelization of sample distribution would be switched to a CUDA kernel. This parallelization is trivially simple because, as explained in the previous implementation's report, there are very few opportunities for data races. Secondly, the cluster sum and count calculations needed to be newly parallelized as well. In the OpenMP implementation, this was achieved using a *reduce* clause to prevent race conditions. A similar approach would have to be taken and tested with CUDA to accommodate for the massive increase in sheer parallelized computation. From this point, profiling tools would be used to detect any remaining opportunities to increase performance via parallelization.

III. Implementation

A. Parallelization Parameters

In OpenMP, a key parameter to evaluate when testing performance is the number of threads utilized in a given parallelized block. In CUDA, parallelization is done by splitting work across *Streaming Multiprocessors*, which in turn manage blocks, which themselves ”manage” individual threads. The amount of threads and blocks, along with the amount of ”problem” each one solves, are all variable and must be studied and analyzed carefully. The approach taken for this was to allow the choosing of the amount of samples each thread would process, and the amount of threads per block. The number of blocks is then calculated at the start of the algorithm based on that information along with the input size.

It's worth noting that because sample totals may not be a multiple of the total number of threads, the last block ID will usually execute with a few idle threads.

B. Sample Distribution

In order to parallelize sample distribution, each thread processes $Samples/Thread$ samples. Since it is possible to divide this work across a very large amount of threads relative to the efficient parallelization threshold of OpenMP, each thread is able to do a very small and simple portion of work assuming $Samples/Thread$ is kept relatively small. This greatly improves the scalability of the algorithm in terms of both total number of samples and total number of clusters, as each thread only has to calculate $Samples/Thread * K$ distances distances.

The only data race encountered in this implementation is that of the convergence flag access. This part of the algorithm uses a flag that keeps track of whether or not any sample has switched cluster in this iteration, and it is stored in device memory for any thread of any block to access and change. Even though threads will only ever set it to true (that is, this doesn't cause a race condition), it is possible to avoid this data race altogether with negligible overhead. In order to do this, threads do the following two things:

- Assign to the flag using CUDA's provided atomicCAS function, making the assignment atomic
- Only ever try to assign to the flag if it is currently read as false, minimizing the thread synchronization overhead of the atomic operation

C. Centroid summation and shared memory support

Because the following optimization utilizes dynamic arrays in kernel, and given that even without it, the sample distribution kernel repeats memory accesses, it was decided to implement shared memory support in this optimization.

Before this optimization was implemented, profiling revealed that the algorithm spent most of its time in the centroid recalculation stage. At that time, this section of the algorithm consisted of the following steps:

1) Copy the sample cluster index information from device to host
2) Reset accumulator variables (centroid sums, cluster sample count) in a K iterations loop
3) Perform centroid coordinate summation and cluster sample counting in an N iterations loop (parallelized with OpenMP)
4) Recalculate centroid coordinates in a K iterations loop
5) Copy new centroid information from host to device

Among these steps, **1** and **5** were deemed the most costly. Step **1** moves $4 * 4$ bytes from device to host (40MB for 10 million samples, for example). Step **5**, by far the most costly of the two, has to perform $3 * N$ additions and assignments (2 for centroid coordinate sums, 1 for cluster sample count). Even though it was still parallelized using OpenMP directives, it is still a very demanding task for the CPU.

To improve this section, the following approach was instead taken:

1) During sample distribution, each thread adds to the sums for that block's samples, and stores results in shared memory
2) After sample distribution but still in the same kernel, K threads for each block add the block's sums to the global sums on the device's global memory
3) After the kernel finishes, back on the CPU, get the calculated sums from the device memory
4) Calculate centroid coordinates
5) Copy centroid coordinates from host to device

Data races, and most importantly race conditions, are avoided in steps **1** and **2** by using CUDA's atomic operations. Step **1** utilizes the _block suffix in its atomic operations, supported in GPUs with compute capability at least 7.0. This means that these operations are only atomic at the block's scope (ideal, since data races are only happening in shared memory which is block specific). Step **2** then utilizes globally atomic additions to add the block's sums to the global totals on the device's global memory. The reason step **2** is done, instead of just adding everything directly into the device's global memory with globally atomic operations, is because the overhead of the consequent global thread syncing weighs more than the overhead of just additionally doing step **2**.

As for memory transfer overheads, since summations are now done on GPU, the amount of data that needs to be transferred from the device has massively decreased. The sums that are transferred from device to host are only made up of $K * 3 * 4$ bytes, which is realistically far less than anything dependant on the sample total.

Finally, since so much of the computation workload has moved to a massively more parallelized GPU kernel, the CPU only has to perform a simple K iterations loop with 2 divisions per iteration to calculate the new centroid coordinates.

D. Sample Generation

Additional profiling revealed that the current bottleneck resided in the usage of *rand* for sample generation (almost 75% of execution time spent on *rand*).

In previous iterations of this projects, sample generation was controlled, and had to be done by sequentially generating each sample with *rand*. This time however, the problem input is arbitrary, and should be chosen according to the testing machine's capabilities/hardware. Therefore, a new, more efficient generation method was used by taking advantage of *curand's* uniformly distributed RNG.

Each thread would now be responsible for using a subset of a sequence of *floats*, which are randomly generated by CUDA's RNG, chosen according to the thread's ID. This ensures that seeding is still supported, aiding in testing and replicability. Additionally, this removes a relatively big bottleneck from the algorithm by massively parallelizing a hefty chunk of sequential workload.

IV. Testing Environment

All testing was performed on a personal desktop computer with the following relevant specs:

GPU	**Micro-architecture**	**Compute Capability**	**Clock Frequency (GHz)**	**Memory (GB)**	**L1 Cache (KB/SM)**
GA104 (RTX 3070)	Ampere	8.6	1.50	8	128

CPU	**Clock Frequency (GHz)**	**Cores**	**L1 Cache (KB)**
Ryzen 5 2600	3.60	6	576

V. Testing and analysis

Because of the large amount of variables that go into each testing scenario, only a limited amount of combinations are shown in this report. In particular, different sample per thread and thread per block rates are not shown. Instead, a rough estimate was done to quickly determine which combination worked best for a particular input size. This combination was then used to perform the tests displayed on the results table.

Sample and cluster totals were chosen such that it is possible to observe and compare the implementations' scalability with input sizes.

No thread distributions with theoretical GPU occupancy below 100% for the used hardware were tested (as confirmed by Nsight Compute's feedback on theoretical occupancy of each kernel). Performance metrics were collected with *perf* and *Nsight Compute*.

Finally, algorithm iterations were limited at 20. The intention is to test the algorithm's implementation with no concern for the intrinsic limitations of the algorithm itself. Testing with unlimited iterations revealed that higher sample counts will inevitably cause a rampant increase in iterations, limiting the information gained about performance when comparing tests with different sample totals.

A. Observations

1) Base: This implementation appears to scale harshly at around this order of magnitude of input size.

2) GPU Sample Distribution: The sheer level of parallelization achievable with the GPU greatly improves the scalability (and overall performance) of the algorithm. In particular, the number of clusters barely weighs on the performance. Additionally, different thread per block and sample per thread totals did not yield performance changes.

3) GPU Sample Sums + Shared Mem: Optimal shared memory usage was found to be around 2560 samples (20KB) per block. We also notice a sharp decline in CPU cache misses and a hit to GPU L1 hit rate because a lot of memory work (cluster sums) was moved to the GPU. This has also cleared up a big bottleneck in centroid recalculation, massively improving scaling.

4) GPU Sample Generation: As expected, parallelizing the randomization of samples has massively improved scaling with N. Furthermore, moving more memory work from CPU to GPU has slightly reduced the CPU's L1 cache misses. However, their total has also become far more volatile, something that is not represented in this table.

Version	**Samples per Thread**	**Threads per Block**	**K**	**N**	**T. exec. (s)**	**Occupancy**	**GPU L1 Hit Rate**	**CPU L1 Misses (E+6)**
Base	N/A	N/A	32	10ˆ7	1.50	N/A	N/A	83.12
				10ˆ8	14.65	N/A	N/A	841.34
			64	10ˆ7	2.56	N/A	N/A	83.46
				10ˆ8	24.27	N/A	N/A	841.34
GPU Sample Distribution	1	128	32	10ˆ7	0.93	91%	85%	60.01
				10ˆ8	7.38	91%	85%	595.61
			64	10ˆ7	0.94	94%	91%	59.63
				10ˆ8	7.44	93%	91%	603.49
GPU Sample Sums + Shared Mem	5	512	32	10ˆ7	0.44	98%	47%	3.38
				10ˆ8	3.60	98%	48%	29.77
			64	10ˆ7	0.45	98%	48%	3.21
				10ˆ8	3.63	99%	48%	28.09
GPU Sample Generation	5	512	32	10ˆ7	0.15	98%	49%	1.31
				10ˆ8	0.50	99%	48%	3.43
			64	10ˆ7	0.16	98%	49%	1.52
				10ˆ8	0.53	99%	48%	6.10

VI. Conclusion

Having now developed several iterations of this clustering algorithm, we conclude that the projects has met our expectations for performance. As expected, the exposure of parallelized work on GPU benefits scalability for this algorithm immensely. While the testing done was oriented towards our own testing environment, the option to specify all the testing parameters in the binary's arguments should theoretically allow any CC>7.0 GPU to run it with its own supported parameters.