# Assignment 2
# Virus Signature Scanning with CUDA

CS3210 – 2023/24 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 paralellisation 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-591:494e496469726563746261e6b554936302e646c6c
```

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

There is no limit to the number of signatures in the virus database, but it will not exceed the size of the RAM in the machine. 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 $4$ KiB ($4096$ bytes)
- At most $8$ MiB ($8\,388\,608$ bytes)
- Total size of all input files (per invocation of your program) is at most $4$ GiB

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 a **brute-force algorithm**, the algorithm from the **GPU Gems article** [link] with some changes hash table lookup sizes; or any other algorithms. We **strongly suggest** to start with one of these first two approaches before trying other algorithms. When choosing an algorithm, consider both exact matching and signature wildcards (?). It is not trivial to extend some 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

The usage of external libraries is subject to approval – please email us prior to using any. Our contact details are available in Section 4.1.

## 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. If you are not using a brute-force algorithm, you should explain at least one method you used to reduce the false negative rate.

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 $F_\beta$ with $\beta = 15$ as follows:

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

**We provide a script** to you that **calculates this** $F_\beta$ 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). Your 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
- Up to 2 bonus marks, described in Section 3.4

## 3.1 Program Requirements

Your CUDA implementation should:

- Be written in C or C++
- Use the CPU minimally – see the FAQ below.
- Give the same result (output) as the provided sequential implementation (only an executable is provided)
- Use at most $11$ GB of memory, with no leaks — the amount available on the A100 machines with MIG configuration (`xgph0-19`)
- Only run on a single GPU at a time (applicable only for the dual-GPU machines)
- Run on Compute Capability 8.0 – specifically we will grade your code on the `xgph` machines, but please feel free to initially develop your code on any of the GPU nodes.
- Scale well with the increase in input size (number of input files, number of signatures, size of input files, length of signatures) – explore this scaling in your report.
- 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 $F_\beta$ metric to measure the correctness of your program. The requirement to obtain full correctness marks is $F_\beta \geq \mathbf{0.5}$.

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. You will not be penalised for printing the same line multiple times.

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 |
|---|---|
| `a2_slurm_job.sh` | Example Slurm job script to run your code. Read this for instructions and examples. |
| `scanner-seq` | Executable file running on host only that identifies exact matches for an input. |
| `check.py` | Python script to output $F_\beta$ 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 Quickstart

To quickly build and run your code on an A100 MIG GPU (and automatically test for correctness with `check.py`), simply run `sbatch a2_slurm_job.sh`. You will have to modify this script (and likely the Makefile) to do other tasks such as: checking for correctness with `check.py`, etc. Some simple recipes have been provided in the script to get your started.

### 3.2.2 Profiling

Since Compute Capability 8.0, NVIDIA has deprecated the use of their previous profiler `nvprof` in favor of the `ncu` tool (NSight Compute CLI) and `nsys` (NSight Systems). You can read more about it here and here. We include a recipe for basic profiling with `nsys` in the Slurm job script, but please explore further.

### 3.2.3 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.
- (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.
- (1.5 mark) **Two** out of the following **three** possible items:
  - A description of one CUDA-based performance optimisation you attempted (if any) with analysis and supporting performance measurements. Refer to an optimisation related to your CUDA parallelization efforts.
  - A description of at least one method that you employed to increase $F_\beta$ for your program, if you employ a method that does not yield $F_\beta = 1$.
  - A description of a CUDA-based memory optimization you attempted.

  For each of these, 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 - Speed and Analysis Contest

You may obtain up to 2 bonus marks for accurate/fast implementation and/or insightful analysis. Minimally, analyse the **impact of the input size on the accuracy and performance, and (if applicable) 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 on the A100 nodes without MIG configuration (xgpg0-9).
  - 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!
  - Note that the provided implementation uses a brute-force solution that runs in O(m*n) time.
  - For your bonus implementation, you can use the full capabilities of the A100, including up to 40GB of GPU memory. Do note that your main implementation (not the bonus one) should still stick to the 11GB limit.
- up to 1 bonus mark - insightful analysis for one parallel algorithm 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 an 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 Theodore (theodoreleebrant@u.nus.edu).

## 4.2 Running your Programs

The link to the documentation on how to use the SoC Compute Cluster for GPU can be accessed at https://nus-cs3210.github.io/student-guide/soc-gpus/. During development you could test these machines on the Compute Cluster, or you can choose to use your personal computer if it has a dedicated NVIDIA GPU.

## 4.3 Deadline and Submission

Assignment submission is due on **Thu, 26 Oct, 2pm**.

- **Canvas Quiz Assignment 2**: Take the quiz and provide the name of your repository and the commit hash corresponding to the `a2-submission` tag; if you are working in a team, only one team member needs to submit the quiz. If both of you submit, we will take the latest submission.
- **GitHub Classroom**: The implementation and report must be submitted through your Github Classroom repository. The GitHub Classroom for Assignment 2 can be accessed through https://classroom.github.com/a/uvcdwKg8.

The GitHub Classroom submission must adhere to the following guidelines:

- Name your team `a2-e0123456` (if you work by yourself) or `a2-e0123456-e0173456` (if you work with another student) – substitute your NUSNET number accordingly.
- Push your **code and report** to your team's GitHub Classroom repository.
- Your report must be a PDF named `<teamname>.pdf`. For example, `a2-e0123456-e0173456.pdf`. `<teamname>` should **exactly match** your team's name; if you are working in a pair, please DO NOT flip the order of your admin numbers.
- Tag the commit that you want us to grade with `a2-submission`; if no such tag exists, we will use the most recent commit.

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

**Final check:** Ensure that you have submitted to both **Canvas** and **GitHub Classroom**. Canvas should contain your commit hash and repository name, and GitHub Classroom should contain your code and report.