PARALLELIZED IMAGE FILTERING

CMP202 Project

By [Redacted]

WHAT DOES PROGRAM DO?

- Image editor for postprocessing effects
- 3 things it can do
- Blur, brightness, contrast at varying intensities
- Show it to you, output to file.
- Both CPU and GPU variations

Like photoshop but with fewer crashes…

HOW DOES IT WORK?

- Provide an Image path
- Output provided image to window
- Choose a function
- Choose how intense you want to effect to be
- Done. Image will display in output window. Can apply multiple filters.
- Output whenever you like

EFFECT EXPLANATIONS

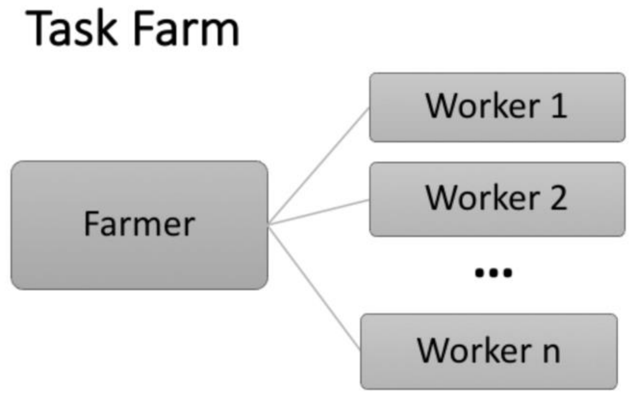
- Blur using convolution and a gaussian kernel. The size we're using for testing is 9x9 (though the user can change this) and uses no standard deviation.
- Brightness applies a scalar on all three channels (RGB) on every pixel directly, increasing the brightness by the specified value.
- The contrast filter applies a contrast factor to each pixel in the image and as it's divided into slices it's considered to be using local contrast. It clamps the colour value to ensure it's within the usable 255,255,255 RGB colour space.

WHY PARALLELIZE?

- Images have large amounts of data and computationally expensive(thousands of pixels, typically)
- Repeated task with limited sharing of resources ensuring a lower synchronization overhead (especially necessary on the GPU)
- More scalable to image sizes we might see more of in future (8K, etc) with better performance
- Likely to be faster, practically speaking.

HOW IS IT PARALLELIZED? (CPU)

- CPU operation takes advantage of task based parallelism through the usage of a task farm, this ensures that the threads that are used are maximized fully.
- Start as many worker threads as there are CPU's (or as many as desired) that takes from a pool of available tasks
- Means that if a section is more algorithmically complex (IE, a block black section vs a section with varying colours) it can handle longer vs shorter tasks.



HOW IS IT PARALLELIZED (GPU)?

- Allocate resources on GPU
- Transfers it to the GPU through a write buffer (Lambda equivalent)
- CL_True is blocking (essentially a GPU wait, waits for buffer write proceed before going onto next instruction)
- Divide image into pixels, execute kernel on them concurrently.
- Using “work items” – essentially the smallest unit of work in OpenCL and represents running a kernel on a specific piece of data, in this case, a single pixel.
- Transfer back once done

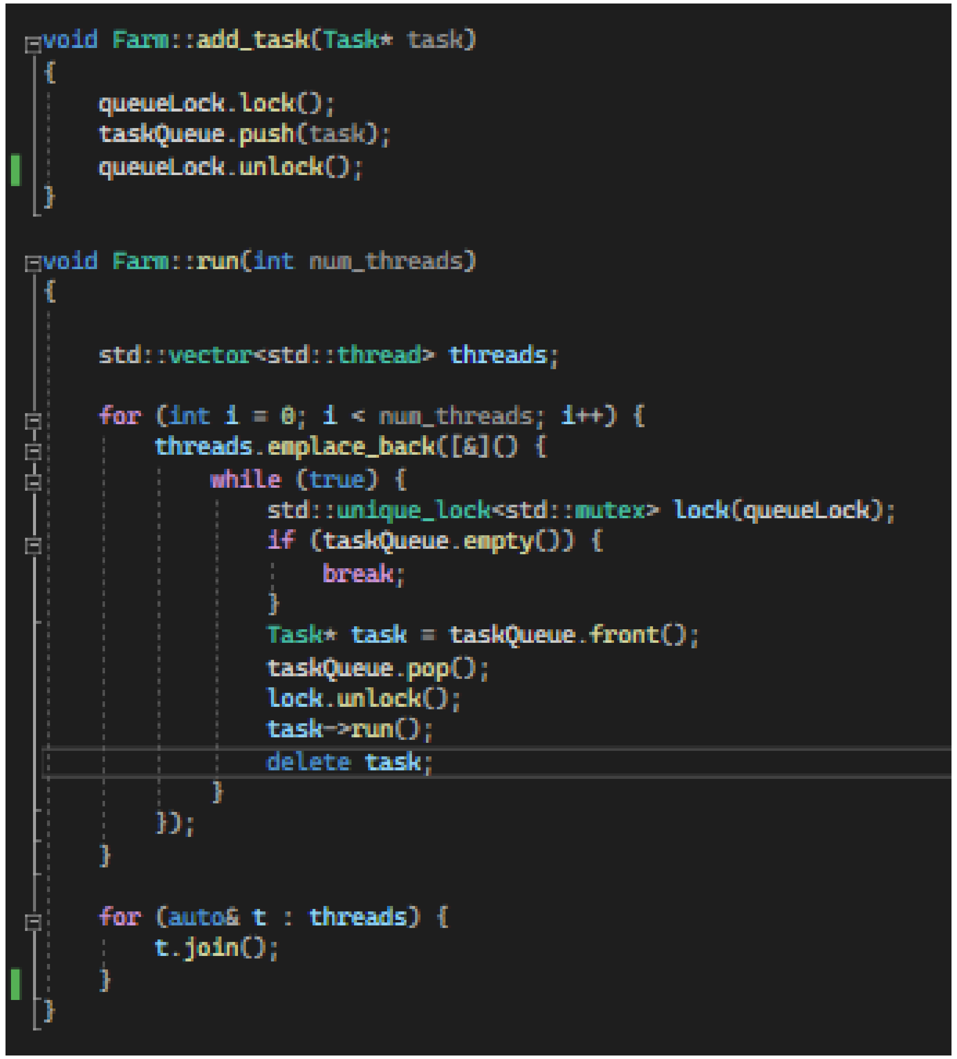
LIBRARIES USED

- OpenCV (Open Computer Vision) was used to implement window GUI functions, image filtering and threads. This was the most efficient way of doing this as it's intended for image processing. Manual implementations of for instance, a convolution filter, would not be materially different and it allowed for other image filters to be easily implemented.
- OpenCL (Open Computing Language) was used for GPU parallelization. Tried Boost but encountered issues building it, CUDA is a nightmare to setup and doesn't come pre-integrated with openCV, TBB has complex syntax.
- AMP is depreciated. As nice as it is, couldn't justify using it.

OpenCL

OpenCV

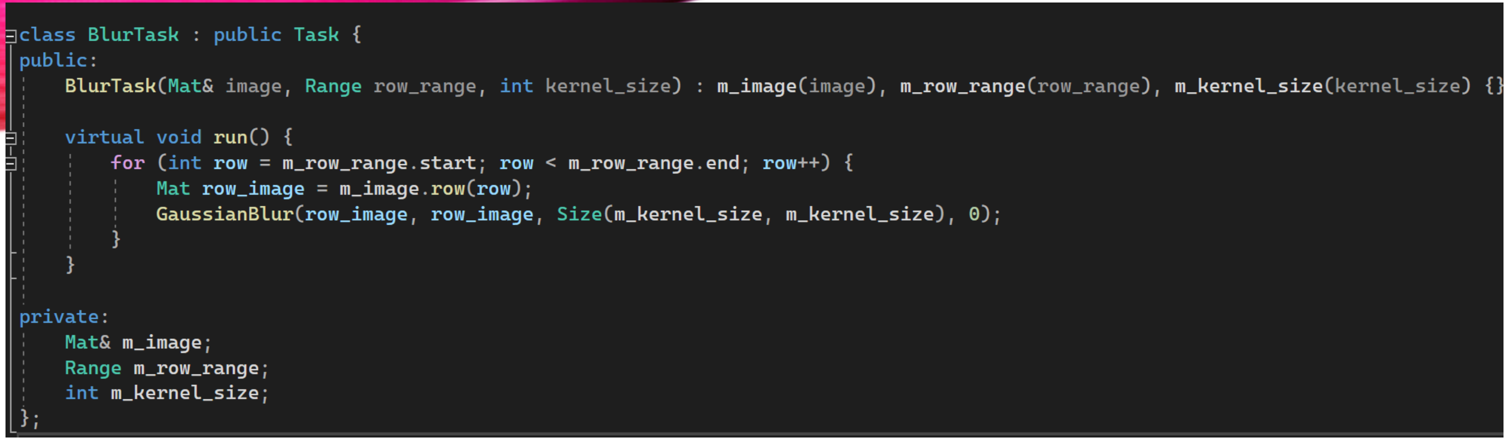
CODE EVALUATION



```
void Farm::add_task(Task* task)
{
    queueLock.lock();
    taskQueue.push(task);
    queueLock.unlock();
}

void Farm::run(int num_threads)
{

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

    for (int i = 0; i < num_threads; i++) {
        threads.emplace_back([&]() {
            while (true) {
                std::unique_lock<std::mutex> lock(queueLock);
                if (taskQueue.empty()) {
                    break;
                }
                Task* task = taskQueue.front();
                taskQueue.pop();
                lock.unlock();
                task->run();
                delete task;
            }
        });
    }

    for (auto& t : threads) {
        t.join();
    }
}
```

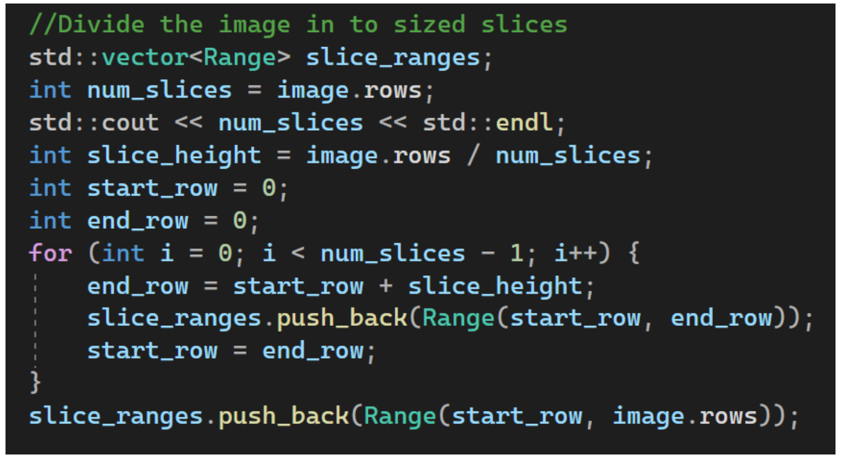
- The taskQueue is a shared resource between multiple threads
- Therefore lock and unlock when modifying it by addition or deletion, using a mutex.

Make some tasks



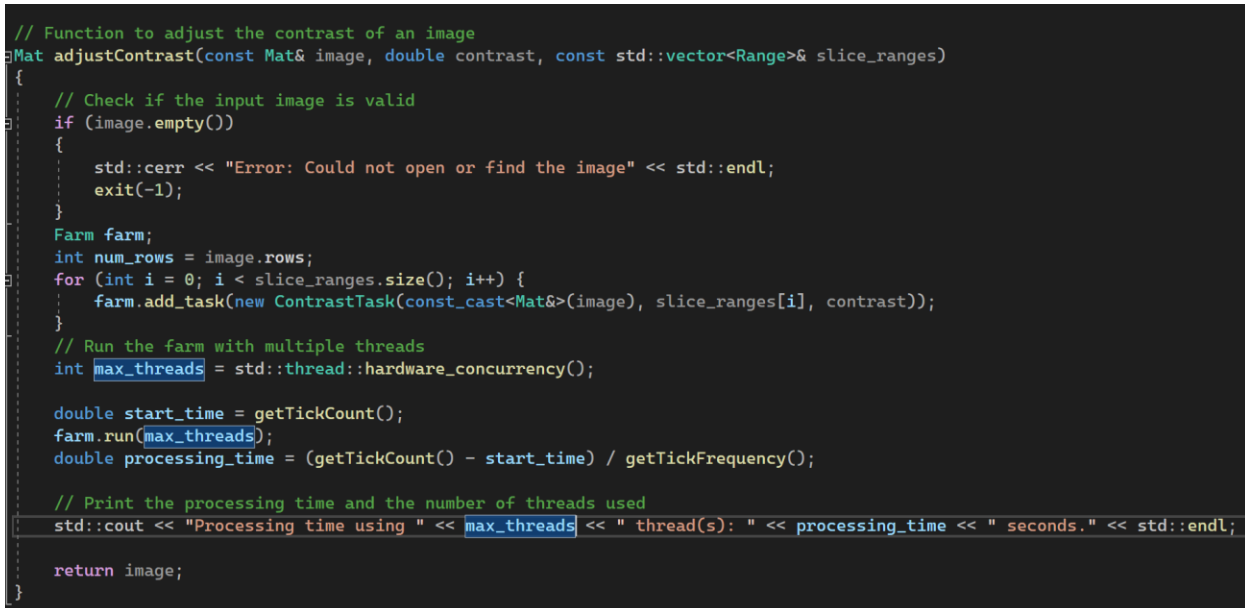
```cpp
class BlurTask : public Task {
public:
    BlurTask(Mat& image, Range row_range, int kernel_size) : m_image(image), m_row_range(row_range), m_kernel_size(kernel_size) {}

    virtual void run() {
        for (int row = m_row_range.start; row < m_row_range.end; row++) {
            Mat row_image = m_image.row(row);
            GaussianBlur(row_image, row_image, Size(m_kernel_size, m_kernel_size), 0);
        }
    }

private:
    Mat& m_image;
    Range m_row_range;
    int m_kernel_size;
};
```

Divide image up into slices (rows of images are ideal) and put into a vector



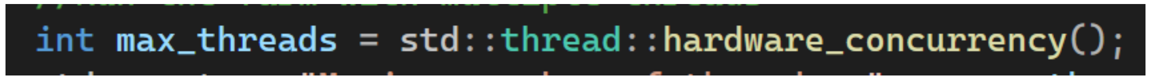
```cpp
//Divide the image in to sized slices
std::vector<Range> slice_ranges;
int num_slices = image.rows;
std::cout << num_slices << std::endl;
int slice_height = image.rows / num_slices;
int start_row = 0;
int end_row = 0;
for (int i = 0; i < num_slices - 1; i++) {
    end_row = start_row + slice_height;
    slice_ranges.push_back(Range(start_row, end_row));
    start_row = end_row;
}
slice_ranges.push_back(Range(start_row, image.rows));
```

Make function that runs the task on a specific slice, adds it to farm



```cpp
// Function to adjust the contrast of an image
Mat adjustContrast(const Mat& image, double contrast, const std::vector<Range>& slice_ranges)
{
    // Check if the input image is valid
    if (image.empty())
    {
        std::cerr << "Error: Could not open or find the image" << std::endl;
        exit(-1);
    }
    Farm farm;
    int num_rows = image.rows;
    for (int i = 0; i < slice_ranges.size(); i++) {
        farm.add_task(new ContrastTask(const_cast<Mat&>(image), slice_ranges[i], contrast));
    }
    // Run the farm with multiple threads
    int max_threads = std::thread::hardware_concurrency();

    double start_time = getTickCount();
    farm.run(max_threads);
    double processing_time = (getTickCount() - start_time) / getTickFrequency();

    // Print the processing time and the number of threads used
    std::cout << "Processing time using " << max_threads << " thread(s): " << processing_time << " seconds." << std::endl;

    return image;
}
```

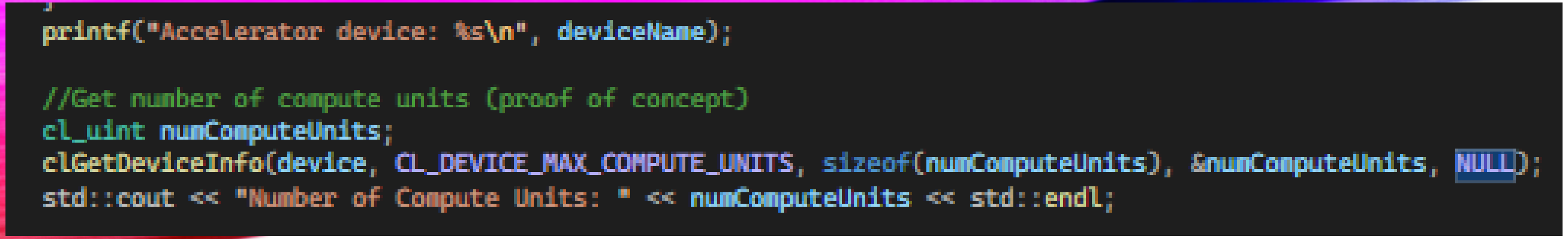
Ascertain thread pool through this



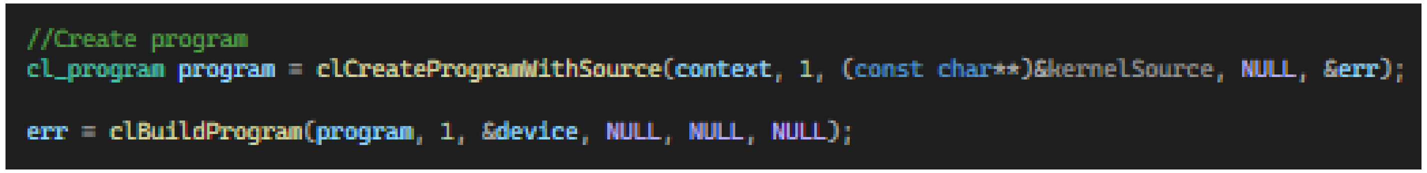
```cpp
int max_threads = std::thread::hardware_concurrency();
```

GPU

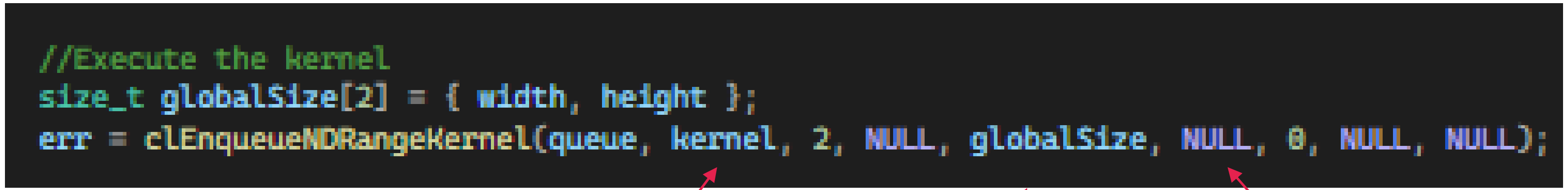
- Make function that essentially takes a kernel as an argument, put two kernels elsewhere because they're big blocks.


```
printf("Accelerator device: %s\n", deviceName);

//Get number of compute units (proof of concept)
cl_uint numComputeUnits;
clGetDeviceInfo(device, CL_DEVICE_MAX_COMPUTE_UNITS, sizeof(numComputeUnits), &numComputeUnits, NULL);
std::cout << "Number of Compute Units: " << numComputeUnits << std::endl;
```

Show compute units and GPU name, to prove it works.


```
//Create program
cl_program program = clCreateProgramWithSource(context, 1, (const char**)&kernelSource, NULL, &err);

err = clBuildProgram(program, 1, &device, NULL, NULL, NULL);
```

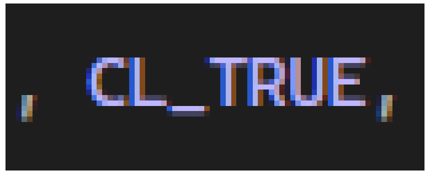
Tell the GPU that our kernel is a program to be ran


```
//Execute the kernel
size_t globalSize[2] = { width, height };
err = clEnqueueNDRangeKernel(queue, kernel, 2, NULL, globalSize, NULL, 0, NULL, NULL);
```

User defined kernel to run (blur or brightness)

2D array of width and height (Pixels in image)

Work Group size – essentially tiling. Ran out of time for this but setting it to NULL lets OpenCL handle it


```
, CL_TRUE,
```

Blocks until the buffer has finished writing

HARDWARE

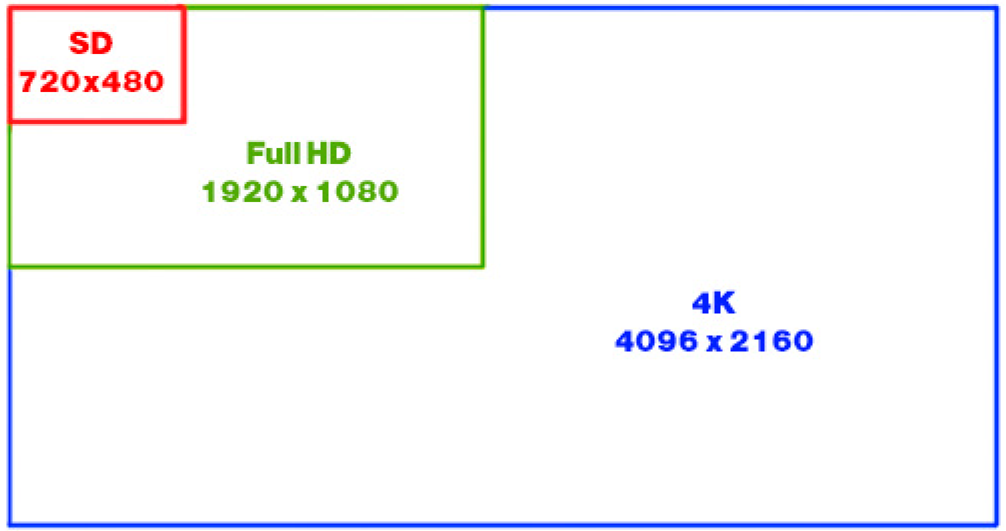
- Ryzen 9 5900X
- 12 cores
- 24 threads

AMD
RYZEN
AMD
RYZEN
9
THIS BOX INCLUDES AN AMD RYZEN™ PROCESSOR AND INSTALLATION INSTRUCTIONS.
THERMAL SOLUTION SOLD SEPARATELY.
5000 SERIES PROCESSOR

- Nvidia RTX 3080ti
- 80 OpenCL compute units

PERFORMANCE TESTING

- Only going to test blur – it's the most computationally expensive so will be the best example – I only have 10 minutes.
- Will depend highly on image size
- Ergo, 2 images tested of varying size
- All 16:9 aspect ratio
- 640x360 (360p) – Small data size
- 3840x2160 (4K)- Large Data Size
- Standard resolutions, more practical.



cat.jpg
(360x640)

city.jpg
(3840x2160)

SLICE DIVISION ON THE CPU

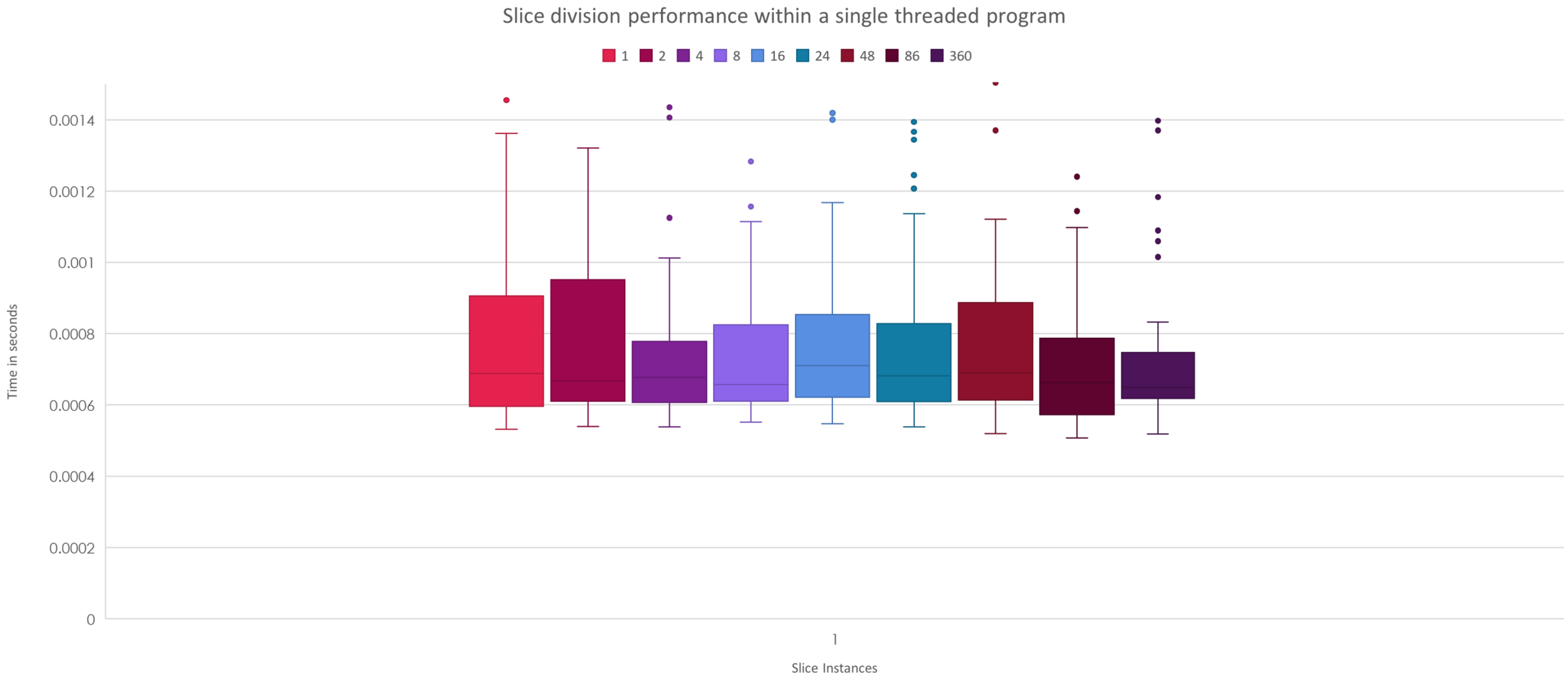
- Run blur operation at kernel size of 9 with 1 thread on cat.jpg
- Run blur operation at kernel size of 9 with 24 threads on cat.jpg
- Should allow us to tell the relationship between threads/slices
- Divide into slices of sizes, 2, 4,8,16 and 24.
- Standardized variables should ensure we're only measuring impact of slice number
- Run 100 times, output to CSV.
- Strictly timing how long it takes the task farm to produce the blurred image

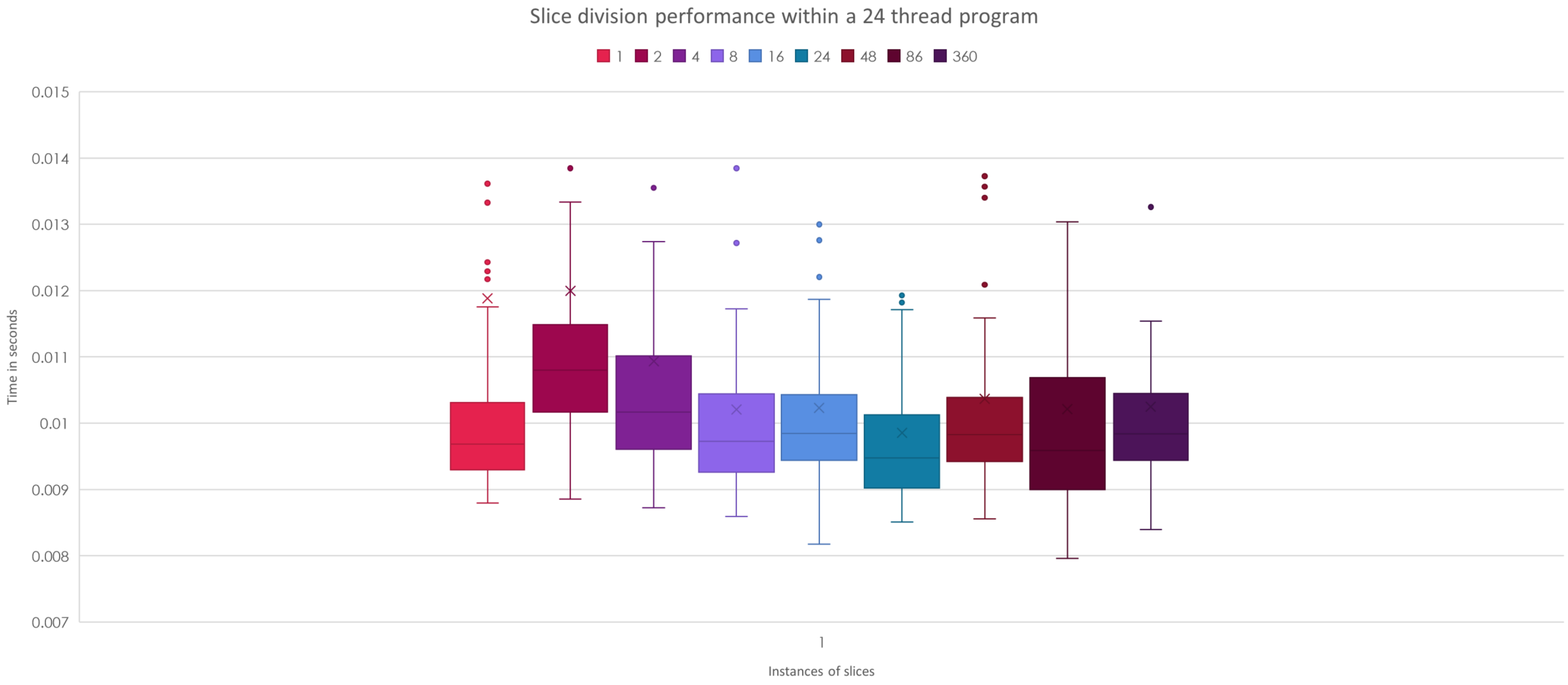
VERDICT

Slices	Average time in MS (1 thread)
1	0.097925
2	0.0997978
4	0.0990156
8	0.0965304
16	0.09708875
24	0.0961467
48	0.09649035
86	0.1000903
360	0.1009315

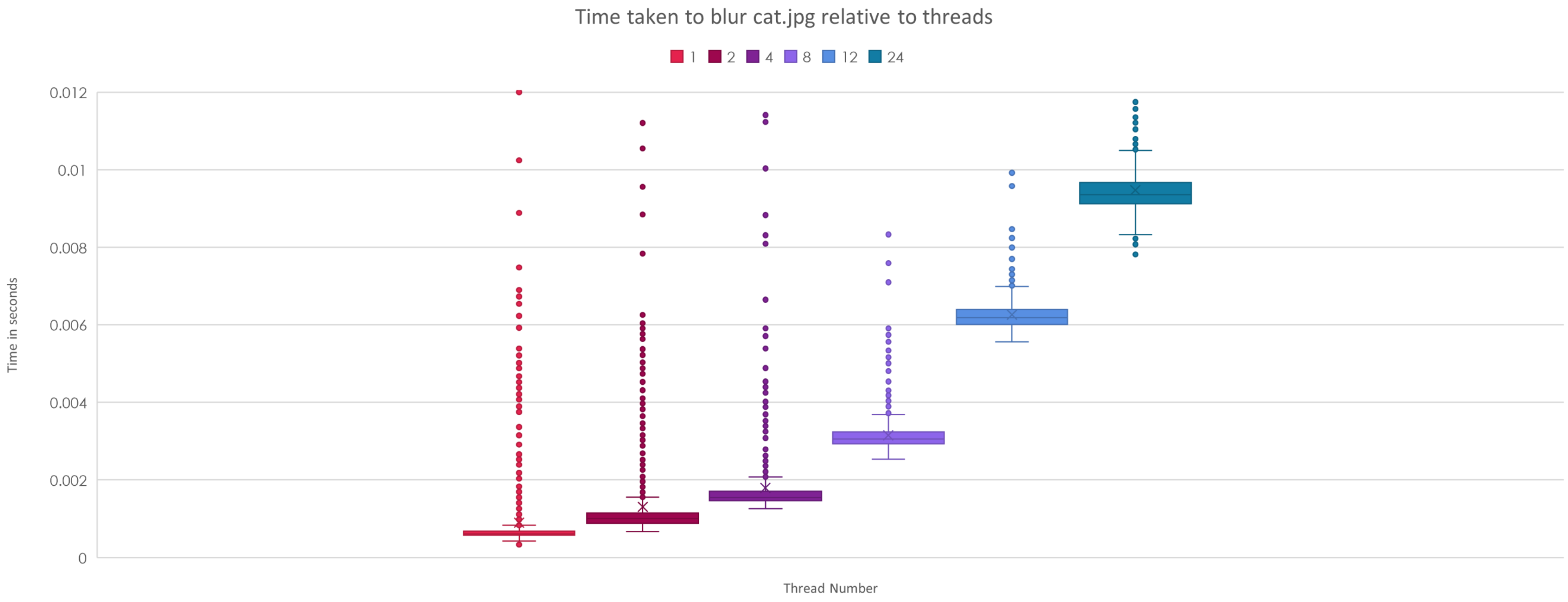
Slices	Average time in MS (24 thread)
1	0.10636375
2	0.06588375
4	0.03823565
8	0.0245279
16	0.0238793
24	0.02235935
48	0.02783815
86	0.0268266
360	0.02168595

- These results are somewhat murky
- A close relationship between thread number and slice number (thread 1 performing well with a single slice), 24 thread and 24 slices performing second best.
- A suggestion that the task farm can effectively handle large datasets well, with 360 slices being the fastest therefore handled well by the 24 threaded Task Farm.

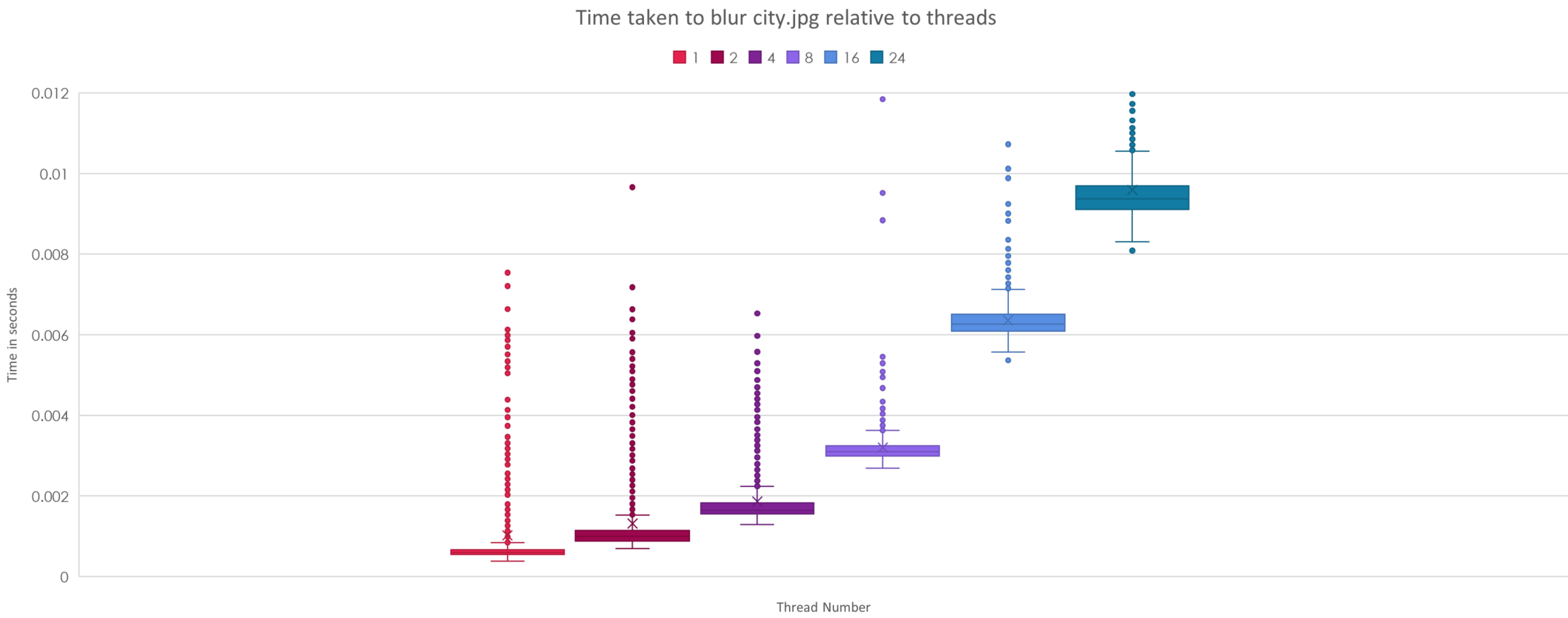

Slice division performance within a single threaded program
1
2
4
8
16
24
48
86
360
0.0014
0.0012
0.001
0.0008
0.0006
0.0004
0.0002
0
Time in seconds
1
Slice Instances


Slice division performance within a 24 thread program
1
2
4
8
16
24
48
86
360
0.015
0.014
0.013
0.012
0.011
0.01
0.009
0.008
0.007
Time in seconds
1
Instances of slices

CAT.JPG (360P IMAGE) – CPU BLUR



CITY.JPG (3840X2160) – CPU BLUR

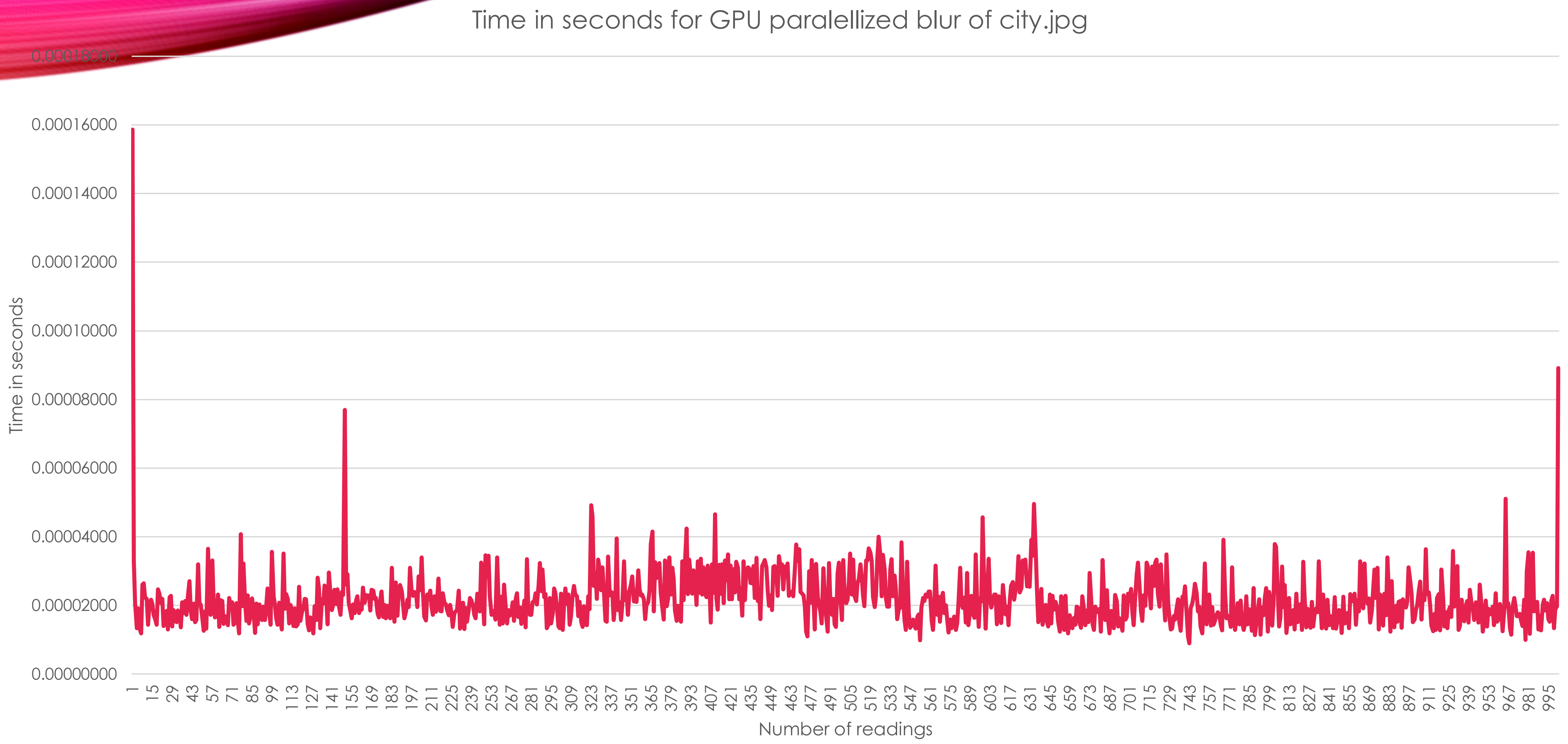


WHY IS THIS THE CASE?

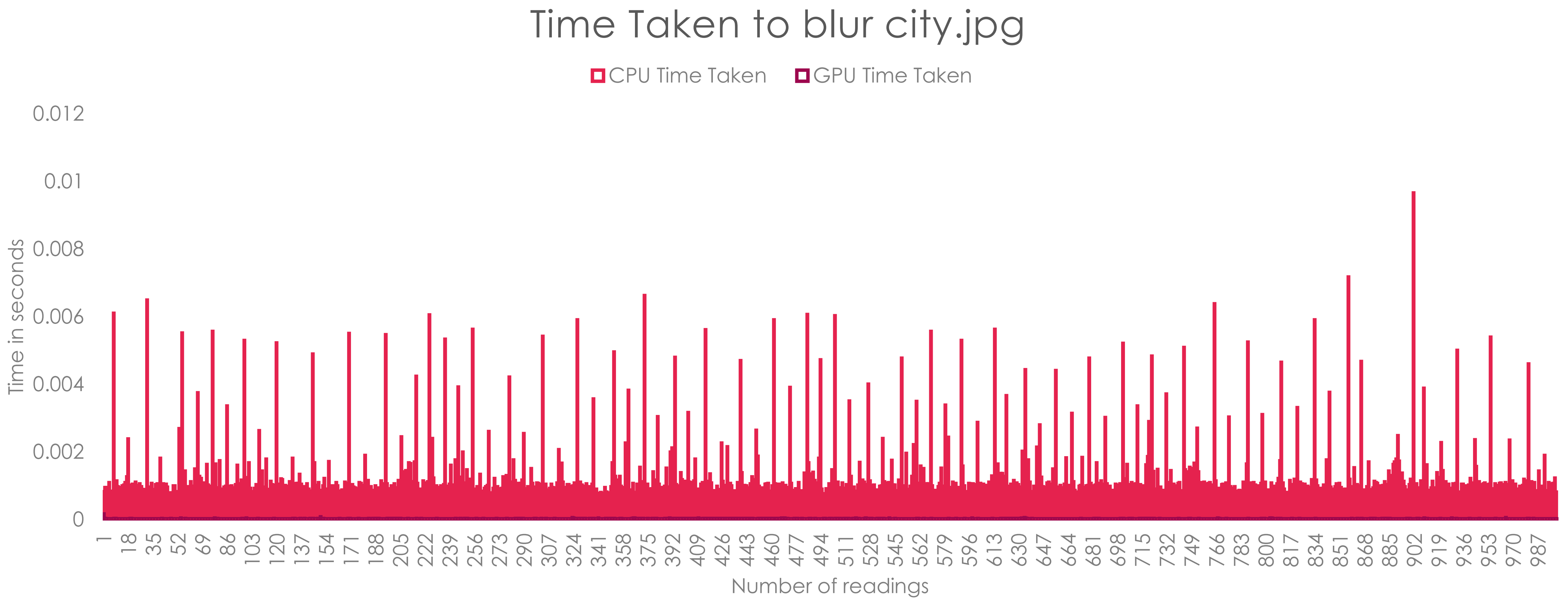
- Unlike mandlebrot, blurring isn’t complex – just a repeated instruction over a large dataset. It’s data heavy vs control or computation heavy.
- We would expect to see limited parallelization benefits through the CPU
- We can also see that CPU overhead is quickly going to overwhelm our benefits
- It objectively does benefit from parallelization – but not in any kind of scalable fashion.

ENTER GPU

- Measure the time it takes the GPU to compute a size 9 kernel on the GPU
- We'd expect it to be faster.
- It's essentially incomparably faster.
- Forced to use scientific notation
- The average time it took the GPU to blur the 4k image on the kernel was 0.00008920 seconds


Time in seconds for GPU paralellized blur of city.jpg
0.00018000
0.00016000
0.00014000
0.00012000
0.00010000
0.00008000
0.00006000
0.00004000
0.00002000
0.00000000
Time in seconds
Number of readings
1
15
29
43
57
71
85
99
113
127
141
155
169
183
197
211
225
239
253
267
281
295
309
323
337
351
365
379
393
407
421
435
449
463
477
491
505
519
533
547
561
575
589
603
617
631
645
659
673
687
701
715
729
743
757
771
785
799
813
827
841
855
869
883
897
911
925
939
953
967
981
995

- If we plot this against the fastest CPU method of blurring City.jpg – 2 threads. Then we can see the absurd discrepancy in speed. Or rather , we can't.



DID WE SEE WHAT WE EXPECTED?

- Yes - We would expect that image blurring and interpretation methods would be far faster on the GPU, as CPU is better at control heavy operations vs a GPU which is better at data heavy operations
- More threads does increase performance, up until a point at which thread overhead exceeds worthiness.
- More slices do improve performance when using a task farm
- This is kind of a failure… but kind of not. We saw what we expected to on the CPU, which wasn’t ideal - but the GPU was incredibly quick.

JUSTIFICATIONS

- If I could go back and do it again, I’d probably not bother with CPU
- The fact I started with the CPU on a data heavy operation was frankly stupid
- Essentially meant I had to do a GPU example to compare against and make this not a failure
- On the upside… makes perfect sense for my GPU implementation
- Ideal for this kind of operation with lots of data and low synchronization

ISSUES

- GPU does suffer on the transfer back to the CPU though, if we measure that it results in markedly slower times.
- The CPU also has an issue with initial loading – a single run will often be markedly slower due to library initialization and caching.

HOW WOULD I IMPROVE IT?

- GPU code does not make effective use of tiles at the minute, it can definitely do so.
- CPU code could ideally implement a method for automatic thread number assignment – just because x number of threads are ideal for me does not mean that is universally true

DOES IT MEET THE REQUIREMENTS? (GPU)

- it can run 2 different kernels
- Doesn’t need to share resources, ergo is not appropriate. It does use waiting though, which whilst not locking is resource safety
- Consideration of strategies to enhance performance Thread divergence is very limited – because there are no branches. No current implementation of tiling.

DOES IT MEET THE REQUIREMENTS? (CPU)

- Runs at least 3 threads (providing they're available) and 3 differing tasks can be ran on the taskQueue.
- Mutexes are used on the taskQueue as it's shared.

THANKS FOR WATCHING