CS426 Project 4 Report

Rüzgar Ayan
21801984

August 6, 2021

1 Implementation Details

For searching the k-mers in the reference string, I have implemented the brute-force algorithm and the Knuth-Morris-Pratt algorithm that I adapted from the pseudo-code in its Wikipedia page. However, the performance difference was quite low compared to searching the k-mers by brute force. I don't have any optimization for extending the hits and computing the extension score, I just compared the characters one by one whenever there was a hit.

1.1 Serial Implementation

For this one, I have used Knuth-Morris-Pratt algorithm for searching the k-mers. Also, different from the parallel implementation I have used a hash-table as suggested in the project document. With this hash-table, I store the hit locations for every different k-mer in this hash-table so that whenever that k-mer is seen again, we don't need to search the reference string again. We just use these hit locations to extend at these locations and compute the extension scores.

We cannot just store the best extension scores in this hash table, because it depends on the read string as well as the k-mer. It would be also possible to maintain hash tables per read string for that purpose, but I did not want to make it that complex. Because I assumed the reference strings would generally be much longer than the read strings, thus the more expensive operation is the searching , not extending.

Note 1: In the file *kmer_serial.cu* there is the function *searchAndExtendSingleRead* that uses the explained hash table to increase the performance. There is also a commented out function *searchAndExtendSingleReadNoHashTable* that you can use to see the differences.

Note 2: I have used the Hash table implementation from https://gist.github.com/phsym/4605704 in the file *hashtable.c*. When compiled with *nvcc*, some parts of it did not work because I guess *nvcc* does not accept the usage of *void ** in some cases and it was causing an error while using *free* on a void pointer. Because of that, I could not use the *destroy* function of the Hash Table. But I was going to use it in the end the program, so I think it does not matter anyway.

1.2 Parallel (CUDA) Implementation

For this program, I was either going to use brute-force searching or Knuth-Morris-Pratt. For Knuth-Morris-Pratt, a transition array of size k is required for each k-mer, that is for each CUDA thread. I could either dynamically allocate this memory or use arrays of fixed size 200 (max size of k-mers). I thought both ways could hurt the performance since there will be a lot of threads having these at the

same time, so I went with the brute-force string searching for the parallel program.

To divide this problem into tasks, I considered each block computing for a single read string and each thread in a block computing for a single k-mer of that read string. So, what each thread does is first searching through all the reference string for a single k-mer and finding extension scores at hits. Each CUDA thread records the highest extension score that they find into an output array. After that, a host function takes care of choosing the k-mer with the highest extension score for each of the read strings. This last step done by the host does not affect the run time much since it is independent from the size of the reference string and the number of iterations in this step is approximately [Length of a read string] $\times$ [Number of read strings]. Another kernel function might be implemented and called to handle this step as well, but I didn't thought this is necessary as we already got a very good performance increase.

All the read strings are copied into the device as a single 1D flattened char array. Similarly, the output array of the device that contains the solutions for each k-mer of each read string is a 1D flattened array.

2 Nvprof Outputs for Parallel Program

The first output of nvprof is given with a reference string of length 100000 and 1000 read strings. This one is shorter with about 700 ms execution time compared to the other output.

```
==14276== NVPROF is profiling process 14276, command: kmer-parallel data/reference_100000bp.txt
    data/reads_1000.txt 5 output.txt
==14276== Profiling application: kmer-parallel data/reference_100000bp.txt data/reads_1000.txt 5
    output.txt
==14276== Profiling result:
            Type  Time(%)      Time     Calls       Avg       Min       Max  Name
 GPU activities:  99.98%  667.29ms         1  667.29ms  667.29ms  667.29ms  kmerSearchAndExtend(char*, char*,
     Solution*, int, int, int)
                   0.01%  90.272us         1  90.272us  90.272us  90.272us  [CUDA memcpy DtoH]
                   0.00%  17.823us         2  8.9110us  8.7360us  9.0870us  [CUDA memcpy HtoD]
      API calls:  76.61%  667.35ms         1  667.35ms  667.35ms  667.35ms  cudaDeviceSynchronize
                  18.71%  162.96ms         3  54.319ms  3.7000us  162.95ms  cudaMalloc
                   4.59%  39.950ms         1  39.950ms  39.950ms  39.950ms  cuDevicePrimaryCtxRelease
                   0.07%  610.90us         3  203.63us  40.400us  387.80us  cudaMemcpy
                   0.01%  50.300us       101     498ns     400ns  1.8000us  cuDeviceGetAttribute
                   0.01%  46.200us         1  46.200us  46.200us  46.200us  cudaLaunchKernel
                   0.00%  40.200us         1  40.200us  40.200us  40.200us  cuDeviceTotalMem
                   0.00%  28.900us         2  14.450us  3.5000us  25.400us  cuModuleUnload
                   0.00%  11.400us         1  11.400us  11.400us  11.400us  cuDeviceGetPCIBusId
                   0.00%  3.8000us         3  1.2660us     300ns  3.0000us  cuDeviceGetCount
                   0.00%  3.7000us         2  1.8500us  1.1000us  2.6000us  cuDeviceGet
                   0.00%  2.0000us         1  2.0000us  2.0000us  2.0000us  cuDeviceGetName
                   0.00%     800ns         1     800ns     800ns     800ns  cuDeviceGetLuid
                   0.00%     700ns         1     700ns     700ns     700ns  cuDeviceGetUuid
```

We can see that %76 of the time is spent in *cudaDeviceSynchronize* which is actually the time spent in the kernel function. %18 of the time is spent with allocating memory in the device, which is surprisingly high that wouldn't be the case in an equivalent serial program. Looking at the Avg, Min and Max values for that, we actually see that almost all the time is spent in only 1 of the 3 *cudaMalloc* calls. There wasn't a very big difference between the sizes of allocated memories in these 3 *cudaMalloc* calls, so I predict that the first cudaMalloc requires some kind of a startup time. I couldn't find out what *cuDevicePrimaryCtxRelease* exactly is, but I believe it is for going back to the executing in CPU after finishing the kernel in GPU. Finally, we see that *cudaMemcpy* calls are very cheap, at least compared

to the *cudaMalloc*.

Next, there is the output of nvprof with a reference string of length 100000 and 9216 read strings. This one is longer with about 5 seconds execution time.

```
==16832== NVPROF is profiling process 16832, command: kmer-parallel data/reference_100000bp.txt
    data/reads_9216_100bp.txt 5 output.txt
==16832== Profiling application: kmer-parallel data/reference_100000bp.txt data/reads_9216_100bp.txt 5
    output.txt
==16832== Profiling result:
            Type  Time(%)      Time     Calls       Avg       Min       Max  Name
 GPU activities:  99.95%  4.76590s         1  4.76590s  4.76590s  4.76590s  kmerSearchAndExtend(char*, char*,
     Solution*, int, int, int)
                   0.05%  2.4863ms         1  2.4863ms  2.4863ms  2.4863ms  [CUDA memcpy DtoH]
                   0.00%  81.313us         2  40.656us  9.1530us  72.160us  [CUDA memcpy HtoD]
      API calls:  95.28%  4.76602s         1  4.76602s  4.76602s  4.76602s  cudaDeviceSynchronize
                   3.82%  191.23ms         3  63.742ms  5.0000us  191.06ms  cudaMalloc
                   0.80%  40.007ms         1  40.007ms  40.007ms  40.007ms  cuDevicePrimaryCtxRelease
                   0.07%  3.3688ms         3  1.1229ms  41.800us  2.7498ms  cudaMemcpy
                   0.03%  1.3921ms         2  696.05us  21.100us  1.3710ms  cuModuleUnload
                   0.00%  52.300us         1  52.300us  52.300us  52.300us  cudaLaunchKernel
                   0.00%  22.400us         1  22.400us  22.400us  22.400us  cuDeviceTotalMem
                   0.00%  19.200us       101     190ns     100ns  1.1000us  cuDeviceGetAttribute
                   0.00%  11.500us         1  11.500us  11.500us  11.500us  cuDeviceGetPCIBusId
                   0.00%  1.6000us         2     800ns     200ns  1.4000us  cuDeviceGet
                   0.00%  1.4000us         3     466ns     300ns     600ns  cuDeviceGetCount
                   0.00%  1.1000us         1  1.1000us  1.1000us  1.1000us  cuDeviceGetName
                   0.00%     500ns         1     500ns     500ns     500ns  cuDeviceGetLuid
                   0.00%     300ns         1     300ns     300ns     300ns  cuDeviceGetUuid
```

The output for *cudaMalloc* is almost the same as before, which is why I suggested it probably has a required startup time that is needed on the first call. Everything seems the same expect the kernel function's execution time which increased as expected.

3 Results

The following experiments were done with:

- CPU: AMD Ryzen 5 2600, 3.4 GHz
- GPU: NVIDIA GTX 1060, 1280 CUDA Cores

All the read strings are of length 100 and R denotes the length of the reference string, N denotes the number of read strings and k denotes the length of k-mers.

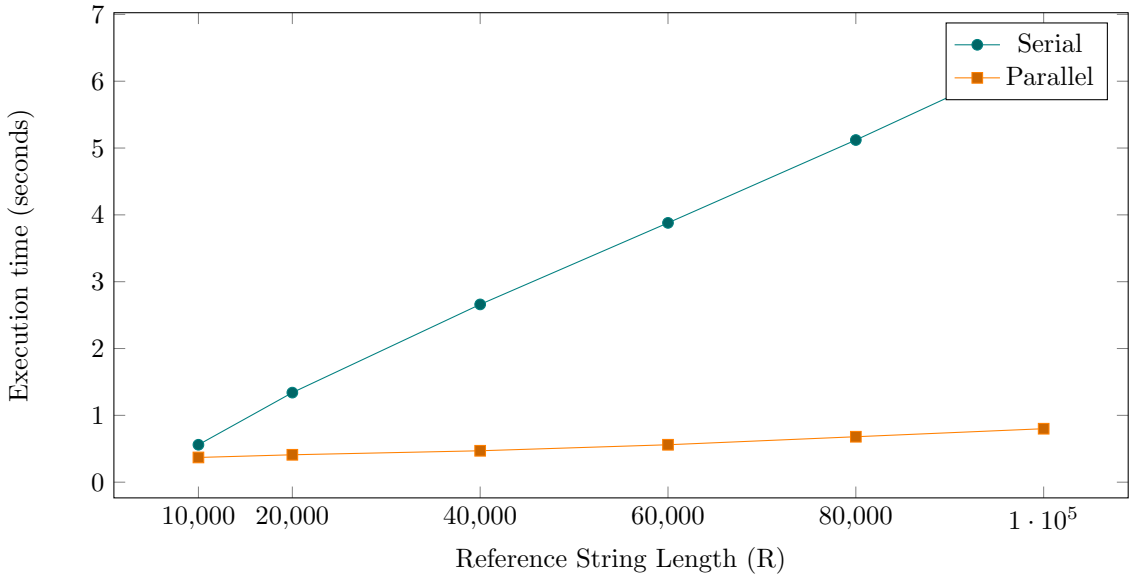

Figure 1: Serial and Parallel Execution Times with $N = 1000, k = 5$ and changing R

For the serial program, we see that 100% increase in R results in approximately 100% increase in the execution time. Since the most expensive part of the program is searching through the reference string, this is expected. It is not very clear in the plot, but the execution time of parallel program increases at a much slower rate. It is 0.37 for $R = 10000$ and 0.8 for $R = 100000$ compared to the 10 times increase in serial case. We see a speedup of about 8 for $R = 100000$ by using the GPU.

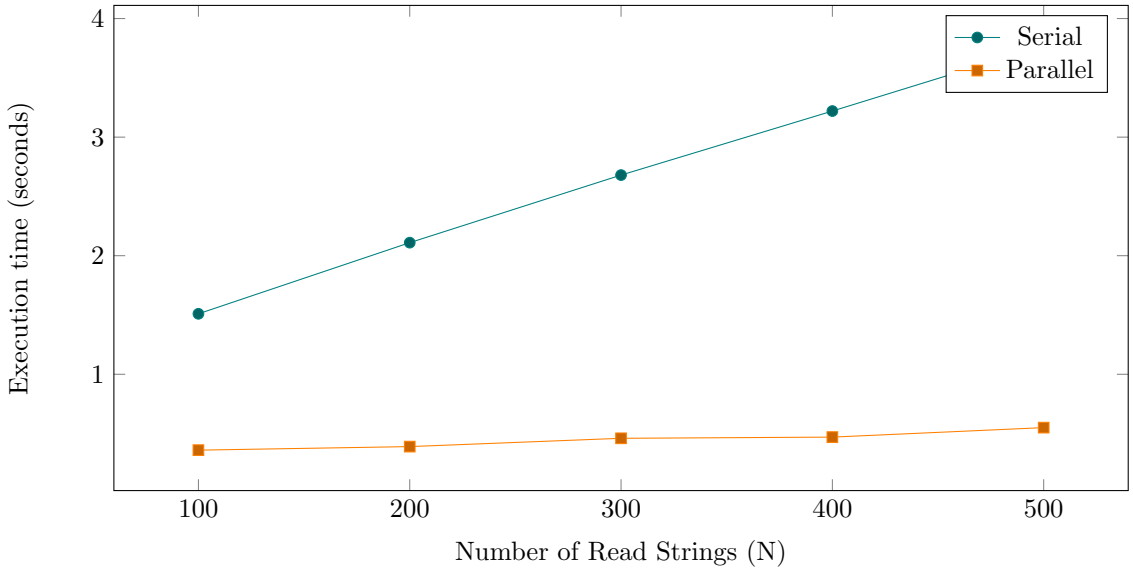

Figure 2: Serial and Parallel Execution Times with $R = 100000, k = 5$ and changing N

For the serial program, we see that 100% times increase in N results in approximately $40 - 50\%$ times increase in the execution time. Compared to the previous plot, this is a smaller increase and the reason is the used hash tables. Even if we add more and more read strings, some k-mers in them are already searched in the reference string and their hit locations are stored. So, we don't need to do the searching part for some k-mers of the newly added read strings. Again, the execution time of parallel program increases at a much slower rate. It is 0.36 for $N = 100$ and 0.55 for $N = 500$ compared to the 2.5 times increase in serial case.

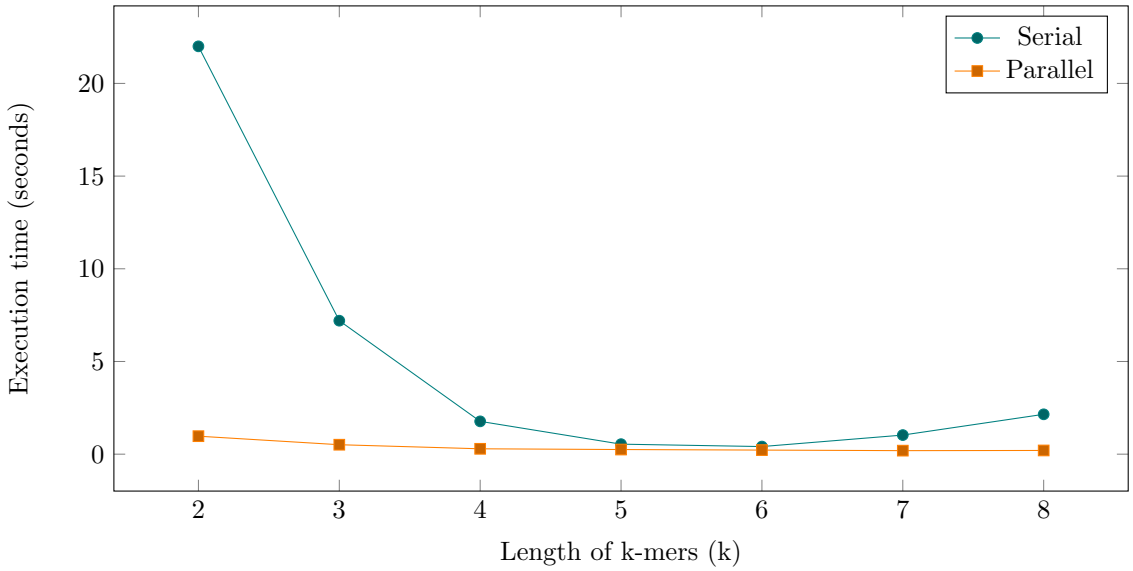

Figure 3: Serial and Parallel Execution Times with $R = 10000, N = 1000$ and changing k

Finally, for the different values of k, we see quite different results. The cases of $k = 2, 3$ are the worst for both cases by far. Although in the serial case, we will have the hit locations for all the possible k-mers in the hash table (because the number of possible k-mers is low for small k) and we will not have to do so many searches, the extension part becomes the bottleneck here. Since there will be a huge number of hits for these small k-mers, we will have to compare the read string with the reference string at all of these hit locations. So, the run times of both programs increase a lot.

We see that $k = 5, 6$ is the best for the serial-case and then the execution time starts to increase again. This is because, there won't be that many hits with these bigger k-mers but still we can effectively use the hash tables to avoid searching reference strings many times. For larger k, the probability of seeing the same k-mers decrease a lot, so we cannot get any performance boost from the hash tables at all.

For the parallel program, this is a decreasing graph, because the probability of hits decrease with larger k-mers.