COM... 21 Parallel Computing with ...ocessing Units (GPUs)

Assignment (80% of module mark)

Deadline: 5pm Friday 17th May (Week 12)

Starting Code: *Download Here*

Document Changes

Any corrections or changes to this document will be noted here and an update will be sent out via the course's Google group mailing list.

Document Built On: 17 January 2024

Introduction

This assessment has been designed against the module's learning objectives. The assignment is worth 80% of the total module mark. The aim of the assignment is to assess your ability and understanding of implementing and optimising parallel algorithms using both OpenMP and CUDA.

An existing project containing a single threaded implementation of three algorithms has been provided. This provided starting code also contains functions for validating the correctness, and timing the performance of your implemented algorithms.

You are expected to implement both an OpenMP and a CUDA version of each of the provided algorithms, and to complete a report to document and justify the techniques you have used, and demonstrate how profiling and/or benchmarking supports your justification.

The Algorithms & Starting Code

Three algorithms have been selected which cover a variety of parallel patterns for you to implement. As these are independent algorithms, they can be approached in any order and their difficulty does vary. You may redesign the algorithms in your own implementations for improved performance, providing input/output pairs rem[illegible]

The refer[illegible]d starting code are available to download from: https://c[illegible]E-Sheffield/COMCUDA_assignment_c614d9bf/zip/re[illegible]

Each of t[illegible]ibed in more detail below.

Standa[illegible]pulation)

Thrust/CUB may not be used for this stage of the assignment.

You are provided two parameters:

- An array of floating point values `input`.
- The length of the input array `N`.

You must calculate the standard deviation (population) of `input` and return a floating point result.

The components of equation 1 are:

- σ: The population standard deviation
- $\sum$ = The sum of...
- x_i = ...each value
- μ = The mean of the population
- N: The size of the population

$$\sigma = \sqrt{\frac{\sum_{i=1}^{N}(x_i - \mu)^2}{N}} \tag{1}$$

The algorithm within `cpu.c::cpu_standarddeviation()` has several steps:

1. Calculate the `mean` of `input`.
2. Subtract `mean` from each element of `input`.
3. Square each of the resulting elements from the previous step.
4. Calculate the `sum` of the resulting array from the previous step.
5. Divide `sum` by `n`.
6. Return the square root of the previous step's result.

It can be executed either via specifying a random seed and population size, e.g.:

```
<executable> CPU SD 12 100000
```

Or via specifying the path to a `.csv` input file, e.g.:

```
<executable> CPU SD sd_in.csv
```

Convolution

You are p… s:

- A 1… y `input` image.
- A 1… ay `output` image.
- The… put.
- The… nput.

Figure 1: An example of a source image (left) and it's gradient magnitude (right).

You must calculate the gradient magnitude of the greyscale image `input`. The horizontal ($\mathbf{G}_x$) and vertical ($\mathbf{G}_y$) Sobel operators (equation 2) are applied to each non-boundary pixel ($\mathbf{P}$) and the magnitude calculated (equation 3) to produce a gradient magnitude image to be stored in `output`. Figure 1 provides an example of a source image and it's resulting gradient magnitude.

$$\mathbf{G}_x = \begin{bmatrix} +1 & 0 & -1 \\ +2 & 0 & -2 \\ +1 & 0 & -1 \end{bmatrix} * \mathbf{P} \quad \text{and} \quad \mathbf{G}_y = \begin{bmatrix} +1 & +2 & +1 \\ 0 & 0 & 0 \\ -1 & -2 & -1 \end{bmatrix} * \mathbf{P} \tag{2}$$

$$\mathbf{G} = \sqrt{{\mathbf{G}_x}^2 + {\mathbf{G}_y}^2} \tag{3}$$

A convolution is performed by aligning the centre of the Sobel operator with a pixel, and summing the result of multiplying each weight with it's corresponding pixel. The resulting value must then be clamped, to ensure it does not go out of bounds.

$$\sum \left(\begin{bmatrix} 1 & 6 & 12 \\ 77 & \mathbf{5} & 34 \\ 56 & 90 & 10 \end{bmatrix} \circ \begin{bmatrix} +1 & 0 & -1 \\ +2 & 0 & -2 \\ +1 & 0 & -1 \end{bmatrix} \right) = \sum \begin{bmatrix} 1 & 0 & -12 \\ 154 & 0 & -68 \\ 56 & 0 & -10 \end{bmatrix} = 121 \tag{4}$$

The convolution operation is demonstrated in equation 4. A pixel with value **5** and it'[illegible] are shown. This matrix is then component-wise mult[illegible]uct) by the horizontal Sobel operator and the compone[illegible]rix are summed.

Pixels at [illegible] do not have a full Moore neighbourhood, and therefore [illegible] such, the output image will be 2 pixels smaller in each d[illegible]

The algor[illegible] in `cpu.c::cpu_convolution()` has four steps performed per non-boundary pixel of the input image:

1. Calculate horizontal Sobel convolution of the pixel.
2. Calculate vertical Sobel convolution of the pixel.
3. Calculate the gradient magnitude from the two convolution results
4. Approximately normalise the gradient magnitude and store it in the output image.

It can be executed via specifying the path to an input `.png` image, optionally a second output `.png` image can be specified, e.g.:


```
<executable> CPU C c_in.png c_out.png
```

Data Structure

You are provided four parameters:

- A sorted array of integer keys `keys`.
- The length of the input array `len_k`.
- A preallocated array for output `boundaries`.
- The length of the output array `len_b`.

You must calculate the index of the first occurrence of each integer within the inclusive-exclusive range `[0, len_b)`, and store it at the corresponding index in the output array. Where an integer does not occur within the input array, it should be assigned the index of the next integer which does occur in the array.

This algorithm constructs an index to data stored within the input array, this is commonly used in data structures such as graphs and spatial binning. Typically there would be one or more value arrays that have been pair sorted with the key array (`keys`). The below code shows how values attached to the integer key `10` could be accessed.

```
for (unsigned int i = boundaries[10]; i < boundaries[11]; ++i) {
    float v = values[i];
    // Do something
}
```

The algorithm implemented within `cpu.c::cpu_datastructure()` has two steps:

1. An intermediate array of length `len_b` must be allocated, and a `histogram` of t[illegible]culated within it.
2. An [illegible]an) operation is performed across the previous ste[illegible]he output array `boundaries`.

Figure 2 [illegible]le of this algorithm.

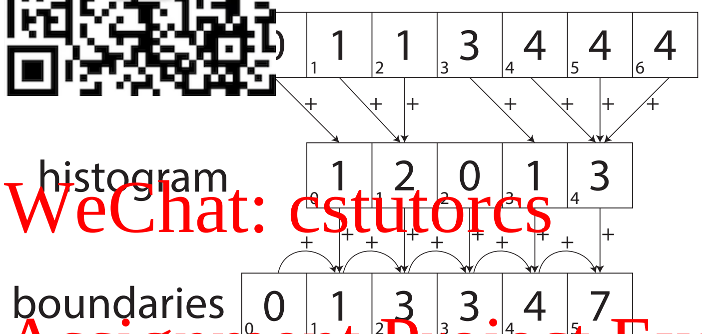

Figure 2: An example showing how the input `keys` produces `boundaries` in the provided algorithm.

It can be executed via specifying either a random seed and array length, e.g.:


```
<executable> CPU DS 12 100000
```

Or, via specifying the path to an input `.csv`, e.g.:

```
<executable> CPU DS ds_in.csv
```


Optionally, a `.csv` may also be specified for the output to be stored, e.g.:

```
<executable> CPU DS 12 100000 ds_out.csv
<executable> CPU DS ds_in.csv ds_out.csv
```


The Task

Code

For this assignment you must complete the code found in both `openmp.c` and `cuda.cu`, so that they perform the same algorithm described above and found in the reference implementation (`cpu.c`), using OpenMP and CUDA respectively. You should not modify or create any other files within the project. The two algorithms to be implemented are separated into 3 methods named `openmp_standarddeviation()`, `openmp_convolution()` and `openmp_datastructure()` respectively (and likewise for CUDA).

You should implement the OpenMP and CUDA algorithms with the intention of achieving the fastest performance for each algorithm on the hardware that you use to develop and test your assignment.

It is imp[illegible] memory as memory leaks could cause the benchma[illegible] the algorithm, to run out of memory.

Report[illegible]

You are e[illegible]ort alongside your code submission. For each of the 6 algo[illegible]ent you should complete the template provided in Appen[illegible]our chance to demonstrate to the marker that you understand what has been taught in the module.

Benchmarks should **always be carried out in Release mode**, with timing averaged over several runs. The provided project code has a runtime argument `--bench` which will repeat the algorithm for a given input 100 times (defined in `config.h`). It is important to benchmark over a range of inputs, to allow consideration of how the performance of each stage scales.

Deliverables

You must submit your `openmp.c`, `cuda.cu` and your report document (e.g. `.pdf`/`.docx`) within a single zip file via Moodle, before the deadline. Your code should build in the Release mode configuration without errors or warnings (other than those caused by IntelliSense) on Diamond machines. You do not need to hand in any other project or code files other than `openmp.c`, `cuda.cu`. As such, it is important that you do not modify any of the other files provided in the starting code so that your submitted code remains compatible with the projects that will be used to mark your submission.

Your code should not rely on any third party tools/libraries **except for those** introduced within the lectures/lab classes. Hence, the use of Thrust and CUB is permitted **except for the standard deviation algorithm**.

Even if you do not complete all aspects of the assignment, partial progress should be submitted as this can still receive marks.

Marking

When marking, both the correctness of the output, and the quality/appropriateness of the technique used will be assessed. The report should be used to demonstrate your understanding of the module's theoretical content by justifying the approaches taken and showing their impact on the performance. The marks for each stage of the assignment will be distributed as follows:

[illegible]	[illegible]penMP (30%)	CUDA (70%)
[illegible]	[illegible]6%	22.4%
[illegible]	[illegible].2%	23.8%
[illegible]	[illegible].2%	23.8%

The CUD[illegible] weighted as it is more difficult.

For each [illegible]he distribution of the marks will be determined by the following criteria:

1. Quality of implementation

- Have all parts of the stage been implemented?
- Is the implementation free from race conditions or other errors regardless of the output?
- Is code structured clearly and logically?
- How optimal is the solution that has been implemented? Has good hardware utilisation been achieved?

2. Automated tests to check for correctness in a range of conditions

- Is the implementation for the specific stage complete and correct (i.e. when compared to a number of test cases which will vary the input)?

3. Choice, justification and performance reporting of the approach towards implementation as evidenced in the report.

- A breakdown of how marks are awarded is provided in the report structure template in Appendix A.

These 3 criteria have roughly equal weighting (each worth 25-40%).

If you submit work after the deadline you will incur a deduction of 5% of the mark for each working day that the work is late after the deadline. Work submitted more than 5 working days late will be graded as 0. This is the same lateness policy applied university wide to all undergraduate and postgraduate programmes.

Assignment Help & Feedback

The lab classes should be used for feedback from demonstrators and the module leaders. You should aim to work iteratively by seeking feedback throughout the semester. If leave your assignment work until the final week you will limit your opportunity for feedback.

For questions you should either bring these to the lab classes or use the course's Google group (COM4521-group@sheffield.ac.uk) which is monitored by the course's teaching staff. However, as messages to the Google group are public to all studen[illegible] including assignment code, instead they should be questi[illegible]iques and specific error messages rather than requests [illegible]

If you are [illegible] questions, you may prefer to use the course's anonymo[illegible]ous questions must be well formed, as there is no possib[illegible]herwise they risk being ignored.

Please do [illegible]sistants or the module leader directly for assignment [illegible]ests for help will be redirected to the above mechanisms for obtaining help and support.

Appendix A: Report Structure Template

Each stag ... ific choice of technique which you have applied in your i ... OpenMP Scheduling, OpenMP approaches for avoiding ... memory caching, Atomics, Reductions, Warp operation ... Each stage should be no more than 500 words and may ... ages.

<Oper ... Algorithm <Standard Deviation/Convolution/Data Structure>

Description

- Briefly describe how the stage is implemented focusing on **what** choice of technique you have applied to your code.

Marks will be awarded for:

- Clarity of description

Justification

- Describe **why** you selected a particular technique or approach. **Provide justification** to demonstrate your understanding of content from the lectures and labs as to why the approach is appropriate and efficient.

Marks will be awarded for:

- Appropriateness of the approach. I.e. Is this the most efficient choice?
- Justification of the approach and demonstration of understanding

Performance

Size	CPU Reference Timing (ms)	<Mode> Timing (ms)

- Decide appropriate benchmark configurations to best demonstrate scaling of your optimised algorithm.
- Report your benchmark results, for example in the table provided above
- Describe which aspects of your implementation limits performance? E.g. Is your code compute, memory or latency bound on the GPU? Have you performed any profiling? Is a particular operation slow?
- What could be improved in your code if you had more time?

Marks will be awarded for:

- Appropriateness of the used benchmark configurations.
- Doe[illegible]h the experimental result?
- Hav[illegible] code been identified?
- Has[illegible]g factors been described or evidenced?