Supplementary material: Mandelbrot set

cal-rmedina

The following document describes the code implementation and the most important funtions implemented in CUDA to generate the Mandelbrot set.

Output image

The output image size (`px`) is defined in the `main` function, the image size will be `width`×`height px` (Fig. 1), while the image dimensions can be chosen arbitrarily, it must not exceed the available **GPU memory** (global memory). To ensure this constraint, a function is included within the `main` function (`limits(width,height);`). If the output dimensions exceed the available memory, the code will terminate without completing the image computation. The screen will display suitable parameters for the available GPU. However, it is worth noting that modern GPUs generally provide sufficient memory to compute big images in size (> 2GB). The memory for the vector with size `size_vec = width*height*sizeof(int)`, used to compute each pixel color is explicitly allocated on the GPU using `cudaMalloc`. A modification on the way to allocate memory should be made in order to compute bigger images (> GPU memory), having memory accessible by both; the CPU and the GPU with `cudaMallocManaged` but that implementation is **out of the scope of this simple code**.

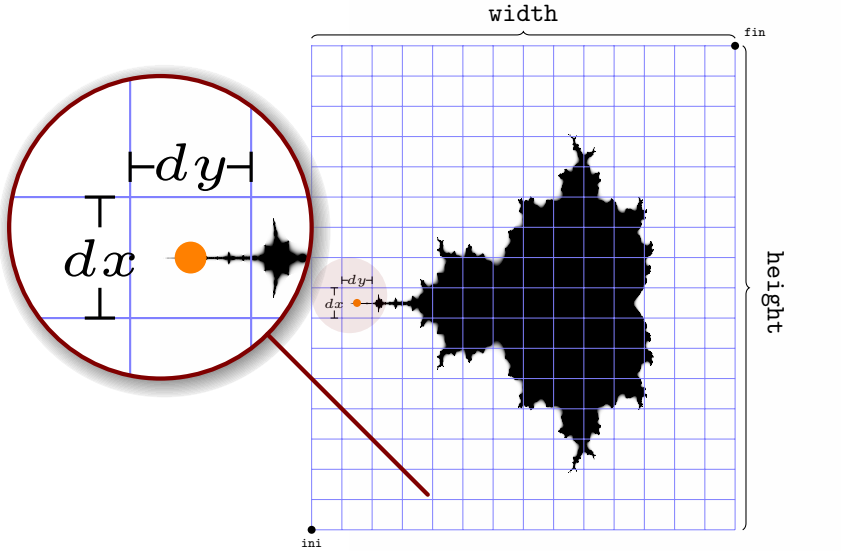


Figure 1: Squares represent `pixels` and the point is the complex number taken to compute the Mandelbrot set.

The output image will have the dimension of `width` × `height` pixels, the size of each `pixel` will be determined by the delimiting points of the rectangular area in the complex plane given by $dx = (x_{fin} - x_{ini})$ /`width` and $dy = (y_{fin} - y_{ini})$ /`height`. It can be seen (Fig. 1) that rectangular pixels are possible ($dx \neq dy$), but the value computed will be the center of the rectangular/square area.

2D Block arrange

The Mandelbrot set is situated in the complex plane, spanning from the point `ini` to `fin`, where the coordinates of `ini` and `fin` represent the lower and upper bounds of the rectangular region that contains the set respectively, to cover the entire image, a kernel with 2D `blocks` is launched. Each block contains a fixed number of threads: `thrX=16` and `thrY=16`, totaling 256 threads per block. The condition that the `width` and `height` must be multiples of `thrX` and `thrY` respectively ensures uniform distribution of threads across the entire image without leaving any gaps. With the thread size fixed (`thrX`,`thrY`= 2^4) it has been decided to use the bitwise left shift operation to set the `width` and `height` values, as an example for `width=1<<10` (`width`=2^{10}) and `height=1<<10` (`height`=2^{10}). For more detail of the block and thread division check Fig. 2.

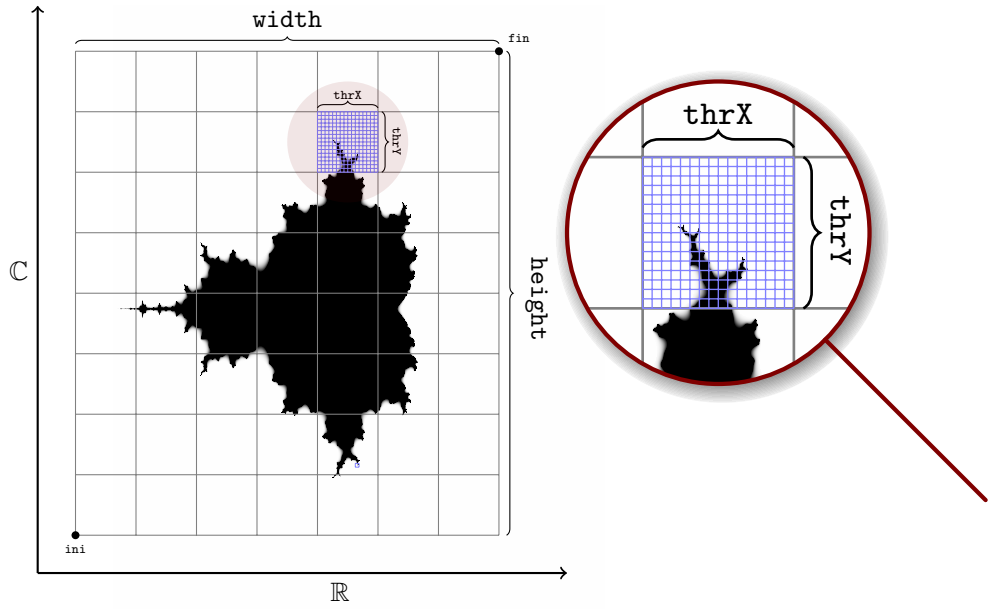
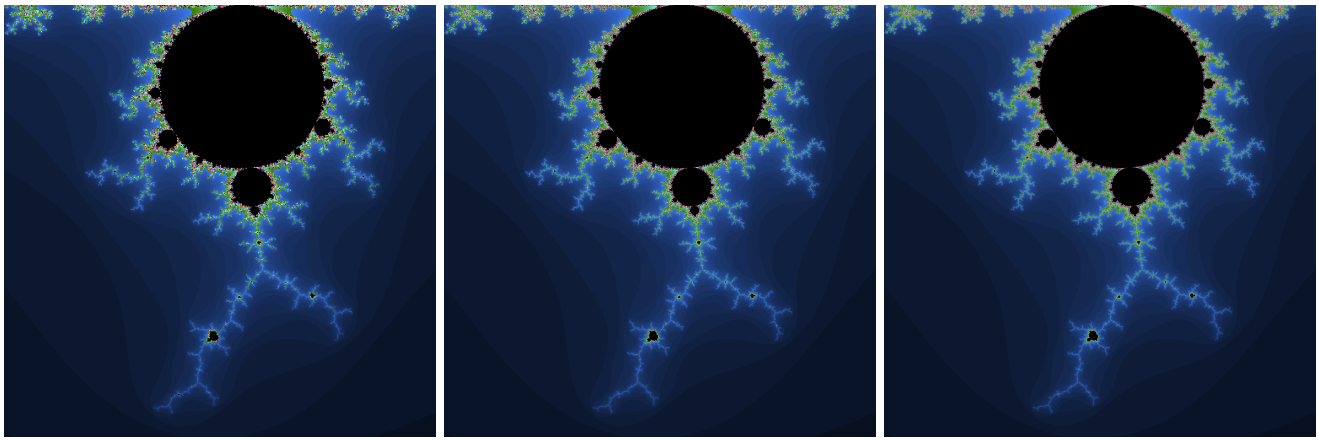


Figure 2: Within the grid, individual squares correspond to distinct objects, a gray square represents a `block`, while a blue square a `pixel`, rectangular arranges are also possible.

Increasing the number of pixels will lead to a more detailed output image but it also increases the size, a comparison for the same region increasing the number of pixels can be seen in Fig. 3, if a more detailed output image is desired or plotting a different section is needed, the points `ini` and `fin` can be changed to reduce/increase the area computed.



(a) 512×512 `px` (b) 2048×2048 `px` (c) 4096×4096 `px`

Figure 3: Images with square region increasing the number of pixels.