CS3210 Assignment 2 CUDA Programming A0244219H

Brief Description of Algorithm

The algorithm used was a simple brute force algorithm where each sample is compared with each signature. The signature is compared to each contiguous substring of the sample.

Parallelisation Strategy, Grid/Block Dimensions

The parallelisation strategy was to have every block be a pair of sample and signature, and every thread to work on comparing a section of the sample with the given signature

Grid Dimensions (no. of blocks) ➔ no. of samples (x) * no. of signatures (y) (2D)

Block Dimensions (no. of threads) ➔ no. of threads per block (1D)

The grid dimensional depended on the input's no. of samples and no. of signatures, but the block dimensional would always be 512, following the cuda recommendation of it being a power of 2. 512 for block dimensional seems to be the highest power of 2 possible that provides the fastest time. Going up to 1028, the code fails as the configuration does not allow it, at least for A100-40 MIG GPU.

Memory Management

For memory, between host and device, I used Unified Memory through cudaMallocManaged, for MatchResult vector as it was the only data that was needed to be copied back to host. Global Memory on the gpu was used to store contents of samples and signatures through cudaMalloc, as they were only needed to be transferred to the gpu and not needed at the end of the program.

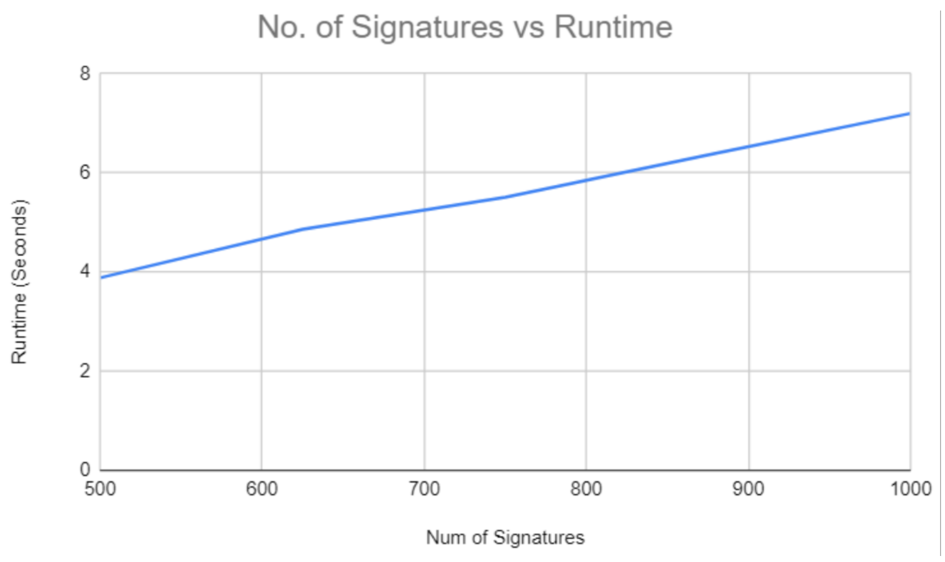
Shared memory was used in each block to store the index of any matches when the threads are comparing contiguous substrings of the sample with the signature. This was so that the threads could see the current matching index and see if the matching index they found is lower than it as we are prioritising matches nearer to the start of the sample string.

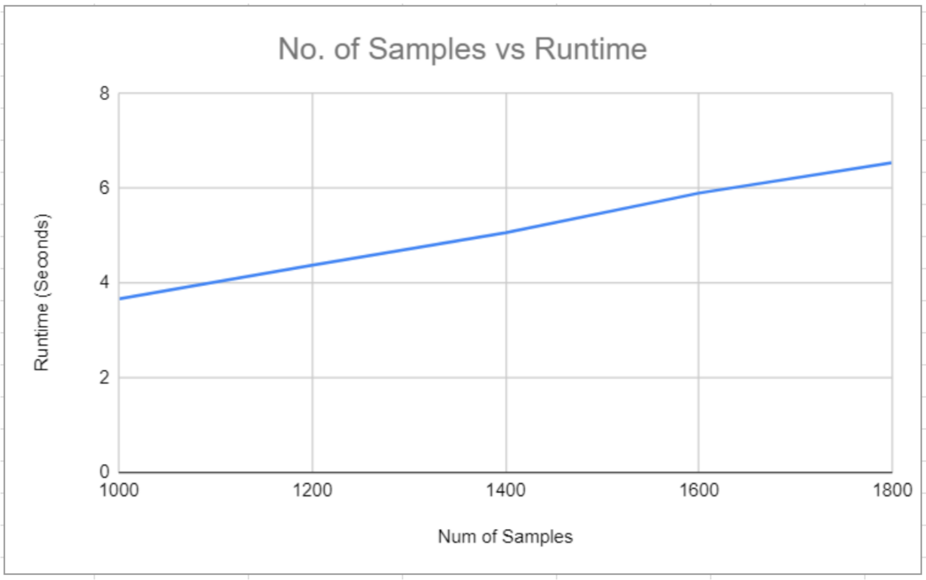
Different Factors of input affecting runtime

1. Number of samples and signatures

   When the number of samples and signatures increases, the runtime increases as expected. There was a linear relationship between the runtime and both number of samples and signature

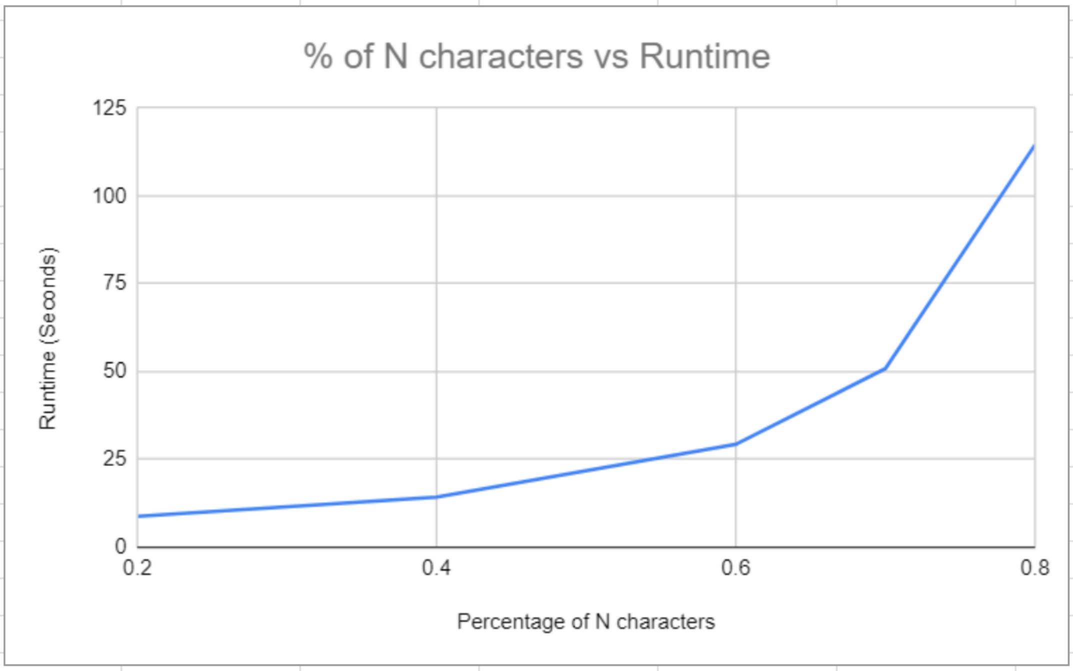
   The rate of increase for an increase in signature was more than that of samples, which is expected as due to the algorithm, more comparison computation would need to be done for that one more signature, as we need to compare it with every contiguous substring of every samples. Whereas for one more sample, we need to compare every contiguous substring of this 1 sample with every signature which is less.





2. Percentage of N characters
   An increase in the percentage of N characters for both sample and signature resulted in an exponential increase in the runtime.
   This is because with every N character added, there will be more matches.

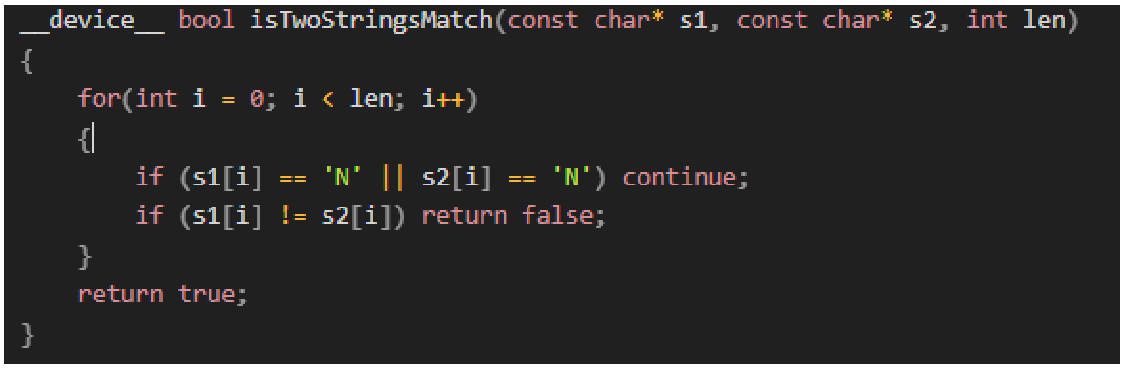
   How the threads determine whether they match they found is the earliest match in the sample is by accessing the shared variable *MatchIndex.
   When more matches are found, this shared variable is accessed more often to be checked and more atomicWrites are done resulting in overhead from synchronization and hence a greater runtime.



Specific Performance Optimizations attempted

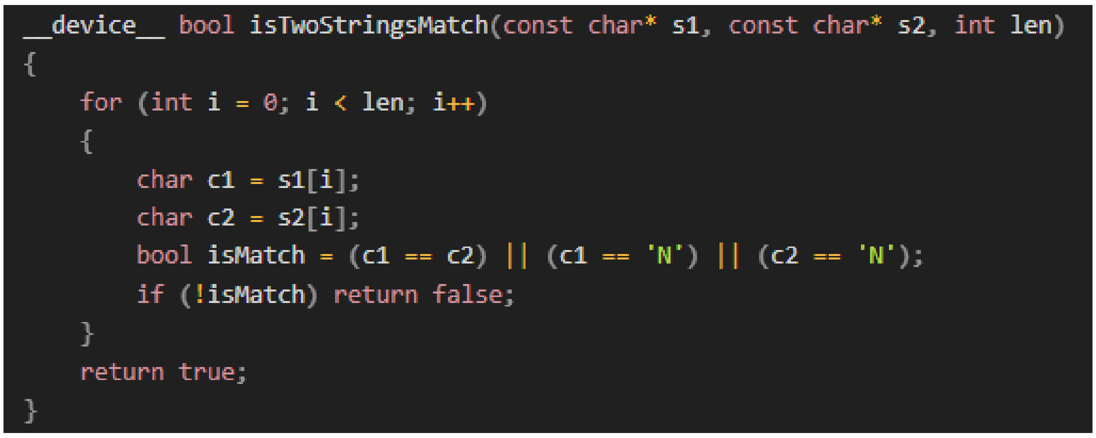
1. Reducing divergence

In the original isTwoStringsMatch() function to check if two strings matched, there were multiple if-else statements



```
__device__ bool isTwoStringsMatch(const char* s1, const char* s2, int len)
{
    for(int i = 0; i < len; i++)
    {
        if (s1[i] == 'N' || s2[i] == 'N') continue;
        if (s1[i] != s2[i]) return false;
    }
    return true;
}
```

In order to avoid divergence within threads in a warp, the if else statements were reduced to

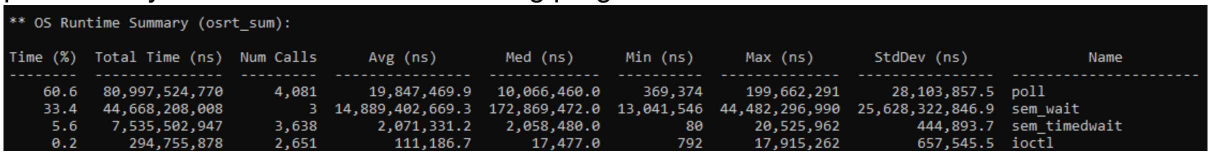


```
__device__ bool isTwoStringsMatch(const char* s1, const char* s2, int len)
{
    for (int i = 0; i < len; i++)
    {
        char c1 = s1[i];
        char c2 = s2[i];
        bool isMatch = (c1 == c2) || (c1 == 'N') || (c2 == 'N');
        if (!isMatch) return false;
    }
    return true;
}
```

However, the runtime speed up did not decrease significantly which prompted me to start using NV tools to optimize.

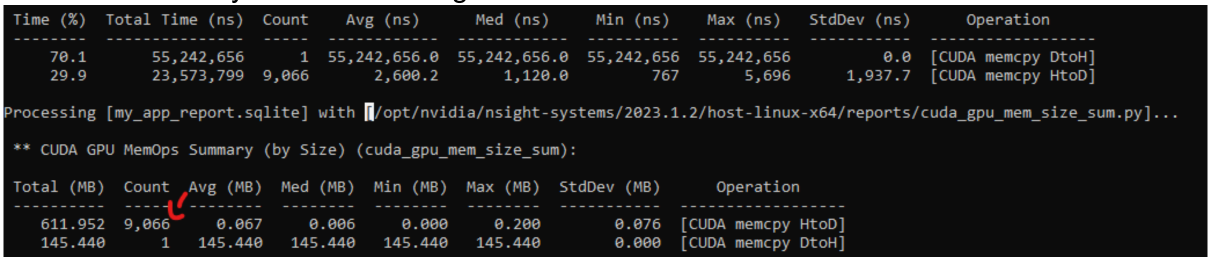
2. Reducing memory transfer delay

To explain the reason for optimizing the memory transfer, I will explain the performance profile analysis I conducted after ensuring program correctness.



```
** OS Runtime Summary (osrt_sum):

Time (%)  Total Time (ns)  Num Calls        Avg (ns)       Med (ns)     Min (ns)       Max (ns)       StdDev (ns)            Name
--------  ---------------  ---------  ---------------  -------------  ----------  --------------  ----------------  ---------------------
    60.6   80,997,524,770      4,081     19,847,469.9   10,066,460.0     369,374     199,662,291      28,103,857.5  poll
    33.4   44,668,208,008          3 14,889,402,669.3  172,869,472.0  13,041,546  44,482,296,990  25,628,322,846.9  sem_wait
     5.6    7,535,502,947      3,638      2,071,331.2    2,058,480.0          80      20,525,962         444,893.7  sem_timedwait
     0.2      294,755,878      2,651        111,186.7       17,477.0         792      17,915,262         657,545.5  ioctl
```

Using Nvidia Night Systems (nsys), I profiled the OS runtime summary and noticed that the code spends a lot of time “polling”. After googling, I realised that in CUDA this is probably due to the communication between the CPU and GPU which prompted me to look at the memory transfer. My initial code would do CudaMemCpy() in a for loop on small chunks which was the result of the long communication time, as each call to cudaMemCpy() had overheads and my code was calling it 9066 times.



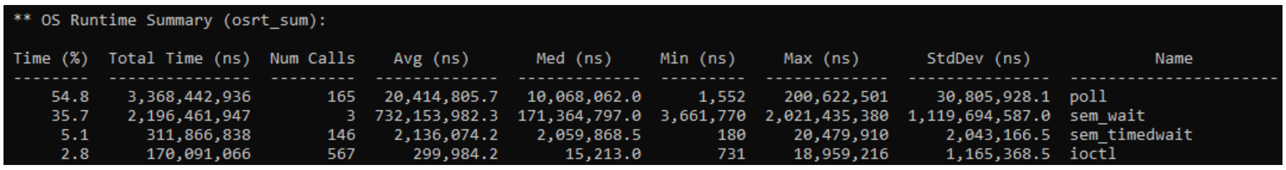
```
 Time (%)  Total Time (ns)  Count    Avg (ns)      Med (ns)      Min (ns)    Max (ns)   StdDev (ns)      Operation
 --------  ---------------  -----  ------------  ------------  ----------  ----------  -----------  ------------------
     70.1       55,242,656      1  55,242,656.0  55,242,656.0  55,242,656  55,242,656          0.0  [CUDA memcpy DtoH]
     29.9       23,573,799  9,066       2,600.2       1,120.0         767       5,696      1,937.7  [CUDA memcpy HtoD]

Processing [my_app_report.sqlite] with [/opt/nvidia/nsight-systems/2023.1.2/host-linux-x64/reports/cuda_gpu_mem_size_sum.py]...

 ** CUDA GPU MemOps Summary (by Size) (cuda_gpu_mem_size_sum):

 Total (MB)  Count  Avg (MB)  Med (MB)  Min (MB)  Max (MB)  StdDev (MB)      Operation
 ----------  -----  --------  --------  --------  --------  -----------  ------------------
    611.952  9,066     0.067     0.006     0.000     0.200        0.076  [CUDA memcpy HtoD]
    145.440      1   145.440   145.440   145.440   145.440        0.000  [CUDA memcpy DtoH]
```

Changes made

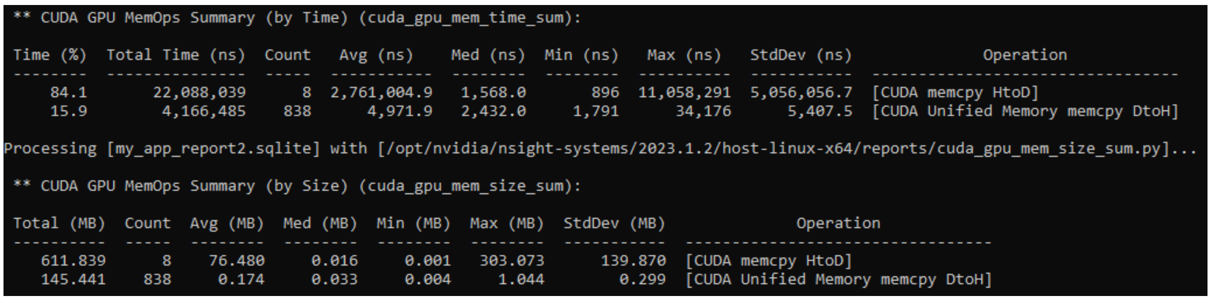
I decided to change the way I allocate and access memory in my kernel so that the CUDA memcpy from host to device can be done in 1 big chunk rather than over loop iterations.

This is seen in my runMatcher() function where contiguous memory is done by combining all of the samples and signatures into one array. Retrieving the correct sample/signature becomes tricky as the char* pointer needs to be retrieved using the correct offset, which is why more arrays sampleOffset and signatureOffset were introduced. (see code comments)

After doing this, the code sped up significantly, with the nsys profiling showcasing this.



```
 ** OS Runtime Summary (osrt_sum):

 Time (%)  Total Time (ns)  Num Calls     Avg (ns)        Med (ns)      Min (ns)      Max (ns)        StdDev (ns)            Name
 --------  ---------------  ---------  -------------  -------------  ---------  -------------  ---------------  ----------------------
     54.8    3,368,442,936        165   20,414,805.7   10,068,062.0      1,552    200,622,501     30,805,928.1  poll
     35.7    2,196,461,947          3  732,153,982.3  171,364,797.0  3,661,770  2,021,435,380  1,119,694,587.0  sem_wait
      5.1      311,866,838        146    2,136,074.2    2,059,868.5        180     20,479,910      2,043,166.5  sem_timedwait
      2.8      170,091,066        567      299,984.2       15,213.0        731     18,959,216      1,165,368.5  ioctl
```



```
 ** CUDA GPU MemOps Summary (by Time) (cuda_gpu_mem_time_sum):

 Time (%)  Total Time (ns)  Count   Avg (ns)    Med (ns)  Min (ns)   Max (ns)   StdDev (ns)               Operation
 --------  ---------------  -----  -----------  --------  --------  ----------  -----------  ----------------------------------
     84.1       22,088,039      8  2,761,004.9   1,568.0       896  11,058,291  5,056,056.7  [CUDA memcpy HtoD]
     15.9        4,166,485    838      4,971.9   2,432.0     1,791      34,176      5,407.5  [CUDA Unified Memory memcpy DtoH]

Processing [my_app_report2.sqlite] with [/opt/nvidia/nsight-systems/2023.1.2/host-linux-x64/reports/cuda_gpu_mem_size_sum.py]...

 ** CUDA GPU MemOps Summary (by Size) (cuda_gpu_mem_size_sum):

 Total (MB)  Count  Avg (MB)  Med (MB)  Min (MB)  Max (MB)  StdDev (MB)               Operation
 ----------  -----  --------  --------  --------  --------  -----------  ----------------------------------
    611.839      8    76.480     0.016     0.001   303.073      139.870  [CUDA memcpy HtoD]
    145.441    838     0.174     0.033     0.004     1.044        0.299  [CUDA Unified Memory memcpy DtoH]
```

The csvs can be found in the appendix.

Appendix

Link to Google sheets: https://docs.google.com/spreadsheets/d/1y--PkMgKHcqrv7eCZLtj9fhXAnxBtdoFxGCa0X6ITIQ/edit?usp=sharing

There are 3 sheets in the link

1. Section 2 -> Diagrams for varying inputs
2. Section 3 Before Memory Optimization -> csv data before contiguous memory
3. Section 3 After Memory Optimization -> csv data after contiguous memory

Link to zip containing all csvs downloaded using nsys: https://drive.google.com/file/d/1gIfxd-_uZUoDtYtyMBOE-0NatwEvL_Mj/view?usp=sharing

AI declaration

For this assignment, I did utilize Chatgpt in order to find out how to run the bash scripts to generate the test cases, as well as the commands to use Nsight Systems nsys to profile my code.

For the memory section, I utilized the CUDA documentation on memory to understand the different syntax. I still ended up using Chatgpt to aid me in some syntax errors that I ran into when trying to code out the memory section of my code, mainly due to CUDA not being able to accept std::string, and needing us to convert it into arrays of char.