Assignment 2 Report

Nguyen Son Linh (A0200705X)
Njal Telstoe (A0260770H)

Program Description

Parallelisation strategy

Our solution is quite similar to the provided code. We run one kernel per file and signature pair, but we have changed the block and grid dimensions such that multiple threads are running per file signature pair. We implemented a brute force signature matching by comparing all substrings in the file to the signature. We split the substrings evenly between the threads. We use a global result array which is initialized to 0, each file signature pair has a position in the array and will write 1 into it if there is a match.

Selection of grid and block dimension

Our block and grid dimensions are 1 dimensional. We use 32 blocks per grid and 32 threads per block. This results in a sufficient amount of threads for each file signature pair such that the matching is very parallelized. We use 32 threads per block to match the warp size.

Shared memory

Our program did not make use of shared memory as the required speedup was reached by just using global memory.

Methods employed to improve FB

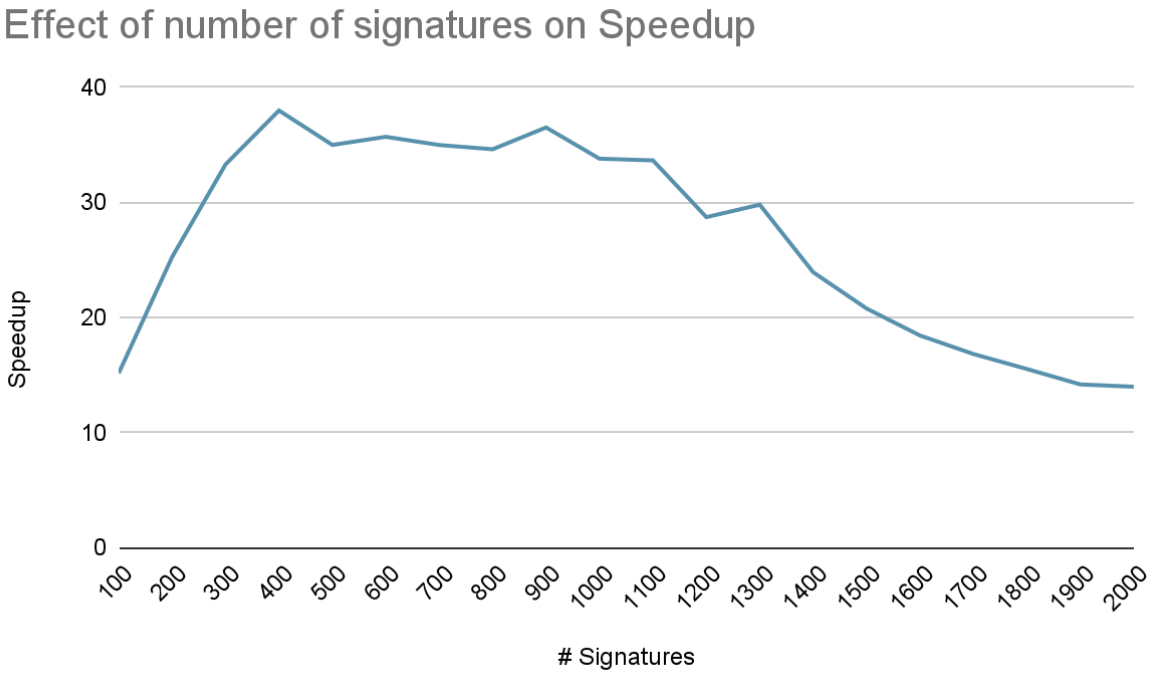
Because brute force was chosen as our algorithm, it is guaranteed to be correct for both exact and wildcard cases, provided that our implementation is bug-free. Therefore, our focus was to do just that, making sure our program runs correctly on different inputs.

Impact of input size

All measurements are taken using node xgpg7.

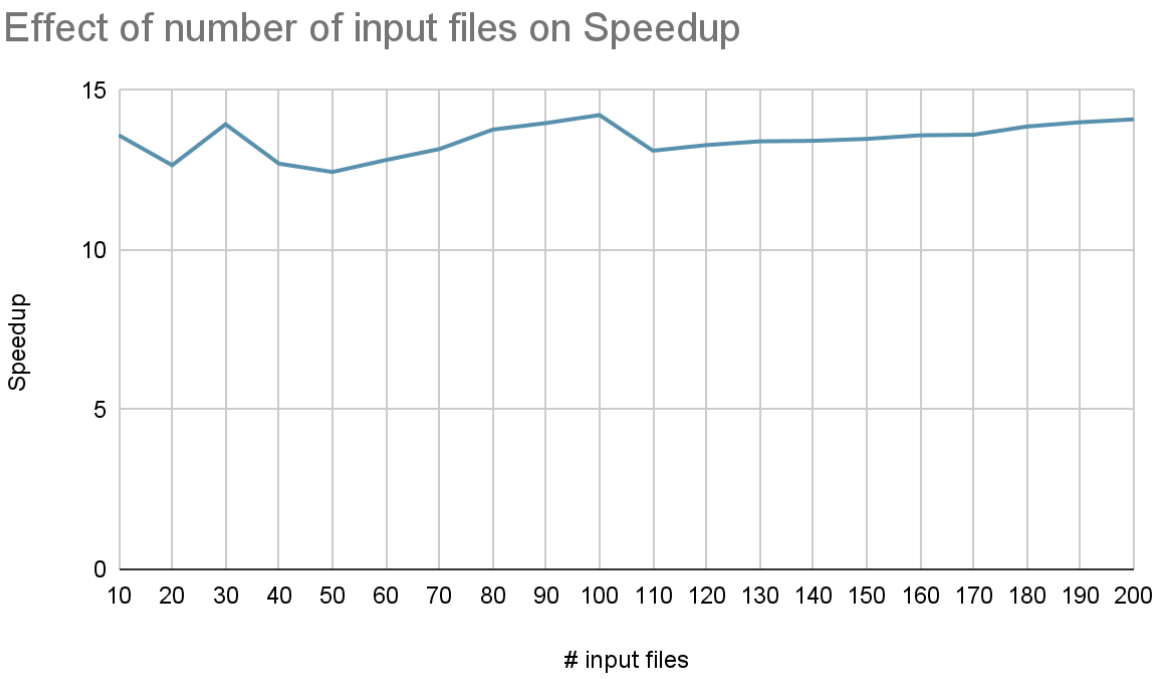
Number of signatures:

We used 200 input files, each having a maximum of 300 viruses. The number of signatures were varied by extracting from sigs-both.txt.



Number of input files:

We used the same file collection as above. This time sigs-both.txt is fixed as the signature database, while different numbers of files are extracted from the collection.



Therefore, the number of signatures affects performance far more than the number of input files as speedup peaks when there are 400 signatures and remains above 30 until 1200 signatures are matched before decreasing rapidly. On the other hand, while keeping the

number of signatures at 2000 and varying the number of input files, speedup remains relatively constant around the 14 mark. Increasing the number of files beyond may give us a better overall view of this situation, however time constraints must be factored in.

We believe that this is due to our implementation, keeping the memory handling as it is in the template code. While increasing the number of files also increases the number of streams, which means that more overlap of memory copy and kernel execution is available, increasing the number of signatures keeps the number of streams constant, leading to each stream having to execute for longer as there are more operations to be done per stream and stream execution is serial.

Performance Optimization

Our original solution would transfer memory to and from global device memory for each kernel, which resulted in a lot of memory transfer overhead because there were many small data transfers. We optimized this by creating a global result array and initializing it to 0. Each kernel would then only write to the array if they found a signature match, and after all kernels had finished, the result was transferred back to the host. This reduced the memory transfer time dramatically, and the speedup of our solution became sufficient.

Original solution:

```
        bool result = true;
cudaMemcpyToSymbol(d_result, &result, sizeof(bool));
        /*
                This launch happens asynchronously. This means that the CUDA driver returns control
                to our code immediately, without waiting for the kernel to finish. We can then
                run another iteration of this loop to launch more kernels.

                Each operation on a given stream is serialised; in our example here, we launch
                all signatures on the same stream for a file, meaning that, in practice, we get
                a maximum of NUM_INPUTS kernels running concurrently.

                Of course, the hardware can have lower limits; on Compute Capability 8.0, at most
                128 kernels can run concurrently --- subject to resource constraints. This means
                you should *definitely* be doing more work per kernel than in our example!
        */
        matchFile<<<blocksPerGrid, threadsPerBlock, /* shared memory per block: */ 0, streams[file_idx]>>>(
                file_bufs[file_idx], inputs[file_idx].size,
                sig_bufs[sig_idx], signatures[sig_idx].size);

        cudaMemcpyFromSymbol(&result, d_result, sizeof(bool));
```

New solution:

Allocate all global memory before starting kernels

```
    uint8_t* dresult;
    size_t result_size = inputs.size() * signatures.size() * sizeof(uint8_t);
    check_cuda_error(cudaMalloc(&dresult, result_size));
```

Move result back to host after all kernels have finished

```
uint8_t* result = (uint8_t*)malloc(result_size);
check_cuda_error(cudaMemcpy(result, dresult, result_size, cudaMemcpyDeviceToHost));
```

Appendix

Reproduction of results

In the commit that is supposed to be marked, there are two bash scripts `run_test_num_signatures.sh` and `run_test_num_input_files.sh`, which can be used to measure the impact of number of signatures/inputs on speedup. What each script does is self-explanatory, and more comments are available in each of the scripts. There are two folders, `performance_tests/` and `measurements/`, which contain the tests used and results for the two charts.

Running 2 scripts mentioned above will not give the exact speedup we obtained as the GPU may not be warmed up, shared with other processes, etc. However, relatively similar speedup is expected.

Chart links (data included)

Speedup relative to number of signatures

Speedup relative to number of inputs