CUDA Assignment 2

Kat Young
High Performance Distributed Systems
Virginia Commonwealth University
youngk6@vcu.edu

I. Original Design Approach

Generally speaking, I implemented KNN in CUDA in a similar algorithm to my implementation of KNN in MPI and sequential KNN. This assignment was different from the others, however, in that the functions we were able to use with the Arff Dataset were not able to be used in the kernel since they were host functions. Due to this, I had to convert the Arff Dataset into a 1D array, which would be able to be indexed in the kernel. I then got metrics that would be useful for indexing the various arrays that I would be using in the kernel such as instancecount, attributecount, and classcount which kept track of how many instances, attributes, and classes there were in the data respectively.

I had four arrays that were important in the kernel for computations: predictions (which was where the final predictions would be placed), Kdist (which was to hold the k nearest distances for each instance), Kclasses (which was to hold the corresponding classes for the k nearest distances in Kdist), and classVotes (which was to keep track of the votes for each class after all of the nearest distances were found). My KNN kernel function ended up taking dpredictions (results array), ddataset (dataset as a 1D array including both attributes and classes), k (nearest neighbors), instancecount, attributecount, classcount, dKdist, dKclasses, dclassVotes, inf (the value infinity for edge case use).

In the kernel, I began by getting the thread ID (tid). The first goal for the kernel was to get the k-nearest-neighbors. I accomplished this by filling up the positions in Kdist and Kclasses with the distances and classes of the first k instances in the dataset, keeping track of the largest distance in the array. As the algorithm continued to iterate through all of the other instances, if it found one of the instances had a distance from the instance being measured that was less than the largest distance in that instance's nearest neighbors array, it replaced the old distance and class with the class of the closer neighbor. The largest distance was recalculated after each time this replacement happened.

A problem occurred, for example, when tid was less than the k value. The distance from that instance to itself was zero, and therefore would never get replaced because there would always be distances greater than zero in the nearest neighbors. To handle this problem, when the first k elements of the nearest neighbors array were selected and tid ¡ k, the distance associated with tid was infinity so that as a result that value would be replaced immediately when the algorithm started replacing elements. When tid ¿ k, the algorithm was told to not look at elements where tid = j (j being the instance we are comparing the original instance with).

The result of this process was an array that contains the distances and an array that contains the classes of the k nearest neighbors for each instance. The next goal of the kernel was to count the votes and find the class with the most votes to be considered the prediction. Classes were voted on by going through the kclasses array for each instance (size k) and incrementing the index position in the classValues array. For example, if an instance's class votes looked like this: [1,0,4,5,1] for k = 5, then the classVotes block allocated for that specific tid (assuming there are 8 classes) would look like this: [1,2,0,0,1,1,0,0]. Since index 1 has the highest value, the most votes went to classify the instance as 1, which would be placed in the predictions array in the spot corresponding to the tid as our prediction.

II. Kernel Configuration, Experimentation, and Results

To experiment with kernel configurations, I looked at changing the number of threads per block and changing my original one kernel to two kernels. To change the configuration to two kernels, all I did was move the class prediction to its own kernel. The first kernel put together the large array of nearest neighbors, and the second kernel essentially counted the votes. I decided to try this because kernel launch cost is negligible and it could potentially be faster due to fewer registers used in each one – more resources. Measuring improvements was somewhat hard because all of the times were in the same general range. I took the average of the first five runs to get the resulting numbers.

When looking at the first implementation I did with just one kernel, I tried two configurations of threads and blocks before I moved to trying a different number of kernels. I tried 256 threads per block as well as 1 thread per block. One thread per block performed better on the small dataset than 256 threads per block, while the reverse was true for the medium dataset.

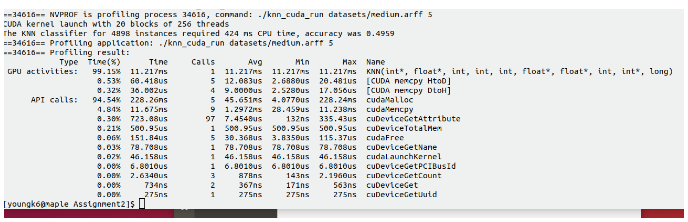
When implementing two kernels, I tested first on 256 threads per block and saw improvements in the times. Then I tried other configurations of threads per block including 1 thread per block, 700 threads per block, 512 threads per block, and 32 threads per block. The results can be seen in the table below, but the best results were from two kernels with 256 threads per block. The differences in times however weren't very large for these datasets. It would potentially be helpful to run these configurations on a massive dataset and see how each performs.

Run Times (ms)			
Method		**small.arff**	**medium.arff**
Sequential		56 ms	9035 ms
MPI	**2 cores**	266 ms	4823 ms
	4 cores	288 ms	2610 ms
	8 cores	318 ms	1634 ms
	16 cores	344 ms	1277 ms
CUDA (one kernel)	**256 threads / block**	179 ms	173 ms
	1 thread / block	149 ms	200 ms
CUDA (two kernels)	**256 threads / block**	117.2 ms	142 ms
	1 thread / block	154 ms	193 ms
	700 threads / block	153 ms	158 ms
	512 threads / block	157 ms	156 ms
	32 threads / block	156 ms	154 ms

Fig. 1. Approximate run times

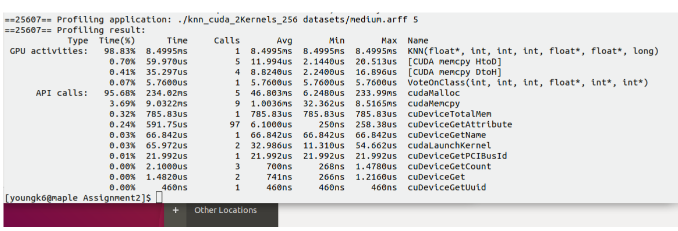
Running the code with the profiler, looking at the code written in one kernel, an average of 11.217 ms was spent in the KNN kernel. Running with two kernels, the first kernel cut its time down to 8.499 ms, and the second kernel runs in 5.7 us, which is equivalent to 0.0057 ms, which means the total time spent in the kernel was reduced by this change.

This was the profiler report from the initial configuration I had with 256 threads per block and one kernel.



```
==34616== NVPROF is profiling process 34616, command: ./knn_cuda_run datasets/medium.arff 5
CUDA kernel launch with 20 blocks of 256 threads
The KNN classifier for 4898 instances required 424 ms CPU time, accuracy was 0.4959
==34616== Profiling application: ./knn_cuda_run datasets/medium.arff 5
==34616== Profiling result:
            Type  Time(%)      Time     Calls       Avg       Min       Max  Name
 GPU activities:   99.15%  11.217ms         1  11.217ms  11.217ms  11.217ms  KNN(int*, float*, int, int, int, float*, float*, int, int*, long)
                    0.53%  60.418us         5  12.083us  2.6880us  20.481us  [CUDA memcpy HtoD]
                    0.32%  36.002us         4  9.0000us  2.5280us  17.056us  [CUDA memcpy DtoH]
      API calls:   94.54%  228.26ms         5  45.651ms  4.0770us  228.24ms  cudaMalloc
                    4.84%  11.675ms         9  1.2972ms  28.459us  11.238ms  cudaMemcpy
                    0.30%  723.08us        97  7.4540us     132ns  335.43us  cuDeviceGetAttribute
                    0.21%  500.95us         1  500.95us  500.95us  500.95us  cuDeviceTotalMem
                    0.06%  151.84us         5  30.368us  3.8350us  115.37us  cudaFree
                    0.03%  78.708us         1  78.708us  78.708us  78.708us  cuDeviceGetName
                    0.02%  46.158us         1  46.158us  46.158us  46.158us  cudaLaunchKernel
                    0.00%  6.8010us         1  6.8010us  6.8010us  6.8010us  cuDeviceGetPCIBusId
                    0.00%  2.6340us         3     878ns     143ns  2.1960us  cuDeviceGetCount
                    0.00%     734ns         2     367ns     171ns     563ns  cuDeviceGet
                    0.00%     275ns         1     275ns     275ns     275ns  cuDeviceGetUuid
[youngk6@maple Assignment2]$
```

Fig. 2. 256 Threads / Block and ONE kernel



```
==25607== Profiling application: ./knn_cuda_2Kernels_256 datasets/medium.arff 5
==25607== Profiling result:
            Type  Time(%)      Time     Calls       Avg       Min       Max  Name
 GPU activities:   98.83%  8.4995ms         1  8.4995ms  8.4995ms  8.4995ms  KNN(float*, int, int, int, float*, float*, long)
                    0.70%  59.970us         5  11.994us  2.1440us  20.513us  [CUDA memcpy HtoD]
                    0.41%  35.297us         4  8.8240us  2.2400us  16.896us  [CUDA memcpy DtoH]
                    0.07%  5.7600us         1  5.7600us  5.7600us  5.7600us  VoteOnClass(int, int, int, float*, int*, int*)
      API calls:   95.68%  234.02ms         5  46.803ms  6.2480us  233.99ms  cudaMalloc
                    3.69%  9.0322ms         9  1.0036ms  32.362us  8.5165ms  cudaMemcpy
                    0.32%  785.83us         1  785.83us  785.83us  785.83us  cuDeviceTotalMem
                    0.24%  591.75us        97  6.1000us     250ns  258.38us  cuDeviceGetAttribute
                    0.03%  66.842us         1  66.842us  66.842us  66.842us  cuDeviceGetName
                    0.03%  65.972us         2  32.986us  11.310us  54.662us  cudaLaunchKernel
                    0.01%  21.992us         1  21.992us  21.992us  21.992us  cuDeviceGetPCIBusId
                    0.00%  2.1000us         3     700ns     268ns  1.4780us  cuDeviceGetCount
                    0.00%  1.4820us         2     741ns     266ns  1.2160us  cuDeviceGet
                    0.00%     460ns         1     460ns     460ns     460ns  cuDeviceGetUuid
[youngk6@maple Assignment2]$
```


Fig. 3. 256 Threads / Block and TWO kernels

III. Speedups

Speedups were calculated using the following equation, where the numerator is the time for the sequential version to run, and the denominator is the time for the parallel version to run.

$$Speedup = \frac{T_S}{T_P} \tag{1}$$

Speedups			
Method		**small.arff**	**medium.arff**
MPI	**2 cores**	0.21	1.87
	4 cores	0.19	3.46
	8 cores	0.18	5.53
	16 cores	0.16	7.08
CUDA (one kernel)	**256 threads / block**	0.31	52.23
	1 thread / block	0.38	45.18
CUDA (two kernels)	**256 threads / block**	0.48	63.62
	1 thread / block	0.36	46.81
	700 threads / block	0.37	57.18
	512 threads / block	0.36	57.92
	32 threads / block	0.36	58.67

Fig. 4. Calculated speedups