Numerical estimation of the area of the Mandelbrot set

Thorsten Förstemann

October 18, 2012

1 Introduction

1.1 Problem statement

An analytical expression of the area of the Mandelbrot set[1] is unknown. There are numerical estimations of the area[2] e.g. by Munfao (2012)[3]. The aim of this approach is to improve the accuracy of the numerical area estimation.

At the moment the most accurate estimation by Munfao uses a Monte Carlo or pixel counting integration method[4]. To increase the precision of the area estimate by a factor of 10 (i.e. one digit) the number of counted pixels must be increased by a factor of $100 = 10^2$. CPU time grows approximately proportional to the number of counted pixels. Thus, Monte Carlo estimations become inefficient at higher precisions.

1.2 Solution statement

Monte Carlo implementations are quite trivial. They can easily be parallelized and implemented using OpenCL[5]. Efficient OpenCL/GPU implementations can run up to 100 times faster on recent hardware compared to GPU implementations. A speedup by 100 yields one more digit precision at a given CPU time, as mentioned earlier.

[1] http://en.wikipedia.org/wiki/Mandelbrot_set
[2] http://oeis.org/A098403
[3] http://www.mrob.com/pub/muency/pixelcounting.html
[4] http://en.wikipedia.org/wiki/Monte_Carlo_method
[5] http://en.wikipedia.org/wiki/OpenCL

Table 2.1: Comparison of current Mandelbrot area estimates

Estimate & error	Details
Förstemann (2012)	
1.506 591 884 9	87 960 930 222 520 calculated pixels
0.000 000 002 8	3 028 136 sec. (35 days) CPU/GPU time
	29 047 880 pixel/sec
Munafo (2012)	
1.506 591 856	16 77 721 600 000 calculated pixels
0.000 000 025 6	691 255 sec. (8.00 days) CPU time
	2 427 066 pixel/sec
Förstemann (2011)	
1.506 591 881	35 046 933 135 360 calculated pixels
0.000 000 006 5	8 640 000 sec. (100 days) CPU/GPU time
	4 056 358 pixel/sec

Table 2.2: Parameters of grids calculated so far

grid size	area estimate	error	CPU [sec.]	av.pixel/s	iter.depth
16 384	1.506 590 913 0	0.000 002 171 7	955	5 621 685	67 108 864
32 768	1.506 591 383 0	0.000 000 678 8	2 539	8 457 990	134 217 728
65 536	1.506 592 037 0	0.000 000 251 4	8 952	9 595 548	268 435 456
131 072	1.506 591 911 7	0.000 000 135 8	23 979	14 329 096	536 870 912
262 144	1.506 591 918 4	0.000 000 040 0	58 394	23 536 486	1 073 741 824
524 288	1.506 591 883 3	0.000 000 021 7	194 027	28 333 985	2 147 483 648
1 048 576	1.506 591 883 1	0.000 000 007 3	745 509	29 496 938	4 294 967 296
2 097 152	1.506 591 884 9	0.000 000 002 8	3 028 136	29 047 880	8 589 934 592

An efficient OpenGL implementation is not trivial due to the chaotic behavior of the Mandelbrot set. The actual OpenCL/GPU implementation made here is about 10 times faster than Munafo's CPU based approach in 2012. The GPU-code runs in a Mathematica 8 environment.

2 Current estimate

The current estimates can be read off table 2.1. The estimate is based on the results listed in table 2.2 (refer to the last row). Table 2.2 is inspired from a similar table by Munafo[6].

The Monte Carlo method used here is based on quadratic grids covering the hole Mandelbrot set. The Method is described in more detail in the next section.

20 grids are calculated at each grid size. Details of the different grid sizes can be read off table 2:

[6] http://www.mrob.com/pub/muency/pixelcounting.html

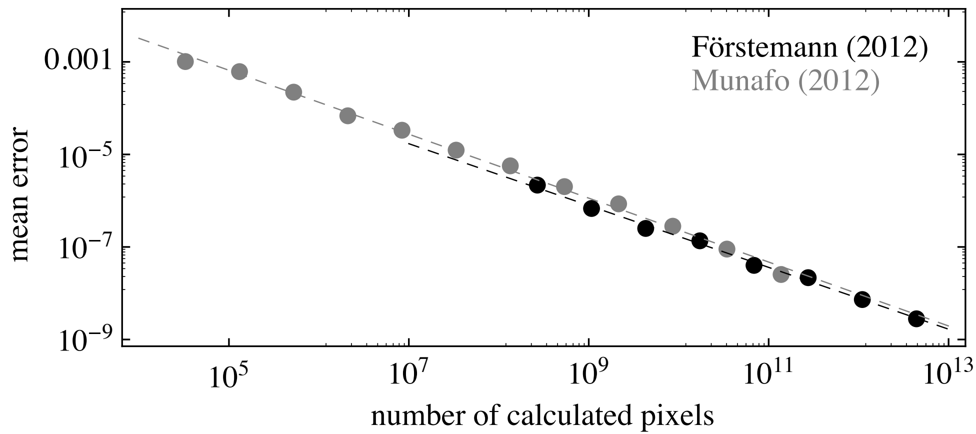

Figure 2.1: A comparison of the mean error versus number of calculated pixels between this and Munafo's approach. The dashed lines correspond to regressions $y \sim x^{-0.68}$.

Grid size The grid size is the number of grid points in both dimensions (quadratic grids).

Area estimate The area estimate and the 95% confidence interval are calculated using the 20 estimates each based on one grid. The grids differ slightly in grid width and offset. The grid widths and offsets are chosen randomly.

Error The 95% confidence interval of the mean: $\sqrt{1.96\sigma\,(20-1)}$ with standard deviation σ

CPU time The total GPU/CPU time in seconds needed to calculate the 20 grids

average pixel rate The number of calculated pixels per second on average

iteration depth The lower bound of the maximum iteration depth

A comparison of the mean error versus number of calculated pixels between this and Munafo's approach is given in figure 2.1. The numbers are taken from table 2.2 and the corresponding table by Munafo mentioned above.

The two main results are:

1. One additional digit of precision is gained by this approach in comparison to Munafo's approach (refer to table 2.1)
2. The actual pixel rate of this approach is about 10x faster than Munafo's approach (refer to table 2.1)

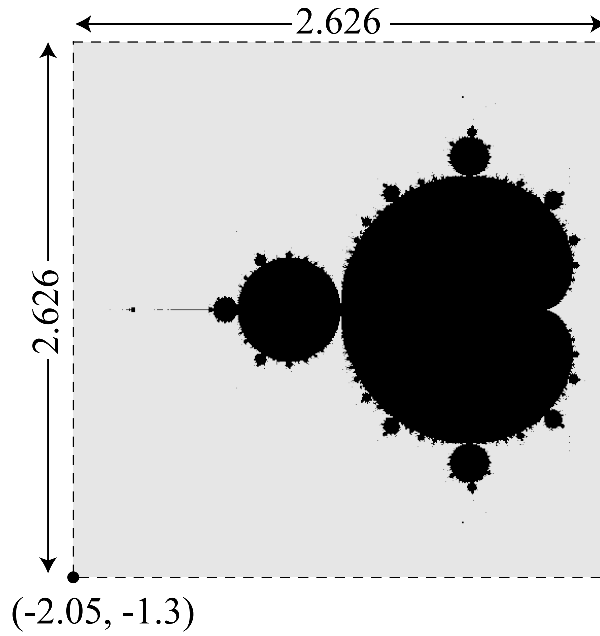

Figure 3.1: The grid for $\text{random}_1 = \text{random}_2 = \text{random}_3 = 0$ is marked gray. Typical values of the grid width are too small to identify individual grid points.

3 Method's details

In this section the used method based on Monte Carlo simulations is illustrated. The implementation with OpenCL and Mathematica is shown in the following sections.

3.1 Grid coordinates and dimensions

Complex numbers z in the complex pane are parametrized by two real coordinates x and y via $z = x + iy$. All calculations are done in double precision.

The Mandelbrot set is sampled by slightly varying grids. The variations are for each grid randomly chosen. The origins of the grids are given by

$$\begin{pmatrix} o_x \\ o_y \end{pmatrix} = \begin{pmatrix} -2.05 \\ -1.3 \end{pmatrix} + w \begin{pmatrix} \text{random}_1 \\ \text{random}_2 \end{pmatrix}$$

where random_1 and random_2 are random numbers between 0 and 1. The grid width is defined by

$$w = 2.6 \frac{1.01 + 0.01\ \text{random}_3}{N}$$

where N is the grid size. The random numbers random_1, random_2 and random_3 are arbitrary but constant for each grid. The coordinates of the (i, j)-th grid point are

$$\begin{pmatrix} x_{ij} \\ y_{ij} \end{pmatrix} = \begin{pmatrix} o_x \\ o_y \end{pmatrix} + w \begin{pmatrix} i \\ j \end{pmatrix}$$

where i and j go from 0 to $N - 1$. Used gird sizes can be read of table 2.2. A typical grid is shown in figure 3.1.

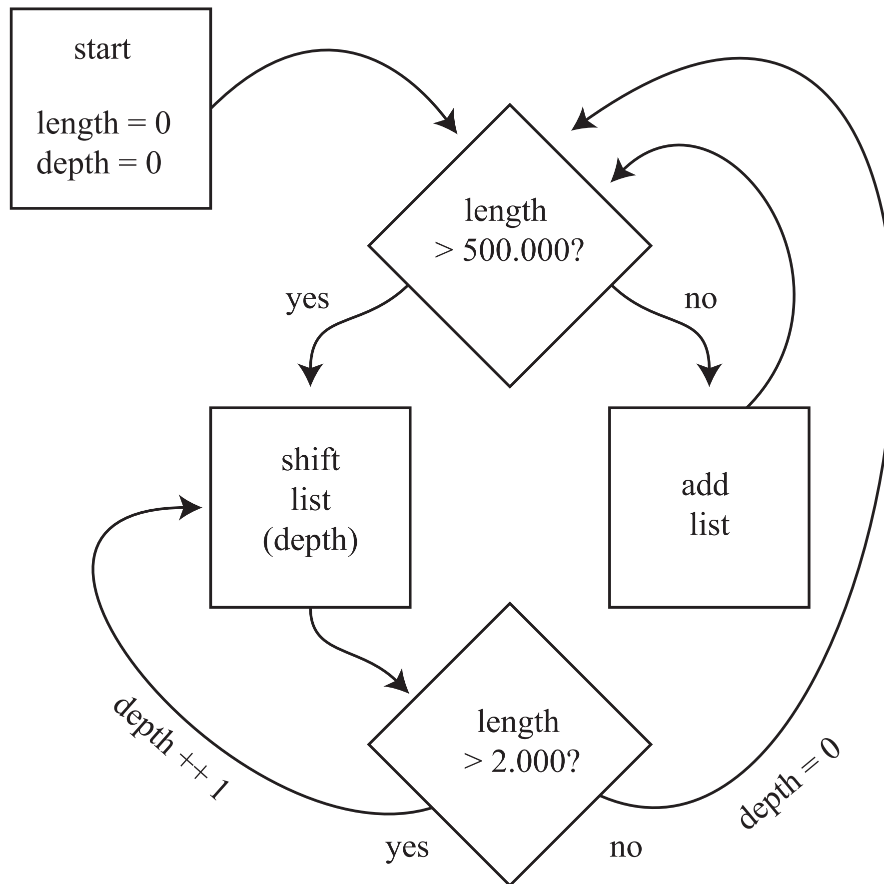

Figure 3.2: Pipeline handling

3.2 GPU-GRIDS AND PARALLELIZATION

Very large grids can not be efficiently handled by the GPU due to memory and timing issues. In this approach the GPU-grid size is set to $n = 1024$. Thus, the original grid has to be partitioned in $m \times m$ GPU-girds, where $N = 1024\,m$.

The calculations of the individual $m \times m$ GPU-grids can be parallelized. In Mathematica four worker kernels process the list of the GPU-girds in parallel. Each worker kernel is bound to an individual GPU. There are two graphic cards with two GPUs each.

3.3 PERFORMACE CONSIDERATIONS

The GPU-gird size is $n = 1024$. Thus, about 1 million pixels are simultaneously loaded into each GPU. By starting iterating on these points more and more points will diverge. Each GPU contains 1600 stream processors. After a certain number of iterations the number of remaining pixels will be smaller than 1600. Thus, the GPU load will decrease rapidly. It is quite sophisticated to keep the pipeline full, especially when dealing with very high iteration depths about 10^{10} and pixel rates about 30 million pixels per second.

3.4 MANAGING THE PIXEL PIPELINE

The pipeline is a list of pixel coordinates with accompanying iteration count and iteration variables (x and y). Each GPU has it's own pipeline. So, there are four pipelines.

In figure 3.2 the applied algorithm is outlined. At the beginning the pipeline is empty (i.e. length = 0) and the parameter depth is set to zero. The *addList* procedure appends a list of pre selected points of a GPU-grid to the pipeline. The details of the *addList* procedure are outlined in the next section. The procedure *addList* is applied until the length of the pipeline is greater than 500.000 points.

The *shiftList* procedure tries to apply 2^{depth} 10000 iterations on each pixel in the pipeline. The details of the *shiftList* procedure are outlined in the section after next. Pixels belonging to the Mandelbrot set and diverging pixels are removed from the pipeline. After appling *shiftList* the parameter depth is increased by one. Thus, more and more iterations are applied to less and less pixels. This balances CPU and GPU load and helps to detect orbits with very long periods. When the length of the pipeline is smaller than 2000 pixels depth is set to zero and again *addList* its applied to the pipeline.

Very high iteration depths can be reached by recycling the less than 2000 non member and non diverging pixels in the next cycle of the algorithm.

3.5 THE ADDLIST PROCEDURE

The pseudocode of the *addList* procedure reads as follows:

```
1 create_GPU_grid()
2 iter(990)
3 test_circle_cardioid()
4 for (n=1;n<4)
5   iter(2^n * 990)
6   orbit(2^n * 1000)
```

The specific definitions of the sub procedures are:

create_GPU_grid(): This sub routine creates an GPU grid with $1024^2 \approx 1$million pixels following the procedure described in section 3.1.

iter (n): This sub routine applies n Mandelbrot iterations to each point of the GPU-grid.

test_circle_cardioid (n): This sub routine tests any point whether it is part of the Mandelbrot set's cardioid or circle.

orbit (n): This sub routine applies n Mandelbrot iterations to each point of the GPU-grid and checks simultaneously for iteration orbits.

Typically the about 1 million pixels divide into the following parts: 20.9% belong to the Mandelbrot set, 77.6% diverge and 1.5% remain unknown. The on average 15000 unknown pixels are appended to the pipeline. $990 + 2 * (990 + 1000) + 4 * (990 + 1000) + 8 * (990 + 1000) = 990 + 14 * 1990 = 28850$ iterations are applied to unknown pixels. This procedure needs about 10 milliseconds GPU and about 15 milliseconds CPU time on average.

Table 3.1: Numbers of remaining pixels for different grids after post processing the pipeline and corresponding errors of the estimated area

grid size	area estimate	error (mean)	iter.depth	remain	error (remain)
16 384	1.506 590 913 0	0.000 002 171 7	67 108 864	520	0.000 000 675 5
32 768	1.506 591 383 0	0.000 000 678 8	134 217 728	1 000	0.000 000 324 8
65 536	1.506 592 037 0	0.000 000 251 4	268 435 456	1 889	0.000 000 153 1
131 072	1.506 591 911 7	0.000 000 135 8	536 870 912	3 571	0.000 000 072 3
262 144	1.506 591 918 4	0.000 000 040 0	1 073 741 824	5 711	0.000 000 029 0
524 288	1.506 591 883 3	0.000 000 021 7	2 147 483 648	8 091	0.000 000 010 2
1 048 576	1.506 591 883 1	0.000 000 007 3	4 294 967 296	8 261	0.000 000 002 6
2 097 152	1.506 591 884 9	0.000 000 002 8	8 589 934 592	9 922	0.000 000 000 8

3.6 The shiftList procedure

The pseudocode of the *shiftList* procedure reads as follows:

```
1 for (n=0;n<depth+1)
2   iter(2^n * 9900)
3   orbit(2^n * 10000)
```

The specific definitions of the sub procedures are the same as for the *addList* procedure.

Following figure 3.2 the shiftList procedure is applied repeatedly to the pipeline. Each time the parameter depth is increased by one. Depth reaches up to 16. This procedure needs about 100 milliseconds GPU and about 10 milliseconds CPU time on average.

3.7 Post processing of the pipeline

As mentioned in section 3.2 there are $m \times m$ GPU-grids. At a certain time all GPU-grids are depleted and integrated into the four pipelines. Then the algorithm mentioned in section 3.4 ends with 4×2000 remaining pixels. A usual CPU based iteration algorithm with orbit detection is applied to these remaining 8000 pixels. The chosen maximum iterations are listed in table 2.2 and 3.1. The maximum iterations are chosen to balance GPU and CPU load and, last but not least, to keep the estimation error of the area due to the remaining pixels smaller than the error due to the sampling of the 20 grids.

The numbers of remaining pixels and the corresponding errors of the estimated area are listed in table 3.1.

4 Implementation

4.1 Hardware configuration

A typical PC (refer to figure 4.1 is used. Specific details of the hardware configuration are:

- CPU: Intel Core i7 2600K

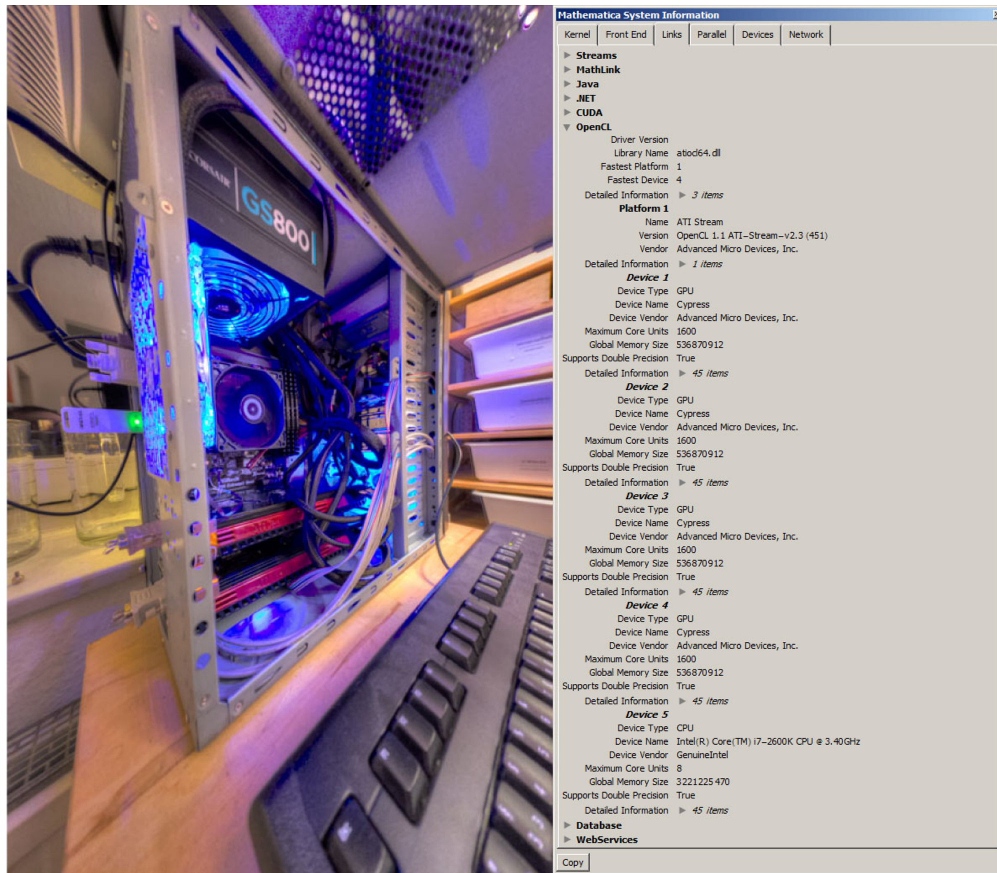

Figure 4.1: Left side: the computer with two graphic cards. Right side: Mathematica's system information

- GPU: 2x Radeon HD 5970. Thus, there are 4 GPUs with 1600 stream processors each (refer to figure 3).
- Power consumption under load: approx. 350 W
- Dummy VGA dongle (refer to e.g. `http://www.vt-tech.info/?p=8379`)

4.2 Software configuration

Specific details of the software configuration are:

- Mathematica 8.0.4.0, Windows 7
- ATI driver Catalyst 11.2 with AMD Stream SDK 2.3 (refer to `http://forums.wolfram.com/mathgroup/archive/2011/Jun/msg00615.html`)
- Adjustment of TDR timeout parameter in Windows 7's registry (refer to e.g. `http://www.cmsoft.com.br/index.php?option=com_content&view=category&layout=blog&id=58&Itemid=95`)

- Installation of a C-compiler for Mathematica (here: Visual Studio 2011 following Mathematica's manual)

4.3 Soucre code

4.3.1 Initialisation

This Mathematica code initializes the calculation and creates the working files.

```
path = NotebookDirectory[];
SetDirectory[path];
kernelString = "kernel" <> ToString[$KernelID];

(* Parameters *)

sizeSample = 20; (* number of rasters to be calculated *)

sizeCPU = 2^4;     (* raster = sizeCPU x sizeCPU tiles handled by CPU *)

sizeGPU = 2^11;(* tile = sizeGPU x size GPU pixels *)

(* Init data *)
tmp = Table[{
     "in_" <>
      ToString[
       AccountingForm[n, {3, 0}, NumberPadding -> {"0", ""},
        NumberPoint -> ""]] <> ".txt",
     {{{}, {}, {}}, {{RandomReal[], RandomReal[],
        RandomReal[]}, {sizeCPU, sizeGPU}}, {0, 0}, 0, 0}
     }, {n, 1, sizeSample}];

(* Create init files *)
Map[Put[#[[2]], #[[1]]] &, tmp];
```

4.3.2 GPU-code for the addList procedure

This is the GPU code of the *addList* procedure. The code is stored as a simple string.

```
sourceBitmap="
#ifdef USING_DOUBLE_PRECISIONQ
#pragma OPENCL EXTENSION cl_amd_fp64 : enable
#endif /* USING_DOUBLE_PRECISIONQ */
__kernel void mandelbrot_kernel(
  __global mint * set, /* housekeeping: data pointer */
  double offx,         /* raster's x offset */
  double offy,         /* raster's y offset */
  double step,         /* raster's grid width */
  mint size)           /* housekeeping: data size */
{
  int xIndex = get_global_id(0);
  int yIndex = get_global_id(1);
  int iter;
  int initMaxiter = 990;    /* initial value of warm up maximum iteration
      depth (empiric) */
```

```
16   int orbitMaxiter = 1000;  /* initial value maximum iteration depth with
        orbit test (empiric) */
17   int maxDoubling = 3;     /* number of maximum iteration's doublings (empiric
        ) */
18   int doubling;
19   double x0 = offx + step * xIndex;
20   double y0 = offy + step * yIndex;
21   double tmp, q, x = 0.0, y = 0.0;
22   double ox, oy;
23   bool card, circ;
24   bool member = false;
25   bool refuge = false;
26
27   /* housekeeping: check index */
28   if (xIndex < size && yIndex < size) {
29     /* warm up iterations */
30     for (iter = 0; (x*x+y*y <= 4.0) && (iter < initMaxiter); iter++) {
31             tmp = x*x - y*y + x0;
32             y = 2.0*x*y + y0;
33             x = tmp;
34         }
35     refuge = (x*x+y*y > 4); /* Escape? */
36     /* Circle and cardioide test */
37     if (!refuge) {
38       q = pow((x0 - 0.25), 2) + y0 * y0;
39       card = q * (q + (x0 - 0.25)) < 0.25 * y0 * y0;
40       circ = pow((x0 + 1.0), 2) + y0 * y0 < 0.0625;
41       member = (card || circ);
42     }
43     /* Maxiter doubling loop */
44     for (doubling = 1; !refuge && !member && doubling<maxDoubling; doubling++)
          {
45       /* normal iteration, maxiter: 2^doubling * initMaxiter */
46       for (iter = iter;
47         (x*x+y*y <= 4.0) && (iter < pow(2.0f,doubling)*initMaxiter);
48         iter++) {
49         tmp = x*x - y*y + x0;
50         y = 2.0*x*y + y0;
51         x = tmp;
52       }
53       refuge = (x*x+y*y > 4); /* Escape? */
54       /* iteration with orbit test, maxiter: 2^doubling * orbitMaxiter */
55       if (!refuge) {
56         ox = x; /* set real part for orbit test */
57         oy = y; /* set imaginary part for orbit test */
58         tmp = x*x - y*y + x0;
59         y = 2.0*x*y + y0;
60         x = tmp;
61         for (iter = iter;
62           (x*x+y*y <= 4.0) && (iter < pow(2.0f,doubling)*orbitMaxiter) &&
63           ((ox != x) || (oy != y)); /* non orbit test */
64           iter++) {
65           tmp = x*x - y*y + x0;
66           y = 2.0*x*y + y0;
67           x = tmp;
```

```
68          }
69          refuge = (x*x+y*y > 4); /* Escape? */
70          member = ((ox == x) && (oy == y)); /* Orbit? */
71        }
72      }
73      if (!refuge && !member) {
74        set[(yIndex + xIndex * size)] = 255; /* mark point as unknown */
75      }
76      if (member) {
77        set[(yIndex + xIndex * size)] = 128; /* mark point as member */
78      }
79      if (refuge) {
80        set[(yIndex + xIndex * size)] = 0; /* mark point as refugee */
81      }
82      }}";
```

4.3.3 GPU-CODE FOR THE SHIFTLIST PROCEDURE

This is the GPU code of the *shiftList* procedure. The code is again stored as a simple string.

```
1  sourceList="
2  #ifdef USING_DOUBLE_PRECISIONQ
3  #pragma OPENCL EXTENSION cl_amd_fp64 : enable
4  #endif /* USING_DOUBLE_PRECISIONQ */
5  __kernel void mandelbrot_kernel(
6    __global double * set, /* housekeeping: data pointer */
7    mint initMaxiter,      /* maxiter without orbit test */
8    mint orbitMaxiter,     /* maxiter with orbit test */
9    mint size)             /* housekeeping: data size */
10 {
11   int xIndex = get_global_id(0);
12   int iter;
13   double tmp;
14   double cx, cy, x, y, ox, oy;
15   bool refuge = false;
16   bool member = false;
17 
18   /* housekeeping: check index */
19   if (xIndex < size) {
20     cx = set[4*(xIndex)];    /* collecting real part of point coordinates */
21     cy = set[4*(xIndex)+1];  /* collecting imaginary part of point coordinates
         */
22     x = set[4*(xIndex)+2];   /* collecting real part of actual iteration value
         */
23     y = set[4*(xIndex)+3];   /* collecting imaginary part of actual iteration
         value */
24     /* normal iteration, maxiter: initMaxiter */
25     for (iter = 0;
26       (x*x+y*y <= 4.0) && (iter < initMaxiter);
27       iter++) {
28             tmp = x*x - y*y + cx;
29             y = 2.0*x*y + cy;
30             x = tmp; }
31     refuge = (x*x+y*y > 4.0); /* Escape? */
```

```
32      /* iteration with orbit test, maxiter: orbitMaxiter */
33      if (!refuge) {
34        ox = x; /* set real part for orbit test */
35        oy = y; /* set imaginary part for orbit test */
36        tmp = x*x - y*y + cx;
37        y = 2.0*x*y + cy;
38        x = tmp;
39        for (iter = iter;
40          (x*x+y*y <= 4.0) && (iter < orbitMaxiter) &&
41          ((ox != x) || (oy != y)); /* non orbit test */
42          iter++) {
43          tmp = x*x - y*y + cx;
44          y = 2.0*x*y + cy;
45          x = tmp;}
46        refuge = (x*x+y*y > 4); /* Escape? */
47        member = ((ox == x) && (oy == y)); /* Orbit? */}
48      if (refuge) {
49        /* mark point as refugee */
50        set[4*(xIndex)] = 0.0;
51        set[4*(xIndex)+1] = 0.0;
52        set[4*(xIndex)+2] = 0.0;
53        set[4*(xIndex)+3] = 0.0;}
54      if (member) {
55        /* mark point as member */
56        set[4*(xIndex)] = 1.0;
57        set[4*(xIndex)+1] = 1.0;
58        set[4*(xIndex)+2] = 1.0;
59        set[4*(xIndex)+3] = 1.0;}
60      if (!refuge && !member) {
61        /* mark point as unknown */
62        set[4*(xIndex)] = cx;
63        set[4*(xIndex)+1] = cy;
64        set[4*(xIndex)+2] = x;
65        set[4*(xIndex)+3] = y;}
66      }}";
```

4.3.4 Mathematica functions for the GPU part

This Mathematica code compiles and loads the GPU functions defined above. The Mathematica functions *addList* and *shiftList* are defined here.

```
1  (* wake up device *)
2  mem=OpenCLMemoryAllocate[Integer,{1024,1024},"Platform"->1,"Device"->$KernelID
      ];
3  OpenCLMemoryUnload[mem];
4
5  (* Load GPU functions *)
6  clBitmap=OpenCLFunctionLoad[
7    sourceBitmap,"mandelbrot_kernel",
8    {{_Integer},"Double","Double","Double",_Integer},
9    {16,16},
10   "ShellOutputFunction"->Print,"ShellCommandFunction"->Print
11 ];
12 clList=OpenCLFunctionLoad[
```

```
13    sourceList,"mandelbrot_kernel",
14    {{"Double"},_Integer,_Integer,_Integer},
15    64,
16    "ShellOutputFunction"->Print,"ShellCommandFunction"->Print];
17
18 (* Helper function for addList, from point index to point coordinates *)
19 gpuRaster[{{rndX_,rndY_,rndW_},{cpuSize_,gpuSize_},idx_}]:=Block[
20    {cpuOffs,cpuWidth,gpuOffsX,gpuOffsY,gpuWidth},
21    If[idx<0||idx>cpuSize^2,Abort[]];
22    cpuOffs={-2.05,-1.3}; (* left bottom *)
23    cpuWidth=2.6*(1.01+0.01*rndW)/cpuSize; (* raster size with random variation
        *)
24    gpuWidth=cpuWidth/gpuSize;
25    {gpuOffsX,gpuOffsY}=cpuOffs-{rndX,rndY}*gpuWidth+{Quotient[idx,cpuSize],Mod[
        idx,cpuSize]}*cpuWidth;
26    {gpuOffsX,gpuOffsY,gpuWidth,gpuSize}
27 ];
28
29 (* fill up work list *)
30 addList[{
31    {fullList_,doneList_,workList_},
32    {{rndX_,rndY_,rndW_},{cpuSize_,gpuSize_}},
33    time_,kernel_,depth_}]:=Block[
34    {idx,ox,oy,step,size,mem,res,out,img,erg,lst,timer,timers},
35    timers={0,0};timer=AbsoluteTime[];
36    idx=Length[doneList]+Length[fullList];
37    If[idx>cpuSize^2/4-1,
38      Return[
39        {{doneList,workList,timers},
40        {{rndX,rndY,rndW},{cpuSize,gpuSize}},
41        time+timers,kernel}]];
42    {ox,oy,step,size}=gpuRaster[{{rndX,rndY,rndW},{cpuSize,gpuSize},4*idx+(
        kernel-1)}];
43    timers=timers+{AbsoluteTime[]-timer,0};timer=AbsoluteTime[];
44    mem=OpenCLMemoryAllocate[Integer,{size,size}];
45    res=clBitmap[mem,ox,oy,step,size];
46    out=OpenCLMemoryGet[First[res]];OpenCLMemoryUnload[mem];
47    timers=timers+{0,AbsoluteTime[]-timer};timer=AbsoluteTime[];
48    img=Image[out,"Byte"];
49    erg=ImageLevels[img,3][[{1,2},2]];
50    lst=ImageData[Binarize[img,0.9]];
51    lst=ArrayRules[SparseArray[lst]][[;;-2,1]];
52    lst=Map[{4*idx+(kernel-1),{ox,oy}+(#-1)*step,{0.0,0.0},0}&,lst];
53    timers=timers+{AbsoluteTime[]-timer,0};timer=AbsoluteTime[];
54    {{fullList,Append[doneList,{4*idx+(kernel-1),erg}],Join[workList,lst]} ,{{
        rndX,rndY,rndW},{cpuSize,gpuSize}},time+timers,kernel,depth}
55 ];
56
57 (* Helper function for shiftList, extract positions *)
58 positionHelper[{doneList_,workList_},bothList_,pattern_]:=Block[
59    {extractedList},
60    extractedList=Extract[workList,Position[bothList,pattern]];
61    extractedList=extractedList[[All,{1,2,3}]];
62    extractedList=Split[Sort[extractedList],#1[[1]]==#2[[1]]&];
63    extractedList=Map[{#[[1,1]],Length[#]}&,extractedList];
```

```
64    extractedList=Map[{Position[doneList[[All,1]],#[[1]]][[1,1]],#[[2]]}&,
          extractedList];
65    extractedList
66 ];
67
68 (* die out work list *)
69 shiftList[{
70 {fullList_,doneList_,workList_},
71 {{rndX_,rndY_,rndW_},{cpuSize_,gpuSize_}},
72 time_,kernel_,depth_}]:=Block[
73    {bothList,refugeList,memberList,unknownList,doneOut,workOut,fullOut,timer,
          timers},
74    timers={0,0};timer=AbsoluteTime[];
75    doneOut=doneList;
76    bothList=Flatten[Map[{#[[2,1]],#[[2,2]],#[[3,1]],#[[3,2]]}&,workList]];
77    timers=timers+{AbsoluteTime[]-timer,0};timer=AbsoluteTime[];
78    bothList=clList[bothList,2^depth*9900,2^depth*10000,Length[bothList]/4];
79    timers=timers+{0,AbsoluteTime[]-timer};timer=AbsoluteTime[];
80    bothList=Partition[First[bothList],4];
81    refugeList=positionHelper[{doneList,workList},bothList,{0.,0.,0.,0.}];
82    Do[doneOut[[refugeList[[i,1]],2,1]]=doneList[[refugeList[[i,1]],2,1]]+
          refugeList[[i,2]],{i,Length[refugeList]}];
83    memberList=positionHelper[{doneList,workList},bothList,{1.,1.,1.,1.}];
84    Do[doneOut[[memberList[[i,1]],2,2]]=doneList[[memberList[[i,1]],2,2]]+
          memberList[[i,2]],{i,Length[memberList]}];
85    unknownList=Position[bothList,Except[{0.,0.,0.,0.}|{1.,1.,1.,1.}],{1}];
86    unknownList=Transpose[{Drop[Extract[workList,unknownList],1],Drop[Extract[
          bothList,unknownList],1]}];
87    workOut=Map[{#[[1,1]],{#[[2,1]],#[[2,2]]},{#[[2,3]],#[[2,4]]},#[[1,4]]+depth
          }&,unknownList];
88    fullOut=Join[fullList,Select[doneOut,Not[gpuSize^2-Total[#[[2]]]>0]&]];
89    doneOut=Select[doneOut,gpuSize^2-Total[#[[2]]]>0&];
90    timers=timers+{AbsoluteTime[]-timer,0};timer=AbsoluteTime[];
91    {{fullOut,doneOut,workOut},{{rndX,rndY,rndW},{cpuSize,gpuSize}},time+timers,
          kernel,depth+1}
92 ];
93 ];
```

4.3.5 Mathematica code for the CPU part

This Mathematica code represents the main loop.

```
1  path=NotebookDirectory[];
2  SetDirectory[path];
3  files=FileNames["in_*"];
4  (* Main loop *)
5  While[files!={},
6     allTime=AbsoluteTime[];
7     file=First[files];
8     set=Get[file];
9     upperLimit=500000;lowerLimit=2000;finalLimit=20; (* list dimensions *)
10    DistributeDefinitions[set,upperLimit,lowerLimit,finalLimit,path,file];
11    (* Begin parallel evaluation *)
12    ParallelEvaluate[
```

```
13   SetDirectory[path];
14   (* init working list *)
15   {{fullList,doneList,workList},{{rndX,rndY,rndW},{cpuSize,gpuSize}},timer,
       kernel,depth}=set;
16   set={{fullList,doneList,workList},{{rndX,rndY,rndW},{cpuSize,gpuSize
       }},{0,0},$KernelID,0};
17   (* CPU raster loop *)
18   While[4*(Length[fullList]+Length[doneList])<cpuSize^2,
19     (* fill up working list *)
20     set=NestWhile[addList,set,Length[#[[1,1]]]+Length
         [#[[1,2]]]<#[[2,2,1]]^2/4&&Length[#[[1,3]]]<upperLimit&];
21     (* die out working list *)
22     set=NestWhile[shiftList,set,Length[#[[1,3]]]>lowerLimit&];
23     (* update working list *)
24     {{fullList,doneList,workList},{{rndX,rndY,rndW},{cpuSize,gpuSize}},timer
         ,kernel,depth}=set;
25     (* print work list status *)
26     Print[Grid[{{
27       DateString[AbsoluteTime[],{"Day",":","Time"}],
28       ToString[Round[(Length[fullList]+Length[doneList])/cpuSize
           ^2*4*100,0.1]]<>"%",
29       ToString[Length[fullList]],
30       ToString[Length[doneList]],
31       ToString[Length[workList]],
32       ToString[Round[timer[[1]],0.1]]<>" s",
33       ToString[Round[timer[[2]],0.1]]<>" s",
34       ToString[depth]
35     }},Frame->All,Alignment->Right,ItemSize->All,Background->Which[
36     kernel==1,LightRed,kernel==2,LightGreen,kernel==3,LightBlue,kernel==4,
         LightGray]]];
37     (* update working list with resetted times *)
38     set={{fullList,doneList,workList},{{rndX,rndY,rndW},{cpuSize,gpuSize
         }},{0,0},kernel,0};
39   ];
40   (* write work list to kernel file *)
41   Put[set,StringReplace[file,{"in"->"out",".txt"->""}]<>"_"<>ToString[
       $KernelID]<>".txt"];
42 ];
43 (* End parallel evaluation *)
44 (* print work list final status *)
45 Print[Grid[{{
46   DateString[AbsoluteTime[],{"Day",":","Time"}],
47   file,
48   ToString[IntegerPart[AbsoluteTime[]-allTime]]<>" s"
49 }},Frame->All,Alignment->Right,ItemSize->All]];
50 (* Read 4 kernel files *)
51 tmp=Map[Get,Table[StringReplace[file,{"in"->"out",".txt"->""}]<>"_"<>
     ToString[i]<>".txt",{i,4}]];
52 tmp=Transpose[tmp[[All,1]]];
53 tmp=Apply[Join,tmp,1];
54 cpu=tmp[[3,All,{2,3}]];
55 cpu=Map[#[[1]]+I*#[[2]]&,cpu,{2}];
56 summ=Total[tmp[[1,All,2]]]+Total[tmp[[2,All,2]]];
57 (* Write result files *)
58 Put[{AbsoluteTime[]-allTime,summ,cpu},StringReplace[file,{"in"->"out",".txt"
```

```
    ->""}]<>"_5"<>".txt"];
59   (* Delete input file, mark as done *)
60   DeleteFile[file];
61   files=FileNames["in_*"];
62 ];
```

4.3.6 MATHEMATICA CODE FOR THE CPU POST PROCESSING

This is the code for the post processing of the pipeline. This code runs entirely on the CPU and in parallel to the main loop above.

```
1  cPostIter =
2    Compile[{{zz, _Complex, 1}},
3     Module[{n = 0, nn = 0, c = zz[[1]], z = zz[[2]], z0 = 0. + 0. I,
4       maxi0 = 2^15, maxi1 = 2^16, refuge = False, member = False,
5       istep = 0},
6      While[istep < 6 + 10 && ! refuge && ! member,
7       nn = 0;
8       While[nn < 2^istep && ! refuge ,
9        n = 0;
10       While[n < maxi0 && ! refuge ,
11        z = z^2 + c;
12        refuge = Re[z]^2 + Im[z]^2 > 4;
13        n++];
14       nn++];
15      If[! refuge,
16       z0 = z;
17       z = z^2 + c;
18       nn = 0;
19       While[nn < 2^istep && ! refuge && ! member ,
20        n = 0;
21        While[n < maxi1 && ! refuge && ! member ,
22         z = z^2 + c;
23         refuge = Re[z]^2 + Im[z]^2 > 4;
24         member = (Re[z0] == Re[z]) && (Im[z0] == Im[z]);
25         n++];
26        nn++];
27       ];
28      istep++;
29      ];
30     Which[refuge, {0, istep, n, nn}, member, {1, istep, n, nn},
31      True, {zz[[1]], istep, n, nn}]
32     ], RuntimeAttributes -> {Listable}, Parallelization -> True,
33    CompilationTarget -> "C", RuntimeOptions -> "Speed"];
34
35 While[True,
36   path = NotebookDirectory[];
37   SetDirectory[path];
38   files5 = FileNames["out_*_5.txt"];
39   files6 = FileNames["out_*_6.txt"];
40   files5 = Map[StringReplace[#, "_5.txt" -> ""] &, files5];
41   files6 = Map[StringReplace[#, "_6.txt" -> ""] &, files6];
42   files = Complement[files5, files6];
43   If[files == {}, Pause[10],
```

```
44    allTime = AbsoluteTime[];
45    file = First[files];
46    data = Get[file <> "_5.txt"];
47    cpu = Chop[cPostIter[data[[3]]]];
48    Put[cpu, file <> "_6" <> ".txt"];
49    summ =
50     data[[2]] + {Length[Select[cpu, #[[1]] == 0. &]],
51       Length[Select[cpu, #[[1]] == 1. &]]};
52    left =
53     Length[cpu] - Length[Select[cpu, #[[1]] == 0. &]] -
54      Length[Select[cpu, #[[1]] == 1. &]];
55    rnd = Get[file <> "_1" <> ".txt"][[2, 1, 3]];
56    area = (summ[[2]] + left)/(Total[summ] +
57         left)*(2.6*(1.01 + 0.01*rnd))^2;
58    Put[{area, summ[[1]], summ[[2]], left, rnd,
59      IntegerPart[AbsoluteTime[] - allTime]}, file <> "_7" <> ".txt"];
60    Print[Grid[{{
61        DateString[AbsoluteTime[], {"Day", ":", "Time"}],
62        file,
63        ToString[
64         AccountingForm[area, {8, 8}, DigitBlock -> 3,
65          NumberPadding -> {"", "0"}]],
66        ToString[summ[[1]]],
67        ToString[summ[[2]]],
68        ToString[left],
69        rnd,
70        ToString[IntegerPart[AbsoluteTime[] - allTime]] <> " s"
71        }}, Frame -> All, Alignment -> Right, ItemSize -> All]];
72    ];
73   ];
```