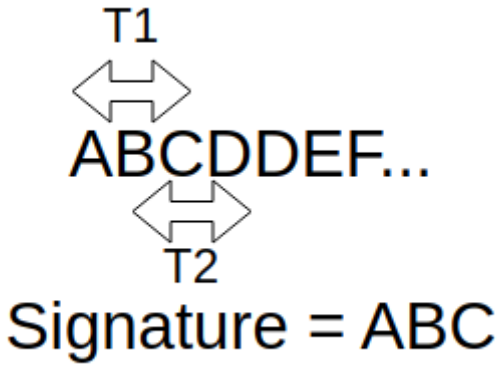
CS3210 Assignment 2 Report

Parallelization Strategy

1. Our program main's method is to use brute force string matching. In this assignment, we convert all the files that are read in bytes to an array of char so that the problem becomes pure string matching.
    a. In this brute force approach, logically, we are just doing two for loops to check for each file to check that whether `file[i..i + len(signature)]` matches with the signature of the virus.
    b. Moreover, each possible combination of (file, signature) pair is launched as a kernel. Each kernel has a variable number of blocks depending on the input size and each block has the maximum possible number of threads that the GPU supports. For example, the Titan V GPU in xgpd5-9 supports max 1024 threads per block.
    c. Each thread will process 1 signature and a portion of file whose length is equal to 1 signature (i.e. `file[i..i + len(signature)]`). All in all, each thread is actually doing a single iteration of the outer for loop in a naive brute force algorithm.

Diagramatically, the strategy can be visualized as the following:



2. We use 1D threads and blocks allocations because its simpler to calculate the index of the file and signature. Each signature and file can be thought of as 1D char array. Moreover.
    a. We also maximize the number of threads. The reason is that based on this discussion: https://forums.developer.nvidia.com/t/is-fewer-threads-better/16746/3, more active threads is desirable because there is very little overhead in switching the threads compared to block. Moreover, 1024 threads is a multiple of 32 (which is the number of threads in a warp). Furthermore, if we don't use max number of threads, the idle threads will be wasted anyway.
    b. Therefore, as we are fixing the number of threads, the number of blocks must be a variable number so that each thread can process 1 iteration of the loop.

Improvement Methods

Since we are using bruteforce, it is then expected our $F_\beta = 1$. Therefore, instead of discussing some methods to increase our F_β, we will discuss two methods to improve our implementation.

1. Our first improvement is to use the Boyer-Moore string search algorithm as explained here: https://en.wikipedia.org/wiki/Boyer%E2%80%93Moore_string-search_algorithm. Boyer-Moore is an improvement over the naive string search algorithm by using bad character and good character heuristic. The heuristic just implies that if some of the characters do not match at certain index, we can skip certain indexes because we know they won't match anyway. The full detail is explained in the wikipedia.
    a. The implementation is commented in our code because tested in xgpd6 (Titan Nvidia 5), the Boyer-Moore implementation is slower than the bruteforce (around ~1.25x slower), even when we already haved preprocessed the bad character first.
2. The second improvement that we have done is to preprocess the conversion of `uint8_t` byte into `char` array using Cuda too. We attempt to parallelize it so that the conversion can be done faster.
3. The last method to improve our performance is to use bitwise operation instead of normal multiplication or division. Since we are dealing with hex and warp that often involves number of multiples of 2 (such as 16 and 32), that we can use bitwise operations which is faster.

Input Factors

All of the experimentations done for this section is done in xgpd5 (2x NVIDIA Titan V GPU, Compute Capability 7.0).

In the first benchmarking effort, we try to fix the number of input files into 20 (using all of the input files given in the "tests" directory) while varying the number of virus signatures in the database.

num signatures	time	sequential time	speedup
100	0.8s	4.679s	5.892x
200	0.9s	9.015s	10.350x
400	1.046s	17.551s	17.548x
800	1.256s	34.772s	27.693x
1600	1.837s	69.163s	37.645x

In the first table, it seems the number of signatures positively correlated to the speedup that our program receives as when we increase the number of signatures, the program also manages to achieve a greater speedup.

In our next effort, we are fixing the number of virus in the signatures database into 2000 signatures while varying the number of input files that our program receives.

num input file	time	sequential time	speedup
1	0.3s	4.632s	13.828x
2	0.4s	5.991s	14.898x
4	0.6s	16.334s	28.000x
8	1.147s	51.029s	44.504x
16	1.791s	79.977s	44.645x

The result also shows a positive correlation between the number of input files and the speedup that our program actually receives.

From the previous 2 results, it is clear to conclude that what effects our program speedup the most is the **number of signatures, number of input file, and the size of each input file**. The reason why is because these 3 things will determine the **problem size** for our implementation. As mentioned before, **each thread in our implementation will process 1 signature and a portion of file whose length is equal to 1 signature**. Thus, it makes sense that our speedup will depends on the number of signatures, number of input file, and the size of each input file as these are the factors that determine the amount of work the threads need to do.

We also observe that our program reached the maximum speedup around 45x - 55x. After this point, even though the problem size is increased, it is very hard to achieve a higher speedup. Here, we run another experiment by varying the minimum input file size and we observe that the speedup no longer increases consistently as the problem size is huge enough to reach maximum speedup already.

min file size	time	sequential time	speedup
4096	2.315s	123.778s	53.479x
8192	2.527s	133.516s	52.827x
16384	2.231s	108.764s	48.759x
32768	2.300s	113.508s	49.344x
65536	2.659s	151.511s	56.974x

To further support this observation, we are running tests with more input files and observe the same phenomenon.

num input file	time	sequential time	speedup
100	10.499s	550.232s	52.406x
200	23.385s	1117.967s	47.806x
400	41.125s	2198.384s	53.456x

Performance Optimization (Preprocessing)

Since we are focusing on the brute force implementation, I am going to elaborate on **second point** in the improvement methods section. This optimization is more related on how we try to increase our parallelization efforts overall.

For context, initially, we are preprocessing the content (i.e. changing it from `uint8_t` byte into `char` array) in the CPU and then use the `cudaMemcpyAsync` in the loop to move the file `char` array into the device. However, after the optimization, we are moving the original `uint8_t` array to the GPU, convert the `uint8_t` byte array into `char` array in the GPU and free the `uint8_t` array in the GPU.

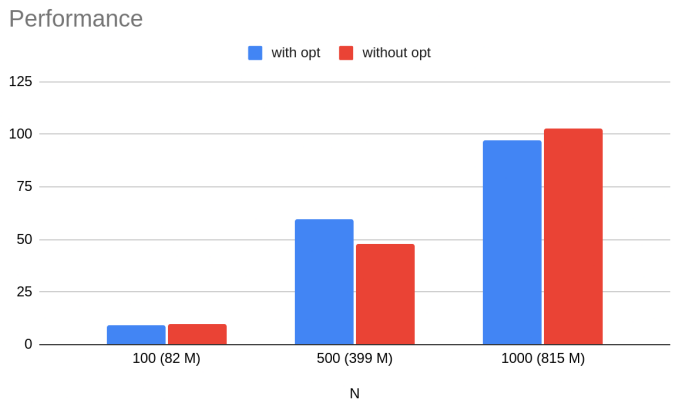
Hypothesis: We hope that by doing this preprocessing in the GPU, the overall virus matching can be done faster as the preprocessing is done in parallel (and we hope the GPU can run faster than CPU is utilized correctly).

Parameter Used for test generation:

```
virus chance: 90.0%
max per file: 5
signature: exact + wildcard (2000 signatures)
```

Result:

N	with opt	without opt
100 (82 M)	9.351	9.533
500 (399 M)	59.823	47.734
1000 (815 M)	97.113	102.898



Conclusion: The result with performance and without performance optimization is very similar to each other. It cannot be conclusively said that the optimization really helps.
One possible explanation about this is that the overhead from mallocing the data in the GPU is pretty big. When we are copying the the `char` array, we need to do the `cudaMalloc` twice as we need to malloc for the `uint8_t` array and also the `char` array in the GPU. The overhead is so big such that the overall performance improvement is not significant, if there is any.

Bonus (implementation)

- We are submitting our naive string matching algorithm implementation because when N > 100, at least it gives around (~1.5x) speed up compared to the parallel implementation already. However, there is not much insightful analysis that I can do for the naive algorithm… :”)