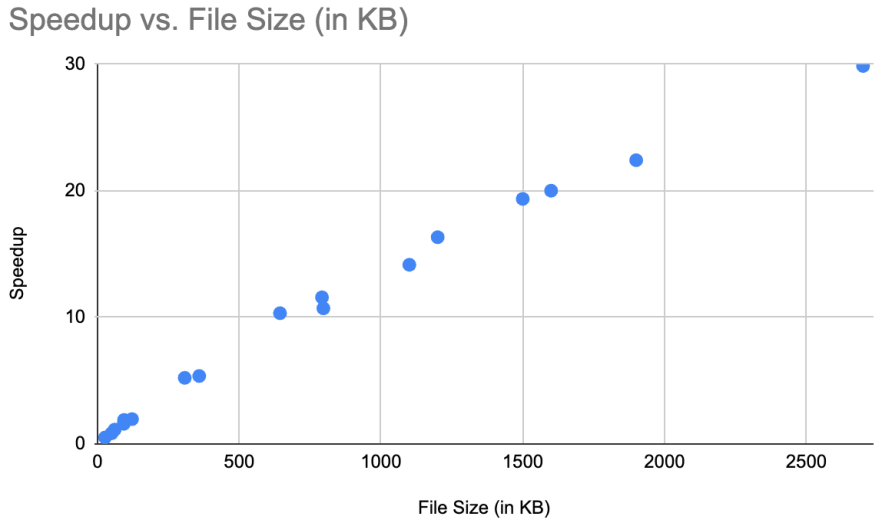
Parallelisation Strategy

In the runScanner function we loop through the input files and signatures. For each input file and signature pair, we launch one kernel with multiple blocks. Within a block, each thread starts from a different position in the input file and checks if the signature matches from that position. This results in parallelism as different parts of the file are checked simultaneously by different threads. The starting index for each thread is calculated based on the block and thread indices (blockIdx.x and threadIdx.x respectively). This approach is basically a sliding window where the window is the size of the signature and slides across the input file.

The blocks are one-dimensional with 1024 threads. 1024 was selected to maximise usage of each SM as it was found to be the maximum number of threads supported by the device. This information was obtained with the `cudaGetDeviceProperties` function. The grids are also one-dimensional with varying size depending on the number of blocks required to cover all possible sliding windows given an input file and signature. One-dimensional grid and blocks are sufficient for our case as we are doing a simple sliding window.

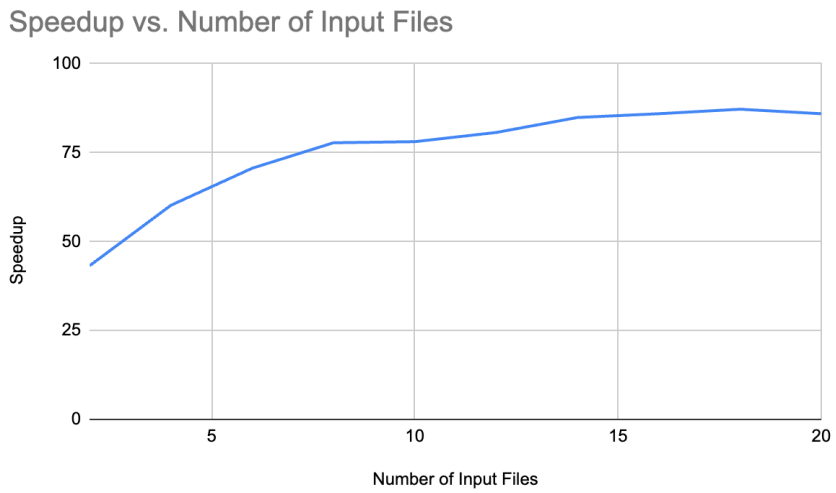
Factors Affecting Speedup

Size of Input Files



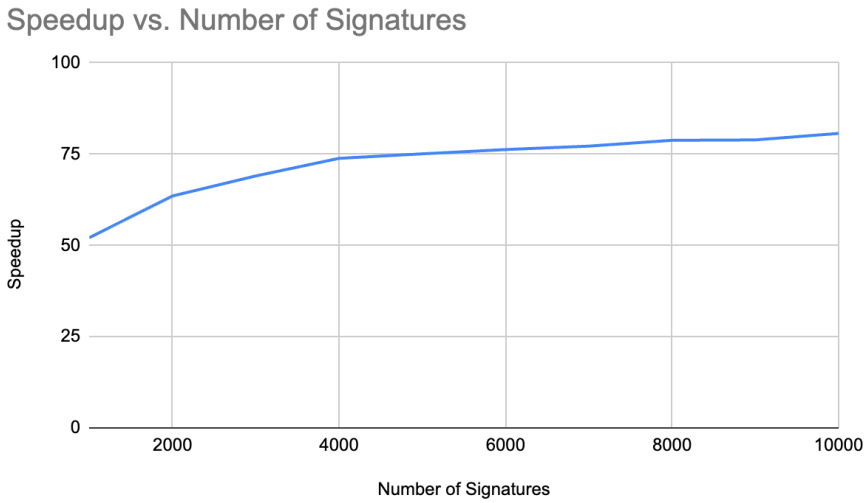
We measured the speedup obtained with varying input file sizes. It is clear that the speedup increases as the input file size increases. For sizes below 100kb, our implementation performed worse. The low speedup with small file sizes can be attributed to a combination of overheads like the high number of kernel launches as well as low overall occupancy in the GPU during execution due to the low workload.

Number of Input Files



We measured the speedup obtained with varying numbers of input files. The total workload given to the provided sequential implementation and our implementation are equal in each run, as such any difference in speedup can be largely attributed to non-computational workload like copying memory from host to device. Since we implemented asynchronous copying with one stream for each file, we can observe the corresponding performance improvements up to 14 files. The plateau can be attributed to the fact that there are only 14 SMs available, as such additional streams beyond that does not increase speedup.

Number of Input Signatures



We measured the speedup obtained with varying numbers of signatures. It is again clear that the speedup increases as the number of signatures increases. Similar to the previous experiments, we see the increase in speedup plateau as we approach saturation.

The conclusion we can draw from the above tests is that the speedup depends on the total amount of workload and not necessarily the type of input size. We cannot conclude that a particular type of input size affects speedup the most as changes in the different types of input sizes affect the total workload in different ways.

Optimisation

Loop unrolling

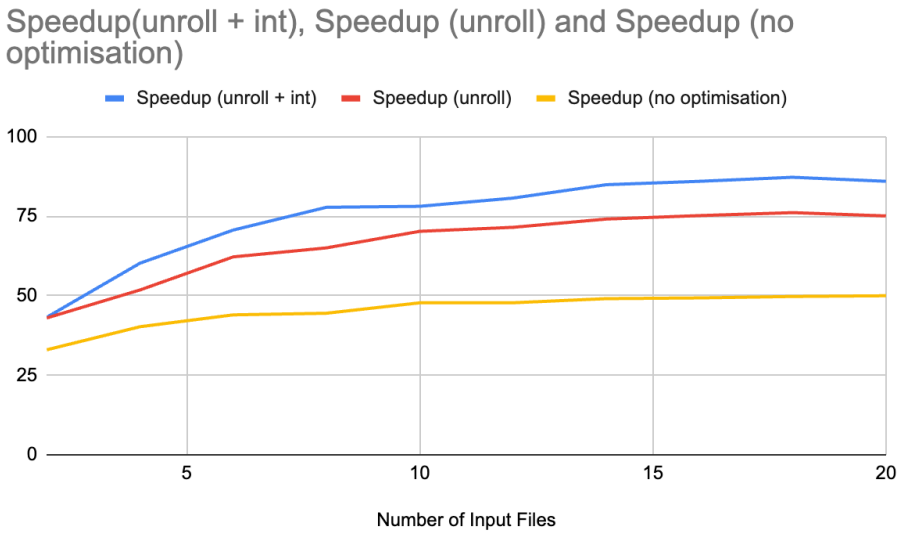
In our first implementation, we did comparisons for each char (4 bits) between the input file data and the signature data. In order to improve this, we implemented a byte level comparison (8 bits) by stepping through the signature 2 chars per iteration. The byte value of the 2 chars are calculated and compared with the file byte. This significantly reduced the number of instructions executed, improving performance.

Casting `size_t` to `int`

Initially the kernel function used indices of type size_t which is an unsigned long (64 bit) under the hood. Upon reading an article on the ampere architecture (NVIDIA Ampere Architecture In-Depth | NVIDIA Technical Blog), we discovered that a warp in an SM of the A100 only had 8 FP64 cores (highlighted in red).

This means that only 8 threads per warp can perform computations done with size_t per cycle. Within the constraints of the assignment, we found that a 32 bit int was sufficient to keep track of the indices. We thus decided to cast size_t to a normal 32 bit int in order for all 32 threads to compute in parallel.

The performance improvements that resulted from each optimization in terms of speedup are shown in the graph below.



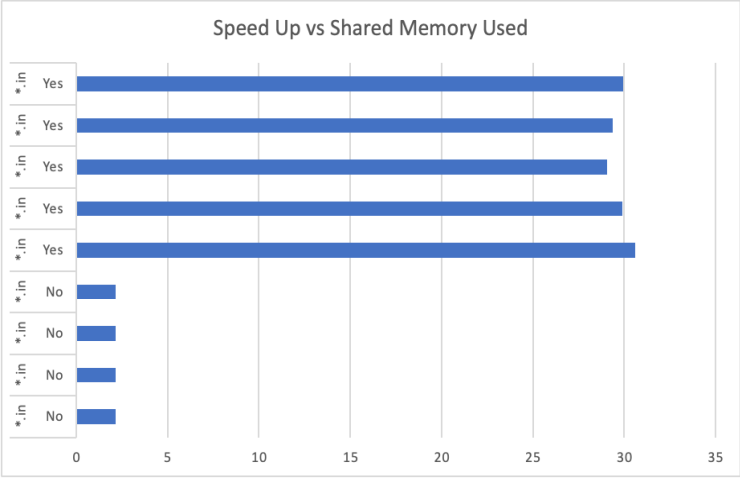
Attempt to utilise shared memory (copy signature data to shared memory)

Shared Memory can help improve performance as the data retrieval latency for shared memory is lower compared to global memory.

For each kernel launch, we brought the relevant signature into shared memory across all the threads in the block.

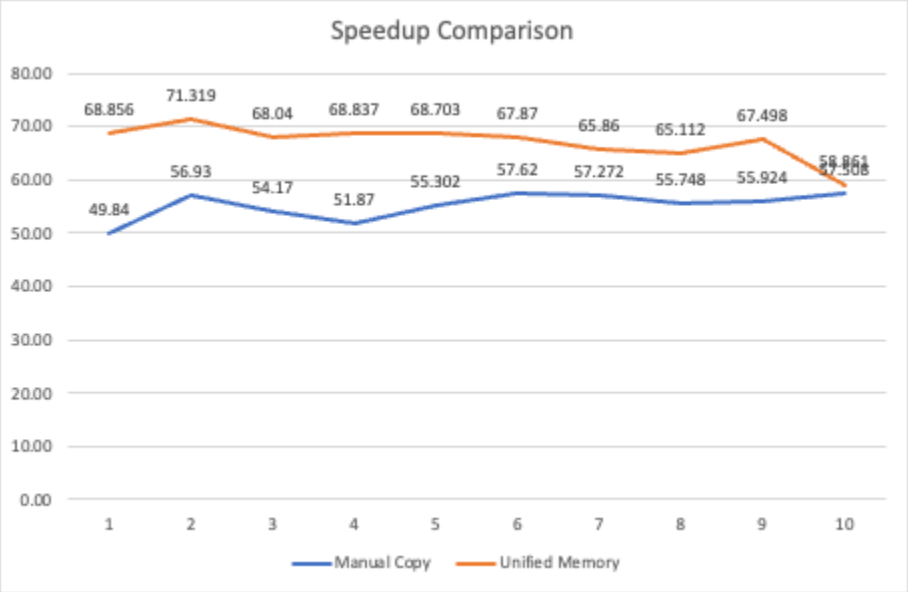
```
__shared__ char shared_sig_data[];

for (size_t i = 0; i < sig_size; ++i) {
    shared_sig_data[i] = sig_data[i];
}
__syncthreads();
```

The performance results are shown below, with a comparison to our original implementation without any use of shared memory.



We observed a drop in performance. This could be due to the higher overhead associated with copying the signature data to shared memory. We also found that there were large shared memory bank conflicts using nsys.

Switch to usage of Unified Memory instead of Manual MemCpy



The usage of unified memory by using `cudaMallocManaged` instead of using `cudaMempcpy` or `cudaMempcpyAsync` improved the performance as seen above. Using the unified memory instead reduced the number of data transfers, especially when it came to transferring the flag_array boolean values.

Appendix

Tests ran for size of input files

```
# File sizes
# ./check.py signatures/sigs-both.txt
tests/virus-0001-Win.Downloader.Banload-242+Win.Trojan.Matrix-8.in
# ./check.py signatures/sigs-both.txt
tests/virus-0002-Win.Downloader.Zlob-1779+Html.Phishing.Bank-532.in
# ./check.py signatures/sigs-both.txt tests/virus-0003-Win.Trojan.Anti-5+Win.Trojan.Mybot-8053.in
# ./check.py signatures/sigs-both.txt
tests/virus-0004-Win.Worm.Gaobot-857+Win.Worm.Delf-1443+Win.Trojan.Mybot-6497.in
# ./check.py signatures/sigs-both.txt
tests/virus-0005-Win.Trojan.Krap-1+Win.Spyware.Banker-4712.in
# ./check.py signatures/sigs-both.txt tests/virus-0006-Win.Spyware.Banker-483.in
# ./check.py signatures/sigs-both.txt tests/virus-0007-Win.Trojan.Matrix-8.in
# ./check.py signatures/sigs-both.txt
tests/virus-0008-Win.Trojan.Agent-35503+Win.Trojan.Mybot-6324+Win.Trojan.Delf-802.in
# ./check.py signatures/sigs-both.txt tests/virus-0009-Win.Worm.Gaobot-688.in
# ./check.py signatures/sigs-both.txt tests/virus-0010-Win.Trojan.Corp-3.in
# ./check.py signatures/sigs-both.txt tests/virus-0011-Win.Trojan.Sdbot-52.in
# ./check.py signatures/sigs-both.txt
tests/virus-0012-Win.Trojan.Bancos-1977+Html.Phishing.Auction-29.in
# ./check.py signatures/sigs-both.txt tests/virus-0013-Win.Trojan.Pakes-518.in
# ./check.py signatures/sigs-both.txt
tests/virus-0014-Win.Trojan.HIV-5+Html.Phishing.Auction-29.in
# ./check.py signatures/sigs-both.txt tests/virus-0015-Win.Spyware.Goldun-31.in
# ./check.py signatures/sigs-both.txt tests/virus-0016-Win.Trojan.Pakes-480.in
# ./check.py signatures/sigs-both.txt
tests/virus-0017-Win.Trojan.Mybot-8097+Win.Trojan.JetHome-2.in
```

Tests ran for number of input files

```
# Number of input files
# ./check.py signatures/sigs-both.txt tests/virus-0011-Win.Trojan.Sdbot-52.in
tests/virus-0011-Win.Trojan.Sdbot-52.in
# for 4,6,8...,20 input files, we added the corresponding number of
tests/virus-0011-Win.Trojan.Sdbot-52.in
```

Tests ran for number of signatures

```
# We duplicated the provided sigs-both.txt file and added the corresponding number of signatures
(1000, 2000, ..., 10000) by copying and renaming the initial 2000 provided signatures
# ./check.py signatures/sigs-both-1000.txt tests/*.in
# ./check.py signatures/sigs-both.txt tests/*.in
# ./check.py signatures/sigs-both-3000.txt tests/*.in
# ./check.py signatures/sigs-both-4000.txt tests/*.in
# ./check.py signatures/sigs-both-5000.txt tests/*.in
# ./check.py signatures/sigs-both-6000.txt tests/*.in
# ./check.py signatures/sigs-both-7000.txt tests/*.in
# ./check.py signatures/sigs-both-8000.txt tests/*.in
# ./check.py signatures/sigs-both-9000.txt tests/*.in
# ./check.py signatures/sigs-both-10000.txt tests/*.in
```

Tests for optimisations

```
# We used the same tests as the ones used for investigating the effect of the number of input
files
# The corresponding implementation of the kernel function is selected for the runs
```

Data

CS3210 Assignment 2 Data