Interactive Mandelbrot

By Jamie Mulvihill

1703169

Purpose of Application

- Create an Interactive Mandelbrot application that would allow the user to move around and that would calculate a new Mandelbrot set based on input controls.
- Make use of Parallel programming techniques and structures.
- Evaluate the performance of both the CPU and GPU used in computing the Mandelbrot set

Implementation

- I used OpenGL to render and visualize the Mandelbrot set, this allowed me to give control of what was being calculated to the user by using keyboard input controls
- Utilised the Microsoft C++ AMP library to implement GPU Threading
- Implemented Parallel design Patterns based

Parallel Design Patterns

- As I tested the performance on the CPU and GPU I used a number of different design patterns, these were:
  - Fork-Join Pattern
  - Task-based Farm Pattern
  - Map Pattern

Fork-Join Pattern

- Splits a task down to a number of sub tasks
- Creates a CPU thread to run each individual task
- When all tasks are complete joins all threads back together
- As each thread is given a set amount of work, once that work is done threads/cores become idle.
- Therefore application is only as quick as slowest thread

```
int slices = 16;
int sliceSize = HEIGHT / slices;

std::vector<std::thread*> threads;

for (int i = 0; i < slices; i++)
{

    // create a task for each slice
    threads.push_back(new std::thread([=] { ComputeMandelbrot(-2.0, 1, -1.25, 1.25, i * sliceSize, (i * sliceSize) + sliceSize);
    }));

}

for (int j = 0; j < 16; j++)
{
    threads[j]->join();
}
```

Task-Based Farm

- Similar to Fork-Join
- Splits work down to a queue of tasks
- Creates a number of threads less than number of tasks
- Each thread takes a task from the queue, executes the tasks and is now available to take another task
- This ensures no idle threads/is load balanced

```
int slices = 16;
int sliceSize = HEIGHT / slices;

// add tasks to farm queue
for (int j = 0; j < slices; ++j) {
    farm.add_CPUTask(new CPUMandelBrotTask(j, -2.0, 1.0, 1.125, -1.125, j * sliceSize, (j * sliceSize) + sliceSize, (TwoDemensional*)image));
}
```

```
for (int i = 0; i < coreCount; i++) {
    farmThreads.push_back(new std::thread([&] { // create a thread
        while (true) {
            mutex_.lock();
            if (!CPUQueue.empty()) { // if theres a task to be done
                CPUTask *t = CPUQueue.front();
                CPUQueue.pop(); // take it from the queue
                mutex_.unlock();
                t->run(); // run task
                delete t;
            }
            else {
                mutex_.unlock();
                return;
            }
        }
    }));
}
for (auto t : farmThreads) {
    t->join();
}
```

Map Pattern

- Used for Embarrassingly Parallel Problems
- Performs the same kernel on every element of an index set
- Each element calculates its own unique calculations
- Elements are unable to communicate between each other
- As work

```
parallel_for_each(view, data->writeExt, [=](concurrency::index<2>idx) restrict(amp)  // runs a kernal for each element within the extent
{
    int x = idx[1]; // x value is set to second value of the index position
    int y = idx[0]; // y value is set tothe firstvalue of the index position

    double  aLeft = left_;
    double aRight = right_;
    double aTop = top_;
    double aBottom = bottom_;

    Complex1 c =
    {
        aLeft + (x * (aRight - aLeft) / 640),
        aTop + (y * (aBottom - aTop) / rows)

    };

    Complex1 z =
    {
        0,
        0
    };
    int iterations = 0;

    while (c_abs(z) < 4.0 && iterations < 500)
    {
        z = c_add(c_mul(z, z), c);

        ++iterations;
    }

    if (iterations == 500)
    {
        imageT[y][x] = 0xff0000ff; // stores calculation back into he array view at index location

    }
    else
    {
        imageT[y][x] = (((iterations / 4) % 255) << 16) | ((iterations % 255) << 8) | (iterations / 4) % 255;   // stores calculation back into he array view at index location
    }
});
imageT.synchronize(); // synchronize values and copy back to cpu
```

Why Used for Mandelbrot?

- Mandelbrot is an embarrassingly parallel problem
- Requires no communication between threads therefore an ideal candidate for the Map Pattern
- Task Farm would allow for the Mandelbrot set to be split into different tasks, sharing the workload over a group of threads to improve performance
- Task Farm would also allow the work to be split over a number of GPUs if available, theoretically further improving performance

Threading and Thread Interactions

- As the Mandelbrot set is an embarrassingly Parallel problem and my design utilised a Map Pattern there was no interaction between threads during the computation of the Mandelbrot set itself
- My Task Farm however used an std::queue where each available thread would attempt to take the font element of the queue, to ensure only one thread had access to this element I created a mutex and used its lock() and unlock() functions.
- This ensured that once a thread was created, the mutex would lock, meaning now only that one thread can proceed past that point and any other threads would be blocked until the threads work was done at which point the mutex would unlock.

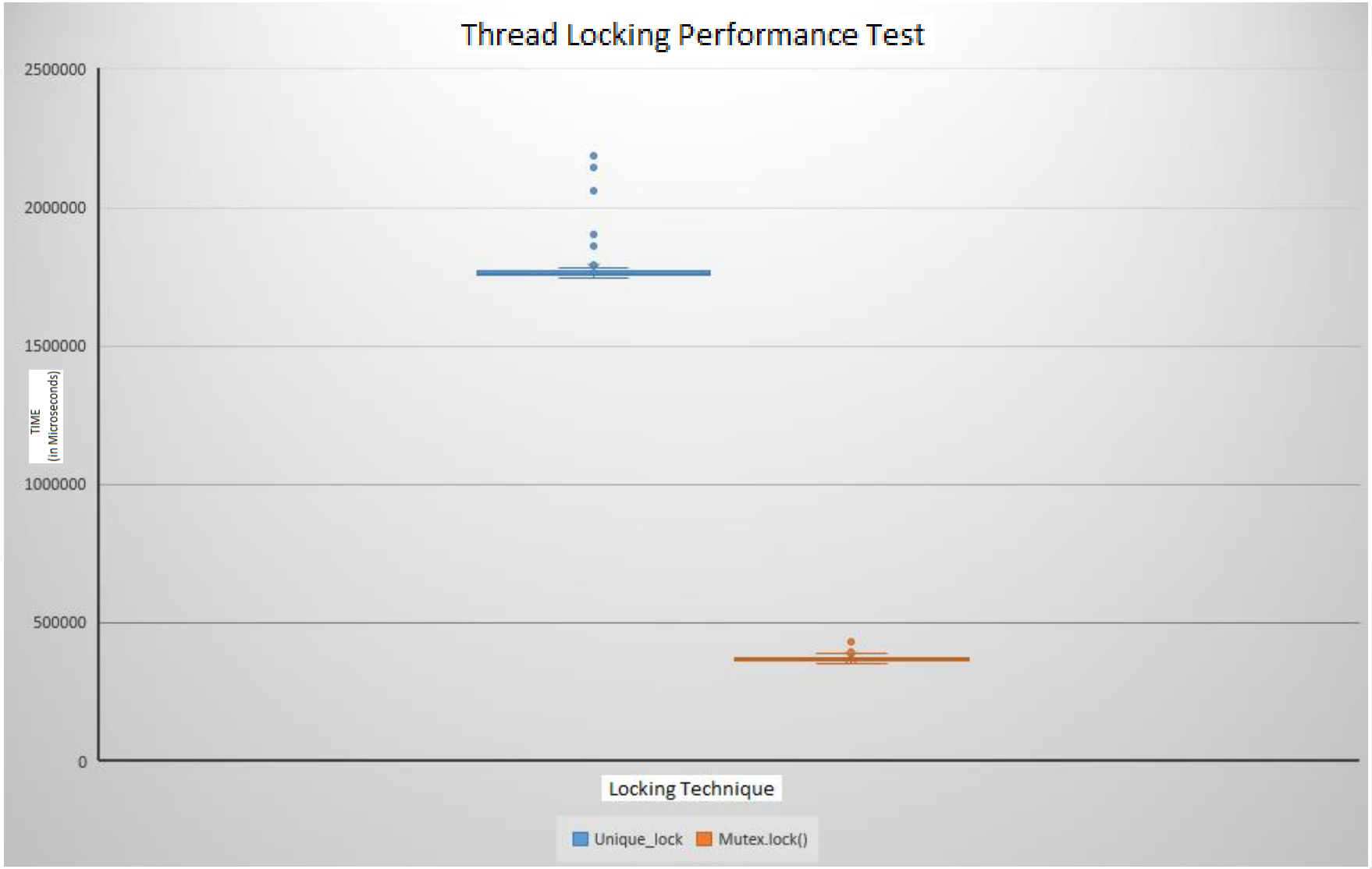
Mutex.lock() vs Unique_lock

```
while (true) {
    mutex_.lock();
    if (!CPUQueue.empty()) {
        CPUTask *t = CPUQueue.front();
        CPUQueue.pop();
        mutex_.unlock();
        t->run();
        delete t;
    }
    else {
        mutex_.unlock();
        return;
    }
}
```

- Using Mutex.lock() allows the mutex to be unlocked before the task has been run and deleted
- This allows another thread to grab a task once the previous thread has started the previous tasks run function

```
while (true) {
    std::unique_lock<std::mutex> lck(mutex_);
    if (!CPUQueue.empty()) {
        CPUTask *t = CPUQueue.front();
        CPUQueue.pop();
        t->run();
        delete t;
    }
    else {
        return;
    }
}
```

- Unique_lock unlocks when it goes out of scope i.e. leaves {}
- Therefore this unique_lock will not unlock until after the task has been run and deleted.
- This will result in a slower performance


Thread Locking Performance Test
2500000
2000000
1500000
1000000
500000
0
TIME
(in Microseconds)
Locking Technique
Unique_lock
Mutex.lock()

Threading and Thread Interactions

- Another technique I used when trying to manage my threads safely was to use a concurrent_queue instead of the std::queue
- The concurrent_queue has a try_pop function that removes an element from the queue if one is available
- By using this I no longer needed to use a mutex as a task could be returned using the try_pop function, a thread could then be created and the tasks run function executed immediately.

Performance evaluation

Hardware Specifications

CPU

- Intel(R) Core™ i7-6700 CPU @ 3.4GHZ
- Quad Core

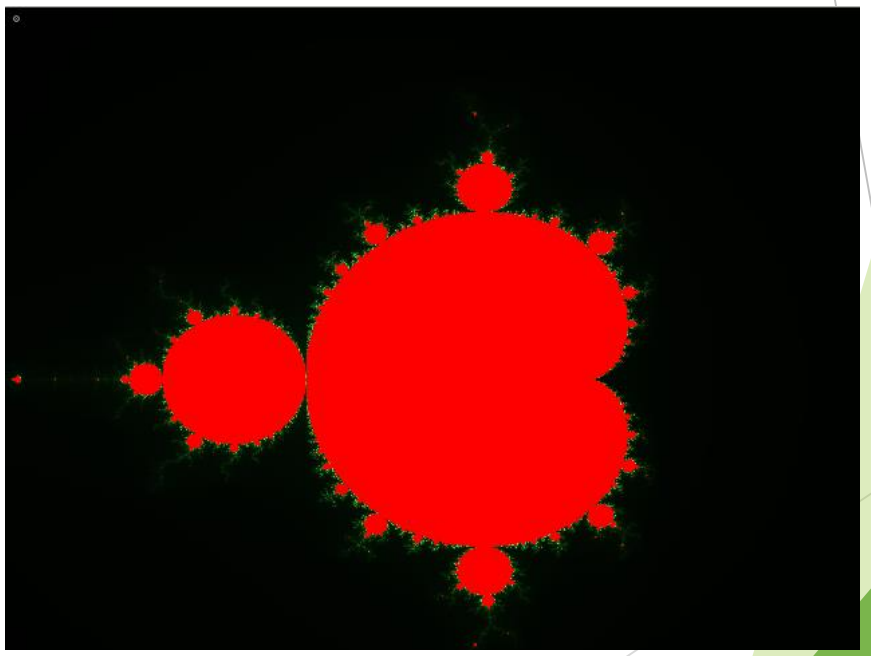
GPU

- NVIDIA GeForce GTX 1080
- Intel(R) HD Graphics 530

Performance evaluation

To find the optimal performance of the hardware I used in my application to compute the Mandelbrot set, I implemented a number of tests on both the CPU and GPU, these were:

- The Fork-Join Pattern vs The CPU Task Based Farm
- CPU Task Farm vs GPU Map Pattern
- Single GPU Varying the number of threads
- Load Balanced Multi GPU varying number of threads
- Comparison of Single & Multi GPU tests



The Fork-Join Pattern vs The CPU Task Based Farm

Hypothesis:

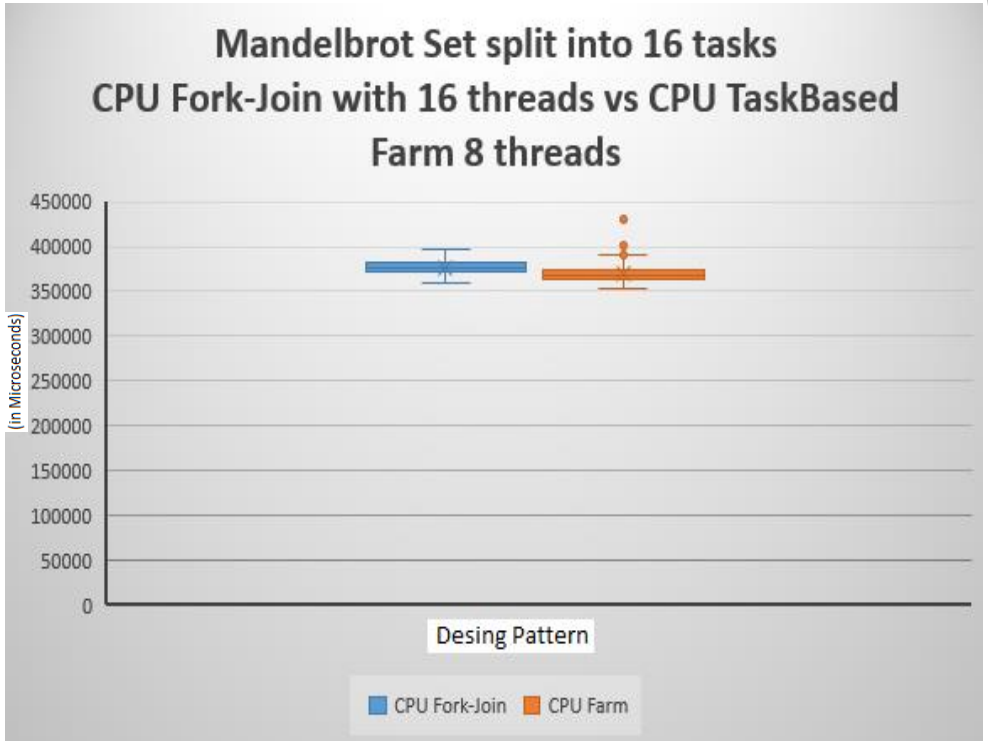
As there should be no idle threads/cores using the Farm Design Pattern, Farm Pattern will perform faster.

Test:

Divide the Mandelbrot Set into 16 tasks, compute using each Pattern 100 times

Result:

Whilst closer performance than originally expected, The farm performed quicker of the two. Possibly closer than expected as Farm must lock a mutex, pop of a queue, create a pointer and unlock a mutex before each task is run



CPU Task Farm vs GPU Map Pattern

Hypothesis:

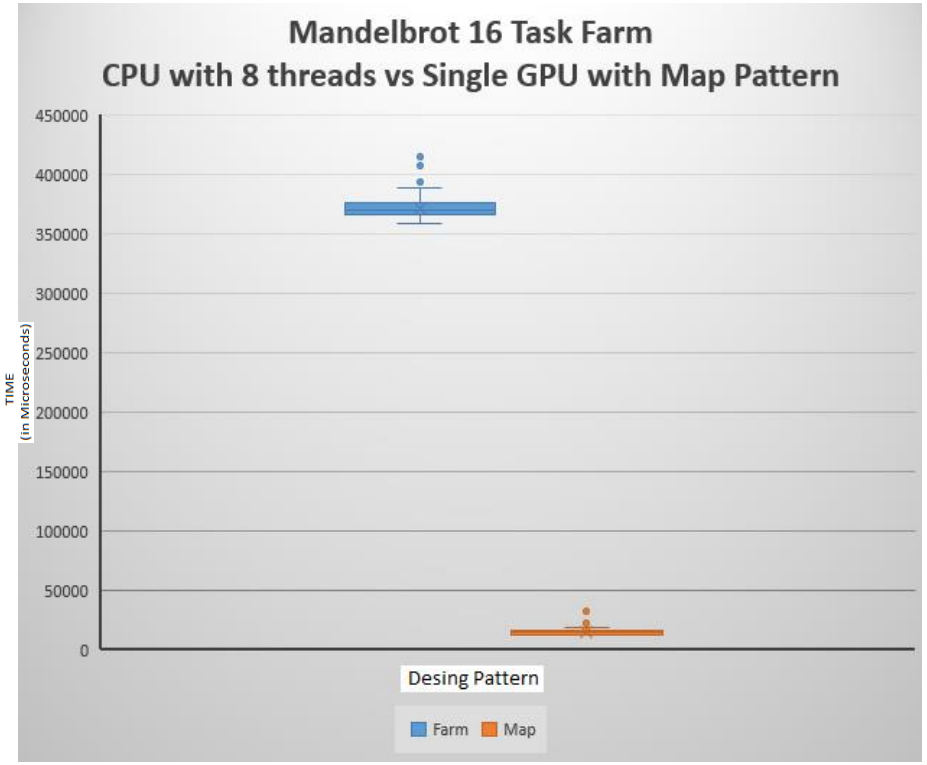
As the Map Pattern utilises thread independent calculations and the Farm has an iterative design, the Map Pattern will perform Quicker

Test:

Divide the Mandelbrot Set into 16 tasks, compute using each Pattern 100 times

Result:

As expected, The GPU Map pattern is the quicker of the two. The GPU used for testing is a high end GPU optimized for mathematic calculations, if a lesser GPU was used the results would be closer.



Single GPU Varying the number of Threads

Hypothesis:

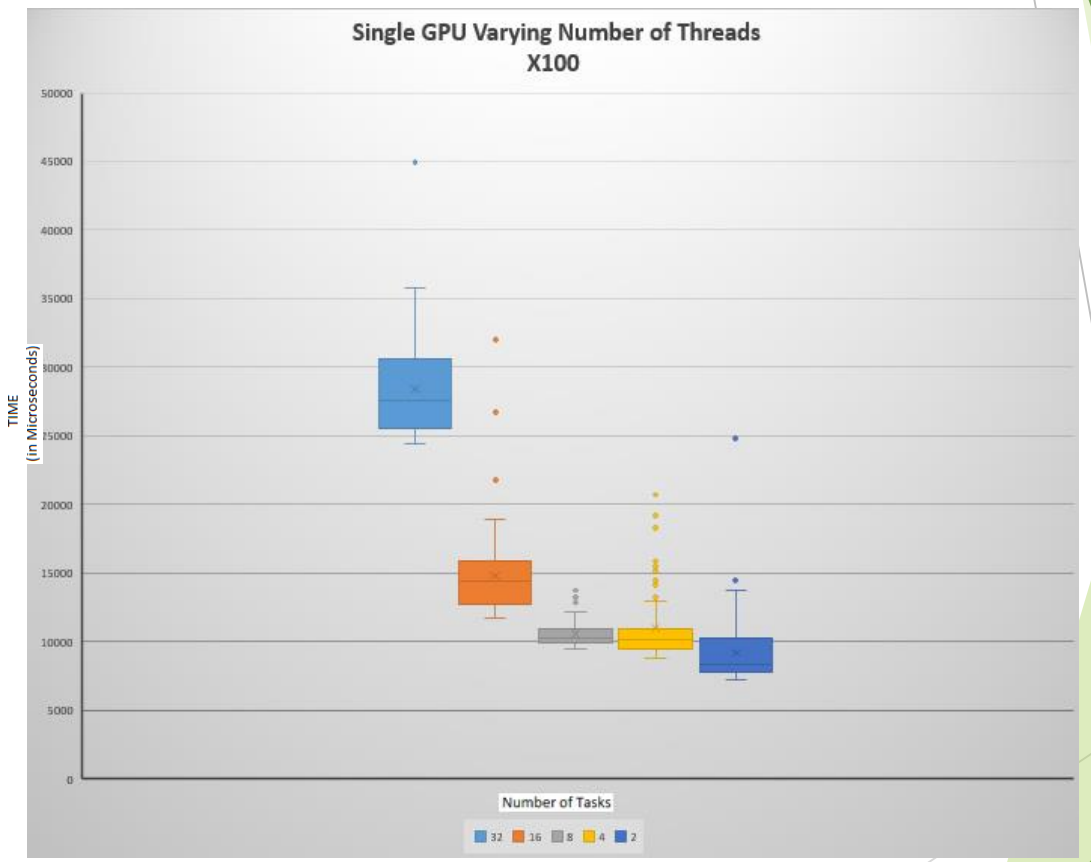
As the number of tasks increases, the number of rows per task and number of threads decreases, with smaller sized tasks the GPU will perform faster

Test:

Compute the same Mandelbrot set 100 times with the set split into 2, 4, 8, 16 and 32 tasks.

Result:

Split into 2 tasks, the GPU performed fastest and with 32 tasks performed slowest. This suggests that the time taken to transfer the data from the CPU to the GPU as a part of each task was the most expensive part of the task and increasing the number of tasks adds more time



Load Balanced Multi GPU varying number of Threads

Hypothesis:

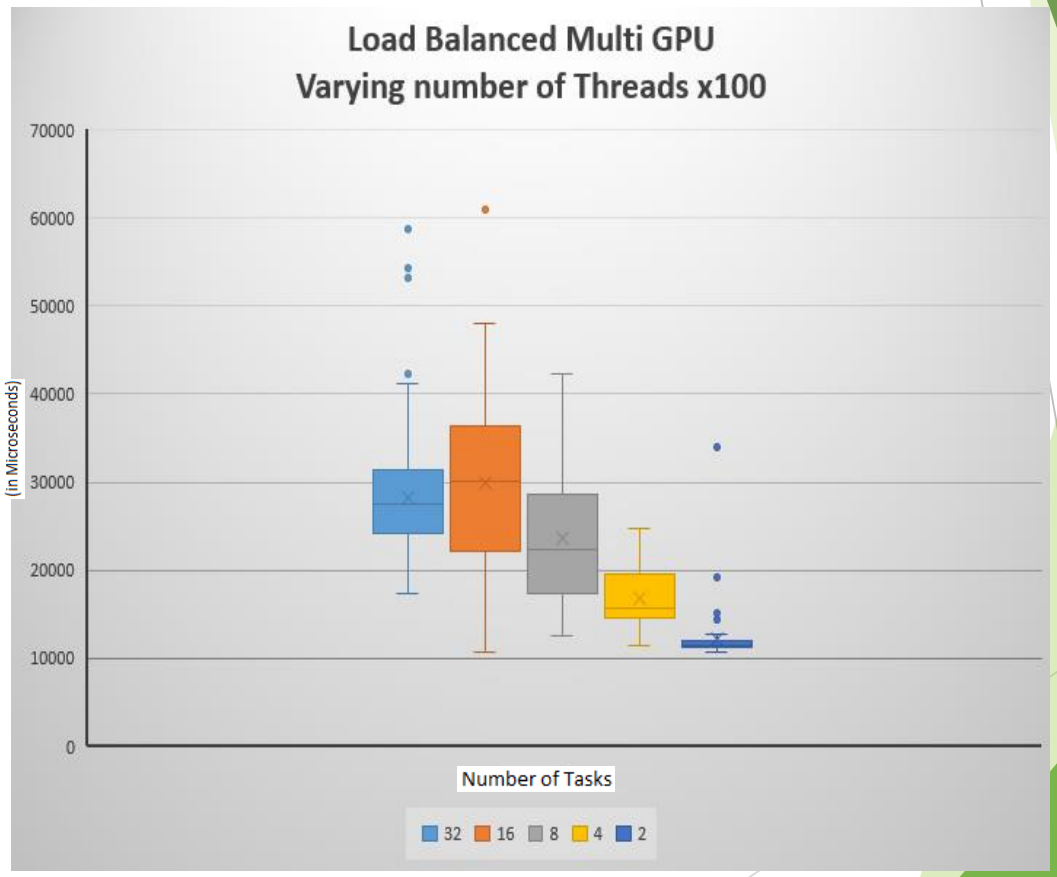
As the number of tasks increased, the faster GPU would take on more of the work and the application would perform faster

Test:

Compute the same Mandelbrot set 100 times with the set split into 2, 4, 8, 16 and 32 tasks.

Result:

With 2 tasks, the application performs fastest, this was not the expected result. As a mutex was added to ensure no two threads tried to take the same task or accelerator, a bottle neck was created. As moving the data to the GPU takes time the increase in tasks increases overall time spent transfer data. There is evidence of the hypothesis being correct however as 16 tasks did return the quickest time



Comparison of Single & Multi GPU tests

Hypothesis:

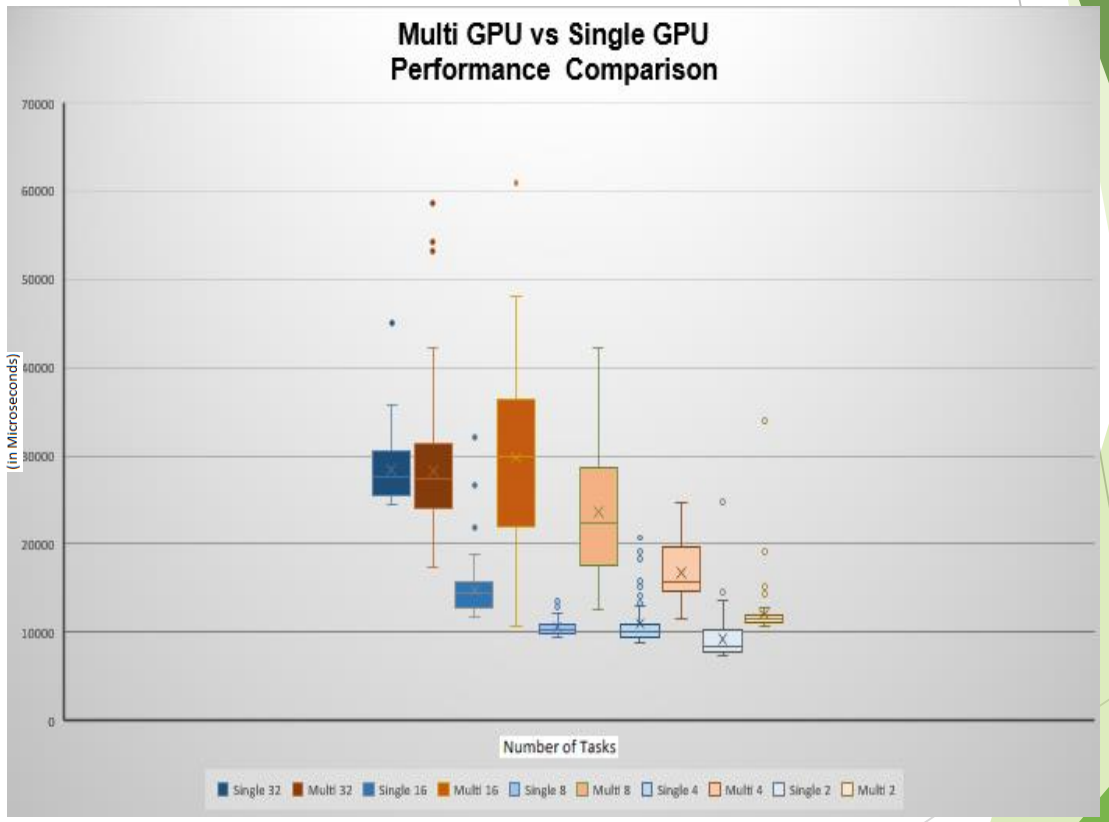
With load balancing the application is splitting the work over multiple GPUs and should perform faster than a single GPU

Test:

Compute the same Mandelbrot set 100 times with the set split into 2, 4, 8, 16 and 32 tasks.

Result:

Single GPU was faster. As Single GPU has fewer bottle necks and doesn't have additional work assigning accelerators it performs the faster of the two, slower times may also be due to lesser GPU picking up tougher tasks.



Critical Evaluation And Conclusions

After carrying out my performance evaluations I came to a number of conclusions about the design of my application and a few ways I could try and solve any issues that arose.

- Performance Results showed the Map Pattern implemented in the application was effective as it allowed for loop acceleration by factoring out sequential loops and created a more data parallel application.

- Whilst the load balancing Multi GPU didn't always return with faster results, it did show evidence to suggest it was behaving as expected, the slower times may have been caused by the lesser GPU taking on the more time complex tasks than the better GPU, this would slow down the overall performance dramatically, the faster times suggests the better GPU took the heavier tasks giving overall much quicker times.

Critical Evaluation And Conclusions

- Did not make use of Tiles. This was a deliberate design choice as the application had no need for Tile shared memory, however using Tiles may have given a slight performance boost as its design exploits underlying hardware and title static variables are not in the global space therefore can be accessed faster

- Transfer of Data, from my results it showed that when the application had smaller tasks, the time performance of the application was slower as the number of tasks was increased, this was due to having to transfer data from the host to the GPU and back more often than when the application had fewer tasks along with the bottleneck created by the mutex.

The End