Introduction

The assessment has been designed against the module's learning objectives. 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 an algorithm 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 the provided algorithm, and to complete a report which documents the justification for the techniques you have used and how profiling and/or benchmarking supports your choices.

The Algorithm and Starting Code

The task is to implement a simple pixelate filter on a photographic image input to produce a photo mosaic. The mosaic should decompose the image into a grid of regular tiles, and average the pixels within each tile. Figure 1 shows the mosaic effect with varying tile sizes (c), for the purposes of the assignment the tile size is fixed to the value found in the starting code.

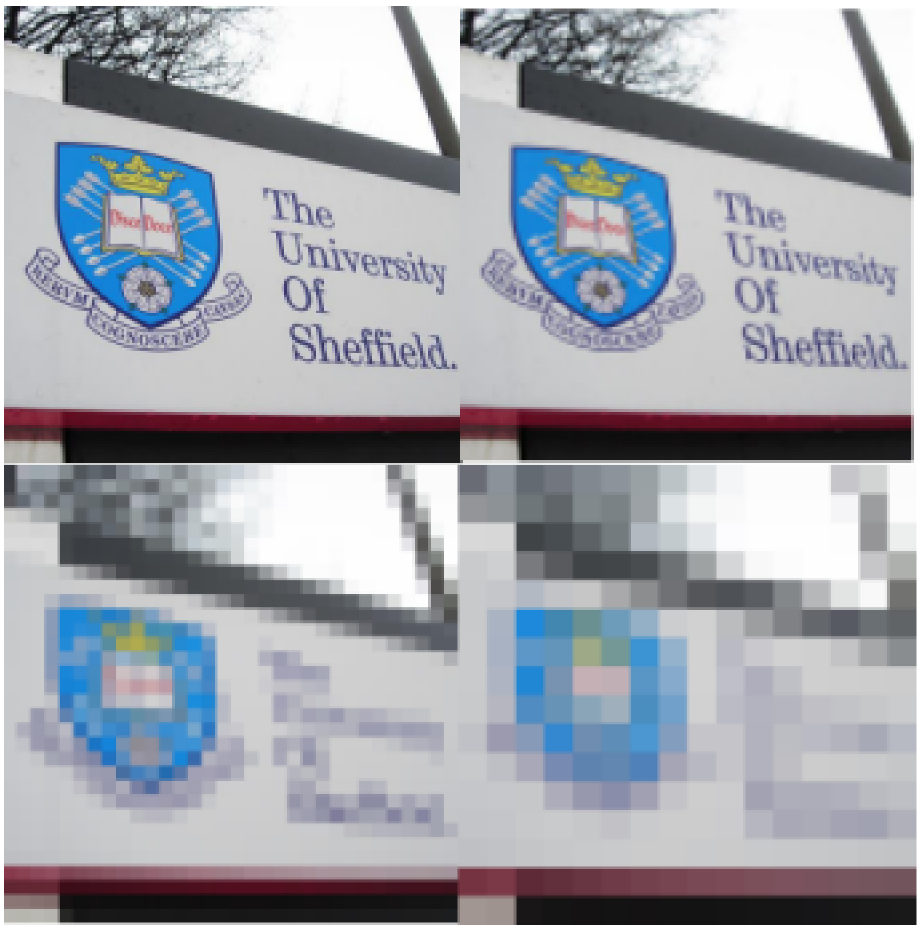

Figure 1: Varying mosaic cell size. Top Left shows the original image, Top Right shows c = 4, Bottom Left shows c = 16, Bottom Right shows c =32.

The Mosaic algorithm has been broken up into 3 stages

1. Tile sum (stage 1) - the pixels of each tile are summed.
2. Compact Mosaic (stage 2) - each tile sum is converted into the average, to produce a compact mosaic image with 1 pixel per tile, the average of the whole image is also returned by this stage.
3. Broadcast (stage 3) - each pixel of the compact mosaic is broadcast to all pixels in the tile at the full image scale, producing the output image.

The provided code only handles 3 channel images (e.g. RGB), other images will not be loaded. Furthermore the tile size is fixed (see `TILE_SIZE` in `config.h`) and images will be cropped to a multiple of the tile size in each dimension. Your algorithms only need to support these provided configurations. The only variable input is that of image size, which may range from a single tile, to hundreds of tiles when your code is tested on submission.

The handout code includes test images (in the `samples` folder) of different input sizes. You can also use your own test images, most RGB jpeg and png images should be supported.

Each of the stages are described in more detail below.

Tile Sum (Stage 1)

Image data is stored in left to right, top to bottom pixel order. Each pixel consisted of 3 `unsigned char`, representing the 3 colour channels red/green/blue. Throughout, these 3 channels must be treated independently.

In this stage, each tile's pixels are loaded and their pixels summed to produce a total for each of the 3 colour channels.

The method `cpu_stage1()` provides the single threaded implementation of this stage of the algorithm. The methods `validate_tile_sum(...)` and `skip_tile_sum(...)` can be used to validate or skip the output of this stage. However, both of these functions require pointers to host memory matching the layout and order of the CPU reference implementation.

Compact Mosaic (Stage 2)

In this stage, the tile sums from the previous stage are divided by the square of `TILE_SIZE` (the number of pixels in a tile) to produce the average pixel value for each tile.

The whole image pixel average must also be returned by this stage, which can be calculated using the individual tile sums or averages.

The method `cpu_stage2()` provides the single threaded implementation of this stage of the algorithm. The methods `validate_compact_mosaic(...)` and `skip_compact_mosaic(...)` can be used to validate or skip the output of this stage. However, both of these functions require pointers to host memory matching the layout and order of the CPU reference implementation.

Broadcast (Stage 3)

The final output image is calculated in stage 3 by broadcasting the pixel averages from the compact mosaic, to a full scale image.

The methods `validate_broadcast(...) and skip_broadcast(...)` can be used to validate or skip the output of this stage. However, both of these functions require pointers to host memory matching the layout and order of the CPU reference implementation.

The Task

Code

For this assignment you must complete the code found in `openmp.c` and `cuda.cu`, so that they perform the same algorithm described above and found inthe 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 5 methods named `openmp_begin()`, `openmp_stage1()`, `openmp_stage2()`, `openmp_stage3()`, `openmp_end()` respectively (and likewise for CUDA). The begin method takes the input image and performs any memory allocations of copies required by the algorithm. The end method, similarly returns the output image and frees allocated memory resources.

You should implement the OpenMP and CUDA algorithms with the intention of achieving the fastest performance for each algorithm on the hardware you use to develop and test your assignment. As the second stage is the most advanced, it is recommended that you address the other stages first. The starting code has helper methods, which allow you to skip and validate the three stages individually.

It is important to free all used memory, as memory leaks could cause the benchmark mode, which repeats the algorithm to run out of memory.

The header `helper.h` contains methods which can be used to skip and validate individual stages of the assignment, to assist you with development and finding problems with the correctness of your code. The `VALIDATION` pre-processor definition is defined only for *Debug* builds. Each of the stage functions (for both OpenMP and CUDA) provides a place for you to call the appropriate validation functions to check that your code is correct. As `VALIDATION` is not defined for *Release* builds this will not affect your benchmarking performance. Note, if you choose to use an alternative memory layout it is not necessary to use the validation methods, however if your code does not pass the final tests, you won't be able to achieve per stage marks from testing.

The Report

You are expected to provide a report alongside your code submission. For each of the 6 stages you should complete the template provided in *Appendix A*. The report is your chance to demonstrate to the marker that you understand what has been taught in the module.

Benchmarks should <u>always be carried out in release mode,</u> 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. It is important to benchmark over a range of inputs, to allow consideration of how the performance scales with the size of inputs.

It is recommended that you provide evidence from the profiler, to support your justification of the performance and limitations of your CUDA implementations.

Deliverables

You must submit your `openmp.c`, `cuda.cu` and your report document (e.g. `.pdf/.docx)` within a single zip file via Mole, before the deadline. Your code should build

in the Release mode configuration provided to you (or via the provided Makefile) without errors or warnings (other than those caused by IntelliSense). You do not need to hand in the project files or any
other 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. You should not create or include any additional header or source files other than those provided.

Your code should not rely on any third party tools/libraries not introduced within the lectures/lab classes. The use of Thrust and CUB is permitted.

Even if you do not complete all aspects of the assignment, partial progress should be submitted as this can still receive marks. I.e. You can use the fallback CPU implementation to move onto the next stage.

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 justify the approach used and demonstrate its impact on the performance.

The marks for each stage of the assignment will be distributed as follows:

	OpenMP (30%)	**CUDA (70%)**
Stage 1 (30%)	10%	30%
Stage 2 (40%)	10%	20%
Stage 3 (30%)	10%	20%

The CUDA stage is more heavily weighted as it is more difficult. The first stage in CUDA also has a greater weight than stages 2 and 3 as there is more opportunity for experimentation in implementing an optimal solution.

For each of the 6 stages in total, the distribution of the marks will be determined by the following criteria;

1) Quality of implementation [50%]
    a) Have all parts of the stage been implemented?
    b) Is the implementation free from race conditions or other errors regardless of the output?
    c) Is code structured clearly and logically?
    d) 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 [25%] a) 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. [25%]. A breakdown of how marks are awarded is provided in the report structure template in Appendix A.

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 and Feedback

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

For questions you should either bring these to the lab or use the course google discussion group (COM4521-group@sheffield.ac.uk) which is monitored by the course demonstrators. However, as messages to the google group are public to all students, emails should avoid including assignment code and should be questions about ideas and techniques rather than requests to fix code.

Please do not email demonstrators or the module leader directly. Any direct requests for help will be redirected to the above mechanisms for obtaining help and support.

Appendix A: Report Structure Template

Each stage should focus on a specific choice of technique which you have applied in your implementation. E.g. OpenMP Scheduling, OpenMP approaches for avoiding race conditions, CUDA memory caching, Atomics, Reductions, Warp operations, Shared Memory, etc.

Each stage should be no more than 500 words and may be far fewer for some

stages.

<OpenMP/CUDA>: Stage <1/2/3>

Description

- Briefly describe how the stage is implemented focusing **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** using your understanding from experience in 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

Problem size	**CPU Reference Timing (ms)**	**<Mode> Stage <N> Timing (ms)**
256 x 256 px		
1024x 1024 px		
2048 x 2048 px		
4096 x 4096 px		

- Report your benchmark results 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;

- *Does the justification match the experimental result?*
- *Have limiting factors of the code been identified?*
- *Has justification for limiting factors been described or evidenced?*