Parallel version of the K-means algorithm using CUDA

Diogo Ramos
Physical Engineering
University of Minho, Portugal
Email: a95109@uminho.pt

Gabriel Costa
Physical Engineering
University of Minho, Portugal
Email: a94893@uminho.pt

***Abstract*—This report analyzes the performance of an implementation of the k-means clustering algorithm using CUDA, as well as a small comparison with an OpenMP implementation.**

1. Introduction

On this final version of the K-means clustering algorithm we decided to follow a new approach, implementing a code that allowed our program to run in a GPU (Graphical Processing Unit), which has a lot more computation power than a CPU, expecting improvements from the previous version.

2. Introductino to CUDA

While a CPU is a generalized processor designed to carry out a vast variety of tasks, a GPU is an accelerator that, differently from TPUs (accelerators optimized for operations using tensors) is highly optimized for number crunching, such examples include vector/matrix operations.

In order to achieve a high computational throughput GPUs are equipped with multiple SMs (streaming multiprocessor), which are, in turn, broken down into blocks (number of blocks per SM is architecture dependant) composed of cores, load/store units, register bank, schedulers, memory, amongst others. We can see the composition of a particular SM in the following picture.

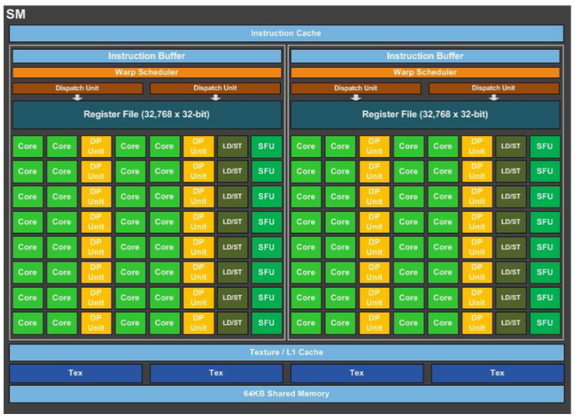


Figure 1: SM unit from a Pascal based GP100

In this report we will analyze performance with the use of a pascal based GP107 (NVIDIA Geforce GTX 1050) with compute capability 6.1 and a Kepler based GK110 (NVIDIA Tesla K20m) with compute capability 3.5, both equipped with a 65536 register bank and 96 KiB of shared memory per SM. While the GTX 1050 sports 5 SMs running at a base speed of 1354 MHz (1493 MHz boost) leading to a FP32 throughput of 1,911 TFLOPS the K20m has 14 SMXs (see attachment [a]) at 706 MHz boasting a throughput of 3,524 TFLOPS.

In order to use GPUs we need to write our program in a C-like language called CUDA C/C++, which serves as conversion layer between normal C/C++ and the GPUs virtual instruction set, also using NVCC to translate *.cu* files (CUDA C/C++ standard code file) into PTX, cubin and object files, it also replaces kernel configurations with the appropriate functions calls (full conversion to C/C++ code), which can then be passed on to a normal C/C++ compiler.

GPUs are also notable for having different programming models, usually associated with SIMT (single instruction multiple threads), which consists of various threads execution a single instruction in lock-step, that can be achieved through the concept of a warp (group of threads, usually 32) executing a single instruction in all it's threads. Another difference is that GPUs are used as accelerators and thus need a host (CPU) which serves as a coordinator for the whole system, telling the GPU what work to do and allowing communications between components such as GPUs and system memory (this particular example can be done without the host in modern systems using NVIDIA GPUs with the use of NVIDIA GPUDirect^{TM}).

3. Code

First order of work would be to rethink the approach to the K means algorithm since we are working under a different programming platform.

A good place to start would be to reorganize the algorithm's stages (distributing the dataset, recalculating the cluster's centroids and verifying the convergence of the dataset). One way to do so started by merging the distribution of the dataset with calculating the sums and sizes of each cluster and after it we would be left with finding the mean of each clusters centroid (a simple division since the sums and sizes where already computed) and checking the iteration for convergence, this last step was done by applying a reduction over an array of values representing the convergence of each thread block, yielding the sum of transitioning elements.

Implementing and profiling said algorithm gives us the following graph.

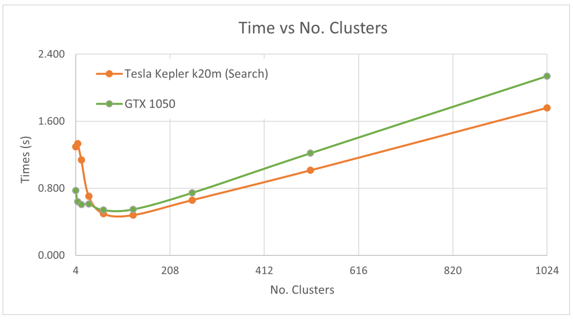


Figure 2: Execution time vs number of clusters for a dataset of 10 million points

Clusters	Kepler k20m Time (s)	CUDA Time (s)
4	1,291	0,075
8	1,334	0,641
16	1,138	0,606
32	0,707	0,614
64	0,496	0,541
128	0,480	0,550
256	0,659	0,747
512	1,015	1,220
1024	1,760	2,140

Table 1: Times obtained in tow different GPU for a dataset of 10 million points

By analysis of the graph above we see that for small cluster amounts the performance isn't great, the reason for this is that when the number of clusters is small the amount of work required to partition the data decreases. In the k means algorithm, each data point must be compared to the centroids of each cluster to determine its assignment. With fewer clusters, there are fewer centroids to compare each data point to, resulting in less work. To understand how this impacts the program we can profile it to a cluster size of 4 in more detail.

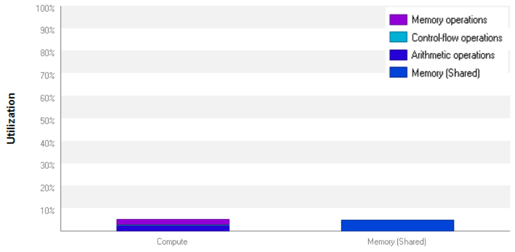


Figure 3: Utilization for a configuration of 10 million points and 4 clusters on GTX 1050

As we can see for 4 clusters the amount of compute done by the GPU is astonishingly low, as expected, since with such little non-predicated work to be done we can't fully utilize a GPU's parallel capability, leading to slower performance.

On the other hand, for a total of 1024 clusters the results fall more in line to what was expected since now the amount of work required for the GPU increases. Each data point must now be compared to more centroids, resulting in more work for the GPU. We can profile the code to see how it impacts the GPU utilization.

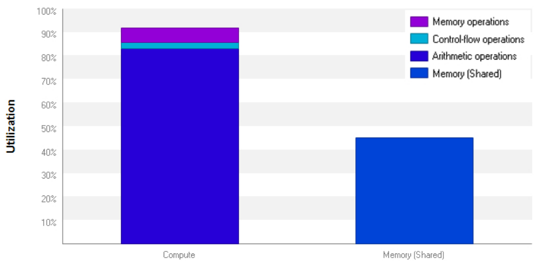


Figure 4: Utilization for a configuration of 10 million points and 1024 clusters on GTX 1050

As we can see for 1024 clusters the amount of compute done by the GPU is nearly 100%, which means that now the work done is limited by the SM's compute, even though memory still plays a part it's now more diluted by the sheer amount of calculations that need to be done for each point.

On parallel implementations of the K means algorithm there's the need to solve data-race conditions, with the way the algorithm is structured they naturally arise. On CUDA one way to do it would be to use a tree reduction over an array on local memory, but we could also use an atomicAdd operation which comes with the CUDA library.

Although atomic operations are generally slower, according to the CUDA Toolkit Documentation since these operations are specially implemented together with the memory and use a fire-and-forget semantic in case of not needing a return value, as happens in this algorithm since we do not care about what value was in memory prior to the alteration.

This fire-and-forget implementation essentially means that the kernel calls the atomic operation and lets the actual atomic operation be executed by the cache (not on the the SM), and the kernel will move on the the next instruction without waiting for the actual atomic operation to complete.

When tested against a version with the tree reduction, the atomic version proved to be faster since normally the atomic operations would be resolved before the next wave of thread blocks reaches the operations, effectively being almost as fast as a normal write to memory, as noted in the Kepler GK110/210 whitepaper[1]. This fire-and-forget way of doing atomics is implement with the use of a memory reduction instruction (RED), as we can see in attachment [b].

[1]Page 12 (Atomic operations) Kepler GK110/210 Whitepaper "https://www.nvidia.com/content/dam/en-zz/Solutions/Data-Center/tesla-product-literature/NVIDIA-Kepler-GK110-GK210-Architecture-Whitepaper.pdf"

3.1. Closer analysis for the peak performance case

We can now take a look at a particular input, 10 million points and 128 clusters, allowing the program to dynamically adjust the number of threads and number of clusters, should make 128 clusters a good balance between compute and memory load, since we can, with the memory usage necessary populate all 64 warps in a SM and have 16 active blocks per SM, yielding the highest occupancy out of all inputs.

The occupancy represents the amount of warps per multiprocessor compared to the hardware's limit with the objective of being as close as possible to 100%.
For this section all results where gather with the use of a NVIDIA GeForce GTX 1050.

Profiling the program can give us a better look at what each kernel represents to total execution time, it's expected the *cycle_kernel* function will take almost all the compute time because it does almost everything that's computationally intense.

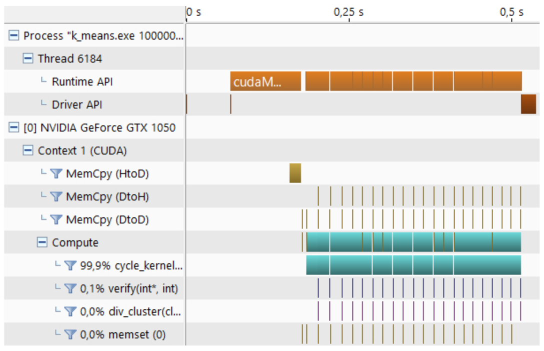


Figure 5: Profiler timeline for 10 Million points and 128 clusters

Looking at the time line we can see a big gap in the beginning which upon further testing proves to be always there and have a very similar execution time, around 200ms, for different inputs, this portion represents the initialization, where the points are generated and all memory allocations and copy operations occur in order for the program to be executed on the GPU. Furthermore we can see that the function *cycle_kernel* does, in fact, take all the execution time of the program being that it takes 99.9% of all the kernels(functions executed by the GPU) time.

Looking now at the compute and memory load.

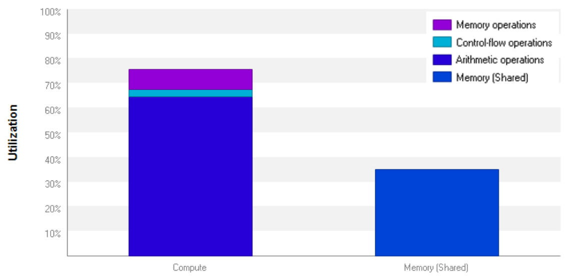


Figure 6: Utilization of computational and shared memory resources in *cycle_kernel*

From the graph we can see the compute is at around 60% utilization (recommended value by the CUDA Toolkit Documentation) which means that compute isn't a bottleneck and combined with the high warp utilization means that each SM runs at almost 100% efficiency. To know more about what might be slowing down performance we should look at the results of profiling by sampling of program counters (PC Sampling).

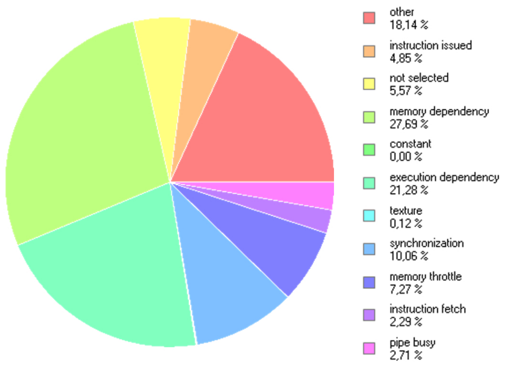


Figure 7: PC Sampling result in *cycle_kernel* function

From the pie chart we have now have a better understanding to what the program is doing. We can see that memory dependency still plays a major role, that arises from the use of predicated instructions (instructions that depend on the result of others to be executed such as if's, and control flow of loops); execution dependency means the thread is doing work (this includes everything the program has to do), finally the other section comes from bank register conflicts and warps waiting for branches to be solved, every other reason is within a reasonable amount for the kind of work the function does. It is to note that PC Sampling doesn't track instruction by instruction it stops the code at intervals and samples what it's currently working on.

We should now get a better view at the kinds of memory transactions the program does.

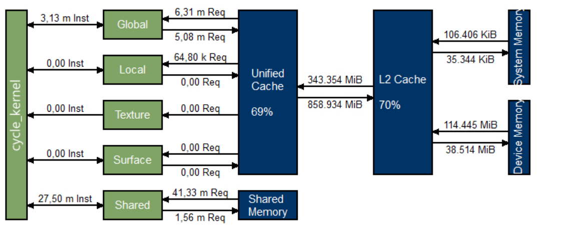


Figure 8: Memory transactions

On the diagram above blue blocks represent real memory, while green blocks represent virtual memory.

We can see that there are a staggering 41,33 million requests from shared memory, which shows just how important it is in order to optimize the code, since the access time to load and store is vastly lower than global memory, doing that many requests to global memory would drastically increase the program's execution time. On the other hand we also see a lot of accesses to global memory with writes almost equalling reads, this naturally occurs since all data is stored in global memory and has to be loaded into shared to be used in a function. The use of the atomic operations is done in memory, even though it's not specified in CUDA's Documentation, one might assume that there are special units for those kind of operations in memory, and so, all the atomic operations done account for a big amount of the write requests into global memory.

4. Comparing with OpenMP

We have now decided to compare this implementation with the previous one (OpenMP from TP2). To do this, we measured the run times of the CUDA implementation on Search's cluster K20m and compared them to the run times of the parallel version of the code using OpenMP also on the cluster, which utilizes a CPU Xeon® E5-2670 V2. The results were then processed and displayed in the following table and bar graph.

Clusters	Parallel (OpenMP) Time (s)	CUDA Time (s)	Speed-Up
4	0,514	1,297	0,40
8	0,581	1,344	0,44
16	0,809	1,138	0,71
32	1,313	0,707	1,86
64	2,366	0,496	4,77
128	4,405	0,480	9,18
256	8,474	0,659	12,86
512	16,568	1,015	16,32
1024	32,880	1,760	18,68

Table 2: Times obtained on "Search" using a 20 thread parallel OpenMP and a CUDA implementation

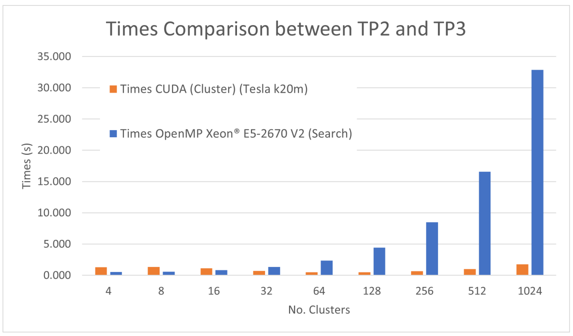


Figure 9: Scalability analysis using 64 clusters

The bar graph illustrates a decline in performance when using a small number of clusters with the new implementation. However, as the number of clusters increases, more of the GPU's parallel capability is utilized, resulting in a significant increase in performance compared to the OpenMP version

However, when using a large number of clusters, a clear difference can be seen due to the computational power of the GPU compared to the CPU, having the CUDA implementation for 1024 resulted in a speed-up of 18.68× compared to the OpenMP version.

5. Conclusion

In summary, using CUDA to run our program on a GPU presented some difficulties, since we where dealing with a new way of problem solving. But by doing so, we were able to create a k-means algorithm that is faster for large samples compared to the OpenMP version, achieving a speedup of 18.68×. For small samples, it would be more beneficial to use an OpenMP implementation, as the GPU would not provide significant performance gains.

References

[1] CUDA Parallel Thread execution Instruction Set Architecture: https://docs.nvidia.com/cuda/pdf/ptx_isa_8.0.pdf

[2] CUDA Binary Utilities: https://docs.nvidia.com/cuda/pdf/CUDA_Binary_Utilities.pdf

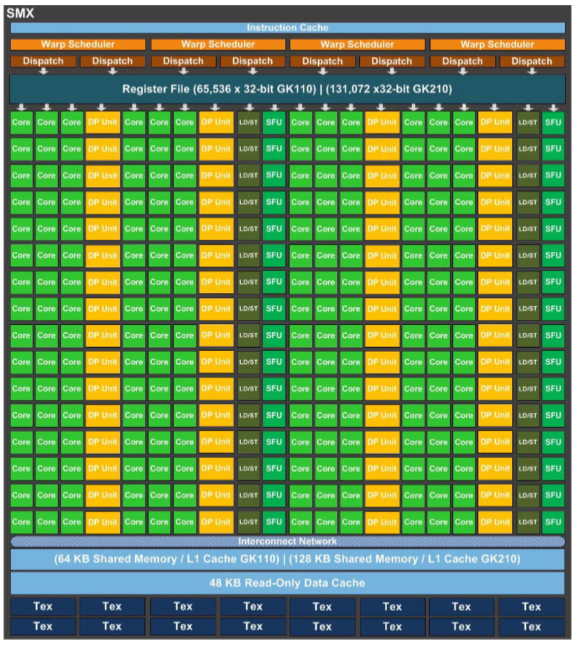
[3] Tesla Kepler GK110/210 Architecture Whitepaper: https://www.nvidia.com/content/dam/en-zz/Solutions/Data-Center/tesla-product-literature/NVIDIA-Kepler-GK110-GK210-Architecture-Whitepaper.pdf

[4] Tesla Pascal GP100 Architecture Whitepaper: https://images.nvidia.com/content/pdf/tesla/whitepaper/pascal-architecture-whitepaper.pdf

[5] Optimizing Parallel Reduciton in CUDA: https://developer.download.nvidia.com/assets/cuda/files/reduction.pdf

6. Attachments

[a] SMX Unit from a Kepler based GK110


SMX
Instruction Cache
Warp Scheduler
Dispatch
Register File (65,536 x 32-bit GK110) | (131,072 x32-bit GK210)
Core
DP Unit
LD/ST
SFU
Interconnect Network
(64 KB Shared Memory / L1 Cache GK110) | (128 KB Shared Memory / L1 Cache GK210)
48 KB Read-Only Data Cache
Tex

[b] Atomic add with fire-and-forget semantic

```
.L_x_19:
{ ISCADD R6.CC, R16.reuse, c[0x0][0x158], 0x2 ;
  @!P2 LDS.U.32 R14, [R14] }
  @!P2 SHR.U32 R2, R13.reuse, 0x1e ;
  SHR R0, R16, 0x1d ;
  IADD.X R7, R7, c[0x0][0x15c] ;
  @!P2 ISCADD R4.CC, R13, c[0x0][0x168], 0x2 ;
  @!P2 IADD.X R5, R2, c[0x0][0x16c] ;
  ISCADD R2.CC, R16, c[0x0][0x150], 0x3 ;
{ IADD.X R3, R0, c[0x0][0x154] ;
  @!P2 STG.E [R4], R14 }
  MOV32I R0, 0x1 ;
  RED.E.ADD [R6], R0 ;
  RED.E.ADD.F32.FTZ.RN [R2], R15 ;
  RED.E.ADD.F32.FTZ.RN [R2+0x4], R12 ;
  NOP ;
-----------------------------------------------
EXIT ;
```