Parallelization Project

Course	**CAB401_25se2 High Performance and Parallel Computing**
Unit Coordinator	A/Prof Wayne Kelly
Assessment	CAB401 Parallelization Project
Author	Oliver Stewart n11588608

This document details the parallelization of a computationally intensive sequence processing task using CUDA. It outlines the challenges encountered, the implementation strategy for GPU acceleration, and potential avenues for future optimization leveraging techniques like GPUDirect Storage.

Overview of Original Sequential Application

The original sequential application computes signatures for genomes - a set of all the chromosomes of an organism or cell. The application does this by employing a series of data splits, matrix addition, hash maps and data casting strategies to iterarively read through a `.fasta` file and build an output file that contains all the signatures.

Changes to the original application

There were a few changes that had to be made to the original sequential algorithm provided by CAB401 as there were bugs with the original code and how the fasta data lines were read. The original application also used Windows C++ functions which needed to be converted to their standard C++ counterparts or completely rehauled and the hashing algorithm was changed to something smaller so that it could easily translated and be implemented on a GPU kernel.

Reading `.fasta` file

The original application read the file line by line using `fgets` however, with my aforementioned adjustment to this. The application now reads the entire file into memory at the start of runtime and then closes the file. Afterwards it parses the file in the intended previous way, ignore every odd line (which just contains metadata) and save the memory position and length of the second line (containing the actual genome sequence) in a vector of custom structs containing a `const char*` and an `int` for processing later. This is sequentially fed into the `partition` function where all the processing starts.

Partitioning of `.fasta` data

After reading the genome sequence, the `partition` function divides it into sections of size `PARTITION_SIZE`, with a 50% overlap with the previous section. If the final section's length is less than `PARTITION_SIZE`, whatever is left is used. The script then iteratively processes each partition using the `compute_signature` function.

Sliding window

After partitioning, `compute_signatures` further splits each genome segment is further processed using a sliding window with a size of `WORDLEN`. These windows are then fed into `signature_add`.

Computing signatures

After being split through a sliding window algorithm the `signature_add` works on each of the sliding window segments iteratively getting the hash of the sliding window (via memory in a hash map or by manual computation) and computing a row-wise addition with all other sliding windows within each partition producing the signature for the partition.

Manual computation of signatures

The original algorithm used `ISAAC-rand.cpp` to generate pseudo-random numbers for the hashing algorithm, to make the pseudo-random number generator CUDA kernel friendly this was replaced with the PGC hash which I adapted from this article. This number generator is used to randomly select `DENSITY` positive and negative numbers in an array of `SIGNATURE_LEN` signed bytes, seeded by the characters in the input sliding window.

Reduction of signatures

After all of the sliding windows for the partition have been got and added with all others in the partition, the partition signature undergoes a reduction. For every 8 bytes in the partition signature, they are bitwise shifted onto a single byte with the condition `(byte > 0)` and outputted into a final condensed array of size `SIGNATURE_LEN / 8` and outputted to the signature file.

Analysis of Parallelism Opportunities

By critiquing the CPU-implementation in the context of a GPU-implementation we can decipher parts of the script that are better suited or can be restructured for GPU compute. While critiquing the CPU-implementation, caution is required to preserve control and data dependencies in the original program to ensure the parallel implementation produces the same result as the sequential algorithm.

Splitting actions

The first opportunity for a parallelism within this script is the partitioning and sliding window actions. As each iteration within the CPU-implementation does not modify the original `.fasta` data there are no data dependencies.

Signature computation

As well as having no data dependencies, the row-wise operator that reduces all the kmer signatures into a single partition signature is a commutative addition (an operation where changing the order of the operands does not change the result). Meaning there are also no control dependencies.

As such, the entire script is safe for parallelism.

Parallelization Strategy and Mapping to Processors

Analyzing the parallelism opportunities from the CPU-implementation exposes a maxima and a later minima data lengths computation occurs at. The maxima data length computation to the kmer-signature computation, after all the partitioning and sliding window splits, the most fine grain computation occurs at the kmer-signature for every `WORDLEN` split of the data. The minima data length computation occurs at the final byte reduction which reduces all the kmer-signatures into their partition through a row-wise addition and finally simplified to fit within `PARTITION_SIZE / 8` bits. As there are no data flow or control dependencies for the algorithm, once the correct number of threads is launched and each thread is mapped to the correct shared memory region, the same maxima and minima computation can be implemented as two separate kernels to do work on the GPU.

Determining the number of threads for kmer signing

To efficiently allocate GPU threads for signing kmers, we need to understand how the genome sequence is divided into smaller chunks. The original algorithm splits the genome in a two-stage process: first into partitions, and then each partition into sliding windows. Here's a breakdown of the process:

1. **Partitioning:** The genome is divided into segments (partitions) of `PARTITION_SIZE` characters. These partitions overlap by 50%.
2. **Sliding Window Split:** Within each partition, a sliding window of size `WORDLEN` is applied. This creates multiple overlapping windows, each yielding a new kmer. It's important that the sliding window doesn't extend beyond the end of the partition.
3. **Iteration:** After processing a partition, the algorithm advances only `PARTITION_SIZE / 2` characters in the genome sequence and repeats steps 1 and 2 until the entire genome is processed.

The number of kmers per partition

Calculating the number of kmers within a partition is straightforward. Because the sliding window creates a new kmer for almost every character (excluding the last WORDLEN - 1 characters), the number of kmers per partition is:

$$l_{\text{partition}} - (L_{\text{word}} - 1)$$

or

$$l_{\text{partition}} - L_{\text{word}} + 1$$

kmers, where $l_{\text{partition}}$ refers to the number of characters within the current partition, and L_{word} refers to the constant `WORDLEN`.

Calculating the total number of partitions

Determining the number of partitions in the genome is more complex, especially when the genome length isn't evenly divisible by `PARTITION_SIZE / 2`. To account for this, we need a function that takes the genome length (x) as input and calculates the estimated number of kmers requiring signatures.

Modelling the kmer count

Initially, the relationship between the number of kmers based on the genome sequence is a linear $y = x - (L_{\text{word}} - 1)$ while $0 \leq x < L_{\text{partition}}$ where $L_{\text{partition}}$ is the constant `PARTITION_SIZE`, and then, the linear relationship "jumps" $\frac{L_{\text{partition}}}{2}$ to $y = x - (L_{\text{word}} - 1) + \frac{L_{\text{partition}}}{2}$ for the period $L_{\text{partition}} \leq x < \frac{3L_{\text{partition}}}{2}$, capturing this relationship, the following formula was constructed which, given a genome sequence length, returns the number of kmer's are signed:

$$f_{\text{signingThreads}}(x) = y = (x - L_{\text{word}} + 1) + \left\lfloor \frac{L_{\text{partition}}}{2} - L_{\text{word}} + 1 \right\rfloor \left\lceil 2 \times \frac{\max(x, L_{\text{partition}}) - L_{\text{partition}}}{L_{\text{partition}}} \right\rceil$$

where y is the number of kmers and x is the genome sequence length. This formula accounts for the linear relationship and the jump caused by the partitioning process.

Allocating shared memory for all kmer signing threads

While it is possible for the CPU to allocate separate regions of VRAM that separate the data the GPU thread is working on, this would add significant CPU overhead and duplicate memory usage to a problem that can be solved using shared memory and some confusing memory mapping for each thread.

As each signing threads just need access to a shared region of memory containing the genome sequence, this makes up once of the allocated memory regions, the second region will contain the output data for all the computed kmer signatures, to determine how much memory to allocate for these, the following functions were used:

$$f_{\text{signingInputBytes}}(x) = y = x$$
$$f_{\text{signingOutputBytes}}(x) = y = f_{\text{signingThreads}}(x) \times L_{\text{signature}}$$

where y is the number of bytes to allocate, and x is the genome sequence length.

Mapping signing threads to shared memory

After allocating the whole genome sequence to a section of shared VRAM for all $f_{\text{signingThreads}}(x)$ each signing thread needs to point to the correct region of that memory containing the kmer to be signed. For this, the same algorithm above is analysed and the flow characteristics are extracted, these characteristics are:

- After $L_{\text{partition}} - L_{\text{word}} + 1$ threads, the next thread should move back $\frac{L_{\text{partition}}}{2}$ characters in shared memory,
- Every $\frac{L_{\text{partition}}}{2}$ thereafter the next thread should move back $\frac{L_{\text{partition}}}{2}$ characters in shared memory.

Capturing this into a function which takes the current thread id x and outputs the memory offset pointing to the kmer signature being computed y, the following function is created:

$$f_{\text{signingInputOffset}}(x) = y = x + \left(L_{\text{word}} - L_{\text{partition}} + \left\lfloor \frac{L_{\text{partition}}}{2} \right\rfloor - 1 \right) \left\lfloor \frac{x}{L_{\text{partition}} - L_{\text{word}} + 1} \right\rfloor$$

As for mapping each thread to the output memory region, the function is very simple:

$$f_{\text{signingOutputOffset}}(x) = y = x \times L_{\text{signature}}$$

Where x is the current thread id and y is the start of the region of memory of size `SIGNATURE_LEN` in the output array to write to.

Determining the number of threads for signature reduction

Like the maxima, kmer signature, compute, efficient GPU thread allocation requires a similar process of understanding how the original CPU-implementation reduces partitions into the outputted bytes. The original algorithm reduces the signatures in two-steps, a row-wise addition for all signatures in a partition and by using binary operators to reduce again to $\frac{L_{\text{signature}}}{8 \times 8}$ bytes. Here's a breakdown of that process:

1. **Row-wise addition**: Add all the kmer signatures ($l_{\text{partition}} - L_{\text{word}} + 1$ signatures) within the partition row by row.
2. **Byte-wise reduction**: For each eight integer values within the row-wise reduced signature ($L_{\text{signature}}$ integers), shift `1` and `0` bits onto a byte based on the result of the condition: `(n > 0)`.

Determining the number of partitions within a genome sequence

A simple conversion of this algorithm to a GPU-implementation would let each thread compute the output bytes for each partition, and to calculate the number of threads for this style of algorithm, the following formula can be used:

$$f_{\text{reducingThreads}}(x) = y = \left\lceil \frac{f_{\text{signingThreads}}(x)}{L_{\text{partition}} - L_{\text{word}} + 1} \right\rceil$$

where y is the number of partitions required for the genome sequence length x.

Determining the number of output bytes from a genome sequence

However, each of these partitions compute multiple bytes and as each byte is computed independent of other bytes within the partition, each byte can be computed separately, to determine the number of bytes required for the genome sequence, a similar function can be used:

$$f_{\text{reducingThreads}}(x) = y = 8 \left\lceil \frac{f_{\text{signingThreads}}(x)}{L_{\text{partition}} - L_{\text{word}} + 1} \right\rceil$$

Allocating shared memory for all reducing threads

As the input for the reducing threads is the already allocated $f_{\text{signingOutputBytes}}(x)$ from the previous function, only the output bytes for the reducing threads needs to be calculated. This is very simple as the output length is the same as the number of threads due to each thread working on a single output byte, the relationship is:

$$f_{\text{reducingOutputBytes}}(x) = y = f_{\text{reducingThreads}}(x)$$

Mapping reducing threads to shared memory

After mapping all the memory for outputting the reduction results from the GPU, the individual threads need to point to the sections of the input memory for the row-wise summation for the output byte, as the row-wise summation for the whole signature is split across the threads, a new approach is required.

Take a theoretical partition of the output from the signing threads:

Constant	Value
$L_{\text{signature}}$	24
$L_{\text{partition}}$	8

```
s1: - 0 + 0 - 0 + 0 - 0 + 0 - 0 + 0 - 0 + 0 - 0 + 0
s2: 0 + 0 - 0 + 0 - 0 + 0 - 0 + 0 - 0 + 0 - 0 + 0 -
s3: + 0 - 0 + 0 - 0 + 0 - 0 + 0 - 0 + 0 - 0 + 0 - 0
s4: 0 - 0 + 0 - 0 + 0 - 0 + 0 - 0 + 0 - 0 + 0 - 0 +
s5: - 0 + 0 - 0 + 0 - 0 + 0 - 0 + 0 - 0 + 0 - 0 + 0
s6: 0 + 0 - 0 + 0 - 0 + 0 - 0 + 0 - 0 + 0 - 0 + 0 -
s7: + 0 - 0 + 0 - 0 + 0 - 0 + 0 - 0 + 0 - 0 + 0 - 0
s8: 0 - 0 + 0 - 0 + 0 - 0 + 0 - 0 + 0 - 0 + 0 - 0 +
```

We need to map each thread to process the following row-wise additions:

```
  | thread i   | thread i+1  | thread i+2  |
s1:|- 0 + 0 - 0 + 0|- 0 + 0 - 0 + 0|- 0 + 0 - 0 + 0|
s2:|0 + 0 - 0 + 0 -|0 + 0 - 0 + 0 -|0 + 0 - 0 + 0 -|
s3:|+ 0 - 0 + 0 - 0|+ 0 - 0 + 0 - 0|+ 0 - 0 + 0 - 0|
s4:|0 - 0 + 0 - 0 +|0 - 0 + 0 - 0 +|0 - 0 + 0 - 0 +|
s5:|- 0 + 0 - 0 + 0|- 0 + 0 - 0 + 0|- 0 + 0 - 0 + 0|
s6:|0 + 0 - 0 + 0 -|0 + 0 - 0 + 0 -|0 + 0 - 0 + 0 -|
s7:|+ 0 - 0 + 0 - 0|+ 0 - 0 + 0 - 0|+ 0 - 0 + 0 - 0|
s8:|0 - 0 + 0 - 0 +|0 - 0 + 0 - 0 +|0 - 0 + 0 - 0 +|
```

To be able to map this, a technique was used that gets the first signature's offset in the partition as well as the offset of where to start adding signatures within the partition, these functions are:

$$f_{\text{partitionOffset}}(x) = y = L_{\text{signature}}(L_{\text{partition}} - L_{\text{word}} + 1) \left\lfloor \frac{x}{8} \right\rfloor$$

$$f_{\text{signatureOffset}}(x) = y = 8x \mod L_{\text{signature}}$$

With these defined at the start of every thread, we can then iterate through the row-wise summation, adding $L_{\text{signature}}$ every 8 integers to add one section of the total computed signature.

As the output size is the same as the number of threads, the output offset is just:

$$f_{\text{reducingOuputOffset}}(x) = y = x$$

Where x is the current thread index.

GPU implementation flow

Evaluating all of the memory and thread allocation and mapping, the following workflow will be implemented in CUDA C++ to implement this GPU algorithm:

1. Copy `.fasta` file content to pinned CPU and GPU memory.
2. Parse `.fasta` content in CPU land to a struct array containing the location and size of all the genome sequences.
3. Allocate an output buffer to shared CPU and GPU memory.
4. Allocate enough buffer VRAM for the worst case, longest genome sequences.
5. Allocate a static number of CUDA Streams for asynchronous job launching.
6. Launch all the jobs for signing and reducing.
7. Wait for all jobs to finish.

8. Write output buffer to output file.

Performance Evaluation: Timing, Profiling and Speedup

To be able to evaluate the performance of the new script, including any speedup, we will use a stopwatch in C++ to time the signature computation, and use Nsight Compute to profile the runtime. To change the compute load dynamically, we will modify the provided `qut3.fasta` in two ways, one test will involve doubling the number of sequences each iteration and another test will involve duplicating each sequence on itself, doubling the length of all the genome sequences in the fasta. The results of this tests are:

These graphs show that on average, the GPU implementation achieves a 4.15x speed up over the CPU-implementation. While this is a decent speed up, on running the script with Nsight Compute, interpreting and extracting the problem areas from the csv output, it can be seen that not much of the GPU is actually being put to work:

Average Signing Kernel Occupancy Results

Row Labels	Min of Metric Value (%)	Average of Metric Value (%)	Max of Metric Value (%)
Achieved Active Warps Per SM	2.14	3.11	5.01
Achieved Occupancy	4.46	6.48	10.43
Theoretical Active Warps per SM	48	48	48
Theoretical Occupancy	100	100	100

Average Reducing Kernel Occupancy Results

Row Labels	Min of Metric Value (%)	Average of Metric Value (%)	Max of Metric Value (%)
Achieved Active Warps Per SM	3.55	3.95	4.69
Achieved Occupancy	7.39	8.23	9.77
Theoretical Active Warps per SM	48	48	48
Theoretical Occupancy	100	100	100

And while these issues don't present themselves when running on a release build due to the CPU being able to launch enough jobs to fill the GPU, this does expose a significant flaw in the implementation. The implementation launches one job per sequence in the `.fasta` and as one might expect, files with roughly the same number of genomes take significantly longer if it is the number of sequences that are changing rather than the sequence length as seen above in the Average Execution Time graphs.

Future Improvement

A better implementation would require the signing and reducing kernels to be able to handle mapping the entire `.fasta` content. Currently, the CPU is orchestrating a job for each line of the `.fasta` file and while the CPU is able to do this for ~2048 jobs concurrently, this is using software to do the job of the significantly more efficient Streaming Multiprocessor. Modern NVIDIA GPU's have many Streaming Multiprocessors (SMs) which can handle massive jobs like launching kernels to process the entire `.fasta` file. As well as properly utilizing SMs, NVIDIA also have functionality for direct data transfer to and from NVMe storage by the GPU, using initial storage coordinates from the operating system, GPUDirect Storage can use the GPU to directly write to storage, bypassing CPU and RAM. Meaning an even better implementation would take advantage of GDS to read the `.fasta` file.

Correctness Verification of Parallel Implementation

To verify the parallel implementation, a simple bash command to get the number of different characters and the number of total characters can be used to get a percentage similarity for the output of the two implmentations:

```
cmp -l qut_small.fasta.part16_sigs03_64 qut_small.fasta.part16_sigs03_64_cuda | wc -l
wc -c < qut_small.fasta.part16_sigs03_64
wc -c < qut_small.fasta.part16_sigs03_64_cuda
```

which outputs:

```
664487
40103896
40103896
```

meaning the GPU output is:

$$1 - \frac{664487}{40103896} = 0.98 = 98\%$$

the same as the CPU output. This small discrepancy is weird due to both file's containing the same total number of characters, as well as on manual inspection, the first and last 16 bytes match perfectly. Manually inspecting the `head` of the `cmp [...]` command shows that the differing bytes don't seem to be character's handled differently by different operating systems and are probably errors due to rounding errors.

Tools, Compilers and Techniques Employed

This project primarily utilized docker images from the GNU Compiler Collection and NVIDIA to make development with different environments easy, make cross-compatible, reproducible results and make installation easy.

Get started with the CPU-implementation

This one is easy, you can do it on anything that'll install docker, but I'd recommend using WSL if you're on Windows:

Step 1: Install docker

Step 2: Clone the repo

```
git clone https://github.com/ostew5/kmer-signatures.git
```

Step 3: Run the run file

```
cd kmer-signatures
sh run:cpu
```

Get started with the GPU-implementation

This is surprisingly easy, I promise, you've got this. You'll need a NVIDIA GPU that supports CUDA version 13.

Step 0: Enable WSL 2 if you're using Windows

`nvidia-smi` needs WSL 2, if it isn't activated open PowerShell as Admin and run:

```
wsl --install
```

Step 1: Install docker

Step 2: Install NVIDIA container tools

Step 3: Clone the repo

```
git clone https://github.com/ostew5/kmer-signatures.git
```

Step 4: Run the run file

```
cd kmer-signatures
sh run:cuda
```

Step 5: Optionally run the profiling script

This uses NVIDIA Nsight Compute, which needs a GPU with Compute Capability 5.0 or newer

```
sh run:cuda_profiling
```

Reflections and Lessons Learned

This parallelisation project was my first time working with CUDA C++. The paradigms associated with programming on GPU are very interesting and I'm surprised with the amount of abstraction that comes with programming with CUDA, I was originally expecting to have to configure a lot more to get anything running on a graphics card. The main lesson learnt was not allocating enough work for the GPU, I didn't realise how powerful Streaming Multiprocessors were and that I had to treat CUDA more like an "API" and that ignorantly flooding the GPU with a massive job would perform better than the meticulous micro jobs that I sent. Other than that I loved coding with CUDA and I can see projects in the past that would benefit from the massively parallel capabilities of GPU compute.