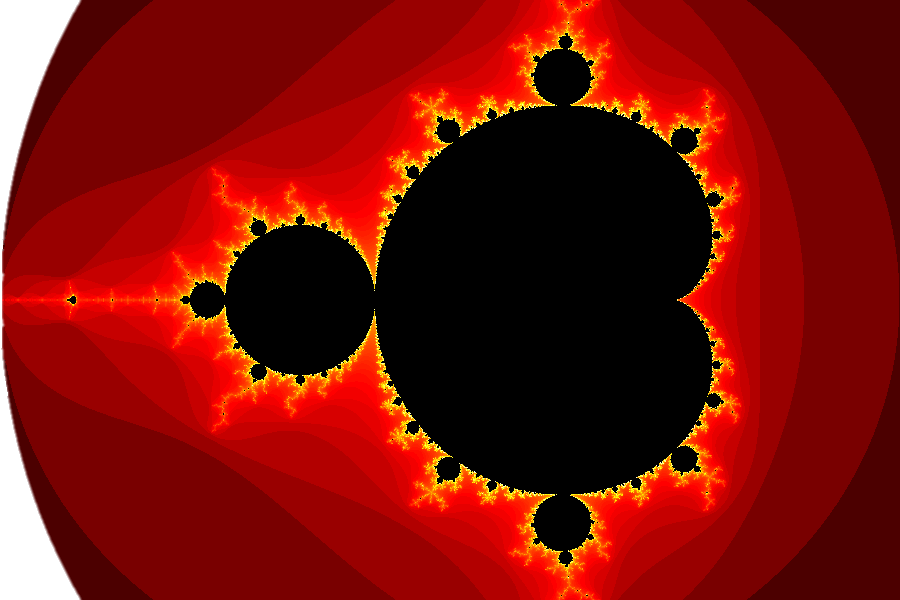
2017

Constantin Toader

B00243868

2/5/2017

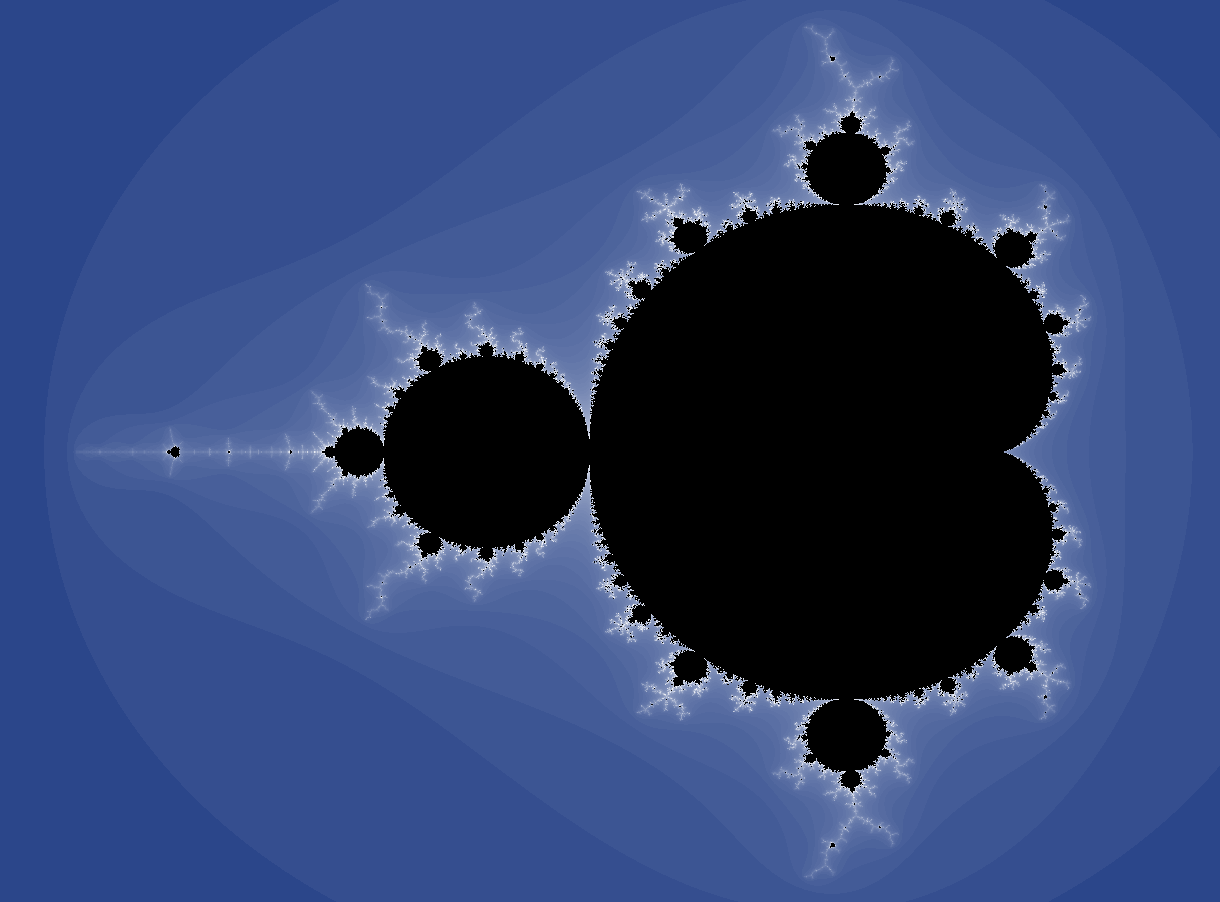
Mandelbrot set



## 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

With , where points in the complex plane for which the orbit of does not tend to infinity are in the set. Setting 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

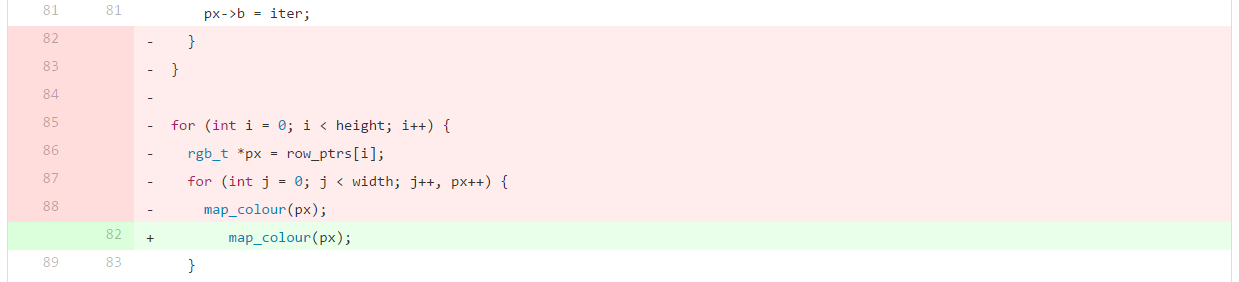
Before optimising the code on the GPU, improvements can be done to the code running on the CPU. To highlight the performance gain, the code was executed 5 times and the average execution time was measured with each improvement. The measurement was done with a high\_resolution\_clock.

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.

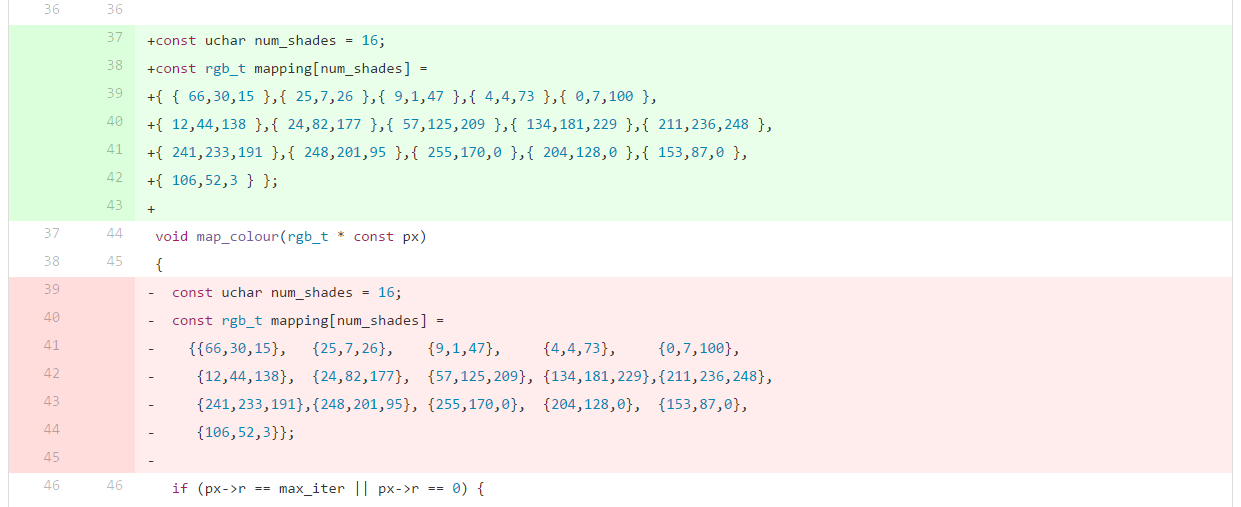


Figure 5. Default Code optimisation - Attempt 2

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.

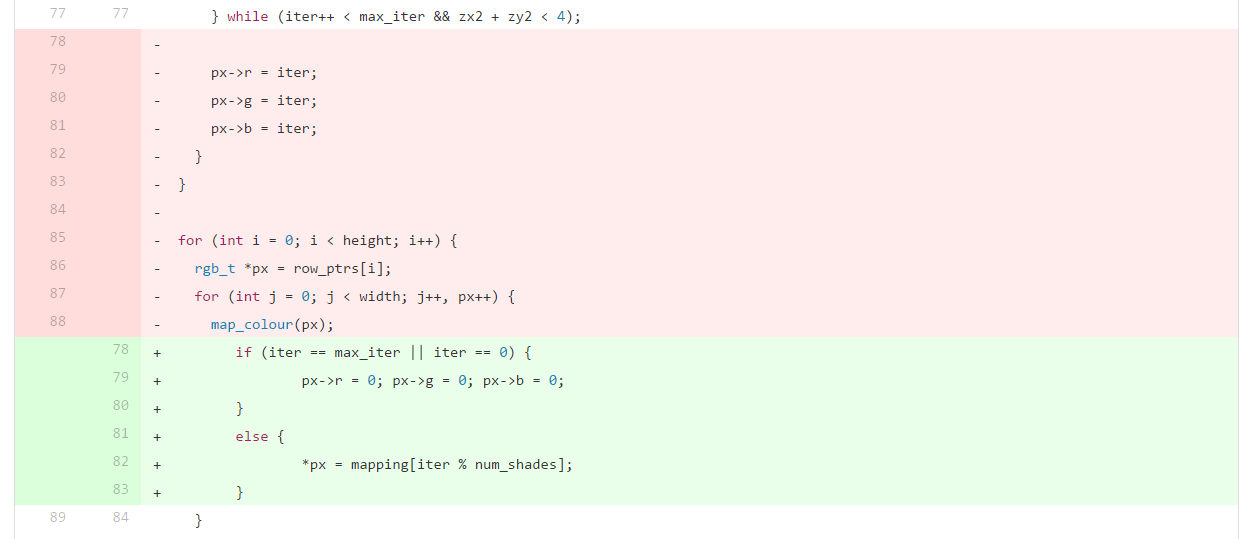


Figure 6. Default Code optimisation - Attempt 3

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.

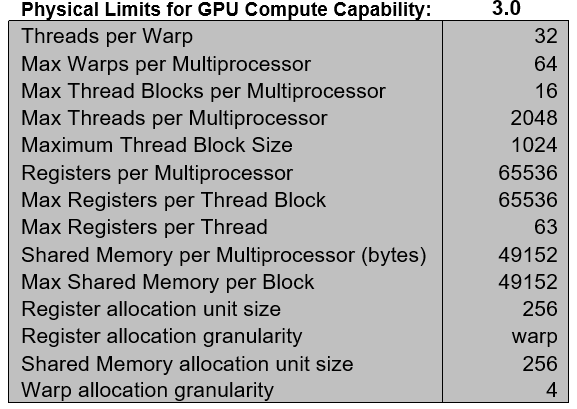
The necessary steps to convert the serial code into code running with the CUDA API are the following:

* 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. 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 theoretical occupancy of 1, for a 4096x4096 image, the following blockSize values are available:

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Blocks: | 16x16 | 8x8 | 4x4 | 2x2 | 1x1 |
| Threads: | 32 | 64 | 128 | 1024x1024 | 1024x1024 |

Arbitrarily, the most balanced values were used for the first execution of the code:

. . . . . . . . .

dim3 blockSize(256, 256);

dim3 gridSize(16, 16);

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 12 registers per thread.

## 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.

During measuring, device limits such as maximum grid size of 16 and maximum threads of 1024 will be ignored.

## 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 realPart imaginaryPart filename

* **width** is the width of the image in pixels
* **height** is the height of the image in pixels
* **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 following section will display 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, but irrelevant data (data collected using more than 16 blocks, more than 1024 threads, less than 32 threads where warp padding would be necessary) will be removed.

### 128x128

For the 128x128 set, CUDA suggested to use 1 block and 1024 threads. This option was not fastest though, as using 1 block of 128 threads showed a better execution time.

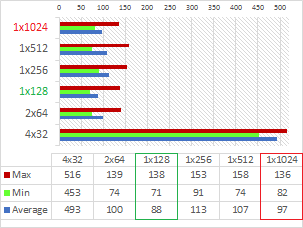
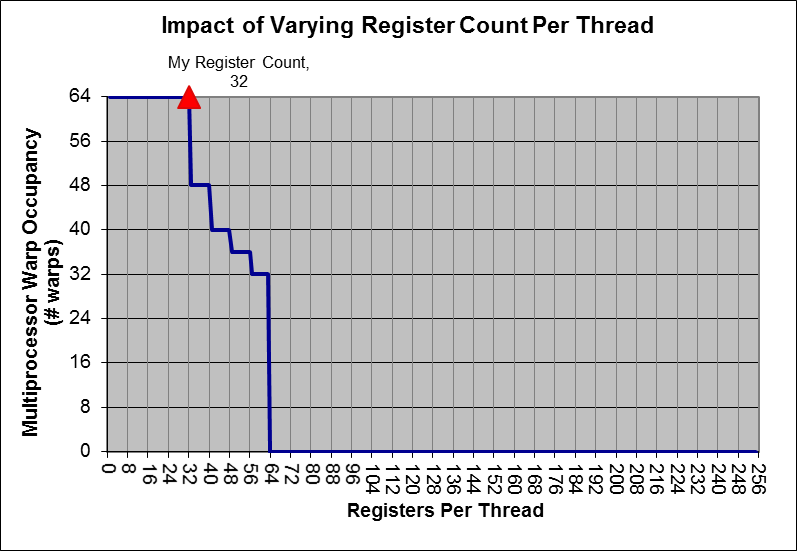
The only improvements that can be done for this kernel are related to the block size and shared memory, as the number of registers per thread is already optimised (32).

Figure 10. Execution time in microseconds for various kernel configurations

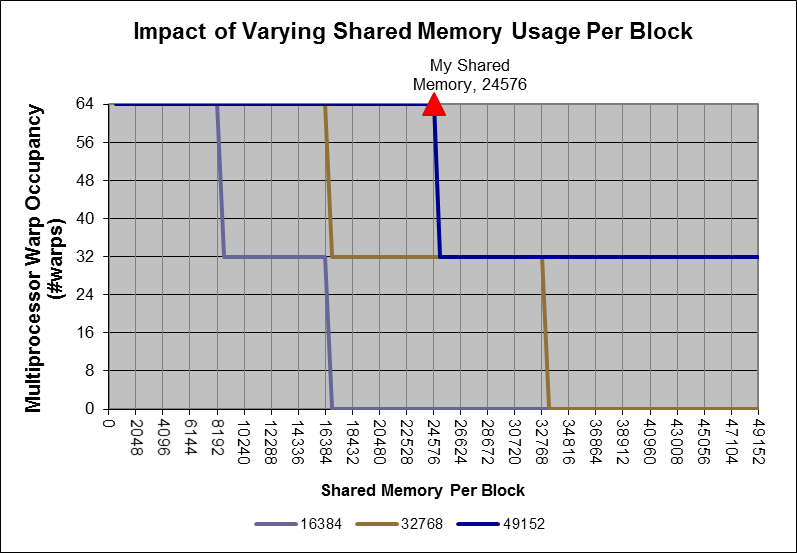
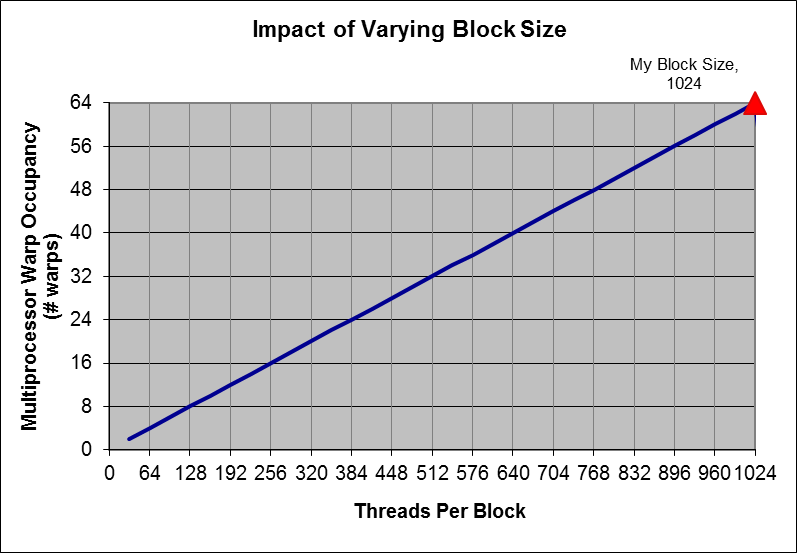
The best configuration for full occupancy is indeed the one suggested by the API, but the shared memory should be increased to 24576 bytes per block.

Figure 11. Optimizations available for 1x1024 kernel configurations

Other performant configurations are:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Blocks | Threads | Registers | Shared Memory | Occupancy |
| 1 | 128 | 32 | 3072 | 100% |
| 1 | 256 | 32 | 6144 | 100% |
| 1 | 512 | 32 | 12288 | 100% |
| 1 | 672 | 32 | 16384 | 98% |
| 1 | 1024 | 32 | 24576 | 100% |

the CUDA

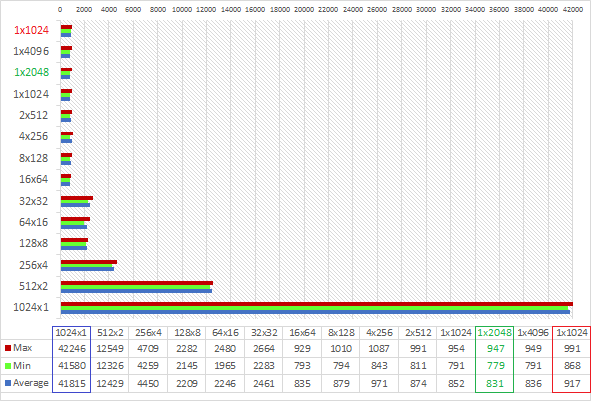
### 256x256

Figure 11. Execution time in microseconds for various kernel configurations for a 256x256 generated set

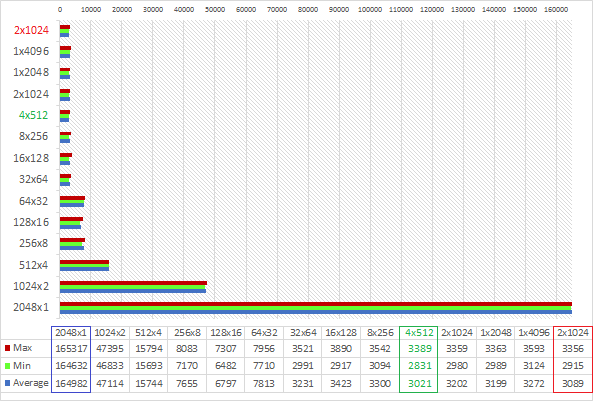
### 512x512

Figure 12. Execution time in microseconds for various kernel configurations for a 512x512 generated set

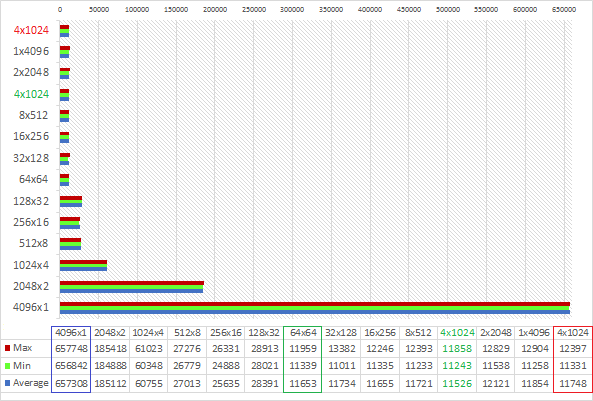
### 1024x1024

Figure 13. Execution time in microseconds for various kernel configurations for a 1024x1024 generated set

### 2048x2048

Figure 14. Execution time in microseconds for various kernel configurations for a 2048x2048 generated set 

### 4096x4096

Figure 15. Execution time in microseconds for various kernel configurations for a 2048x2048 generated set 

## Results processing

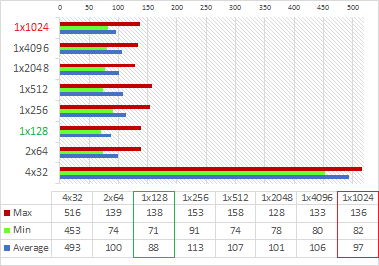
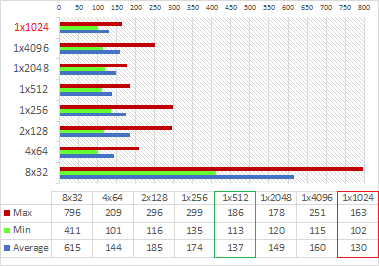
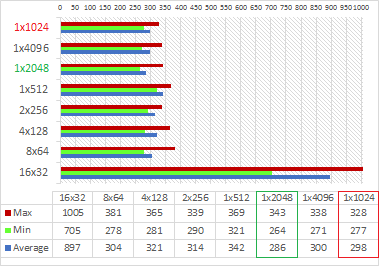
To visualise better the difference in charts, irrelevant data (data collected using more than 16 blocks, data collected using configurations where warp padding was necessary, and data generated using the same configuration as the suggested one) was removed.

Figure 16. Relevant data for 128x128 set on the left and 256x256 on the right



512x512

It is interesting to observe that the CUDA API did not always have the best average time. In fact, for most of the outputs, the API failed to provide the best performance.

After collecting all this data, it can be combined to create different graphs.

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.

# 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/>