Assignment 2

Virus Signature Scanning with CUDA

CS3210 – 2024/25 Semester 1

Learning Outcomes

This assignment lets you explore the intricacies of building a parallel application using NVIDIA CUDA. Your task is to detect **biological viruses** with the DNA sequences of human samples (e.g., cheek swabs, etc). The objective is to accurately detect viruses with a high degree of parallelization on a GPGPU.

Contents

1 Problem Description

1.1 Introduction

The COVID-19 pandemic showed that rapid and accurate detections of viral infections are critical for public health. Scientists and public health professionals successfully tracked the spread of the virus and its variants by analyzing vast amounts of genetic data in near real-time. This data was generated by genomic technologies like Next-Generation Sequencing (NGS) and Real-Time PCR (RT-PCR), allowing scientists to determine the DNA and RNA of pathogens quickly, and obtain the DNA sequences of potentially infected individuals at scale.

However, these technologies generate **enormous amounts of data**! In many cases, simple parallel programs on CPUs are too slow to match detect viruses within human DNA in real-time. We need a better option.

1.2 Assignment Overview

In this assignment, your goal is to use CUDA on a GPU to **rapidly identify viral infections in human patients**. You will receive files containing DNA sequences that represent **samples** taken from patients (e.g., from cheek swabs, nasal swabs, etc.), and a database of **signatures** – known viral DNA sequences. Your goal is to identify which patients have which viruses, and also output an overall "confidence score" for each match.

1.3 Definitions: Samples, Signatures, and Matches

There are two types of input you will have to handle, **patient DNA sequences** (**"samples"**) and **viral DNA sequences** (**"signatures"**). However, the data represented by both are ultimately **DNA sequences**, which we define below.

DNA Sequence

- A DNA sequence (in this assignment) is a string where each character may be one of **five possible characters**: 'A', 'T', 'C', 'G', or 'N'.
- Each of 'A', 'T', 'C', 'G' represents a *nucleotide*, one of the four unique possible molecules present as part of a DNA string.
- The 'N' character represents an *unspecified* nucleotide – **any** of the four nucleotides are legally permitted to be in that specific position.

Below is an example of a DNA sequence with the 5 possible characters.

TNACGGTNCGATCTCGAAANTTCATACCGCGAAANTTCATACGCTATTCAN

Figure 1: Example of a DNA sequence with unspecified nucleotides, color-coded by character.

Your goal is to detect and output **matches** between **"samples"** and **"signatures"**.

Matches

- **Samples and Signatures:** A DNA sequence obtained from a patient (a "sample") can *match a* DNA sequence representing a specific virus (a "signature"). This indicates that a particular patient has a particular virus.
- **Matches:** A match exists between a signature and a sample if the entire *signature* is a *contiguous substring* of the *sample*.
- **Matching Unspecified Characters:** A 'N' character within a sample can match any value in a signature (i.e., may represent any of A/T/C/G).
- **Multiple Matches:** If there are multiple valid matches of a particular signature within a sample, only return the **positionally first** match (closest to the start of the string). One sample can contain (and match) multiple different signatures, though.

Let's look at some examples of matches.

Sample: GGGGATCGGGGGCCCCATCT
Signature: ATC

Figure 2: **Match**: The signature is a contiguous substring of the sample. We ignore the second ATC match.

Sample: GGGGATGGGGGCCCCTTTT
Signature: ATC

Figure 3: **No match**: The signature is not a contiguous substring of the sample.

Sample: GGGGATNGGGGCCCCTTTT
Signature: ATC

Figure 4: **Match**: The sample contained an 'N' that can be interpreted as a 'C'

Sample: GGGGATGGGGGCCCCTTTT
Signature: ATN

Figure 5: **Match**: The signature contained an 'N' that can be interpreted as a 'G'

In Figure 2 and Figure 3, we observe cases where the signature is a contiguous substring of the sample, and where it is not. In Figure 4 and Figure 5, we see how unspecified (N) nucleotides are allowed to match any nucleotide, whether as part of the signature or the sample. Note that if *both* the sample and signature contain an N at the same position, the match is still valid.

1.4 Definitions: Read Error Probability, Phred Scores, and Phred+33 Encoding

While this looks pretty straightforward, in the real world, **patient DNA sequences ("samples")** are not so perfect. Patient samples are "read" by a *DNA sequencing* machine, which takes a biological sample (e.g., cheek swab), and outputs the DNA sequence associated with that sample. Each time a nucleotide is read, it has a **read error probability**, which defines the probability that the nucleotide was read incorrectly. In the output file from the DNA sequencing machine, this is represented as a *Phred score*.

Example Sample DNA Sequence	T	G	G	C	A	N
Example Read Error Probability (P)	0.01	0.1	0.005	0.79433	0.06310	1
Phred (Q) Score ($-10\log_{10}(P)$)	20	10	33	1	12	0
Phred+33 Encoded ($Q+33$ in ASCII)	5	+	B	"	-	!

Table 1: Example of a DNA sequence with example error probabilities, and their corresponding Phred scores.

Phred Score and Phred+33 Encoding

- **Phred Score:** A Phred score is a measure of the quality of single read of a nucleotide. It is defined as $Q = -10\log_{10}(P)$, where P is the probability that the nucleotide was read incorrectly. A higher Phred score indicates a lower probability of error.
- **Phred+33 Encoding:** The Phred score is represented in text files by an ASCII character with the value $Q+33$. For example: a Phred score of 20 is represented by the character '5' (ASCII value 53), a score of 10 is represented by '+' (ASCII value 43), and so on.
- **Phred Score for N:** The Phred score of an N nucleotide is always 0.

The takeaway for this section is: *each nucleotide in a patient DNA sequence has an associated quality score*.

1.5 Definitions: Match Confidence

Given that each nucleotide in a patient DNA sequence has an associated quality score, we can define a **match confidence** metric as a measure of how likely a match is to be correct.

Match Confidence

- **Match Confidence:** The match confidence is the **average Phred score** of the nucleotides in the sample that matched the signature. That is: for each nucleotide in the sample that matched the signature, we take the Phred score of that nucleotide, and average them to get the match confidence.
- **Multiple Matches:** As above, if there are multiple matches of a given signature within a sample, only consider the Phred scores for the first match within the sample (closest to the start of the sample string).

We show a worked example of deriving such a match confidence score below:

Example Sample DNA Sequence	N	A	T	C	G	G	T	T
Example Phred Scores	0	10	20	30	10	20	30	5

Table 2: Example of a patient DNA sequence with Phred scores.

Sample: NATCGGTT
Signature: ATC

Phred Scores for Matched A T C sequence:
10 20 30
Match Confidence:
$\frac{10+20+30}{3} = 20.000$

Figure 6: Example of a match between the sequence in Figure 2 and a signature (ATC), with match confidence.

For each match, we will ask you to calculate its match confidence score.

2 Matcher Program Details

2.1 How your program is called

We assume that calling `make` at the *top level* (not within a subfolder!) of your submission will compile your program into an executable named `matcher`. Your program will then be called (as part of a Slurm GPU job, details mentioned below in Section 4.1), with two input file paths, as such:

`./matcher <sample_file_path>.fastq <signature_file_path>.fasta`

Your program should then print match information to `stdout` (see format in Appendix B). In reality, we will do the printing for you, and you just have to fill up a vector with the matches you detected.

2.2 What you need to implement

In this assignment, we will do all the I/O for you (see Appendix A and B for details) and time your code within **common.cc** (which contains the main entrypoint for the executable). At grading time, we will replace common.cc with our own version, so we will ignore any of your modifications to it.

Your main task is to implement a specific function within the file **kernel_skeleton.cu**. This is its signature:

```
void runMatcher(const std::vector<klibpp::KSeq>& samples,
                const std::vector<klibpp::KSeq>& signatures,
                std::vector<MatchResult>& matches);
```

The arguments to this function are:

- `samples`: A vector of `klibpp::KSeq` objects, each representing a patient DNA sequence.
- `signatures`: A vector of `klibpp::KSeq` objects, each representing a viral DNA sequence.
- `matches`: An empty vector of `MatchResult` objects, which you will populate with your matches.

The `klibpp::KSeq` struct describes a single DNA sequence – either a sample or a signature.

```
struct KSeq {
    // Sequence identifier
    std::string name;
    // Not used in this assignment
    std::string comment;
    // The actual DNA sequence containing A/T/C/G/N
    std::string seq;
    // (If it's a sample) The Phred+33 encoded quality string
    // e.g., where the character '5' refers to a Phred score of 20.
    std::string qual;
};
```

To report your results to us: You must **fill the** `matches` **vector** we pass to you as an argument, with `MatchResult` values. You can put your results in any order. A single `MatchResult` is defined as follows:

```
struct MatchResult {
    std::string sample_name;
    std::string signature_name;
    double match_score;
};
```

After your function finishes executing, we will (in common.cc) **(a) print out how long it took**, and **(b) print out the match results to stdout for you**, so that it does not count towards your execution time.

3 Implementation Guidelines

3.1 General Guidelines

To solve this problem, we recommend starting (and potentially ending) with a **brute-force algorithm**, which is ideal for GPU optimization due to its massively parallel nature. If you try other algorithms, consider carefully the impact of the unspecified nucleotide (`N`). It is not trivial to extend some algorithms to handle "wildcards".

You may want to carefully consider the following, at least:

- What grid and block dimensions to use for your kernel
- 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
- How to maximize occupancy on the GPU
- How to optimize memory transfer

The usage of external libraries is subject to approval – please email us prior to using any. Our contact details are available in Section 5.1.

3.2 Profiling

Remember that your should always *profile, and then optimize*.

You can consider using the two GPU profiling tools provided by NVIDIA: `ncu` (NSight Compute CLI) and `nsys` (NSight Systems). You can read more about them here and here. These tools are sometimes a bit difficult to use and the best practices change. Some tips: you may need to use `--clock-control none` for `ncu`, and if possible you may want to use one of the NSight Systems UI tools for profiling (if you can configure UI forwarding on the SoC cluster correctly).

Another possibility for fine-grained debugging is to use `cudaEventRecord` and associated methods to time individual components of your CUDA code. In general – you should try to find ways to make guided optimizations of your CUDA program instead of randomly changing values.

4 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.). *This also includes use of AI tools such as ChatGPT, Copilot, etc.* Please refer to our policy in the introductory lecture on AI tool usage.

The grades are divided as follows:

- 6 marks – the implementation in CUDA, split into:
  - 3 marks – the *correctness* of your solution (see Section 4.1)
  - 3 marks – the *performance* of your solution (see Section 4.2)
- 3 marks – the report (see Section 4.3)
- Up to 1 bonus mark for your report, described in Section 4.4

4.1 Correctness Requirements

Your CUDA implementation should follow these requirements in addition to anything mentioned elsewhere in this document or FAQ:

- **Language/Compiler:** Be written in C or C++ and compiled with `nvcc` (OpenMP is not allowed).
- **Executable:** Produces an executable called `matcher` when compiled with `make`. We will `make clean`, `make`, and run your `matcher` program twice during testing, once on an A100 and once on an H100.
- **Resources:** Run on the **SoC Compute Cluster**, using a **single CPU core**, using at most **20 GB of RAM**, and with *either* a single **a100-40 MIG GPU** (`xgph` node), or a single **h100-96 GPU** (`xgpi` node). That is, we will run your executable with the equivalent of
  `srun --ntasks 1 --cpus-per-task 1 --cpu_bind core --mem 20G --gpus a100-40 --constraint xgph`, or
  `srun --ntasks 1 --cpus-per-task 1 --cpu_bind core --mem 20G --gpus h100-96 --constraint xgpi`

  Both configurations should work correctly.
- **Output:** Give the same result (output) as the provided `bench-*` executables, assuming that it is implemented correctly by us. We may update this during the assignment if we find any errors. Note that correctness is heavily emphasized in this assignment!
- **CPU Usage:** You should *minimize the use of the CPU* to focus on implementing your code via the GPU. While creating a comprehensive list is hard to do (and may be further clarified in the FAQ over time), you should avoid such activities in the CPU:
  - Converting Phred+33 characters to the corresponding Phred score.
  - Actually doing any comparisons between the signature and sample (i.e., any actual matching work)
  - Computing the average match confidence scores

  For some clarity, some *non exhaustive* examples of things that *are* allowed include: preprocessing the vectors we give you into different data structures to pass into the GPU, allocating memory, calculating the lengths of things to allocate memory, type conversions, launching CUDA kernels, calling any CUDA APIs, filling in the match result vectors etc. The spirit of this requirement is to avoid doing any actual significant computation on the CPU that could be done on the GPU.

- **Input Requirements (important):**
  - **Number of signatures:** In the range $[500, 1000]$
  - **Number of samples:** In the range $[1000, 2200]$
  - **Sample DNA Length:** In the range $[100000, 200000]$.
  - **Signature DNA Length:** In the range $[3000, 10000]$.
  - **Probability that any given nucleotide is a specific DNA sequence is 'N':** In the range $[0, 0.1]$
    - For the avoidance of doubt: reference our implementation in `gen_sample.cc` or `gen_sig.cc`, that uses a `std::bernoulli_distribution`.
    - Note that while our generators use the same probability value across an entire sample or signature, we may not stick to this at grading time (e.g., different DNA sequences could have different N probabilities).
  - **Maximum percentage of samples with viruses:** 1% (however, you are not allowed to use this to implement any logic e.g., to exit early, we will do correctness checking with varied percentages)

    You may generate files matching these requirements using our provided helpers as such:

    ```
    ./gen_sig 1000 3000 10000 0.1 > sig.fasta
    ```

    and then

    ```
    ./gen_sample sig.fasta 2000 20 1 2 100000 200000 10 30 0.1 > samp.fastq
    ```

    Note that the sample generation will create a file over 500 MB and might take a while. You may want to run this via Slurm.
- Not have any race conditions, data races, deadlocks, or any other synchronization issues.
- Not use any external utilities or libraries (e.g., those not provided with the C/C++ environment) that are not otherwise approved by the teaching team.

4.2 Performance Requirements

We provide two benchmark executables for you to beat: **bench-a100** and **bench-h100**, that can only be run on the A100-40 and H100-96 respectively.

These are the performance requirements:

- **Input Files:** We will measure performance given the same inputs for the benchmarks and your CUDA implementation, for at least the input sizes mentioned in Section 4.1.
- **Benchmark Executables:** We will test your code against **both** bench-a100 and bench-h100, running on the appropriate GPU types. We do not specify the performance mark distribution.
- **Correctness affects Performance marks:** Note that not getting full marks on correctness will impact your performance score - as it's very easy to get a "fast" solution by doing nothing!

4.3 Report Requirements

4.3.1 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 4.3.2.
- 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.

4.3.2 Content

Your report should contain:

- (1 mark) A brief description of:
  1. the algorithm used by your program
  2. the parallelisation strategy – for example (not exhaustive): how work (signatures, input files) are divided into kernels, threads, and blocks
  3. your choice and justification for grid and block dimensions
  4. how you handle memory in your program, and if applicable, how shared memory 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. How do different factors of the input (sequence lengths, number of samples and signatures, percentage of N characters, percentage of signatures matching samples, etc) affect the overall runtime of the program?
  2. How can you explain these observations?
- (1 mark) Describe at least **two** specific performance optimisations you attempted with analysis and supporting performance measurements. For each of these, include at least one summary graph in your report and link to your *raw* supporting measurements (for example, a .csv file in the repository, a Google sheet, or appendix section). **You must include the raw data somewhere** so that we can verify it.

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, including **queuing on the SoC cluster**. **Start early** and test selectively.

4.4 Bonus - Algorithmic Improvements and Analysis

You may obtain 1 bonus mark for writing another matcher implementation, and adding another section to your report. These are the requirements:

- You must implement a **significantly different** algorithm than your original submission. For instance, a "more CUDA optimized" bruteforce will not be considered.
- This implementation must be **considerably faster** than the original implementation.
- You must include a **clear analysis** of the performance differences and *explore why this is the case*.
- The bonus executable must be created via running `make bonus`, producing an executable called `matcher-bonus`
- You should include an extra 1 page in your report with your analysis (not counted towards page limit).

4.5 Template Code

We provide skeleton code that can be used as a starting point for your CUDA implementation.

File name	Description
`kernel_skeleton.cu`	The file you should modify to add your CUDA code. Contains the runMatcher function that you should be implementing.
`common.h / common.cpp`	The main entrypoint to the program, where we do the I/O for you, call your runMatcher function, and time the execution of your program. `common.h` also includes important definitions for your program. Any modifications you make to these files will be ignored.
`kseq/`	Library for reading input files, do not modify.
`bench-a100 and bench-h100`	Our benchmark executables.
`Makefile`	The build script. You may modify this as necessary. Remember to build the code on an actual GPU node!

Table 3: The list of main provided code files and executables in the skeleton

We also include some helper scripts and executables These are purely for some convenience – we don't guarantee their correctness and you don't have to use them.

File name	Description
`download_fasta.sh`	If you'd like more interesting testcases, this downloads a real FASTA virus signature file from https://www.ncbi.nlm.nih.gov/nuccore/, given an `accession number` as an argument. For instance, given this virus at https://www.ncbi.nlm.nih.gov/nuccore/NC_045512.2, the accession number is `NC_045512.2`
`gen_sample`	Executable that generates a FASTQ sample file from a set of arguments including a FASTA virus file. Read the gen_sample.cc file for more information.
`gen_sig`	Executable that generates a FASTA signature file from a set of arguments. Read the gen_sig.cc file for more information.

Table 4: Some helper code and executables

5 Admin

5.1 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 Sriram (srirams@comp.nus.edu.sg).

5.2 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/.

5.3 Deadline and Submission

Assignment submission is due on **Tue, 22 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/TVaexnZj.

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`.

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 **(in the right format!)**, and GitHub Classroom should contain your code and report.

Appendices

A Input Formats

While we handle the I/O for you, you might find information about the input file format useful, as it may help in generating test cases.

Your program will take two input files. The first is a list of **patient DNA sequence ("samples")** in the **FASTQ** format. The second is a list of **viral DNA sequences ("signatures")** in the **FASTA** format.

A.1 Samples: FASTQ files

The **FASTQ** file format (file extension `.fastq`) is an industry-standard format for storing actual DNA sequences from DNA sequencing machines.

FASTQ file format A FASTQ file is a text file that describes the result of reading multiple DNA sequences, where each DNA sequence is represented by four line-separated fields:

- Field 1: begins with a '`@`' character and is followed by a string that is called the *sequence identifier* – a unique string describing the current sequence. This could be the name of a patient, for example. The string may contain one or more spaces, in which case, the string is separated by the first space into a `name` entry (this is the "sequence identifier" in this case) and a `comment` entry. You can ignore the `comment`.
- Field 2: The actual DNA sequence data. Sometimes, this field is split across multiple lines (to keep each line to around 70 characters).
- Field 3: The '+' character (and typically nothing else, can be ignored)
- Field 4: A string with the same length as Field 2, in the Phred+33 format.

Below is an example of a FASTQ file that describes two *two* patient DNA sequences (hence, $2 \times 4 = 8$ lines).

```
@PATIENT-ZERO-CHEEK-SWAB-1
GATTTGGGGTTCAAAGCAGTATCGATCAAATAGTAAATCCATTTGTTCAACTCACAGTTT
+
!''*((((***+))%%%++)(%%%%).1***-+*''))**55CCF>>>>>>CCCCCCC65
@TINKY-WINKY-NASAL-SWAB-5
TCGCACTCAACGCCCTGCATATGACAAGACAGAATC
+
<>;##=><9=AAAAAAAAAA9#:<#<;<<<????#=
```

Figure 7: Example of a FASTQ file with two sequences

A.2 Signatures: FASTA files

The **FASTA** file format (file extension `.fasta`) is another industry-standard format for storing DNA sequences.

FASTA file format A FASTA file is a text file that describes the result of reading multiple DNA sequences, where each DNA sequence is represented by two line-separated fields:

- Field 1: begins with a '>' character and is followed by a string that is called the *sequence identifier* – a unique string describing the current sequence.
- Field 2: The actual DNA sequence data. Sometimes, this field is split across multiple lines (to keep each line to around 70 characters).

Below is an example of a FASTA file that describes two *two* viral DNA sequences (hence, $2 \times 2 = 4$ lines).

```
>COVID-19
GATTTGGGGTTCAAAGCAGTATCGATCAAATAGTAAATCCATTTGTTCAACTCACAGTTT
>EBOLA
TCGCACTCAACGCCCTGCATATGACAAGACAGAATC
```

Figure 8: Example of a FASTA file with two sequences

B Output Format

Similarly, while we handle the I/O for you, you might find the output format useful for parsing / verification purposes.

The output format is relatively simple — output one line per match to `stdout` including the sample sequence identifier, signature sequence identifier, and match confidence score. There is no requirement that the lines are sorted, or out in in any particular order. Each line should follow the format from Figure 9:

`sample-sequence-identifier: signature-sequence-identifier (match-confidence)`

Figure 9: Format of one output line

For the avoidance of doubt:

- The `sample-sequence-identifier` is the `.name` field of the sample.
- A space character
- The `signature-sequence-identifier` is the `.name` field of the signature.
- A space character and an opening parenthesis
- The `match-confidence` value to *3 decimal places*.
- A closing parenthesis

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

```
TINKY-WINKY-NASAL-SWAB-5: Ebola (50.050)
PATIENT-ZERO-CHEEK-SWAB-1: COVID-19 (12.452)
```

C Potential Optimization: Asynchronous CUDA Kernels

There are many options to optimize CUDA applications, which can be found in the CUDA Best Practices Guide. One such option for memory intensive applications is the idea of **asynchronous (concurrent) kernel launches** and execution, described more in this section of the 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.

Note that **this isn't like the spatial partitioning recommendation we gave you the Assignment 1 PDF** – CUDA is complex, and there's no guarantee what we've mentioned here will make your code faster vs using other techniques. In general, please refer to the CUDA Programming Guide for more ideas if necessary.