COM4521/COM6521 Parallel Computing with Graphical Processing Units (GPUs)

Assignment (80% of module mark)

Deadline: 5pm Friday 19th May (Week 12)

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: 9 February 2023

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 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 to document and justify the techniques you have used, and demonstrate how profiling and/or benchmarking supports your approach.

The Algorithm & Starting Code

Modern video games regularly use particle effects to visually represent phenomena such as smoke and fire. Hundreds or thousands of translucent particles, represented by either coloured shape or sprite billboards, are rendered each frame with their overlap blended to create the desired effect. Processing many particles and graphical rendering are both inherently suited to parallel computation.

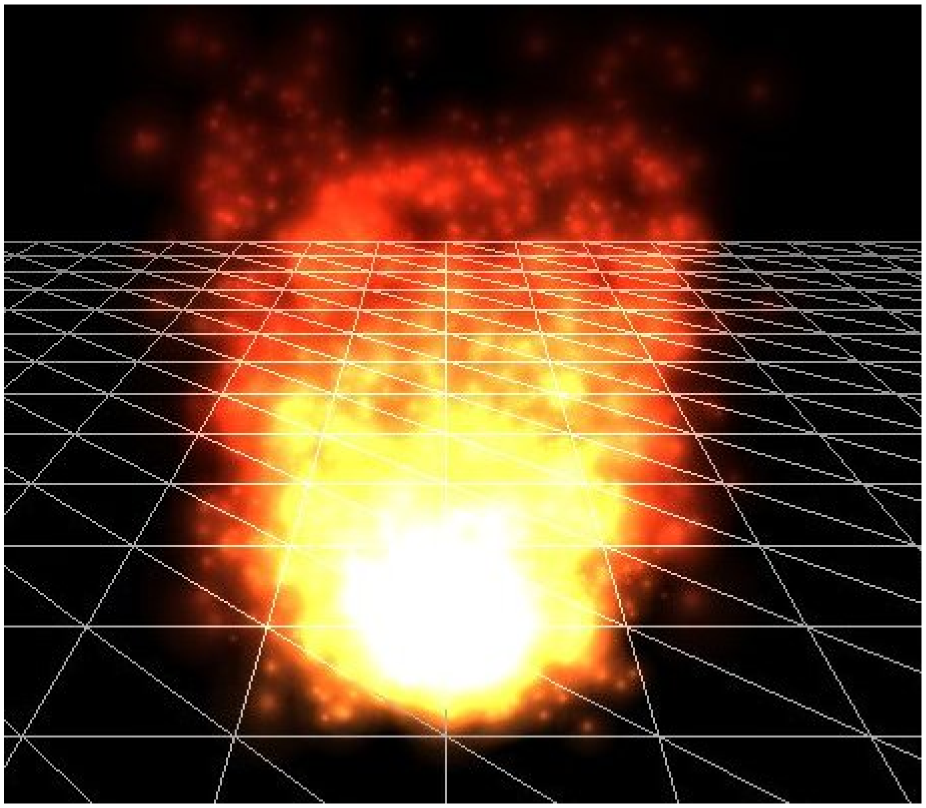

Figure 1: A fire simulated with particle effects. *LearnOpenGL.com (CC BY 4.0)*

The algorithm you are provided, and must optimise for this assignment, implements a software renderer for a single frame of translucent circular particles. The algorithm is provided an input array of `Particle` structs, each containing a `color`, `location` and `radius` of a circle and image dimensions for the output image. The algorithm must then position and render these translucent circles in back-to-front order to produce an image to be output. It is important that blending of colours to each pixel occurs in the correct order, otherwise an incorrect image will be output.

You do **not** need to understand the graphics pipeline or other particle effects information to complete this assignment. **All required information can be found in this assignment brief and the reference implementation.**

The reference implementation and starting code are available to download from: https://github.com/RSE-Sheffield/COMCUDA_particle_assignment/archive/master.zip

The software renderer is broken up into 3 stages, and has been structured in a manner suitable for GPU parallelisation:

1. Particle Contribution (Stage 1) - The number of particles contributing to each pixel of the output image is calculated.
2. Pixel Data (Stage 2) - The colours which contribute to each pixel are sorted by particle depth and stored.
3. Blending (Stage 3) - The sorted colours for each pixel are combined using the blend formula to calculate the pixel's colour.

There are two inputs parameters to the problem:

- The number of particles
- The output image dimensions (hence the number of pixels)

Your code only needs to support changes to these two input parameters. Other configuration parameters can be found in `config.h`, however they do not need to be changed for the assignment.

On submission your code will be tested on a range of these input parameters to assess it's performance scaling and correctness. Naturally, you may find extreme input parameters lead to out of memory errors or potentially very slow computation.

Each of the stages are described in more detail below.

Stage 1 - Particle Contribution

During the particle contribution, the pixels covered by each particle (circle) are calculated and a counter for each covered pixel is incremented. This essentially calculates a histogram, where each pixel has a corresponding bin.

Figure 2 demonstrates how a pixel is considered covered by a particle if the centre of the pixel falls within the bounds of the circle.

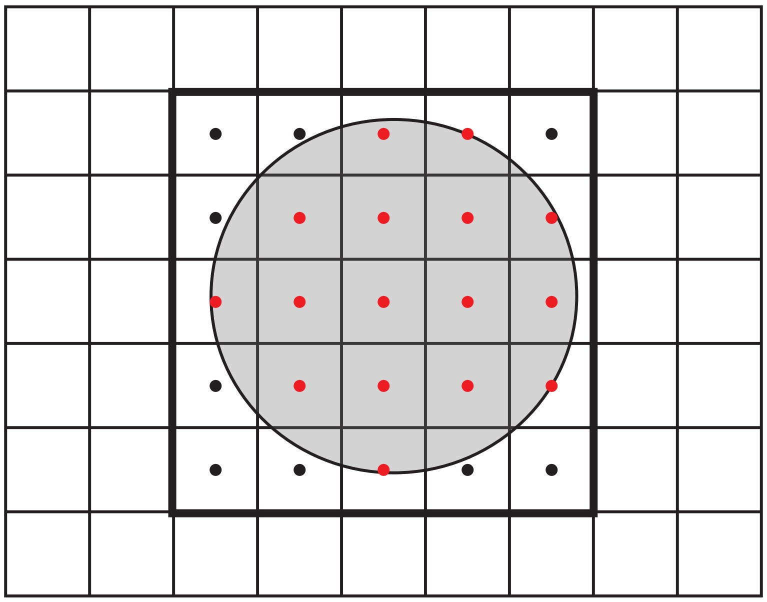

Figure 2: Computing the contribution of a particle to the output image. Pixels with a red dot are considered covered by the particle (circle).

When this has been performed for all of the particles, you should be left with a histogram of contributions, as shown in Figure 3.

Stage 2 - Pixel Data

This stage has several components.

1. Scan the Histogram

Firstly, the histogram from stage 1 has an exclusive prefix sum (scan) operation performed across it. The result of this operation is a lookup table, providing a unique index to each pixel's contributing colours.

2. Allocate Colour & Depth Storage

Next, the storage for all the colours contributing to every pixel must be allocated. The final value from the prefix sum holds the total number of colours.

3. Write Colours to the Storage

The histogram from stage 1 is now repeated, however as each particle contributes to the histogram they also copy their colour to a unique location in the pixel contribution storage.

4. Pair Sort

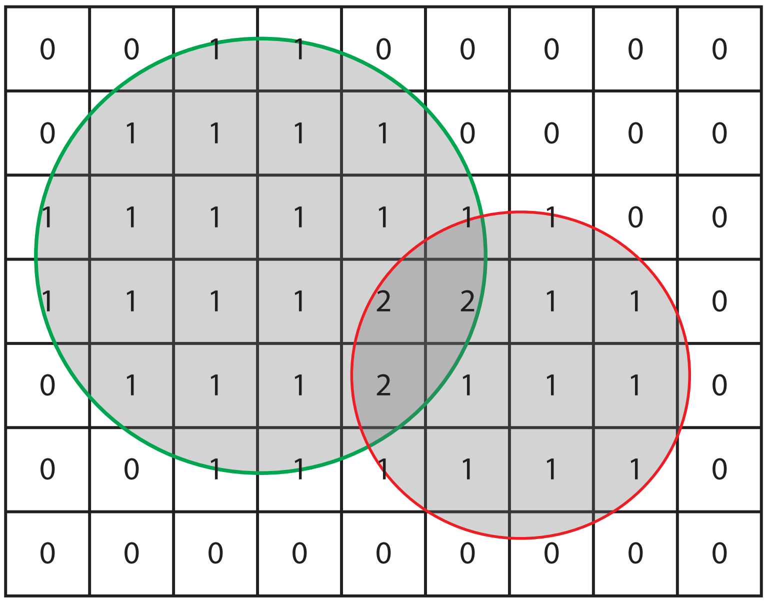

Figure 3: Two overlapping particles and the resulting contribution histogram

The colours for each pixel are pair sorted according to their depths in ascending order. A pair sort is like a regular sort, however key-value pairs are sorted according to their keys.

Stage 3 - Blending

Each pixel iterates it's colours from the colour storage (in back to front order) and blends them one-by-one to the corresponding output image pixel using the formula in Equation 1. This calculates the final blended colour of each pixel of the image.

$$\vec{C_r} = \vec{C_s} A_s + \vec{C}d(1 - A_s) \tag{1}$$

The resulting colour $\vec{C_r}$ from Equation 1 equals the source colour $\vec{C_s}$ multiplied by source alpha $\vec{A_s}$ added to the destination colour $\vec{C_d}$ multiplied by 1 minus the source alpha $\vec{A_s}$.

The first iteration, the destination colour $\vec{C_d}$ begins white (255, 255, 255), each subsequent iteration the previous iteration's resulting colour $\vec{C_r}$ becomes the destination colour $\vec{C_d}$. Whereas the source color $\vec{C_s}$ and source alpha $\vec{A_s}$ each iteration are provided by one of the pixel's colours from the colour storage.

Figure 4 demonstrates how the blending order impacts the resulting colours. Each colour blended into a pixel has a greater influence than those before it, hence the last blended colour has the greatest influence on the pixel's final colour.

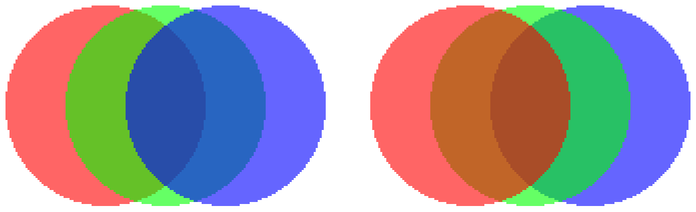

Figure 4: The impact of blending order. Red, Green, Blue (left) vs the same circles blended in reverse order Blue, Green, Red (right). The last blended colour is most visible where all three overlap Blue (left), Red (right).

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 5 methods named `openmp_begin()`, `openmp_stage1()`, `openmp_stage2()`, `openmp_stage3()` and `openmp_end()` respectively (and likewise for CUDA). The begin method takes the inputs and performs any memory allocations 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 methods (for both OpenMP and CUDA) provides a place for you to call the appropriate validation methods to check that your code is correct. As `VALIDATION` is not defined for Release builds this will not affect your benchmarking performance.

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 **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 Mole, 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.

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:

	OpenMP (30%)	**CUDA (70%)**
Stage 1 (30%)	10%	20%
Stage 2 (45%)	10%	35%
Stage 3 (25%)	10%	15%

The CUDA stage is more heavily weighted as it is more difficult. The second stage in CUDA also has a greater weight than stages 1 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%]

- 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 [25%]

- 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 & 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 demonstrators. However, as messages to the Google group are public to all students, emails should avoid including assignment code, instead they should be questions about ideas, techniques and specific error messages rather than requests to fix code.

Please do not email demonstrators or the module leader directly for assignment help. 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 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

Particle Count	Output Image Size	CPU Reference Timing (ms)	<Mode> Stage <N> 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.
- Does the justification match the experimental result?
- Have limiting factors of the code been identified?
- Has justification for limiting factors been described or evidenced?