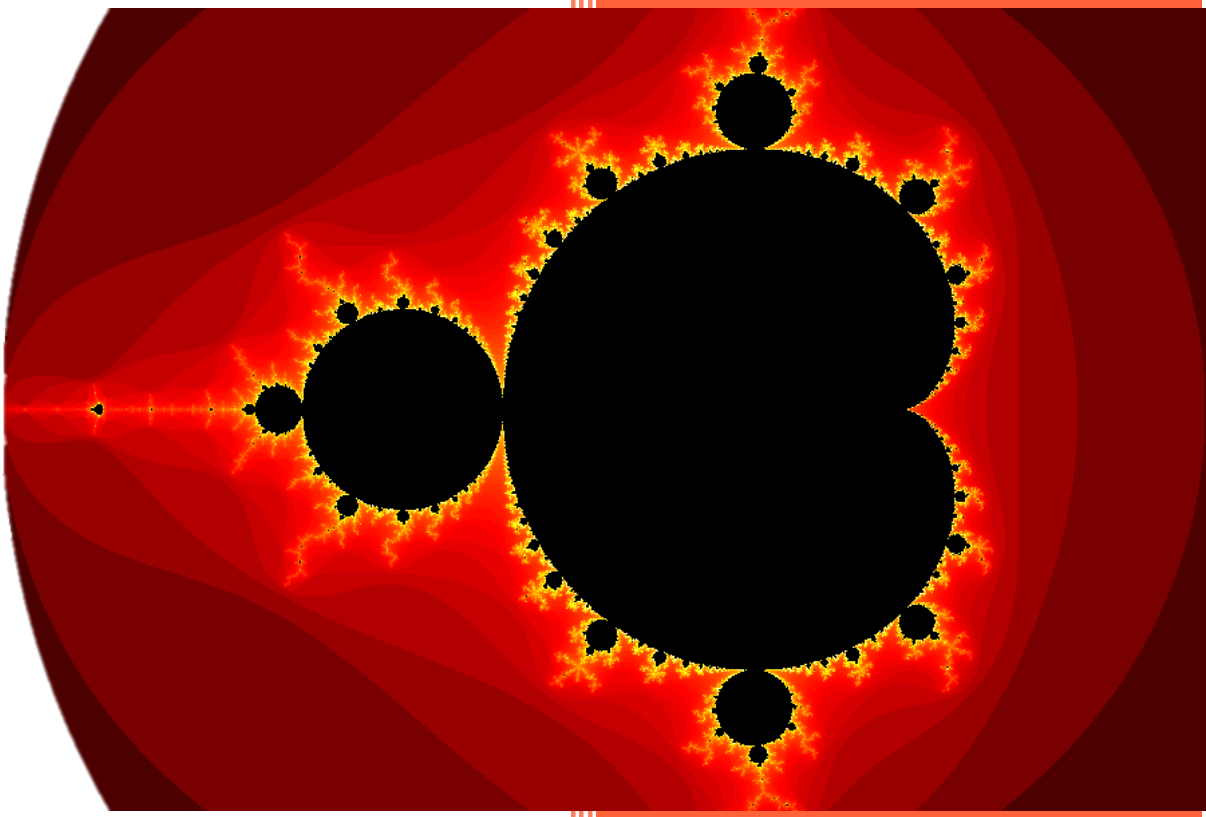
2017

# Mandelbrot set



Constantin Toader

B00243868

2/5/2017

# Contents

## The Mandelbrot Set

The term Mandelbrot set is used to refer both to a general class of fractal sets and to a particular instance of such a set. In general, a Mandelbrot set marks the set of points in the complex plane such that the corresponding Julia set is connected and not computable.

The Mandelbrot set is the set obtained from the quadratic recurrence equation

$$z_{n+1} = z_n^2 + \mathrm{C}$$

With $z_0 = \mathrm{C}$, where points C in the complex plane for which the orbit of $z_n$ does not tend to infinity are in the set. Setting $z_0$ equal to any point **in the set** that is not a periodic point gives the same result (Weisstein, n.d.).

The Mandelbrot set is the dark glob in the centre of the picture. The colour of the pixels outside indicate how many iterations it took for each of those pixels until the condition for being outside the Mandelbrot set was satisfied.

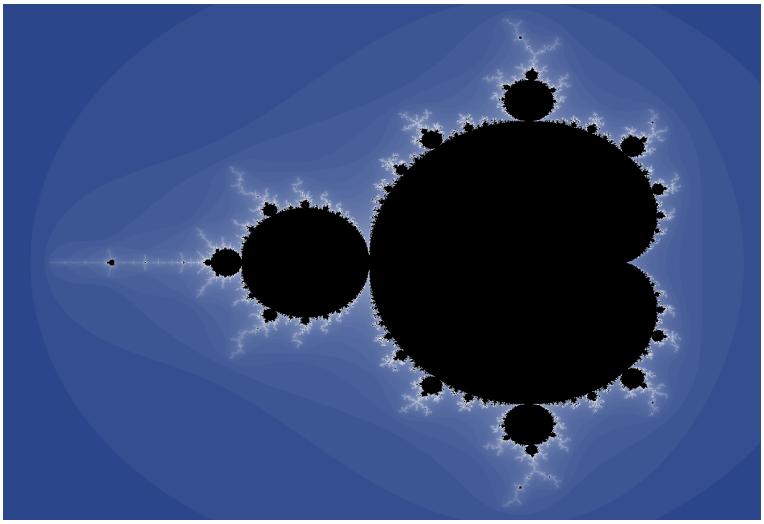


*Figure 1. Mandelbrot Set (Mandelbrot Set, n.d.)*

## The Algorithm

In pseudocode, the algorithm looks as the following:

```
For each pixel (Px, Py) on the screen, do:
{
  x0 = scaled x coordinate of pixel (between (-2.5, 1))
  y0 = scaled y coordinate of pixel (between (-1, 1))
  x = 0.0
  y = 0.0
  iteration = 0
  max_iteration = 1000
  while (x*x + y*y < 2*2  AND  iteration < max_iteration) {
    xtemp = x*x - y*y + x0
    y = 2*x*y + y0
    x = xtemp
    iteration = iteration + 1
  }
  color = palette[iteration]
  plot(Px, Py, color)
}
```

*Figure 2. Pseudocode (En.wikipedia.org, 2017)*

## The Problem

Given a sample code that can generate a section of the Mandelbrot set using a Central Processing Unit, there is a requirement to optimise and port the execution of the application to a General Purpose Graphic Processing Unit using Nvidia's CUDA API.

## The Sample Code

Compiling the code generates an executable, which, when executed, saves a section of an instance of the Mandelbrot set to a ppm image file with a size of 4096x4096 pixels.

The main program is composed of three functions with the following execution times:

- *alloc_2d* – **13 ms:** reserves memory for the image to be generated;
- *calc_mandel* – **3420 ms:** fills the pixels with the right colours reflecting the Mandelbrot set;
- *screen_dump* – **236 ms:** saves the generated data to an image file, for a better visualisation;

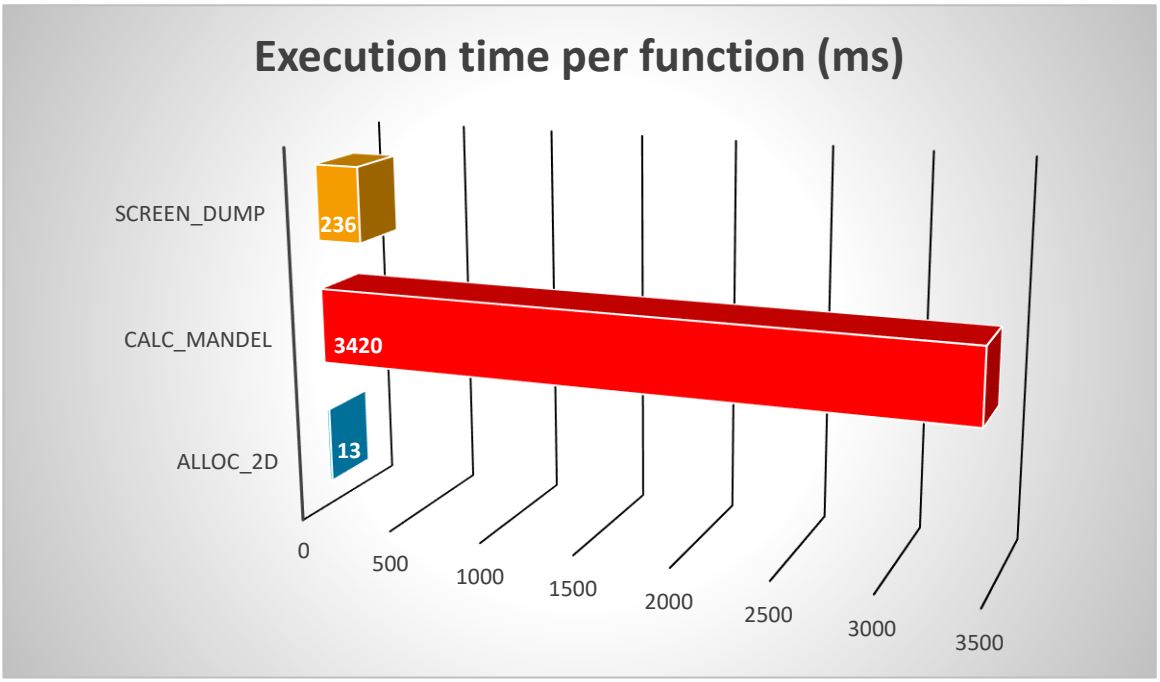



*Figure 3. Execution time for each function in CPU-based solution*

During the execution, *calc_mandel* calls another function *map_colour.* This is the code of the function:

```
void map_colour(rgb_t * const px)
{
  const uchar num_shades = 16;
  const rgb_t mapping[num_shades] =
    {{66,30,15},   {25,7,26},    {9,1,47},     {4,4,73},     {0,7,100},
     {12,44,138},  {24,82,177},  {57,125,209}, {134,181,229},{211,236,248},
     {241,233,191},{248,201,95}, {255,170,0},  {204,128,0},  {153,87,0},
     {106,52,3}};

  if (px->r == max_iter || px->r == 0) {
    px->r = 0; px->g = 0; px->b = 0;
  } else {
    const uchar uc = px->r % num_shades;
    *px = mapping[uc];
  }
}
```

With italic bold is highlighted the code that slows the execution time.

This code is executed in the *calc_mandel* function for height *x* width iterations, and with each iteration we create the shades in the body of the map_colour, and with each iteraration a pixel is evaluated and given a colour.

During the execution of the *calc_mandel* we already traverse all the pixels, so the shade attribution for each pixel could be done inside the for loops belonging to the *calc_mandel*.

## Sample code optimisation

Before optimising the code on the GPU, improvements can be done to the code running on the CPU. To highlight the performance gain, the code was executed 5 times and the average execution time was measured with each improvement. The measurement was done with a high_resolution_clock.

```
81  81        px->b = iter;
82          -   }
83          - }
84          -
85          - for (int i = 0; i < height; i++) {
86          -   rgb_t *px = row_ptrs[i];
87          -   for (int j = 0; j < width; j++, px++) {
88          -     map_colour(px);
        82  +       map_colour(px);
89  83        }
```

*Figure 4. Default Code optimisation - Attempt 1*

The first attempt was to move the map_colour in the for loops from the calc_mandel function as shown in Figure 3. Interestingly after doing this, the performance dropped and the execution time grew from 3630 milliseconds to 3789 milliseconds. Seeing that this was not a good result the code was reverted.

```
36  36
        37  +const uchar num_shades = 16;
        38  +const rgb_t mapping[num_shades] =
        39  +{ { 66,30,15 },{ 25,7,26 },{ 9,1,47 },{ 4,4,73 },{ 0,7,100 },
        40  +{ 12,44,138 },{ 24,82,177 },{ 57,125,209 },{ 134,181,229 },{ 211,236,248 },
        41  +{ 241,233,191 },{ 248,201,95 },{ 255,170,0 },{ 204,128,0 },{ 153,87,0 },
        42  +{ 106,52,3 } };
        43  +
37  44    void map_colour(rgb_t * const px)
38  45    {
39          - const uchar num_shades = 16;
40          - const rgb_t mapping[num_shades] =
41          -   {{66,30,15},   {25,7,26},    {9,1,47},     {4,4,73},     {0,7,100},
42          -    {12,44,138},  {24,82,177},  {57,125,209}, {134,181,229},{211,236,248},
43          -    {241,233,191},{248,201,95}, {255,170,0},  {204,128,0},  {153,87,0},
44          -    {106,52,3}};
45          -
46  46    if (px->r == max_iter || px->r == 0) {
```

*Figure 5. Default Code optimisation - Attempt 2*

The second attempt was to move the declaration of the shades outside of the map_colour function as seen in Figure 4. This means that the shades would only be declared only one time, even though they would sit in the global scope. The results show a performance gain, with the execution time dropping from 3630 milliseconds to 3325 milliseconds.

```
77  77          } while (iter++ < max_iter && zx2 + zy2 < 4);
78  -
79  -       px->r = iter;
80  -       px->g = iter;
81  -       px->b = iter;
82  -     }
83  -   }
84  -
85  -   for (int i = 0; i < height; i++) {
86  -     rgb_t *px = row_ptrs[i];
87  -     for (int j = 0; j < width; j++, px++) {
88  -       map_colour(px);
    78  +         if (iter == max_iter || iter == 0) {
    79  +                 px->r = 0; px->g = 0; px->b = 0;
    80  +         }
    81  +         else {
    82  +                 *px = mapping[iter % num_shades];
    83  +         }
89  84      }
```

*Figure 6. Default Code optimisation - Attempt 3*

The last step was to try again to move the colouring step in the calc_mandel for loop, but this time by eliminating the map_colour function, and to evaluate the pixel using the value of iter, as shown in Figure 5. Also, to calculate the right index of the shade, the local declaration of the uc has been removed. The results highlight another performance gain, with the execution time dropping from the previous result of 3325 milliseconds to 2959 milliseconds.

Overall with the current changes, the code runs faster, on average, with 671.4 milliseconds, a 18% performance gain.

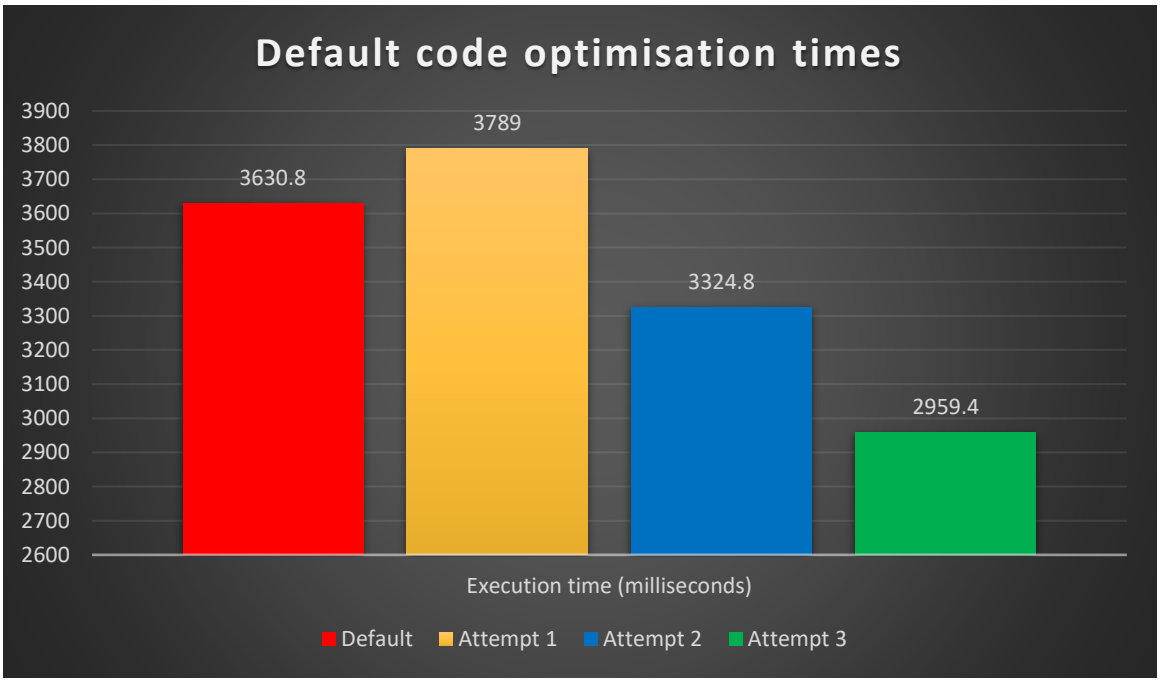



*Figure 7. Default code optimisation graph (lower is better)*

## Cuda version

With CUDA, *host* CPU code can launch GPU *kernels* by calling *device* functions that execute on the GPU.

The necessary steps to convert the serial code into code running with the CUDA API are the following:

- create handles for the image data for both device and host
- allocate device memory for the device image data
- prepare kernel
    - find grid size and block size
    - calculate index
    - run Mandelbrot algorithm
- retrieve image from device
- output image to file
- clean-up

To abstract away the functionality related to the creation of a Mandelbrot set, the code related to it has been encapsulated in a separated class. The class is defined by the width, height and the starting complex number which is tested if it is inside a such set. Also, two pointers are being declared to easy identify the image data on the host and on the device.

The constructor is responsible for instant allocation of memory on the device using an init() function, but also an array of Pixels is reserved on the host. The destructor is responsible for cleaning up the memory. Memory from both host and device is freed when the destructor is called.

Other function is fetch() which syncs the GPU processing units and copies the image data from device to host. The last useful function (ignoring the getters and setters) is the saveAs() function, which takes a string as a parameter to use as a name when writing the image to disk. This function does not check if there is any data on the host data, so it should be used only after the data has been fetched from the device.

The Util.h header file holds directives and definitions for some constants and structs. A check() function improves logging the cudaError_t. Another useful utility is the measure template which records the execution time for any method passed to it.

## The hardware

The specification of the used GPU with Compute Capability of 3.0 is visible on the right in Figure 8 (NV_Quadro_K4000, 2013). The device has a maximum block size of 1024 threads, a warp size of 32, maximum registers 63, max shared memory per block 49152 bytes and the maximum grid size is 16. These limitations will reduce the number of combinations that can be used to output the best performance.

| Physical Limits for GPU Compute Capability: | 3.0 |
|---|---|
| Threads per Warp | 32 |
| Max Warps per Multiprocessor | 64 |
| Max Thread Blocks per Multiprocessor | 16 |
| Max Threads per Multiprocessor | 2048 |
| Maximum Thread Block Size | 1024 |
| Registers per Multiprocessor | 65536 |
| Max Registers per Thread Block | 65536 |
| Max Registers per Thread | 63 |
| Shared Memory per Multiprocessor (bytes) | 49152 |
| Max Shared Memory per Block | 49152 |
| Register allocation unit size | 256 |
| Register allocation granularity | warp |
| Shared Memory allocation unit size | 256 |
| Warp allocation granularity | 4 |

*Figure 8. Hardware limitations for Quadro K4000*

## Running the first CUDA version

There are two important aspects to be considered before starting to use the CUDA API: index calculation and kernel parameters identification.

```
int row = blockIdx.y * blockDim.y + threadIdx.y;

int column = blockIdx.x * blockDim.x + threadIdx.x;

int index = row * width + column;
```

The other aspect involved calculating the grid size and block size. A 2D grid will be used to calculate each direction (weight or height) in its own dimension. To try to find the right values, we need to consider the warp size of 32 to avoid padding, so for the block sizes, a multiple of 32 seems to be a good candidate (talonmies, 2012), but using our limitations it should be maximum 1024. Also, the grid can have maximum 16 blocks. Once a block size has been identified, the number of blocks, the grid size will be computed by dividing the image data to the number of threads activated for each block.

So, for example, to obtain a measure the execution time for a 4096x4096 image, the following experiment was done ignoring the hardware limits using the following blockSize values:

| Blocks: | 4096 | 2048 | 1024 | 512 | 256 | 128 | 64 | 32 | 16 | 8 | 4 | 2 | 1 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Threads: | 1 | 2 | 4 | 8 | 16 | 32 | 64 | 128 | 256 | 512 | 1024 | 2048 | 4096 |

. . . . . .

```
dim3 blockSize(4, 4);

dim3 gridSize(1024, 1024);

calc_mandel << < gridSize, blockSize >> >();
```

. . . . . .

Another aspect was to make sure that the shades used at each iteration are immediately available to be used on each block. For this, the array of shades has been globally declared as __constant__.

The application was executed with 32 registers per thread and 256 bytes of shared memory per block. Later these values will be tweaked to improve performance.

## Averaging and measuring – Methodology

To try to find the best execution times for a piece of code, micro-benchmarking has been performed using the following procedure:

Given a target code which requires its execution time to be measured, we surround it between a start and end timer, record the difference and repeat this process for an arbitrary chosen number of cycles and average the differences. This process is again repeated for different kernel configurations and each average time is saved along with the configuration it has been used. The lowest average should highlight the best kernel configuration. The process is again repeated for different image resolutions. All timings are recorded in different files (one file with timings for each resolution) and through observation, the fastest times will be highlighted.

For each cycle
    Assign current time instance to start
    Run block of code
    Assign current time to end
    Add difference to total cycle time
End for
Record time/cycles as average execution time.

## Cuda Occupancy API

To ease the optimisation, Harris (2014) suggests to use the Occupancy API if a CUDA 6.5 or higher is available. The API makes it possible to compute an efficient execution configuration for a kernel.

Using two variables for the block and grid size and the cudaOccupancyMaxPotentialBlockSize() we can calculate the grid size depending on the width (or height) of the image:

- a variable minGridSize and the block size is passed by reference to the above-mentioned function, together with the kernel and the maximum block size is obtained.
- based on the block size, gridSize is calculated using (width + blockSize -1) / blockSize
- this way a rounded (integer) number of blocks is used.
- these two values are stored in a KernelProperties struct and can be further used.

Just for logging, the theoretical occupancy is also calculated after the deviceProperties have been retrieved:

occupancy = (maxActiveBlocks * blockSize / warpSize) / (maxThreadsPerMultiProcessor / warpSize);

This can be further simplified to:

occupancy = maxActiveBlocks * blockSize / maxThreadsPerMultiProcessor;

Even though Luitjens & Rennich (2011) suggest that 66% occupancy is enough to saturate the bandwidth, our kernel configuration will try to use it at 100% occupancy. During the measuring process, the recommended configuration will be also timed and recorded.

## Data collection

To collect minimum, maximum and the average execution time for each kernel configuration we run the application via command line passing the following arguments:

```
gpu.exe width height true realPart imaginaryPart filename
```

- **width** and **height** represent the value of the width and height of the image in pixels
- **true** is a flag that enables output comparison against cpu output
- **realPart** is the real component of the complex number to check if it is in the set
- **imaginaryPart** is the imaginary component of the complex number to check if it is in the set
- **filename** is the name of the image to be generated.

All arguments have default values so when we want to run the program for a certain image size, we just pass the width at height:

```
gpu.exe 128 128
```

Images will be generated carrying the output, and in the filename the image will have the kernel configuration and the execution time. Also, same data is collected in an excel file with these headers:

| Width | Height | Blocks | Threads | Millis | Nanos |
|---|---|---|---|---|---|

*Figure 9. The headers of the Excel generated tables*

Repeating the execution of the program for different images sizes, enough data is generated that can be further processed to reveal the minimum, maximum and average execution time.

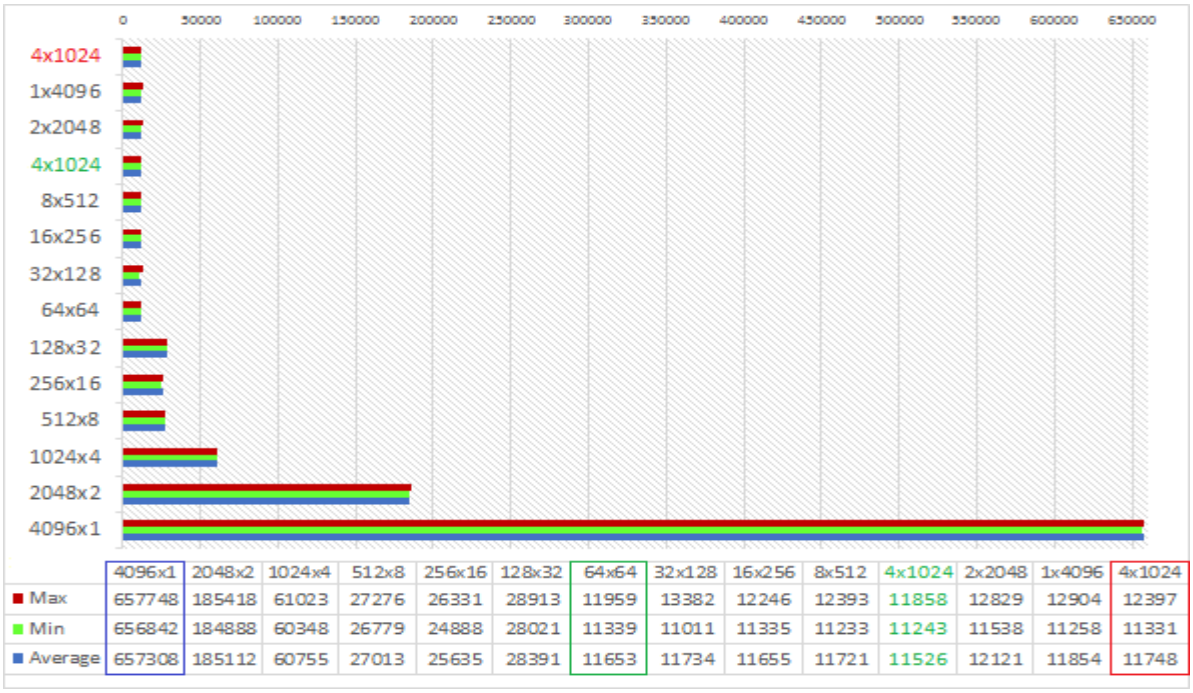
## Results

The most results are stored in the appendix section which will have charts that display the **minimum**, **maximum** and **average** execution time needed to generate Mandelbrot sets of different sizes, for various kernel configurations in format blocks x threads, visible on the table-legend at the bottom of each chart. Always the topmost entry in the chart, displayed with red text, will represent the recommended configuration (using the CUDA occupancy API). We normally care about the average

time as it is more representative. The fastest average time is highlighted with green text in the data table, the recommended configuration time has a red outline, the fastest configuration that does not use the CUDA occupancy API is displayed using a green outline, while the lowest execution times are represented with a blue outline. The application generates a lot of data, even irrelevant data (data collected using more than 16 blocks, more than 1024 threads, less than 32 threads where warp padding would be necessary). Other image sizes execution times for each configuration can be found in the appendix.

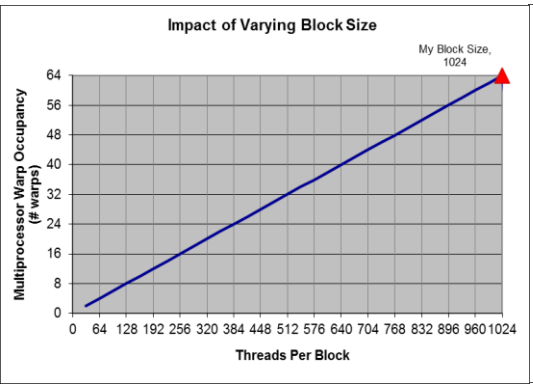
### 4096x4096

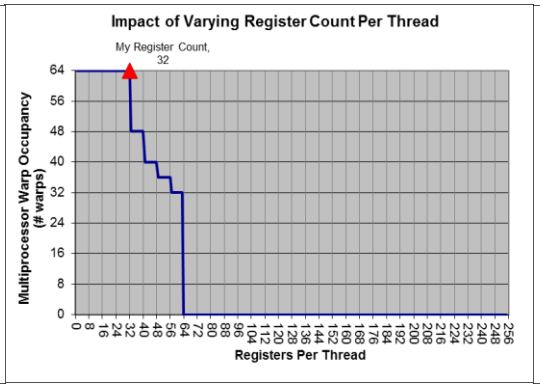
For the 4096x4096 set, CUDA suggested to use minimum 4 blocks of 1024 threads each. The only improvements that can be done for this kernel configuration are related to the block size, shared memory made available, and the number of registers, as currently running with 12 registers per thread.




| | 4096x1 | 2048x2 | 1024x4 | 512x8 | 256x16 | 128x32 | 64x64 | 32x128 | 16x256 | 8x512 | 4x1024 | 2x2048 | 1x4096 | 4x1024 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Max | 657748 | 185418 | 61023 | 27276 | 26331 | 28913 | 11959 | 13382 | 12246 | 12393 | 11858 | 12829 | 12904 | 12397 |
| Min | 656842 | 184888 | 60348 | 26779 | 24888 | 28021 | 11339 | 11011 | 11335 | 11233 | 11243 | 11538 | 11258 | 11331 |
| Average | 657308 | 185112 | 60755 | 27013 | 25635 | 28391 | 11653 | 11734 | 11655 | 11721 | 11526 | 12121 | 11854 | 11748 |

*Figure 10. Execution time in microseconds for various kernel configurations*



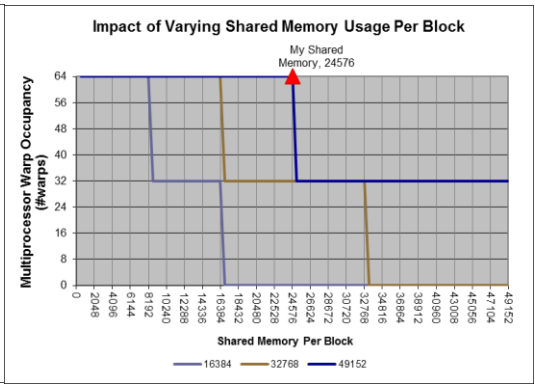




*Figure 11. Optimizations available for 2x1024 kernel configurations*

The best configuration for full occupancy is indeed the one suggested by the API, but the shared memory could be increased to 24576 bytes per block and the maximum register count should be 32. Any superior value will have an impact on the occupancy. Other performant configurations are:

| Min Blocks (max 16) | Threads | Max Registers | Max Shared Memory | Occupancy |
|---|---|---|---|---|
| **16** | 128 | 32 | 3072 | 100% |
| **8** | 256 | 32 | 6144 | 100% |
| **4** | 512 | 32 | 12288 | 100% |
| **2** | 1024 | 32 | 24576 | 100% |

If the compute ability would not limit the hardware to a maximum block size of 1024 active threads, and 2048 threads could be used at once, it is easy to predict what the configuration for a full occupancy would be:

| Min Blocks (max 16) | Threads | Max Registers | Max Shared Memory | Occupancy |
|---|---|---|---|---|
| **1** | 2048 | 32 | 49152 | 100% |

It is interesting to notice that even configurations outside the GPU specification limits generated correct images and could be successfully measured. Probably, for example, in the 32 blocks of 128 threads configuration, the application completed 2 cycles with 16 blocks of 128 threads. Some incompatible configurations obtained decent times, while other obtain disastrous times (4096 x 1).

Overall comparing the best average GPU execution time against the average CPU execution time, it can be noted a performance improvement of 2948 milliseconds, making the GPU application 269 times faster, which is quite impressive.

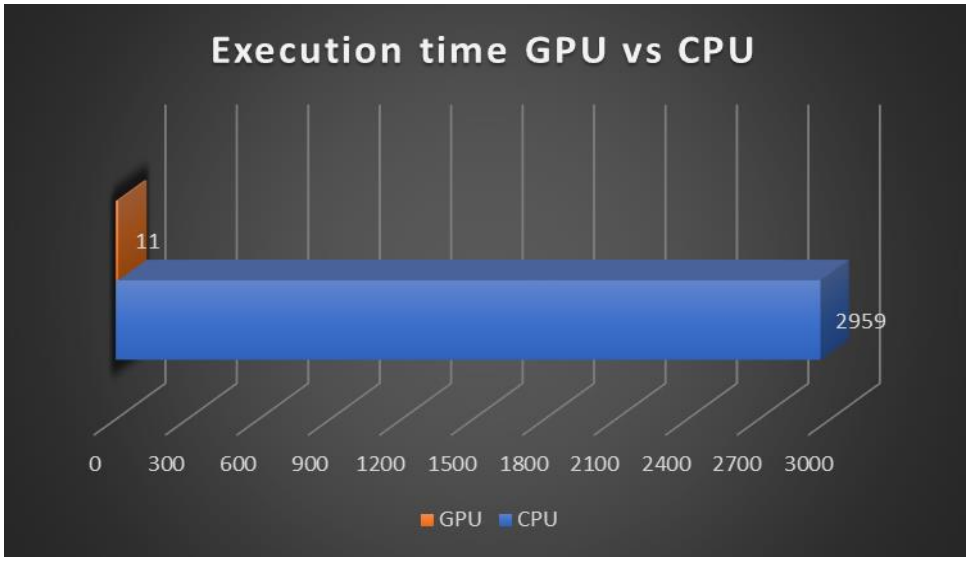



*Figure 12. GPU vs CPU execution time*

Other configurations have been checked for different image size, and in a few cases (512x512 image and 1024x1024 image), incompatible configurations (1x2048) obtained better times than the CUDA recommended configuration.

Also, it can be noted that in all suggested configurations, CUDA recommended 1024 threads, only varying the minimum number of blocks. The next image shows the suggested minimum number of blocks for different image sizes:

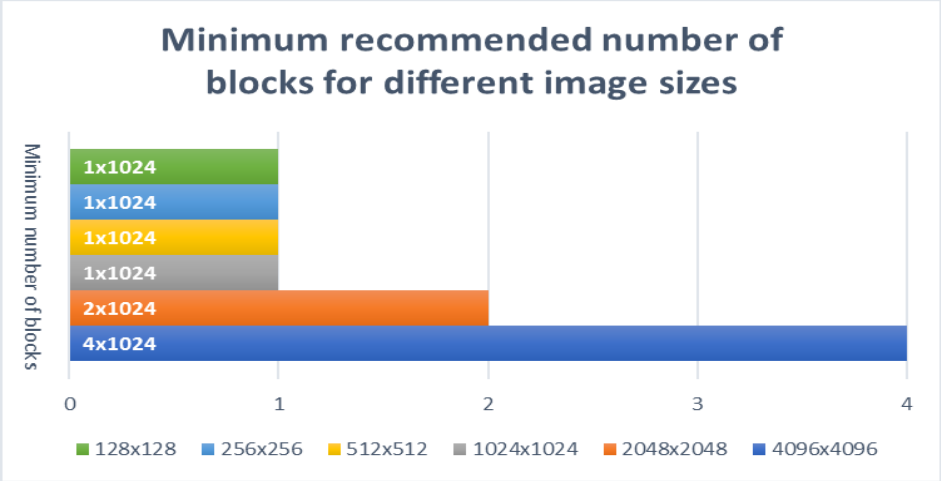



*Figure 13. Minimum number of blocks suggested for different image sizes*

To generalise a better configuration for all different sizes, the minimum number of active blocks should be 4. Because it is a minimum, it should work efficiently with configurations that used to be 1x1024 or 2x1024.

## Comparing the output

A wrongly calculated index, or a misused shade, could totally corrupt the resulted Mandelbrot set. To verify the validity of the image, on request, using the third command line argument set to "true" the application can be executed normally and before the last step a CPU generated Mandelbrot Set can be compared pixel by pixel with the GPU set generated with CUDA recommended kernel settings.

```
bool isEqual(Pixel* first, Pixel* second) {
	return first->b == second->b
		&& first->g == second->g
		&& first->r == second->r;
};
```

The comparison checks if two pixels are equal. This happens when all their red, green, blue channels have the same values. This check is performed for all the pixels in the compare function.

```
void compare(Pixel* first, Pixel* second, int width, int height) {
	int size = width * height;
	for (int i = 0; i < size; i++) {
		if (!isEqual(first++, second++)) {
			cout << endl << "Images are different";
			cin.get();
			return;
		}
		if (i && i % PIXEL_COUNT_REPORT == 0) {
			cout << endl << "So far " << i << " pixels are identical";
		}
	}
	cout << endl << "All " << size << " pixels are identical";
	cout << endl << "Images are identical";
	cin.get();
};
```

## Conclusions

Obviously using CUDA API efficiently can significantly increase performance. In most cases, efficient results are returned when occupancy of the threads is 100%, except for the accepted cases when the data has finished processing in 1 cycle (smaller images).

One aspect that can be observed, is that even if the maximum blocks per SM is 16, CUDA had no problems generating output for grids with more than 16 blocks, even though in most cases it was a lot slower. Also, it is interesting to observe that the CUDA API did not always have the best average time and, for some of the outputs, the API failed to provide the best performance.

It is important to saturate the hardware in the best possible way. Using the CUDA Occupancy Calculator it is possible to observe for which kernel configurations the theoretical 100% occupancy can be achieved. Setting a maximum number of registers and maximum amount of share memory as shown in the Occupancy Calculator makes the application easier to obtain best efficiency. Generalising a configuration to work for different image sizes can be performed with an insignificant, tolerable performance loss for some of the images.

A performance achievement like the one obtained when comparing the GPU against CPU execution time, is outstanding, as the Cuda application is 269 times faster, and probably can be even further optimised.

## How to reproduce the results:

In the application folder, run the gpu.exe in command line using the following arguments:

```
gpu.exe 4096 4096
```

Repeat the command for the remaining image sizes: 2048, 1024, 512, 256, 128.

To compare GPU output against CPU output run the application with the following arguments:

```
gpu.exe 4096 4096 true
```

This will compare the GPU output generated by the kernel using the recommended configuration against the CPU output.

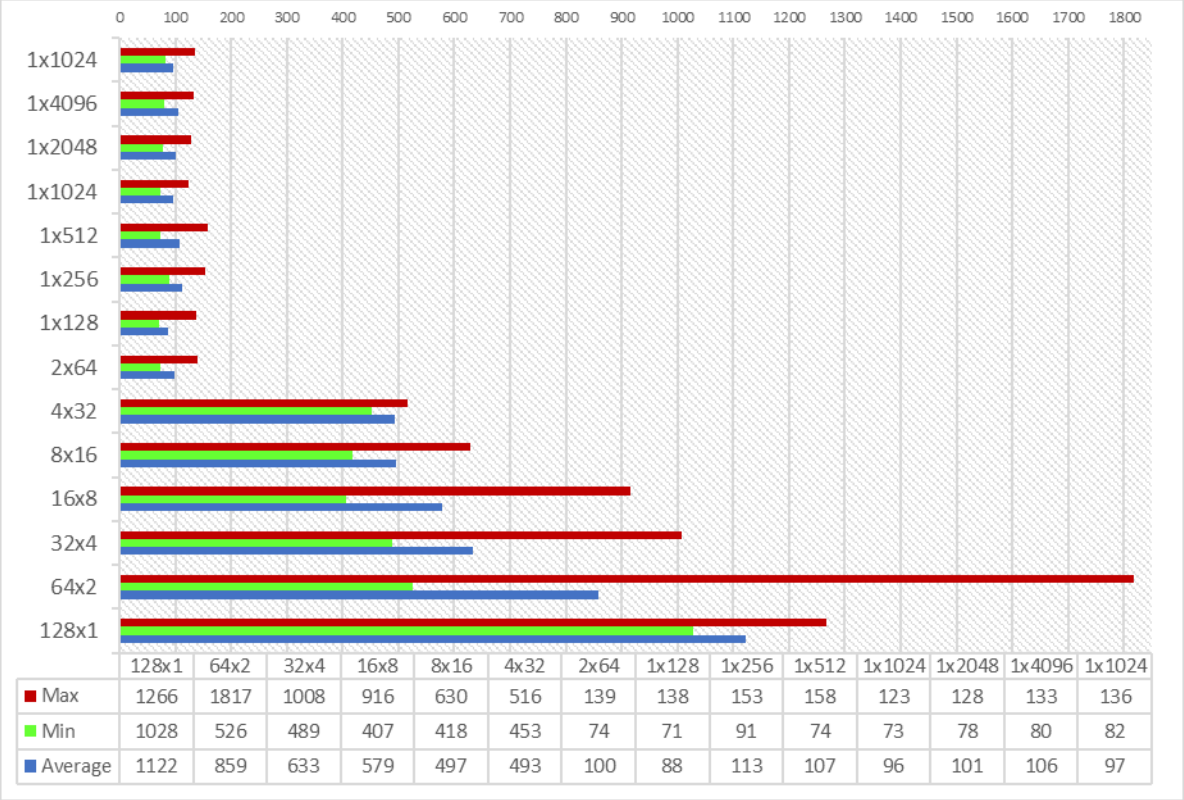
To manually enter the complex number value, run the application with the following arguments:

```
gpu.exe 4096 4096 false -0.6 0
```

-0.6 is the real part and 0 is the imaginary part of the complex number.

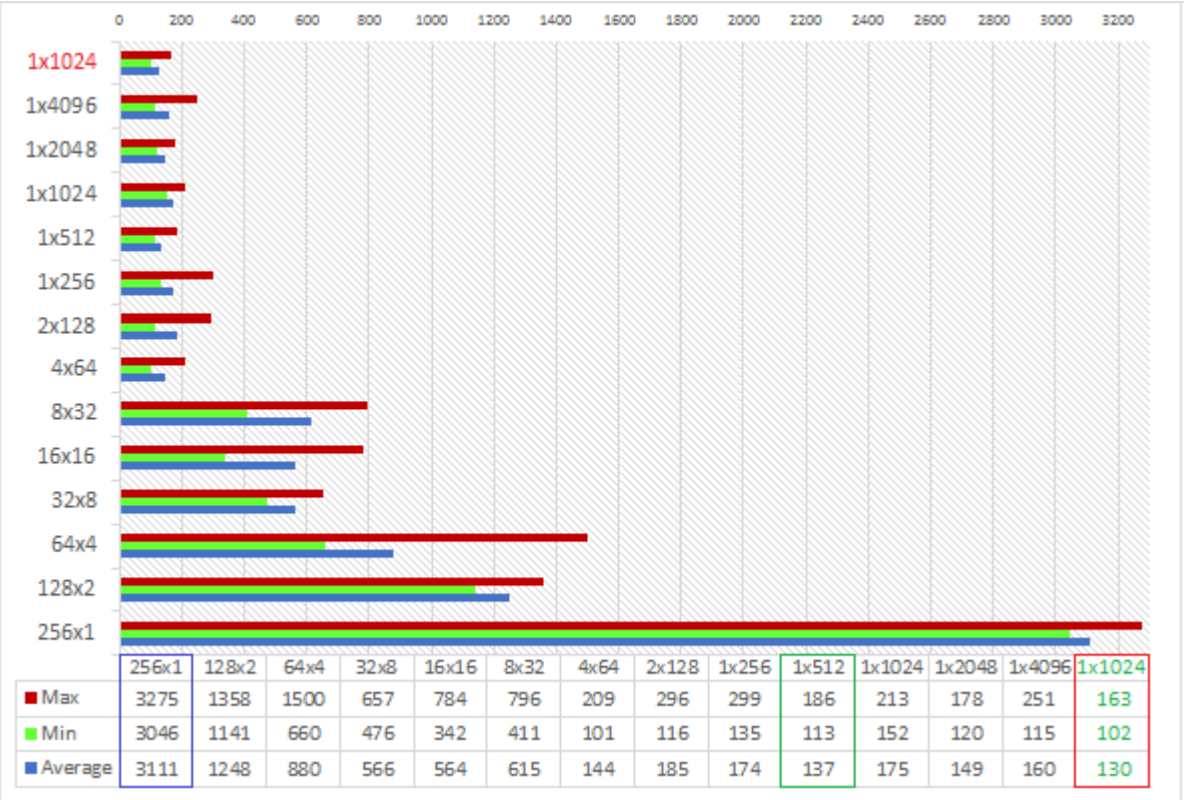
## Appendix (charts for various kernel configuration and their obtained timings)

### 128x128



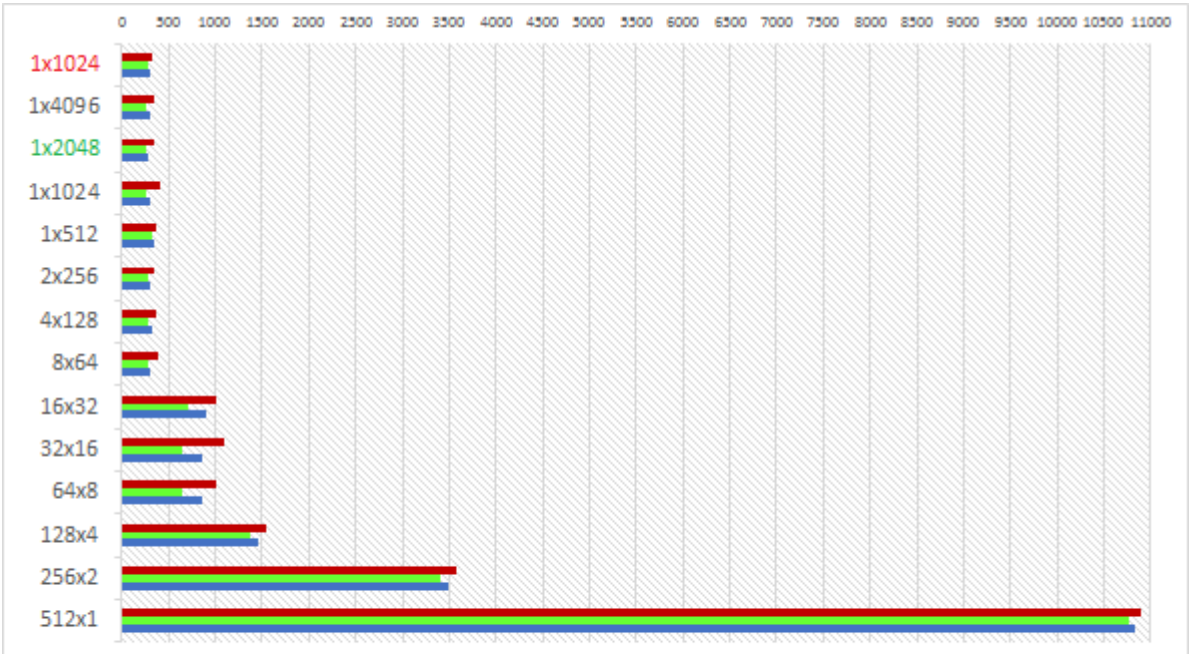

| | 128x1 | 64x2 | 32x4 | 16x8 | 8x16 | 4x32 | 2x64 | 1x128 | 1x256 | 1x512 | 1x1024 | 1x2048 | 1x4096 | 1x1024 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Max | 1266 | 1817 | 1008 | 916 | 630 | 516 | 139 | 138 | 153 | 158 | 123 | 128 | 133 | 136 |
| Min | 1028 | 526 | 489 | 407 | 418 | 453 | 74 | 71 | 91 | 74 | 73 | 78 | 80 | 82 |
| Average | 1122 | 859 | 633 | 579 | 497 | 493 | 100 | 88 | 113 | 107 | 96 | 101 | 106 | 97 |

### 256x256



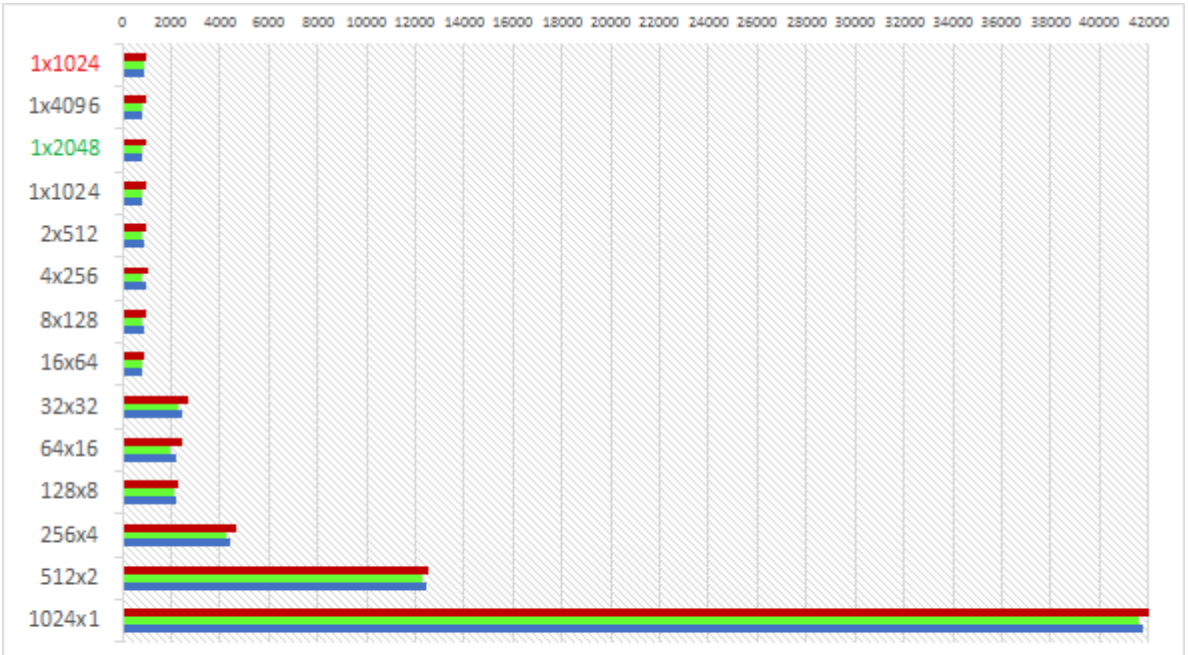

| | 256x1 | 128x2 | 64x4 | 32x8 | 16x16 | 8x32 | 4x64 | 2x128 | 1x256 | 1x512 | 1x1024 | 1x2048 | 1x4096 | 1x1024 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Max | 3275 | 1358 | 1500 | 657 | 784 | 796 | 209 | 296 | 299 | 186 | 213 | 178 | 251 | 163 |
| Min | 3046 | 1141 | 660 | 476 | 342 | 411 | 101 | 116 | 135 | 113 | 152 | 120 | 115 | 102 |
| Average | 3111 | 1248 | 880 | 566 | 564 | 615 | 144 | 185 | 174 | 137 | 175 | 149 | 160 | 130 |

## 512x512



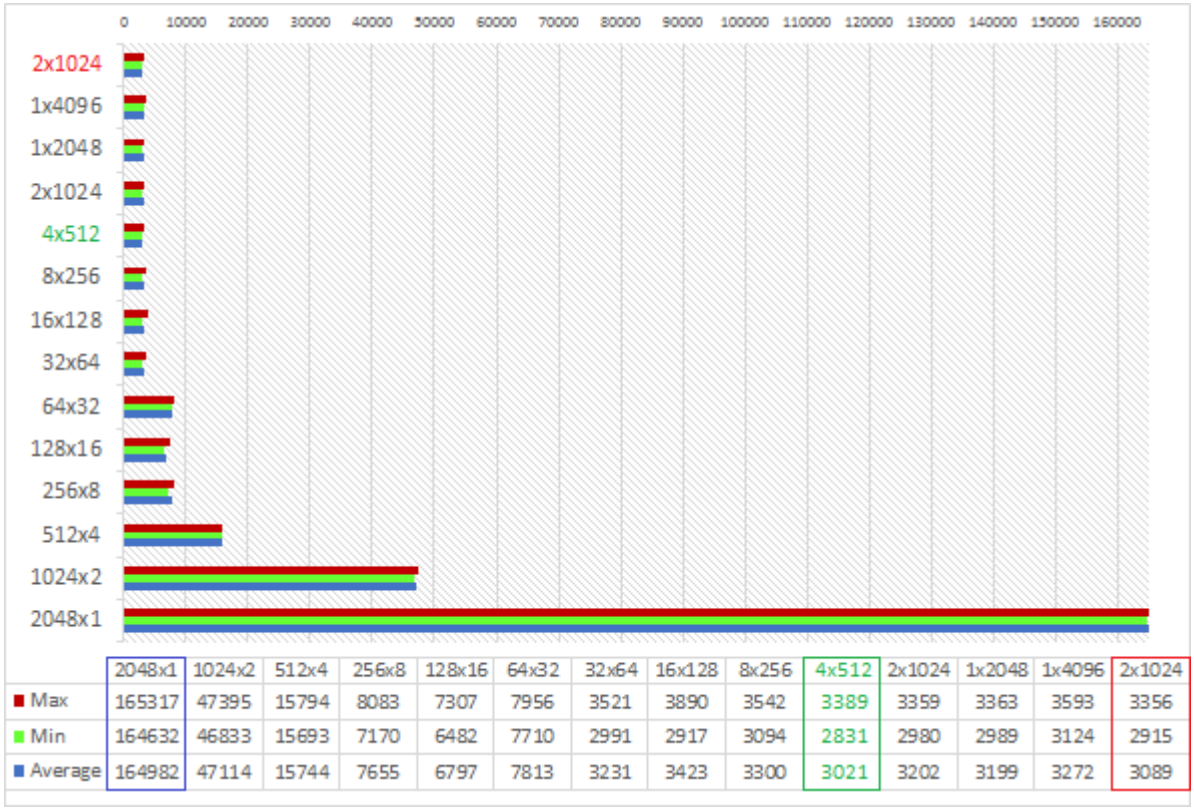

| | 512x1 | 256x2 | 128x4 | 64x8 | 32x16 | 16x32 | 8x64 | 4x128 | 2x256 | 1x512 | 1x1024 | 1x2048 | 1x4096 | 1x1024 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Max | 10885 | 3577 | 1540 | 1000 | 1096 | 1005 | 381 | 365 | 339 | 369 | 405 | 343 | 338 | 328 |
| Min | 10758 | 3411 | 1372 | 646 | 647 | 705 | 278 | 281 | 290 | 321 | 265 | 264 | 271 | 277 |
| Average | 10828 | 3481 | 1458 | 856 | 849 | 897 | 304 | 321 | 314 | 342 | 309 | 286 | 300 | 298 |

## 1024x1024



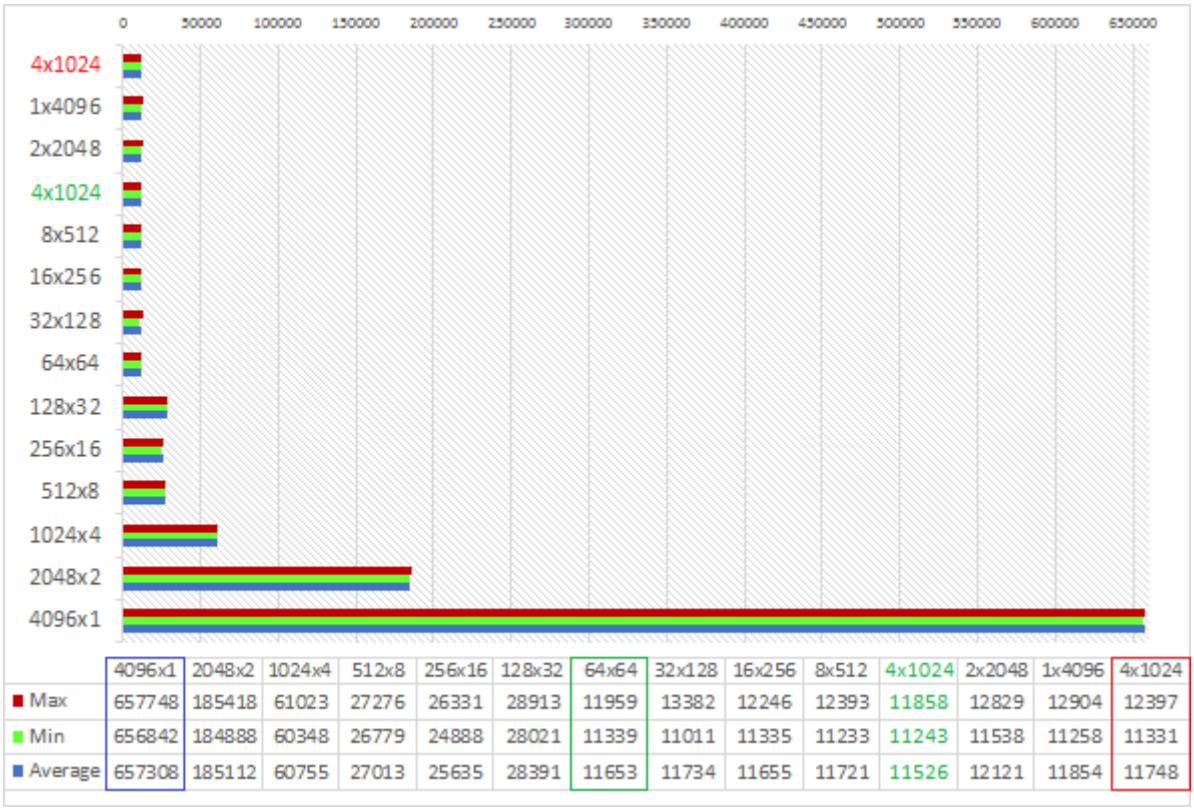

| | 1024x1 | 512x2 | 256x4 | 128x8 | 64x16 | 32x32 | 16x64 | 8x128 | 4x256 | 2x512 | 1x1024 | 1x2048 | 1x4096 | 1x1024 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Max | 42246 | 12549 | 4709 | 2282 | 2480 | 2664 | 929 | 1010 | 1087 | 991 | 954 | 947 | 949 | 991 |
| Min | 41580 | 12326 | 4259 | 2145 | 1965 | 2283 | 793 | 794 | 843 | 811 | 791 | 779 | 791 | 868 |
| Average | 41815 | 12429 | 4450 | 2209 | 2246 | 2461 | 835 | 879 | 971 | 874 | 852 | 831 | 836 | 917 |

## 2048x2048




| | 2048x1 | 1024x2 | 512x4 | 256x8 | 128x16 | 64x32 | 32x64 | 16x128 | 8x256 | 4x512 | 2x1024 | 1x2048 | 1x4096 | 2x1024 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Max | 165317 | 47395 | 15794 | 8083 | 7307 | 7956 | 3521 | 3890 | 3542 | 3389 | 3359 | 3363 | 3593 | 3356 |
| Min | 164632 | 46833 | 15693 | 7170 | 6482 | 7710 | 2991 | 2917 | 3094 | 2831 | 2980 | 2989 | 3124 | 2915 |
| Average | 164982 | 47114 | 15744 | 7655 | 6797 | 7813 | 3231 | 3423 | 3300 | 3021 | 3202 | 3199 | 3272 | 3089 |

## 4096x4096




| | 4096x1 | 2048x2 | 1024x4 | 512x8 | 256x16 | 128x32 | 64x64 | 32x128 | 16x256 | 8x512 | 4x1024 | 2x2048 | 1x4096 | 4x1024 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Max | 657748 | 185418 | 61023 | 27276 | 26331 | 28913 | 11959 | 13382 | 12246 | 12393 | 11858 | 12829 | 12904 | 12397 |
| Min | 656842 | 184888 | 60348 | 26779 | 24888 | 28021 | 11339 | 11011 | 11335 | 11233 | 11243 | 11538 | 11258 | 11331 |
| Average | 657308 | 185112 | 60755 | 27013 | 25635 | 28391 | 11653 | 11734 | 11655 | 11721 | 11526 | 12121 | 11854 | 11748 |

# References

DS_NV_Quadro_K4000_OCT13_NV_US_LR (2013). [image] Available at: http://www.nvidia.co.uk/content/PDF/data-sheet/DS_NV_Quadro_K4000_OCT13_NV_US_LR.pdf [Accessed 13 Feb. 2017]

En.wikipedia.org. (2017). *Mandelbrot set*. [online] Available at: https://en.wikipedia.org/wiki/Mandelbrot_set#Escape_time_algorithm [Accessed 5 Feb. 2017].

Harris, M. (2014). *CUDA Pro Tip: Occupancy API Simplifies Launch Configuration*. [online] Parallel Forall. Available at: https://devblogs.nvidia.com/parallelforall/cuda-pro-tip-occupancy-api-simplifies-launch-configuration/ [Accessed 13 Feb. 2017].

Luitjens, J & Rennich, R (2011) *CUDA Warps and Occupancy*. [online] Available at: http://on-demand.gputechconf.com/gtc-express/2011/presentations/cuda_webinars_WarpsAndOccupancy.pdf [Accessed 11 Feb. 2017].

Mandelbrot Set. (n.d.). [image] Available at: http://www.math.utah.edu/~alfeld/math/mandelbrot/large.gif [Accessed 5 Feb. 2017].

Weisstein, E. (n.d.). *Mandelbrot Set -- from Wolfram MathWorld*. [online] Mathworld.wolfram.com. Available at: http://mathworld.wolfram.com/MandelbrotSet.html [Accessed 5 Feb. 2017].

talonmies,. (2012). *How do I choose grid and block dimensions for CUDA kernels?*. *Stackoverflow.com*. [Online] Available at: http://stackoverflow.com/questions/9985912/how-do-i-choose-grid-and-block-dimensions-for-cuda-kernels [Accessed 13 Feb. 2017].

# Extra reading materials

http://www.stuffedcow.net/research/cudabmk

http://docs.nvidia.com/cuda/cuda-driver-api/group__CUDA__OCCUPANCY.html#group__CUDA__OCCUPANCY_1g04c0bb65630f82d9b99a5ca0203ee5aa

https://devblogs.nvidia.com/parallelforall/cuda-pro-tip-occupancy-api-simplifies-launch-configuration/