Parallelization Strategy

Our CUDA program consists of only a single `matchFile` function, that scans a single input file for any viruses in the provided signature database. We created separate CUDA streams for each input file and called the `matchFile` kernel on each of them, concurrently scanning all input files.

Work Distribution in `matchFile`

The kernel consists of a 1D grid with the number of blocks being equivalent to the number of signatures in the virus signature database. Each block is assigned a different signature to scan the given input file for. Since each virus signature only needs to be detected once, we only need coordination when looking for the same virus signature. Our 1 signature to 1 block arrangement divides the task into the units that require intra-block coordination while removing any need for inter-block coordination. Additionally, this approach provides a good amount of blocks so that we may maintain high occupancy on the GPGPU.

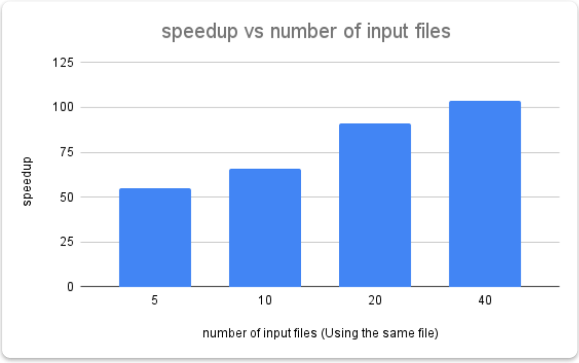
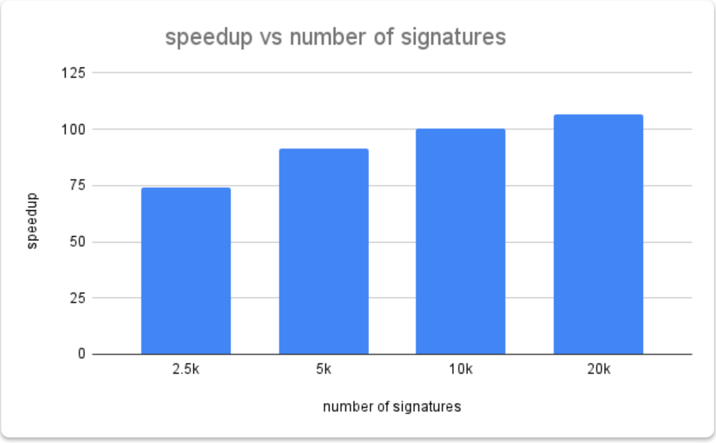
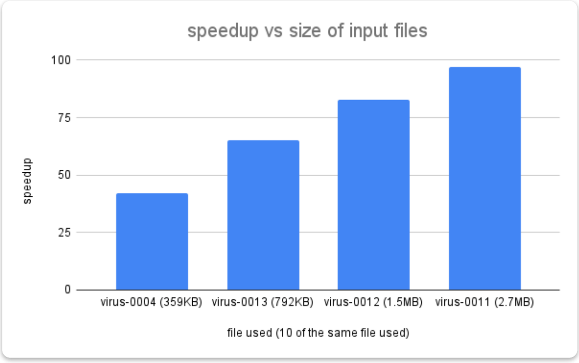
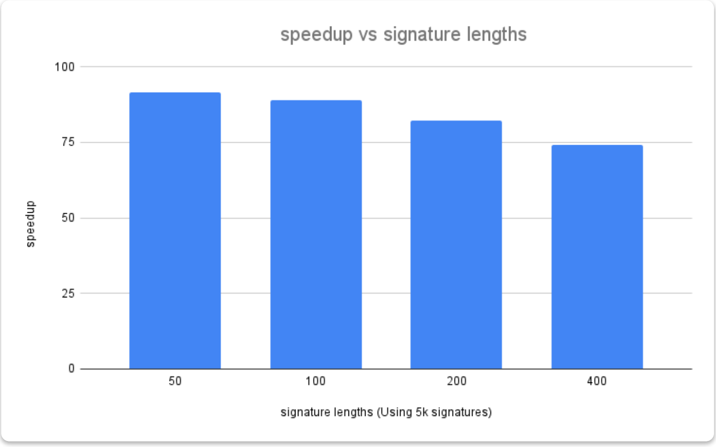
The pseudocode below shows the sequential version of the algorithm we employed in `matchFile`:

```
for i = 0 to fileLength:
    for j = 0 to signatureLength:
        // Assume that signature contains bytes instead of hexadecimal characters
        if (file[i + j] != signature[j]) break;
    if (j == signatureLength) // Successfully scanned the signature
```

In our parallel algorithm, we distributed the outer loop among all the threads in a block in a cyclic distribution fashion. In order to communicate to all threads in a block that the signature has been found, we use a variable `found` that can be accessed by all threads to denote if the signature has been found. Our implementation can be found in `kernel.cu`. The reason we chose to use 512 threads is because from 512 to 1024 threads, which is the maximum number of threads per block, there was no noticeable change in performance for large input sizes.

Effects of Input Sizes on Speedup

The aspects of input size we investigated are the number of input files, the size of input files, the number of signatures and the length of each signature.

Varying Input Files	Varying Signatures
	
	

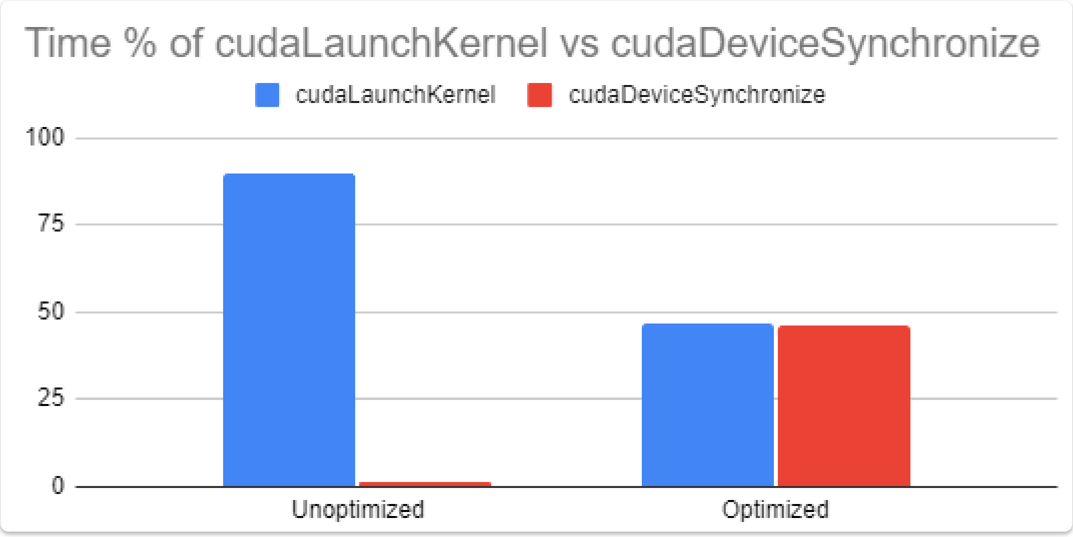
The size of the input files seem to have the greatest impact on speedup, most likely because there is more work that can be done in parallel as the size of the input file increases. Following that train of thought, we expected a change in the number of signatures to also lead to a greater change in speedup, but that was not the case. Since each signature-file pair produces a block, it is possible that occupancy on the GPGPU is sufficiently high, such that an increase in the number of blocks does not affect speedup very much. It is also possible that our implementation is memory-bound, but our memory optimization attempt, that is discussed later on, did not show any performance improvements.

Among the four aspects, only signature length has a negative relationship with speedup. For our implementation, it is possible that longer signatures lengths may lead to more divergent control flows for threads in a warp, which might explain the steady decrease in speedup.

Optimizations

Better Work Distribution for Better Device Utilization

Our initial implementation launches a kernel per file and signature combination. Launching a kernel on a device incurs quite a bit overhead, thus launching such a large number of kernels will lead to a deterioration in performance. Furthermore, the number of kernels that can execute concurrently on a device is limited, hence this approach does not utilize the device well. Instead, we changed our approach to use a kernel per input file and a signature per block. We ran both approaches using `nsys nvprof` with all the provided test inputs and `sigs-both.txt`. The results are shown below:



cudaDeviceSynchronize is used to wait for all kernels to finish executing before printing the results. It can be used as an approximate for how long the kernels take to finish executing. For the first approach (unoptimized), 90% of the time is spent on launching kernels rather than doing work. For the second approach (optimized), the amount of time spent on kernel launches and the amount of time spent doing work is quite even. The speedup increases from about 8 times for the first approach to about 75 times for the second approach.

Global-Memory Access Optimization

We designed our program to have a favorable global memory access pattern with minimal transactions. There are only 3 unique memory accesses in our implementation:

```
__global__ void matchFile(...) {
    for (int i = start; i < end; ++i)
        for (int j = 0; j != sigBytesLength; ++j)
            const char hex_char_1 = signature[j * 2]; //(1)
            const char hex_char_2 = signature[j * 2 + 1]; //(2)
            const curr_file_byte = file_data[i + j] //(3)
}
```

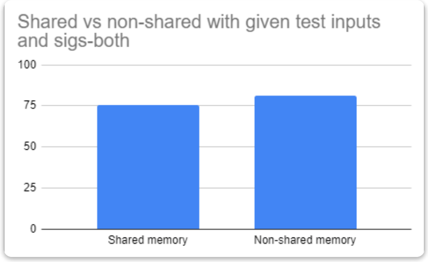
Threads in a warp will always access the same character in `signature`, hence the data is broadcasted to all threads. Threads in a warp always access consecutive bytes in `file_data`, thus the accesses are coalesced into a single transaction.

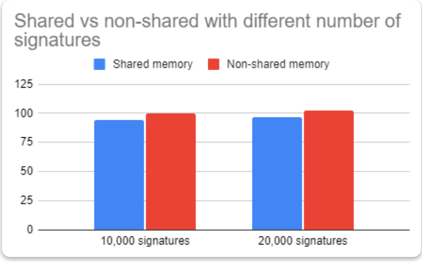
Shared-Memory Optimization Attempt

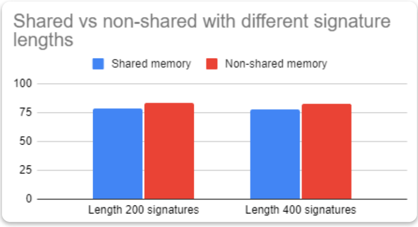
Threads in the same block scan the same input file for the same virus signature. The threads will access both the input file and the signature multiple times over the kernel execution, thus we thought it might improve performance if we brought the data into shared memory. Input files are too large, but virus signatures are small enough, hence we opt for storing just the signature in the shared memory for each block. This is shown in the pseudocode below and the full implementation can be found in `kernel-shared.cu`.

```
__global__ void matchFile(...)
{
    __shared__ char sigShared[];
    // At the start, copy the signature into shared memory
    for (int i = threadIdx.x; i < signatureLength; i += blockDim.x)
        sigShared[i] = sigGlobal[i]

    // Rest of the computation
    ...
}
```


Shared vs non-shared with given test inputs and sigs-both


Shared vs non-shared with different number of signatures


Shared vs non-shared with different signature lengths

As shown by the leftmost figure, we initially tested this implementation using all the provided test inputs and with `sigs-both.txt`. Unlike our expectations, the shared memory version turned out to have slightly lower speedup. We further tested this with some variation in the number of signatures and signature length shown by the second and third figures respectively. There was no noticeable difference in the change in speedup for both shared memory and non-shared memory implementations for both cases. These findings indicate that global memory access is most likely not a bottleneck for our current implementation. Our implementation exhibits a great deal of sequential and temporal locality since it mainly involves traversing a contiguous memory space in sequence. Thus, global memory access latency may have already been reduced greatly via caching and shared memory becomes an overhead rather than a speedup.

Other optimizations

Making use of cudaMemcpyAsync() instead of cudaMemcpy() allows for asychronous data transfer of the file between the CPU and the GPU. GPU computations can then be done without waiting for the transfer to complete, reducing the time the GPU will remain idle.

Appendix

How to Reproduce Results

All results were produced using `a2_slurm_job.sh` without any changes to the given slurm options except for `--time`. `check.py` was used to obtain speedup value for each test. `kernel.cu` contains our final implementation, while `kernel-shared.cu` contains our shared memory optimization attempt.

To test the effects of change in number of input files. We used only the virus-0017 input file that was provided in the starter code and varied the number for each run. To test the effects of change in input file size, we used the files stated in the relevant bar chart. 10 of the same input files were used for each test.

To vary the number of signatures and signature length, we wrote a custom script that generates a virus signature database with random virus signatures. These generated signature files can be found in the `signatures/` directory. Files used to vary the number of signatures are named `sigs-x-samelen.txt` and files used to vary signature length are named `sigs-len-x.txt`. `gen_tests.py` was used to generate input files for these signature databases with the options `num_files = 10, virus_chance = 1, max_viruses_per_file = 10`. All 10 input files generated were used in the respective tests.

Nodes Used

As mentioned before, all slurm options except for `--time` were not changed in `a2_slurm_job.sh`. The only nodes used were the A100 machines with MIG configuration (xgph0-19).

Measured Data

Collected data can be found in the `data.csv` file in our repository.