

Assignment 2

Virus Signature Scanning with CUDA

CS3210 – 2022/23 Semester 1

Learning Outcomes

This assignment lets you explore the intricacies of building a parallel application using NVIDIA CUDA. You will parallelise known algorithms for pattern matching to detect virus files. The objective is to achieve accurate detection with a high degree of parallelisation on GPGPU.

1 Problem Description

1.1 Introduction

In this problem, you are provided with a list of virus signatures, together with several input files that contain 0 to 5 viruses. The task is to use CUDA to scan each of the input files to check for potential viruses, using the signature database provided. The signature database was adapted from the ClamAV database, and input files are randomly generated. The problem is inspired by this GPU Gems article.

1.2 Virus Signatures

The “signature” of a virus is simply a set of contiguous bytes that can be used to identify a virus within an input file. For example, if we had the following virus signature, and an input file (in hexadecimal):

Signature: 7f 45 4c 46 02 01 01 03
Input file: 4f 7c 8a 9d 7f 45 4c 46 02 01 01 03 69 10

Figure 1: Example signature and file with a virus

The example above (Figure 1) shows an input file that *contains* a virus; the part highlighted in red matches the virus signature bytes exactly. Any number of leading and trailing bytes can be ignored, as long as the complete signature byte string appears somewhere in the input.

On the other hand, the following file (Figure 2) *does not* contain the virus above:

Input file: 4f 7c 7f 45 4c 46 69 8a 9d 02 01 01 03 69

Figure 2: Example of a file without the virus

While it does contain all the bytes of the signature, and in the correct order at that, the signature is not contiguous — there are bytes in between the two halves (highlighted), which means this file should not match the virus signature.

1.2.1 Signature Wildcards

To make the problem more interesting, we introduce wildcards into the equation. A wildcard is represented by a “?” character in the signature, and it matches **any 4 bits** of the input file.

In the first case, we have e?, which can match any of e0, e1, ..., ee, ef. Similarly, the second byte ?f matches any of 0f, 1f, ..., ef, ff. The last case is when both nibbles in a byte are wildcards; then, the input file can contain any of the 256 possible 8-bit values.

For example:

```
Signature:      fe e? fa ?f
Input file 1:   1a fe e7 fa 4f 69 10
```

Figure 3: Example of matching with wildcards (1)

In Figure 3 above, we have two wildcards attached to two bytes, and the wildcard bytes matched e7 and 4f respectively. For input file 2 in Figure 4 below, the byte a0 did not match the signature, so the file is (probably) not a virus — not this particular virus, at least.

```
Signature:      fe e? fa ?f
Input file 2:   70 fe a0 fa 4f 10 69
```

Figure 4: Example of matching with wildcards (2)

1.3 Input Format

There will be two inputs given to your program: a signature database, and one or more input files; they will be passed as filenames in the command-line arguments, although the provided skeleton code already handles this for you.

1.3.1 Signature Database Format

The signature database consists of one virus signature per line, with each line having this format (Figure 5):

```
virus-name:01af7419...
```

Figure 5: Format of the signature database

Figure 6 shows some example lines from the signature database:

```
Linux.ELF-420:7f454c4602010103
Win.Trojan.Small-4379:606a026a01e8??000000536a
Win.Trojan.Castova-591:494e4964697265637462616e6b554936302e646c6c
```

Figure 6: Example signatures

Each line contains a virus-name and a signature string. The signature string is given in hex characters (like Python's `hex` function), such that two hexadecimal characters (0-9, a-f) represent one byte. We give some guarantees for the signature string:

- At least 64 characters (32 bytes) long
- At most 8192 characters (4096 bytes) long
- At most 10% of the characters will be wildcards ("?")
- No more than 4 wildcards can appear consecutively

The number of signatures in the virus database is not limited; you can expect that at least 20 000 signatures will be used during grading.

There is no need to parse the virus names — they are just there to identify the signature — but they will be used when outputting. Only one signature database will be given to your program.

1.3.2 Input File Format

Input files are simply binary files (not text!); there is no fixed format for them, other than these guarantees:

- At least 4 kB (4096 bytes)
- At most 8 MB (8 388 608 bytes)
- Total size of all input files (per invocation of your program) is at most 4 GB

For the most part, we will use random data to generate our input files for grading; the script we use is provided in the skeleton code as well. Unlike the signature database, we will potentially pass many input files at once to your program.

1.3.3 Output Format

The output format is relatively simple — simply output one line per matching signature, per file. There is no requirement to sort the lines, or output them in any particular order. Each line should follow the format from Figure 7:

```
input-file:  virus-name-1
```

Figure 7: Format of one output line

For the avoidance of doubt:

- The name of the input file
- A single space
- The name of the virus that the file contains

For example, this is a valid output from the program:

```
tests/virus-0003-Win.Trojan.Remut-1.in: Win.Trojan.Remut-1
tests/virus-0004-Win.Trojan.Hi-3+Win.Trojan.Integrator-1.in: Win.Trojan.Integrator-1
tests/virus-0006-Win.Trojan.SdBot-3967: Win.Trojan.DNSChanger-123
tests/virus-0004-Win.Trojan.Hi-3+Win.Trojan.Integrator-1.in: Win.Trojan.Hi-3
tests/benign-0003.in: Email.Trojan.Ecard-23
```

Note that the two virus matches for virus-0004 are not printed together, and not in any particular order. This output also contains both a false identification (wrong virus) as well as a false positive (the benign file). Also, as mentioned above, one input file can contain multiple viruses.

2 Implementation Guidelines

To solve this problem, you can try several approaches, including but not limited to:

- The algorithm from the GPU gems article [link]; you might want to modify the hash table lookup sizes
- A brute force $O(nm)$ solution
- Rolling hash (eg. Rabin-Karp [link])
- Finite automata-based algorithms (eg. Deterministic FSA [link])
- Any other approach you can think of

When choosing an algorithm, consider both exact matching and signature wildcards (?). It is not trivial to extend some of these algorithms to handle wildcards.

Applying some of these algorithms does not produce 100% accurate results. Specifically, the algorithms incorrectly identify inputs as viruses (false negatives and false positives) — see Section 2.1 for details. You should focus on improving the accuracy while implementing a high degree of concurrency (parallelism) in your programs.

To adapt your algorithm of choice to a CUDA program, you should consider the following:

- What grid and block dimensions to use for your kernel
- Whether or not your kernels can be launched concurrently
- How much shared memory to allocate per block, and how to split it up
- How to best utilise the total (global) memory on the GPU

2.1 Correctness Grading

Your implementation does not need to be 100% accurate to obtain full correctness marks. You may detect some benign files as viruses (false positive), and miss some actual viruses (false negatives).

False positives will not be penalised heavily; you should focus on reducing the false negative rate as much as possible, ie. err on the side of reporting something as a virus. This follows what we would expect a real virus detection program to do; we would prefer to get benign files marked as viruses rather than completely miss an actual virus.

While it is possible to get the false negative rate to 0 (using the right algorithm and strategies), this is **not necessary** to obtain full marks for correctness. You should apply at least 2 methods to reduce false negatives in your program, and explain these in the report.

The starter code provides a sequential implementation that is 100% accurate (ie. no false positives or negatives), but is not necessarily as fast as it can be. You should use this implementation to check for the correctness of your program, and use the provided script to get a sense of your false positive/negative rate.

Using the number of false positives and false negatives, we compute F-score for each set of inputs. Specifically, we compute F_β with $\beta = 15$ as follows:

$$F_\beta = \frac{(1 + \beta^2) \times \text{true positives}}{(1 + \beta^2) \times \text{true positives} + \beta^2 \times \text{false negatives} + \text{false positives}} \quad (1)$$

We provide a script to you that produces this F_β for a set of inputs and outputs. Again, feel free to use an inexact algorithm (eg. hash-based approaches).

2.2 Hint: Asynchronous CUDA Kernels

For asynchronous (concurrent) kernel launches and execution, you can look at this resource: [Cuda Best Practices Guide](#). In particular, this method allows you to copy memory between the CPU and GPU, as well as launch CUDA kernels, asynchronously and concurrently; this can be done by using CUDA streams and the “async” version of functions.

```
// make streams
std::vector<cudaStream>_t streams {};
streams.resize(100);

for(size_t i = 0; i < streams.size(); i++)
    cudaStreamCreate(&streams[i]);

// computation:
for(int k = 0; k < 100; k++)
{
    cudaMemcpyAsync(gpu, cpu, 256, cudaMemcpyHostToDevice, streams[k]);
    kernel<<<10, 10, 0, streams[k]>>>(...);
    cudaMemcpyAsync(cpu, gpu, 256, cudaMemcpyDeviceToHost, streams[k]);
}

// synchronise streams (like a join)
// not strictly necessary, destroying also forces a synchronisation.
for(size_t i = 0; i < streams.size(); i++)
    cudaStreamSynchronize(streams[i]);

// clean up the streams
for(size_t i = 0; i < streams.size(); i++)
    cudaStreamDestroy(streams[i]);
```

In the example listing above, the iterations of the computation loop can run concurrently — up to the hardware limit of the GPU. How this works is that a call to an “async” function will return to the CPU immediately, without waiting for the GPU to finish executing.

Note that we have passed a stream to the kernel launch as the last argument (`<<<..., streams[k]>>>`); this makes the kernel launch asynchronous as well.

3 Grading

You are advised to work in groups of two students for this assignment (but you are allowed to work independently as well). You may have a different teammate compared to Assignment 1. You may discuss the assignment with others but in the case of plagiarism, both parties will be severely penalised. Cite your references or at least mention them in your report (what you referenced, where it came from, how much you referenced, etc.).

The grades are divided as follows:

- 8.5 marks – the implementation in CUDA, split into:
 - 6.0 marks – matching signatures **without** wildcards (“?”)
 - 2.5 marks – matching signatures **including** wildcards (“?”)
- 3.5 marks – the report

3.1 Program Requirements

Your CUDA implementation should:

- Be written in C or C++
- Give the same result (output) as the provided sequential implementation (only an executable is provided)
- Use at most 12 GB of memory, with no leaks — the amount available on the Titan V GPU (xgpd5-9)
- Only run on a single GPU (applicable only for the dual-GPU machines)
- Run on compute capability 7.0
- Scale well with the increase in input size (number of input files, number of signatures, size of input files, length of signatures)
- Be accurate, with $F_\beta \geq 0.5$ for $\beta = 15$.
- Run faster than our sequential implementation on any machine for a reasonably large input size

Specifically, to obtain full marks for the performance component, your CUDA implementation should achieve a speedup greater than 1x when running with the following parameters:

- At least 10 000 signatures in the database
- At least 10 input files, each at least 512 kB large

We will use the F_β metric to measure the correctness of your program. The requirement to obtain full correctness marks is $F_\beta \geq 0.5$.

When grading, we will test for correctness by checking the outputs of your implementation (as mentioned in the output format above). You **do not** need to exactly follow the output order of our sequential implementation; our grading script will account for these differences automatically.

We have tested our sequential implementation but as always it is possibly incorrect. If you notice any issues, please do let us know. Our contact details are available in Section 4.1.

3.2 Template Code

We provide skeleton code that can be used as a starting point for your CUDA implementation. The code is written using C++, but feel free to change this code in any way, including changing to C. Furthermore, you might consider writing your program using a mix of C and C++.

The template code provided includes the files listed in Table 1:

File name	Description
scanner-seq	Executable file running on host only that identifies exact matches for an input.
check.py	Python script to output F_β for your input/output pairs.
gen_tests.py	Python script to generate new testcases based on a set of signatures.
Makefile	The build script.
src/defs.h	Common header file that includes the definition of the InputFile and Signature structs.
src/common.cpp	Provided code to parse the signature database and read the input files to (host) memory.
src/kernel.cu	The provided skeleton CUDA code.

Table 1: The list of provided files in the skeleton

3.2.1 Provided “Sequential” Implementation

You might realise that our provided sequential implementation is not exactly sequential — in fact, it is parallelised across all CPU cores! This reduces the amount of time that you need to spend waiting to check your own program for correctness.

If you wish to run it manually with parallelisation, set the environment variable `PARALLEL_HAX=1`. The last line of output is the number of nanoseconds elapsed, summed over *all threads* — this represents the amount of time that a sequential implementation *would have taken*.

To be clear: you only need to achieve a speedup against the *reported sequential time*, **not** the wall-clock time of our program. The provided `check.py` script already accounts for this when showing your speedup.

3.3 Report Requirements

3.3.1 Format

Your report should follow these specifications:

- Four pages maximum for main content (excluding appendix).
- All text in your report should be no smaller than 11-point Arial (any typeface and size is ok so long as it's readable English and not trying to bypass the page limit).
- All page margins (top, bottom, left, right) should be at least 1 inch (2.54cm).
- Have visually distinct headers for each content item in Section 3.3.2.
- It should be self-contained. If you write part of your report somewhere else and reference that in your submitted “report”, we reserve the right to ignore any content outside the submitted document. An exception is referencing a document containing measurement data that you created as part of the assignment - we encourage you to do this.
- If headers, spacing or diagrams cause your report to *slightly* exceed the page limit, that's ok - we prefer well-organised, easily readable reports.

3.3.2 Content

Your report should contain:

- (1 mark) A brief description of:
 1. your program's parallelisation strategy, eg. how work (signatures, input files) are divided into kernels, threads, and blocks
 2. your choice and justification for grid and block dimensions
 3. how block shared memory (if applicable) is used in your kernel

Diagrams are not required but may help you explain something clearly without taking much space.

- (0.5 marks) A description of at least two methods that you employed to increase F_β for your program.
- (1 mark) Answers to the following questions :
 1. Which aspect of the input size affects speedup the most for your implementation?
 2. Why do you think this is the case?

Take measurements to support your answers. You may vary the input size (number of input files, number of signatures, size of input files, length of signatures), and use the different GPUs available to test your program.

- (1 mark) A description of ONE performance optimisation you attempted (if any) with analysis and supporting performance measurements. Refer to an optimisation related to your parallelization efforts. Include at least one summary graph in your report and link to your supporting measurements (for example, a .csv file in the repository or a Google sheet).

Additionally, your report should have an appendix (does not count towards page limit) containing:

- Details on how to reproduce your results, e.g. inputs, execution time measurement, etc.
- A list of nodes you used for testing and performance measurements.
- Relevant performance measurements, if you don't want to link to an external document.



Tips:

- There could be many variables that contribute to performance, and studying every combination could be highly impractical and time-consuming. You will be graded more on the quality of your investigations, not so much on the quantity of things tried or even whether your hypothesis turned out to be correct.
- Performance analysis may take longer than expected and/or run into unexpected obstacles (like your program failing halfway). **Start early** and test selectively.

3.4 Bonus - Analysis Contest

You may obtain up to 3 bonus marks for accurate implementation and/or insightful analysis. Analyse the **impact of the input size on the accuracy and performance, and the trade-off between accuracy and performance**. The input characteristics include number of input files, number of signatures, size of input files, length of signatures. Presenting measurements without any interesting insights will not count for bonus.

- up to 1 bonus mark - 100 % accurate CUDA implementation that is faster than our our multi-threaded "sequential" implementation:
 - you should compare against our parallel execution time, not the sequential execution time
 - your program should have **no** false negatives **nor** false positives to qualify!

- up to 1 bonus mark - insightful analysis for one parallel algorithm used for signature matching.
- up to 2 bonus marks - insightful analysis for two (or more) parallel algorithms used for signature matching.

To participate in the contest, you must add a maximum 2-page section at the end of your report (before the appendix) entitled “Bonus” where you include your analysis. In the Bonus section, you may refer to the sections in the main report. This section does not count towards the main report page count.

If you have additional implementation, you must create a makefile recipe named `bonus` such that when `make bonus` is run, the additional executable files are generated. Explain in the Bonus section how to use these additional files.

4 Admin

4.1 FAQ

Frequently asked questions (FAQ) received from students for this assignment will be answered [here](#). The most recent questions will be added at the beginning of the file, preceded by the date label. **Check this file before asking your questions.**

If there are any questions regarding the assignment, please use the Discussion Section on Canvas or email Zhia Yang (zhiayang@u.nus.edu) or Walter (kwalter@u.nus.edu).

4.2 Github Classroom and Starter Code

We will use Github Classroom to provide starter code and for submission. Use this link to get started: <https://classroom.github.com/a/i080EdMM>

The first member of the team to access this link will be prompted to accept the assignment that gives your team access to the new repository. Create a new team by typing `A2-A0123456Z` or `A2-A0123456Z_A0173456T`, using the student numbers of the students forming a team. Note that the naming convention must be followed strictly, e.g. capitalisation, dash, and spacing. Use the same name for your report file (e.g. `A2-A0123456Z_A0173456T.pdf`) for easy identification.

The other member in the team will be able to see an existing team with your team id under “Join an existing team” section.

The repository is automatically populated with the starter code provided for this assignment.

4.3 Running your Programs

As mentioned in the CUDA lab, our lab machines do not have GPUs; use the compute cluster machines to test your program. Consult Lab 3 for details about how to connect to the SoC Computer Cluster. The list of machines reserved for CS3210 are:

- `xgpd[5-9]`: 2x NVIDIA Titan V GPU, 12 GB memory, Compute Capability 7.0
- `xgpf[6-9]`: NVIDIA Tesla T4 GPU, 15 GB memory, Compute Capability 7.5
- `xgpg[5-9]`: NVIDIA A100, 40 GB memory, Compute Capability 8.0

Please note that the Compute Cluster machines are shared among all students, *without* Slurm. Try to use nodes that are currently unused for more accurate benchmarking results. You can use `nvidia-smi` to see if there are any other CUDA programs currently running.

During development you should test with these machines on the SoC Compute Cluster, or you can choose to use your personal computer (if it has a dedicated NVIDIA GPU).

4.4 Deadline and Submission

Assignment submission is due on **Mon, 17 Oct, 2pm**.

The implementation and report must be submitted through your Github Classroom repository. The **commit hash** that you intend to use for submission must be submitted in the text entry for Assignment 2 in Canvas.

- Push your **code and report** to your team's Github Classroom repository.
- Your report must be a PDF named `<team_name>.pdf`. For example, `A2-A0123456Z_A0173456T.pdf`. `<team_name>` should **exactly match** your team's name; if you are working in a pair, please DO NOT flip the order of your admin numbers.
- If you are working in a team, please register your group under People (Assignment 2 Groups) before submitting your **commit hash** to Assignment 2 on Canvas. Failing to do this step would show a "Missing Submission" for Assignment 2 in Canvas, as only one of the team members is required to submit.
- Submit the **commit hash** you want us to grade (need not be the latest one in your repo) using the text entry box for Assignment 2 on Canvas. **Failing to submit this commit hash would be considered an incomplete submission** (penalties apply), and we will grade the latest commit found in your repository dated before the deadline of the assignment.
- The time of latest assignment attempt (including late submissions) **on Canvas** will be taken as the assignment submission time. Multiple attempts are allowed if re-submission is needed. Note that for submissions made as a group, only the most recent submission (from any of the students) will be graded, and both students receive that grade.

A penalty of 5% per day (out of your grade) will be applied for late submissions.