KNN Optimization on CUDA

Javier Grandio
Computer Science Department, Virginia Commonwealth University, Richmond, USA
grandiogj@vcu.edu

***Abstract*—This assignment presents an implementation of the KNN classifier using Parallel programming with CUDA to speed up the process. It compares the runtime of the single-thread CPU version, MPI version and GPU parallel version, considering two different datasets sizes. Finally, it shows a comparison between different implementations with CUDA.**

I. Introduction

The computational complexity of a KNN classifier is $O(n^2d)$, being n the number of instances, and d the number of attributes of each instance.

Working with a CPU, the limitation of threads is the main constraint. However, working with the GPU allows to work with a number of threads bigger than n^2d, so ideally the problem could be reduced to a complexity of O(1).

The following approaches using the GPU were studied:

1. One thread to compute every instance. It would reduce the complexity by n.
    a. Working with a single stream.
    b. Working with multiple streams.
2. One thread for every distance of every instance. It reduces the complexity by n^2, but a new array has to be allocated (memory allocation also consumes time).
    a. Working with a single stream.
    b. Working with multiple streams.
3. One thread to compute the difference of every attribute of every distance. It would reduce the complexity by n^2d. But since d here is very small, it does not make a big difference.

II. Runtime Study

In this section, the results obtained in the first assignment are compared to the runtimes obtained while working with the GPU. The runtime is only measured during the execution of the KNN algorithm itself, because it is the part of the code that has been optimized in all cases. Also, the algorithm is tested with k = 1 to test the runtimes, and as runtime value is chosen the average over 5 different runs of each approach. The results are shown in Table 1.

As can be seen in Table 1, the small dataset runtime is shorter for the single threaded version. This is due to the overhead needed to start the parallelization, which in this case can be observed that is smaller for the CUDA implementations than for the MPI implementation.

On the other side, while working with the medium dataset, the results are much different. In this case, the runtimes for all the CUDA implementations are almost the same than for the small datasets, achieving in all the cases speedups around 50. From these results can be inferred that the running time working with the GPU is not comparable to the time needed to allocate arrays, even working with the medium dataset, so the improvement that could be achieved with one CUDA implementation with respect to the others CUDA implementations is not possible to measure just looking at the runtimes. In consequence, in order to study the differences between approaches, the results of the performance profiler will be shown in the next section.

Lastly, it is convenient to highlight that in this case, working with GPU, the speedups obtained are greater than the maximum speedup calculated in the last assignment for MPI (30), mainly because the overhead to parallelize with GPU is smaller. However, these speedups can be also interpreted as maximum speedups for GPU, because the computation time is insignificant compared to the overhead (Getting same runtimes for small and medium datasets).

III. Profiler Study

To study the behavior of the CUDA approaches using the profiler, all the tests are run using the medium dataset and a k=1.

The first approach studied is the 1. a. This approach is the simplest one, working only with one stream and an only level of parallelization.

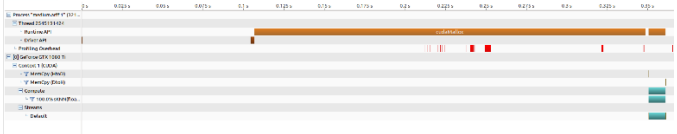

Fig. 1. Timeline for approach 1.a.

Table 1. Runtimes and Speedups

		Single-thread	MPI 16 processes	Cuda 1. a.	Cuda 1. b.	Cuda 2. a.	Cuda 2. b.	Cuda 3.
Small Dataset	Runtime (ms)	30	360	170	160	173	165	174
	Speedup	1.000	0.083	0.176	0.188	0.173	0.182	0.172
Medium Dataset	Runtime (ms)	9000	1300	170	165	163	174	183
	Speedup	1.000	6.923	52.941	54.545	55.215	51.724	49.180

In Fig 1. is shown the timeline for the run of this approach. An interesting fact about this timeline, is that as guessed in the previous section, the longest part is used to allocate memory, and the run of the kernel (blue line) is only about 10 ms.

sKNN(float*, int*, int*, int, int, int, float*, int*)	
Queued	n/a
Submitted	n/a
Start	350.54803 ms (350,548,03
End	361.1151 ms (361,115,100
Duration	10.56707 ms (10,567,070 r
Stream	Default
Grid Size	[20,1,1]
Block Size	[256,1,1]
Registers/Thread	32
Shared Memory/Block	0 B
Launch Type	Normal
▾ Efficiency	
Global Load Efficiency	⚠ 13.5%
Global Store Efficiency	⚠ 20.5%
Shared Efficiency	n/a
Warp Execution Efficiency	99.1%
Not-Predicated-Off Warp Execution Efficiency	93.5%
▾ Occupancy	
Achieved	⚠ 12%
Theoretical	100%
▾ Shared Memory Configuration	
Shared Memory Executed	0 B
Shared Memory Bank Size	4 B

Fig. 2. Kernel details 1.a.

In Fig. 2, the kernel details show that the load and store efficiency have low values. However, since the computation time is not comparable to memory allocation time, for this size of problem the time that takes to optimize the code is not worth the performance improvement.

Working with 1.b. the only difference is that 4 streams are used, however, since every stream has to have loaded in device memory all the instances and attributes when they start, an asynchronous load is not possible and the runtime is the same that for the first case, because all the streams have to start at the same time. Only the loading of the memory to the host can be done asynchronously, however, since all the streams will finish the kernel at about the same time and the load is for all of them in the same direction, no advantage is achieved with this approach.

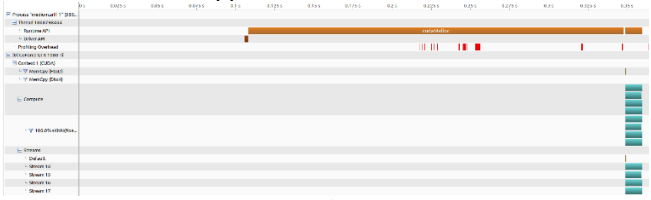

Fig. 3. Timeline for approach 1.b.

The timeline shown in Fig 3. reinforces the previous explanation, showing that the only difference is the number of streams.

Approach 2.a. adds a new level of parallelization, calculating all the distances with different threads.

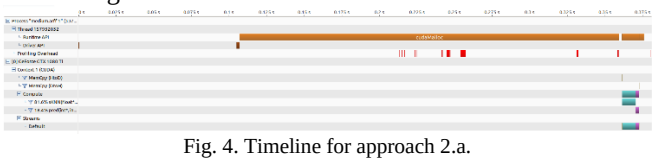

Fig. 4. Timeline for approach 2.a.

In the timeline of Fig. 4 can be seen that for this case two different kernels are used, but the main part is also for memory allocation.

sKNN(float*, int*, int, int, float*, int, float*)	
Queued	n/a
Submitted	n/a
Start	361.31459 ms (361,314,58
End	370.52826 ms (370,528,26
Duration	9.21368 ms (9,213,679 ns)
Stream	Default
Grid Size	[307,307,1]
Block Size	[16,16,1]
Registers/Thread	23
Shared Memory/Block	0 B
Launch Type	Normal
▾ Efficiency	
Global Load Efficiency	⚠ 12.5%
Global Store Efficiency	⚠ 60.8%
Shared Efficiency	n/a
Warp Execution Efficiency	99.8%
Not-Predicated-Off Warp Execution Efficiency	93.7%
▾ Occupancy	
Achieved	93.4%
Theoretical	100%
▾ Shared Memory Configuration	
Shared Memory Executed	0 B
Shared Memory Bank Size	4 B

Fig. 5. Kernel details 2.a.

pred(int*, int, int, float*, float*, int*, int*)	
Queued	n/a
Submitted	n/a
Start	370.5291 ms (370,529,096
End	372.60194 ms (372,601,93
Duration	2.07284 ms (2,072,843 ns)
Stream	Default
Grid Size	[20,1,1]
Block Size	[256,1,1]
Registers/Thread	32
Shared Memory/Block	0 B
Launch Type	Normal
▾ Efficiency	
Global Load Efficiency	90.7%
Global Store Efficiency	⚠ 19.6%
Shared Efficiency	n/a
Warp Execution Efficiency	97.8%
Not-Predicated-Off Warp Execution Efficiency	88%
▾ Occupancy	
Achieved	⚠ 12%
Theoretical	100%
▾ Shared Memory Configuration	
Shared Memory Executed	0 B
Shared Memory Bank Size	4 B

Fig. 6. Kernel details 2.b.

As seen in Fig. 5 and Fig. 6, besides getting a new parallelization layer with this approach, the Global Store Efficiency is improved for one kernel, and the Global Load Efficiency is improved for the other. Taking into account this factor, can be inferred that for bigger datasets this new approach would be faster than the first one.

When working with approach 2.b. happens the same than with 1.b. So for the solution coded in this assignment, it is not worth to use several streams.

Lastly, approach 3. adds a new level of parallelization and the usage of shared memory. This solution was designed specifically for this case and is not very scalable, so if it was wanted to work with much bigger datasets, some coding optimization should be done. With this new approach, the percentage of memory utilization is similar to case 2, so the only improvement would be obtained because a new parallelization layer is added.

IV. Conclusions

After studying all the results, in the first place can be observed that the GPU implementation runtime is not comparable to the CPU implementation when the dataset size grows. Also, it can be seen that the medium dataset size is small for the GPU implementation, so the naïve approach would be enough, and the time to optimize the code is not worth the runtime improvement in this case. However, for much bigger datasets, it would be worth to optimize the code to take advance of all the computational resources of the GPU, the capability to work with different streams and improve memory efficiency.