ACS Lab 3 - Accelerating a clustering algorithm

Ingimundur Vilhjalmsson (4911326, ivilhjalmsson) Naveen Jakka (4770145, njakka)
Thomas Makryniotis (4625501, tmakryniotis)

Report contains 9 of maximum 10 pages.

1 Introduction

For lab 3 we sought out to accelerate a k-means clustering algorithm using techniques that we learned from the previous labs.

In essence, the objective of k-means clustering is to group data points together into k number of clusters based on their attributes or features. These features can be represented by values in a Euclidean space. Determining which cluster a data point belongs to is done by finding the closest centroid to it. Centroids signify the center of a cluster, and are randomly generated initially. Since our data set exist in a Euclidean space we use the Euclidean distance formula to calculate the distance between points and centroids. Which is given by equation 1.

$$d(\boldsymbol{p}, \boldsymbol{q}) = \sqrt{\sum_{i=1}^{n} (q_i - p_i)^2} \tag{1}$$

Once the closest centroid to a data set element is found we assign a label to that element corresponding to the centroid. Then after every data set element has been assigned a label we refine the values of each centroid. This is done to find more accurate positions for the centroids, which is accomplished by finding the average position of the elements within their cluster. This process requires multiple iterations, but will eventually converge when the labels remain unchanged.

2 Experimental setup

Throughout the whole lab we used the INSY cluster to benchmark our work, on which used CUDA 9.2. The specifications of the hardware we ran on are as follows:

Element	Desc
Name	GTX 1080 Ti
Global memory	11.7 GB
Shared Memory Per Block	384 KB
Total Constant Memory	1024
Registers Per Block	512 KB
Maximum Threads Per Block	1024
Multiprocessor Count	28
Warp Size	32

Table 1: GPU specifications

Every optimization we made was profiled using five different benchmark configurations:

Setting	B1	B2	B3	B4	FOM
Data set size	500	2048	10^4	10^5	10^6
Features	2	3	8	20	42
Clusters	3	5	10	20	24
Iteration Th.	10	15	18	25	30
Scaling factor	0.1	0.01	1e-3	1e-4	1e-5

Table 2: Benchmark Configurations

Note: Any dataset that is referred to in the report is the figure-of-merit (FOM) benchmark, unless otherwise stated.

3 Initial Analysis

At the beginning phases of this lab we carefully studied the provided code and made detailed notes on how it worked. We have captured the serial algorithm in a flowchart which is shown in the below figure.

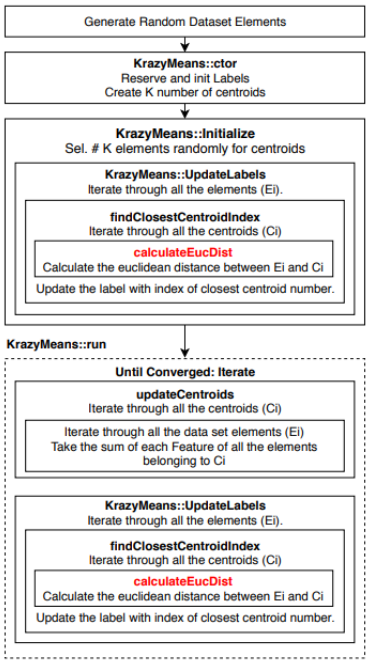

Figure 1: Flowchart for Serial version

Furthermore, after comprehending the objective of the code we analyzed each function in terms of how they could be optimized. This began by identifying which parts had heavy computational loads, while also looking into ways to achieve minimal memory transfers between the CPU and GPU.

3.1 Compute-intensive Areas

Using *gprof* we were able to profile the initial code. This allowed us to quickly see how time consuming each function of the program was when it is ran. The following table shows the obtained data from using *gprof*:

% time	name
92.76	calculateEuclideanDistance
4.97	findClosestCentroidIndex
1.95	updateCentroids
0.32	Other

Table 3: Gprof data

From Table-3, it is quite apparent that the most time consuming task is calculating the Euclidean distances as it takes approximately 93% of the total runtime of the program. Although the function `calculateEuclideanDistance` only computes the distance between two points, it is called incredibly often. Figure-1 illustrates this section marked in red letters, which is enclosed in a loop which iterates through all the dataset elements. For this reason we checked where the function calls are being made and found out that they occur in the function `findClosestCentroidIndex`. However, this functions purpose is only to determine what centroid a data point is closest to, and only checks for one data point. This is because the function `updateLabels` calls `findClosestCentroidIndex` for every data set element, one at a time. Thus, implementing simple parallelization, such as OpenMP, would undoubtedly be most effective if done in `updateLabels`, rather than in the other previously mentioned functions.

4 Optimization 1: OpenMP Implementation

4.1 Description

After having gone through the code and figuring all the ins and outs to it we inserted OpenMP pragmas to the part we believed would benefit from parallelism the most. That is, parallelizing the for loop in the `updateLabels` function using 8 threads. The main purpose for doing this was to verify that we had correctly identified the compute-intensive areas. The part of the code where the change is made is illustrated in the Figure-2. The parts outlined in green are the places where OpenMP was implemented.

4.2 Result and Profile

The CPU baseline execution took 753.857 seconds whereas the 8 threaded OpenMP execution took 200.204 seconds; that is almost 3.8 times faster. This speedup sounds great on paper, but when considering the overall execution time, even with OpenMP, it is still too long. Our overall timing results after conducting benchmarks for all five execution configurations, as shown in table 2, are presented in figure 3.

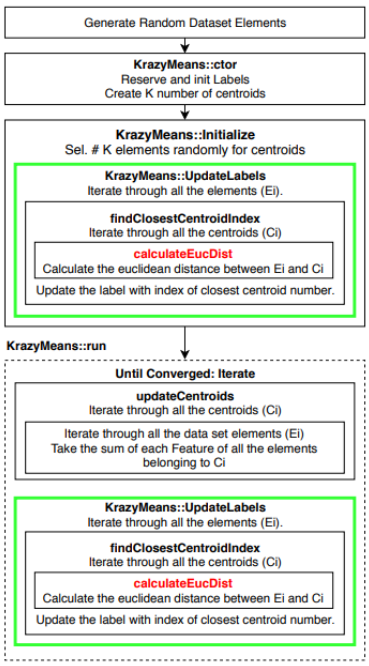

Figure 2: Flowchart indicating where OpenMP was used

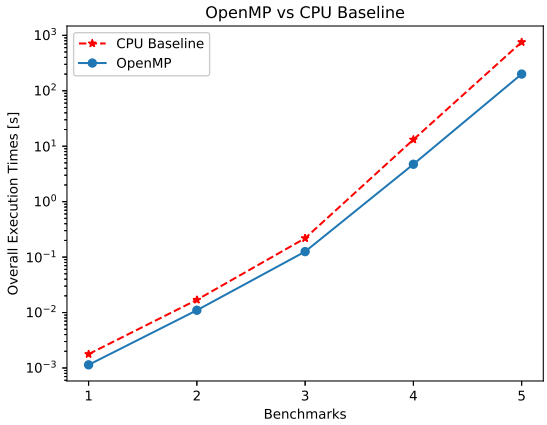

Figure 3: OpenMP vs CPU Baseline

From figure 3 it is evident that the OpenMP optimization increases the performance for small and large data sets. Furthermore, as the data set grows the speedup from OpenMP does as well.

4.3 Discussion

Even though we observed good speedup after implementing OpenMP, we knew that it could be improved vastly by using CUDA or OpenCL instead. *The main objective of the OpenMP optimization was only to identify areas that benefit from parallelization and not to achieve the best performance possible.* That is why we did not devote any more time or effort in building on this optimization.

5 Optimization 2: Naive CUDA Implementation

5.1 Description

Following our first optimization, we began working on first CUDA implementation. From the profiling data obtained from the serial version, we identified the function `calculateEuclideanDistance` as the bottleneck and decided to implement the function `updateLabels` on the GPU. This was achieved by assigning each dataset to a thread on the GPU and updating its corresponding label. The flowchart with the implementation can be seen from Figure-2 outlined in green which is the same as the previous optimization.

5.2 Challenge

The baseline algorithm uses a flag named `updated` which is used to notify the subsequent program that a label value was updated. This is easy to implement when implemented sequentially. The actual challenge was implementing the flag in a parallel version. This challenge was solved by extending the labels container by one element and using the last element as a place holder to keep the update count. With initializing it to zero and each time an update happens the counter is incremented. After the kernel call is returned and after the labels are copied back, the last value of the labels container can be examined to determine if the label has been updated or not. (Greater than zero: value has been updated, Equal to zero: no label was updated). By doing this, we also reduced a separate storage for the `update` flag.

5.3 Profile

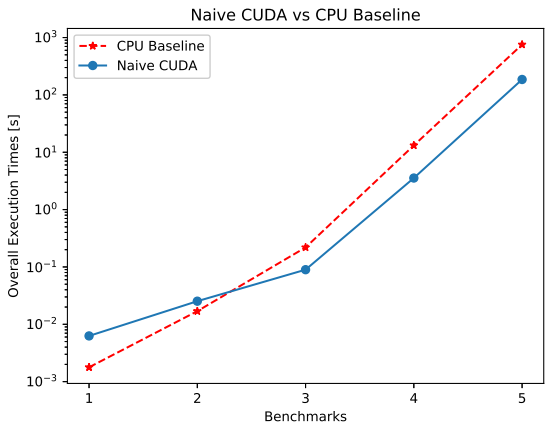

Figure 4: Naive CUDA vs OpenMP CPU

Having the GPU work on `updateLabels` reduced the overall runtime significantly; for FOM we attained a speedup of 4.06. Comparing this naive CUDA implementation to the results yielded from using OpenMP on the CPU we get a small speedup of 1.08. This result was expected since the implementation is quite naive and we are obviously not utilizing the GPU capabilities efficiently.

5.4 Estimation

Building on this implementation would mean to perform `updateCentroids` on the GPU and also to optimize the memory transfer from the host Host to Device. When taking these factors into account we would expect far higher throughputs in memory transfers and computations which would lead to drastically improved speedups.

5.5 Design and implementation

Looking at the algorithm, the dataset elements are only being read. This observation led to the decision of storing the dataset elements in the GPU during initialization phase and leave them in the memory until the program finishes execution. This decision reduced the frequent memory transfer of about 176 MBytes (referring to FOM benchmark) which takes about 50 ms.

The actual data of centroids and dataset elements are wrapped in two levels. This data cannot be handled by the GPU directly. Therefore, for the GPU to access this data, just before when there is a need to transfer the data to the GPU, these data is linearized, which transforms the data into a one-dimensional array. The GPU kernel can make use of thread-block indexing and feature size to find out the dataset element it needs to access. After the kernel execution, the label data is copied to the CPU. To obtain 100% thread occupancy, we have used 1024 threads per block.

5.6 Discussion

In this section we have looked into which parts of the serial program can be implemented on the GPU. An obvious speed-up was seen with this implementation compared to the serial baseline. In the subsequent sections we look further into optimizing the implementation.

6 Optimization 3: Algorithm Enhancement

6.1 Description

In this section's implementation we have developed on the idea of assigning each thread a dataset element for calculation. We have made an algorithm enhancement with which we could achieve calculation on the GPU for the function `updateCentroids`. We have noticed a significant improvement in performance.

6.2 Algorithm Revision

Aside from the enhancements done as part of moving the calculations to the GPU, we found out that the algorithm used by function updateCentroids can be enhanced. Assume that there are K-centroids, N-datasets and F-features. The outer for-loop iterates through each centroid, the second for-loop iterates through each dataset element and inner most for-loop iterates through all the features. This takes K*N*F iterations to accomplish the job.

The process of iterating through every data set ele-

ment k number of times to calculate refined feature values for each centroid seemed inefficient. Therefore we modified it to iterate through each element and do all relevant feature accumulations, then calculate the average values for the new centroids in a seperate loop. This is improved in the implementation by using a for-loop to iterate through each data_set index and another for-loop to iterate through each data_set feature. Which does N*F number of iterations in total. We have analyzed the time taken by these two variants and the time taken has been noted in the table below. It can be observed that there is no significant improvement in the amount of time taken. However, the for loops are restructured. With the baseline algorithm it would be very inefficient to run the same code on the GPU. But with this restructuring, algorithm can leverage the GPU and produce higher throughput. Subsequent Section focuses on implementing the algorithm on the GPU.

```
// Naive algorithm
void KrazyMeans::updateCentroids()
{
 clearCentroids();
 for (size_t cen = 0; cen < centroids.size();
     ...  cen++) {
  size_t num_assigned = 0;
  for (size_t ds_elem = 0; ds_elem < data_set
      ... ->size(); ds_elem++) {
   if (labels[ds_elem] == cen) {
    num_assigned++;
    for (size_t f = 0; f < centroids[c].size
        ... (); f++) {
     centroids[c][f] += data_set->vectors[v][
         ... f];
    }
   }
  }

  // Average each centroids with number of
      ... assignments
  if (num_assigned != 0) {
   for (size_t feature = 0; f < data_set->
       ... num_features; f++) {
    centroids[c][f] /= (float) num_assigned;
   }
  }
 }
}
```

6.3 CUDA Implementation of Revised Algorithm

With the new algorithm it is feasible to assign each dataset element to a GPU thread and calculate the new centroids which would have not been possible otherwise.

```
// Enhanced algorithm
void KrazyMeans::updateCentroids()
{
 clearCentroids();

 vector<float> assignments(centroids.size()
     ... ,0);

 for(size_t ds_elem = 0; ds_elem < data_set->
     ... size(); ds_elem++) {
  assignments[labels[ds_elem]]++;
  for(size_t feature = 0; feature < data_set
      ... ->num_features; feature++)
   centroids[labels[ds_elem]][feature] +=
       ... data_set->vectors[ds_elem][feature
       ... ];
 }

 // Average each centroids with number of
     ... assignments
 for(size_t cen = 0; cen < centroids.size();
     ... cen++)
  for(size_t feature = 0; feature < data_set
      ... ->num_features; feature++)
   centroids[cen][feature] /= (float)
       ... assignments[cen];
}
```

6.4 Design and Implementation

The dataset elements, centroids and labels are transferred to the GPU. Along with this a new data container called `assignments` (with size equal to number of centroids) is also passed to the GPU. This data container stores the number of dataset elements assigned to a particular centroid. In the kernel, each time a label of a particular dataset element is read, the assignment is incremented by one using atomicAdd.

After the execution of the kernel call, a for-loop runs through all the centroids and their features to calculate average of each element. Which results in the final centroids. Figure-5 shows parts of the code uses CUDA (outlined in green). But the lines in blue are executed after the kernel call in the CPU.

6.5 Profile

Our benchmarking results from this optimization are graphed in the following figure.

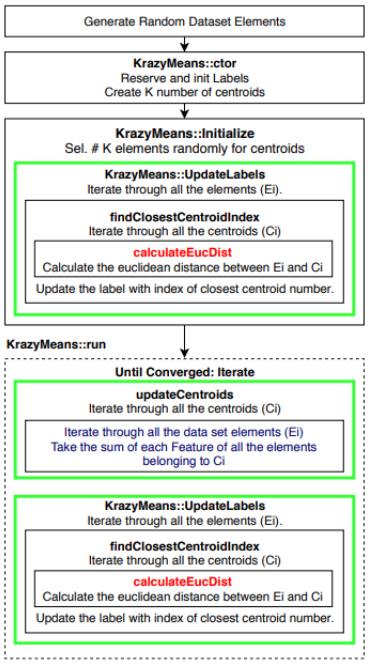

Figure 5: Full-Implementation CUDA

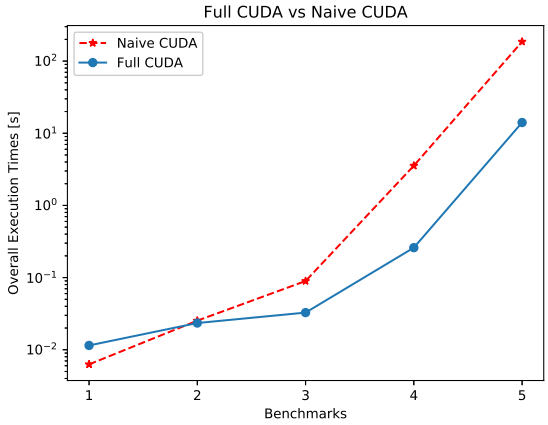

Figure 6: Full CUDA vs Naive CUDA

As we expected, the effects of utilizing the GPU for both `updateLabels` and `updateCentroids` we achieved major speedups in the execution time. From the FOM benchmark on the Full CUDA implementation we got an overall execution time of 14.08 seconds, while the naive CUDA implementation took 185.80 seconds; a speedup of 13.2.

6.6 Estimation

Because we are still doing computations on the CPU, averaging features for each centroid, we suspected that it would be feasible to handle them with AVX or OpenMP to increase the performance. While having that in consideration we also discussed and analyzed optimal data transfer strategies.

7 Optimization 4: Inclusion of AVX / OpenMP

7.1 Description

In the previous optimization we have split the function updateCentroids into two components and assigned one for the GPU and another for the CPU with a for-loop to iterate over all the centroids to perform averaging of each feature. In this optimization we focus on ways to improve the iteration made by the for-loop and reduce the time spent on calculating the averages of each feature. To achieve this we figured that we could use either AVX or OpenMP for calculating averaging over the features. There is an insignificant improvement by implementing AVX to perform averaging. The primary reason being the overhead included when transferring data back and forth the AVX registers

7.1.1 Design and implementation with AVX In general AVX is good for the tasks which for perform a single operation on multiple data (otherwise know as SIMD). The for-loop first iterates through all the centroids and for each centroid it iterates through each feature and divides it with assignment that was obtained by execution of kernel function. This part can be rewritten using AVX. Out of all the centroid features present, eight of these elements can be considered each time and loaded into an AVX register and broadcast the assignment into another register and perform a division which would divide each content of the previous AVX register with *assignment* and repeat the process until division for all the features is completed.

7.1.2 Implentation with OpenMP Similarly to the proposed AVX implementation we also pondered on what kind of effect OpenMP would have on this particular computational part. That is, performing the averaging of `updateCentroids` with OpenMP. After some testing we were unable to see any improvements whatsoever. In fact, the 8 threaded OpenMP implementation had a negative effect. A possible reason for this might be due to not enough data for OpenMP to be effective. Moreover, the time that `updateCentroids` consumes is miniscule. Thus, any successful improvements regarding this area would only contribute to small speedups. Because the return of investment in time spent on optimizing `updateCentroids` seemed poor, we decided to shift our focus to other parts of the program.

8 Optimization 5: Reducing Data Transfers

8.1 Description

With implementation of Optimization 3 after implementing both the functions `updateLabels` and `updateCentroids` on the GPU, and running the *nvprof*, it was clear that there were many redundant transfers happening back and forth the device memory, which is expensive. In this section we focus on ways of reducing the number of data transfers and redundant

memory allocation that take place.

8.2 Design and Implementation

Since we are going to keep using device memory for `labels`, `centroids` and `assignments` our first improvement was achieved by avoiding device memory allocation before every kernel call and memory freeing after the call.

The device memory allocation for `centroids, labels and assignments` are being done in the initialization stage and these pointer are used throughout the program whenever a memory transfer to that particular data is needed to the device memory.

From the flowchart shown in Figure-1, it can be observed that `iterate` function is being called until the dataset elements converge. iterate function further calls `updateCentroids` and `updateLabels` in a sequence.

The function `updateCentroids` finds the new centroids on the GPU and after the kernel call the memory is transferred back to the CPU but the new centroids data is left on the GPU. Similarly, the function `updateLabels` finds the new labels values for each dataset and copy its data back to the CPU and the data of labels is left on the memory for the consecutive usage. The algorithms shows that `updateCentroids` use the labels data and `updateLabels` use the centroids data. We consider this inter-dependence of data as an advantage and leave the data in the memory so that the next consecutive function call can use the data without having to copy the data to the device.

8.3 Profile

For the above implementation, using NVPROF we measured and tabulated some important parameters for both Full-CUDA implementation and Reduced Memory Transfers.

Time[s]	**Time[%]**	**Description**
5.5236	84.02%	cuda_updateLabels
0.6373	9.70%	cuda_accumulate_features
0.3501	5.33%	[CUDA memcpy HtoD]
0.0631	0.96%	[CUDA memcpy DtoH]
API Calls	**Calls**	**Description**
76.88%	103	cudaDeviceSynchronize
16.10%	825	cudaMemcpy
5.88%	516	cudaMallocManaged
1.14%		Other

Table 4: Nvprof data in Full-CUDA optimization

Time [s]	**Time[%]**	**Description**
12.5434	94.35%	cuda_updateLabels
0.6085	4.58%	cuda_accumulate_features
0.0728	0.55%	[CUDA memcpy HtoD]
0.0703	0.53%	[CUDA memcpy DtoH]
API Calls	**Calls**	**Description**
90.27%	104	cudaDeviceSynchronize
5.98%	726	cudaMemcpy
2.84%	3	cudaMallocManaged
0.91%		Other

Table 5: Nvprof data after reducing memory transfers

It can be clearly seen that the number of memory operation have been reduced in the latter case. Consequently, the time spent on each kernel function is also reduced.

8.4 Discussion

In this section we have tried to reduce the memory transfers which are redundant and achieved performance gain in terms of time spent on memory transfers. However, there are many memory transfer which are still happening. In the subsequent section we try to reduce the time spent on data transfers.

9 Optimization 6: Using Pinned Memory

9.1 Description

We have seen in the previous section that there are substantial memory transfer that happen which cannot be avoided. To solve the problem we chose to reduce the time taken for memory transfers with pinned memory. The necessary data that needs to be copied to the device is first copied to the pinned memory and move to the device from it. Using pinned memory show a significant bandwidth utilization.

9.2 Design and Implementation

We started of by finding out which data is being transferred frequently and their sizes. Depending on this data we tried to use pinned memory in the most applicable cases where the performance would be a real gain. Figure-7 [2] shows a graph with amount of data to be transferred with the time it takes to transfer that data.

9.3 Profile

Our benchmarking results from this optimization are graphed in the following figure.

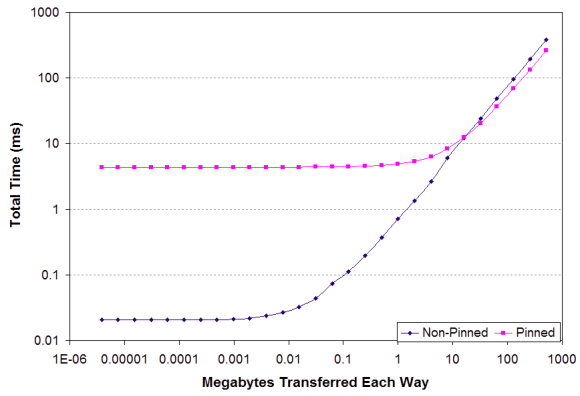

Figure 7: Amount of data transfer vs. Time taken [2]

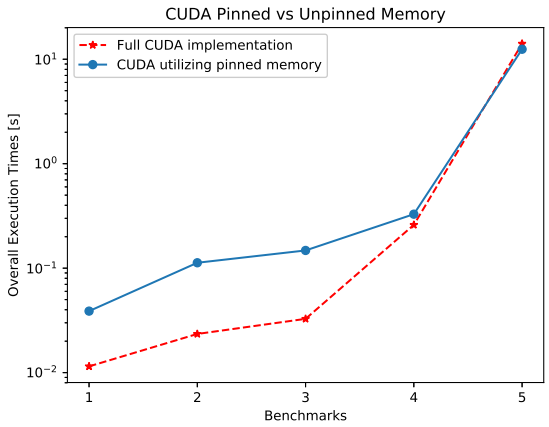

Figure 8: Pinned vs Unpinned Memories

From Figure-7, it can be concluded that pinned memory transfers for small data sizes has worse performance than paged memory transfers. This has been observed in the experiment which uses different benchmarks with varying sizes of dataset elements.

The bandwidth variation when pinned memory is used is shown in the Table-6.

Mem. Type	Bandwidth
Paged memory	3.2834 GB/s
Pinned memory	10.253 GB/s

Table 6: Bandwidths for transferring 168MB dataset

9.4 Estimation

One of the many possible optimizations to increase performance, involves converting complex functions to simpler ones. An example like that is the power function being used in the *updateLabels()* which has been replaced by a simple multiplication. This change offered a speedup of 2 to our implementation for the reason that a single multiplication is a more fundamental and simple instruction down to the ISA level, rather then using the power function which requires specific ISA instructions more complex and less effective for small calculations.

9.5 Results from multiplication optimization

Benchmarking the program using the power function and with multiplication instead produced the following results:

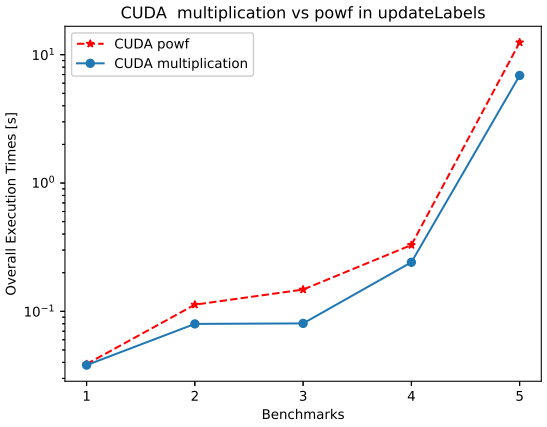

Figure 9: CUDA Pinned powf vs mul.

As seen on figure 9, the speedup is phenomenal for such a small modification. This discovery led us to think about other mathematical function calls and look into possibilities to optimize them.

10 Memory Optimizations in CUDA

10.1 Description

Despite CUDA offering a great advantage in terms of speed and throughput, implementing it in a naive way it is definitely not the optimal solution. The reason for this is that CUDA imports a whole new set of properties and features that can be used and combined in such a way, in order to produce optimal results. Even though different applications benefit by different combinations of these features, it is universally accepted that optimizing the memory usage plays a crucial role. In our implementation of the k-means algorithm we have been experimenting with many different memory optimizations, some of them yielding remarkable results in theory but had little contribution in practice, while others had a negative impact on performance. In the following paragraphs we will discuss these optimizations and provide some thorough explanations.

10.2 Usage of faster memory

Our final efforts on memory optimization were focused on using different types of memory that would fit specifically to our algorithm and data. For that reason we decided to use the GPU constant memory in order to store the centroid data. Constant memory is a special off-chip memory which is being cached, is read-only and can be accessed both by the host and the thread pool. It has a size of 64kB and is the faster available memory after the register file and the shared memory. Our centroid data, according to nvprof tool,

Grid Size	Runtime (s)	Iterations
32	5,33	102
64	6,81	101
128	6,80	101
256	6,84	102
512	6,91	101
1024	6,85	102

Table 7: Runtime without Constant memory

Grid Size	Runtime (s)	Iterations
32	6,59	103
64	9,11	101
128	9,46	101
256	9,46	102
512	9,49	102
1024	9,43	103

Table 8: Runtime with Constant memory

have a size of around 5kB so they could be stored in that memory. The logic of using that memory was that the centroid data are being accessed at least 24 times in the *CUDA_updateLabels()* function so it is a resource that can be used in a more efficient way.

Another thing that should be noted though, which is that the constant memory has some actual advantage when its data are being used multiple times. For that reason we implemented a "randomization" mechanism which makes different threads access different parts of the centroids allowing it to cache more data in small number of iterations. Thereby increasing the rate of cache-hits.

10.2.1 Constant Memory Observed Effects During the usage of constant memory we had some interesting remarks. Its usage doesn't seem to produce better results at all, since in all cases we observed a slower runtime. Our explanation about that involves cache memory misses or cache memry probably not being utilized at all because of one-time access to the data.

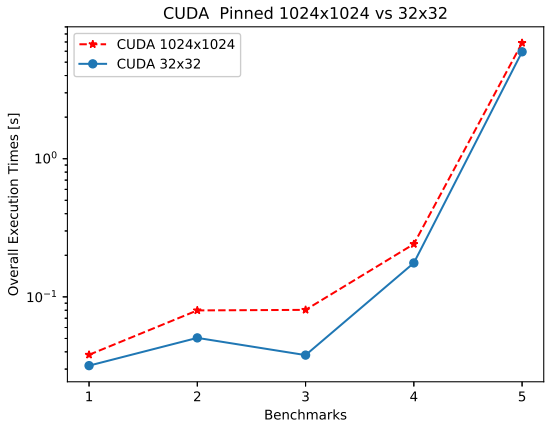

Figure 10: Graph comparison of 32 vs 1024 threads

Here we can see the difference between running 32 threads and 1024. The results are astounding. As this was the final optimization we made successfully, let us look at how they progressed in a single graph.

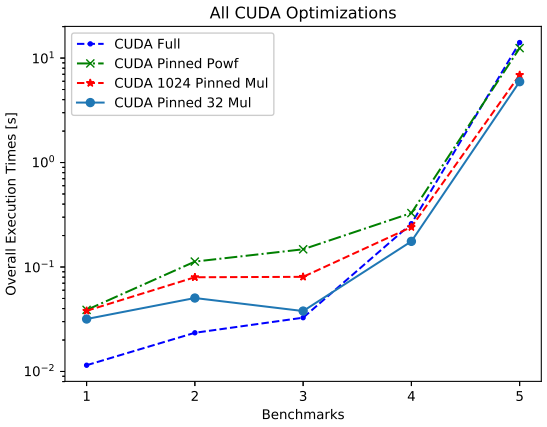

Figure 11: All CUDA Optimizations

11 Other Considerations and Remarks

11.1 Asynchronous transfers

The problems being solved most of the time by CUDA kernels are data-intensive which means that big amounts of data need to be transferred often to and from the GPU. In the most basic kernel implementations those data transfers are being done sequentially the one after the other, with calculations being done in the mean time. So a sequence of operations like this would look like in the Figure 7

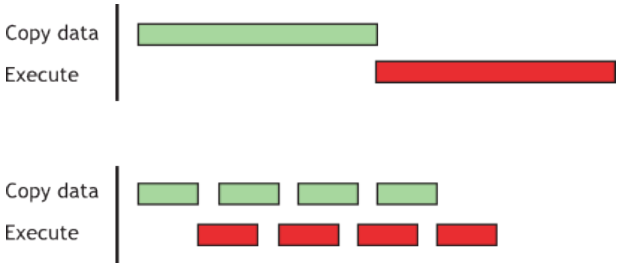

Figure 12: Synchronous vs Asynchronous streams

In order to create asynchronous transfers, we need to create multiple CUDA streams, so each stream will be responsible for one data transfer or one kernel execution. We can have multiple asynchronous transfers, or transfers and computations at the same time, as long as each one of them lies on a different stream. In our application we weren't able to utilize effectively multiple streams, but this could be done if the algoritmh itself could be modified.

11.2 Algorithm Enhancements

As we noticed throughout the lab `updateLabels` was the most time consuming component of the overall execution, we focused primarily on optimizing it as much as possible. After having discovered how much calling `powf` cost us we were able to replace it with simple

multiplication. Such a simple modification provided a significant speedup. From that point we decided it would be worth our time to investigate whether it would be possible to use a different algorithm to calculate a square root than using `sqrtf`. Our brief research in square root algorithms led us to a very interesting way to calculate the inverse square root of a float. The algorithm is called "Fast inverse square root" which originated from the video game Quake III Arena, released in 1999. The inverse square root is calculated by utilizing the structure of floating-point numbers and doing some peculiar operations with them. Although we managed to test and get accurate computations with this method, the number of iterations were way off. Thus we did not pursue this approach since it most likely would take too much time to fix. Other square root algorithms were tested but to no avail.

12 Verification

In every stage of our implementation, we plotted the points and labels of the benchmark with two features to visual verify that our results obtained are correct. Once we had that established, we started benchmarking for larger datasets. Later on, we manually looked into the CSV files generated and checked whether the labels have matched. Although, we have noticed that the performance was increased by using 32 threads, we were not able to verify it exhaustively. Hence, we have chosen 1024 thread as the final version in order to not compromise on the data integrity.

13 Conclusion

The first part before any memory optimization, is identifying the hot-spots of the algorithm, which functions and routines cause the greatest memory load. In the case of k-means algorithm, the first and most obvious hot-spot is the data transfer from the host (CPU memory) to the device (GPU memory). In our particular case we had to transfer the entire dataset from the host to the device, which translates roughly in transferring 168 MBytes of data. The throughput is being limited by the bandwidth of the PCI express bus (approximately 16 GB/s). By profiling the naive implementation using the nvprof tool, we found out that during the initial transfer the maximum throughput was approximately 1.3665GB/s, which is way lower than the actual bus capabilities.

The first optimization we did in order to tackle this problem was to make usage of the *page-locked* or pinned memory. The logic behind pinned memory is quite simple and boils down to reserving part of the system memory and make it unavailable to other processes, by page-locking it. In that data are always available in ram memory and are never being paged. By using this mechanism we managed to achieve a higher throughput in transferring the dataset to the device. Practically, this is implemented by using *cudaMallocHost()* instead of a normal *malloc()* and two subsequent copies, one from the paged memory to the pinned, and one from the pinned memory to the GPU.

In terms of performance, this technique has yielded approximately eight times faster memory transactions, with a throughput of 10,7344 GB/s. Using this new implementation as a basis we can proceed with further optimizations.

References

[1] Mark Harris *How to Optimize Data Transfers in CUDA C/C++*
Guide on using and profiling pinned memory in CUDA.
https://devblogs.nvidia.com/how-optimize-data-transfers-cuda-cc/

[2] Michael Boyer *Choosing Between Pinned and Non-Pinned Memory*
Analyzing effectiveness of pinned memory.
https://www.cs.virginia.edu/ mwb7w/cuda_support/pinned_trad

[3] Robert Crovella *Using Constant Memory*
NVIDIA discussion on usage of constant memory.
https://devtalk.nvidia.com/default/topic/910290/using-constant-memory/

[4] Mark Harris *Using Shared Memory in CUDA C/C++*
NVIDIA devtalk on shared memory.
https://devblogs.nvidia.com/using-shared-memory-cuda-cc/

[5] Steve Rennich *CUDA C/C++ Streams and Concurrency*
NVIDIA slides on usage of streams and concurrency in CUDA
https://developer.download.nvidia.com/CUDA/training/Streams

[6] Wikipedia *Fast Inverse Square Root*
https://en.wikipedia.org/wiki/Fast_inverse_square_root