Numerical estimation of the area of the Mandelbrot set using Monte Carlo Integration

T. Förstemann[*]

(Dated: November 1, 2016)

An analytical expression of the area of the Mandelbrot set is unknown. So far, the most accurate numerical estimation of the area by R.P. Munafo (2012) based on Monte Carlo integration has a precision of 7 decimals and takes 8 days on a standard quad core PC. Gaining one more digit would take about $10^2 = 100$ times the CPU load or about 800 days, theoretically. Practically, in this approach 8 digits of precision are reached with about 50× the CPU load. Mathematica with OpenCL on a standad PC system provides approximately a 12× speedup, resulting in a decent CPU time of about 35 days.

CONTENTS

* thorsten@foerstemann.name

I. INTRODUCTION

A. Problem Statement

An analytical expression of the area of the Mandelbrot[1] is unknown. There are numerical estimations of the area, e.g. by Munafo (2012)[2] or [3]. The aim of this approach is to improve the accuracy of the numerical area estimation using Monte Carlo integration.

At the moment the most accurate estimation by Munafo 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.

B. Solution Statement

The basic idea of Monte Carlo implementations is quite trivial. Monte Carlo implementations 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 single core CPU implementations. A speedup by 100 yields one more digit precision at a given CPU time, as mentioned section I A.

In this case an efficient OpenGL implementation is not trivial due to the chaotic behavior of the Mandelbrot set. This OpenCL/GPU implementation is about 10 times faster than Munafo's quad core CPU approach in 2012. Thus, the actual GPU implementation provides an approximate speedup of factor 40 compared to an actual single core CPU implementation. The achieved pixel rate amounts to about 30 million pixels per second, while the maximum iteration depth is about 10^{10}.

Calculations were made in 2012 [7]. The GPU code ran in a Mathematica 8 environment on a standard PC with two dual core graphic cards.

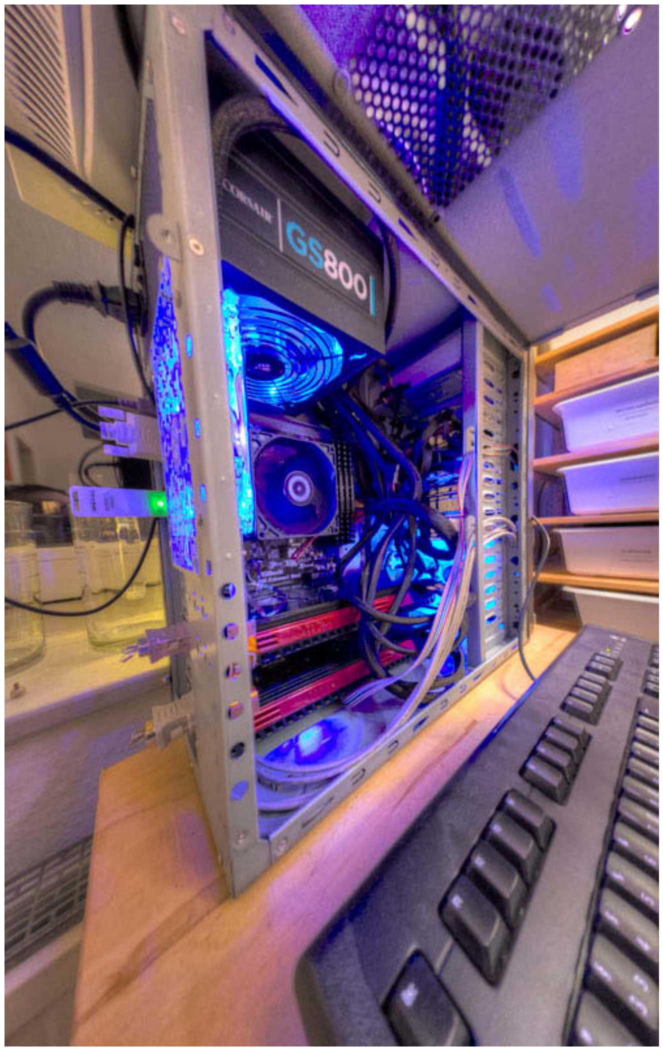


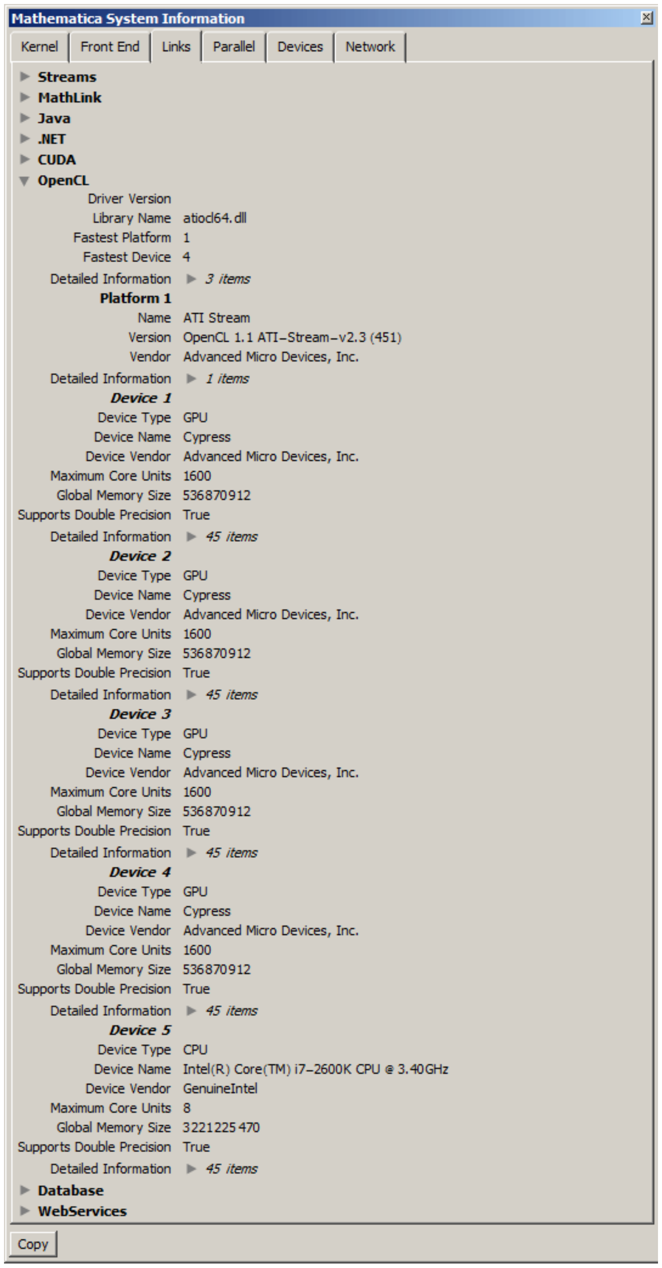
FIG. 1. Standard PC with two graphic cards.

II. MATERIALS AND METHODS

A. Hardware and Software

A typical PC (refer to figure 1) is used. Specific details of the hardware and software configuration are:

- CPU: Intel Core i7 2600K (about $700 full system, as of 2012)
- GPU: 2x Radeon HD 5970. Thus, there are 4 GPUs with 1600 stream processors (Cypress) each. Important feature: double precision; Ebay prices as of 2012: $250 each.
- Power consumption under load: approx. 350 watts. Thus, after 1 month we have about 300 kWh energy consumption and about $70 energy costs (Germany).
- Mathematica 8.0.4.0, Windows 7 (as of 2012)



```
▶ Streams
▶ MathLink
▶ Java
▶ .NET
▶ CUDA
▼ OpenCL
              Driver Version
                Library Name  atiocl64.dll
            Fastest Platform  1
              Fastest Device  4
        Detailed Information  ▶ 3 items
                  Platform 1
                        Name  ATI Stream
                     Version  OpenCL 1.1 ATI-Stream-v2.3 (451)
                      Vendor  Advanced Micro Devices, Inc.
        Detailed Information  ▶ 1 items
                    Device 1
                 Device Type  GPU
                 Device Name  Cypress
               Device Vendor  Advanced Micro Devices, Inc.
        Maximum Core Units    1600
          Global Memory Size  536870912
    Supports Double Precision True
        Detailed Information  ▶ 45 items
                    Device 2
                 Device Type  GPU
                 Device Name  Cypress
               Device Vendor  Advanced Micro Devices, Inc.
        Maximum Core Units    1600
          Global Memory Size  536870912
    Supports Double Precision True
        Detailed Information  ▶ 45 items
                    Device 3
                 Device Type  GPU
                 Device Name  Cypress
               Device Vendor  Advanced Micro Devices, Inc.
        Maximum Core Units    1600
          Global Memory Size  536870912
    Supports Double Precision True
        Detailed Information  ▶ 45 items
                    Device 4
                 Device Type  GPU
                 Device Name  Cypress
               Device Vendor  Advanced Micro Devices, Inc.
        Maximum Core Units    1600
          Global Memory Size  536870912
    Supports Double Precision True
        Detailed Information  ▶ 45 items
                    Device 5
                 Device Type  CPU
                 Device Name  Intel(R) Core(TM) i7-2600K CPU @ 3.40GHz
               Device Vendor  GenuineIntel
        Maximum Core Units    8
          Global Memory Size  3221225470
    Supports Double Precision True
        Detailed Information  ▶ 45 items
▶ Database
▶ WebServices
```

FIG. 2. Mathematica's OpenCL system information. Most important information: support for double precision.

- ATI driver Catalyst 11.2 with AMD Stream SDK 2.3 [6] (as of 2012)

B. Methods

In this section the applied method to estimate the area of the Mandelbrot set based on Monte Carlo integration is illustrated. The implementation with OpenCL and Mathematica is detailed in the following sections.

1. Grid Coordinates and Dimensions

The Mandelbrot set is sampled by slightly varying rasters. Complex numbers z in the complex plane are parametrized by two real coordinates x and y via $z = x + iy$. All floating point calculations are done in double precision (refer to figure 2).

Large rasters can not be efficiently handled by the GPU due to memory and timing issues. Thus, the complete raster has to be partitioned in smaller, so-called GPU grids. These GPU grids can be arranged in so-called CPU grids.

To summarize this, rasters are partitioned into:

1. CPU grids of variable size $M \times M$;
   Grid points are indexed by (n, m). The indicies (n, m) are used to index the different GPU grids.

2. GPU grids of fixed size $N \times N$ where $N = 2048$;
   Grid points are indexed by (i, j).

The GPU gird size is $N = 2048$. Thus, about 4 million pixels are simultaneously loaded into each GPU. The calculations of the $M \times M$ GPU grids can be parallelized by the i7 quad core CPU using all four GPUs. Thus, these grids are called CPU grids.

The coordinates for each raster point indexed by i, j, n and m are

$$\begin{pmatrix} x_{nm,ij} \\ y_{nm,ij} \end{pmatrix} = \begin{pmatrix} \text{offs}_x \\ \text{offs}_y \end{pmatrix} + wM \begin{pmatrix} n \\ m \end{pmatrix} + w \begin{pmatrix} i \\ j \end{pmatrix}$$

Thus, the size of the raster is $(NM) \times (NM)$. Since N is fixed the size of the raster can be parameterized by M. The offset is given by

$$\begin{pmatrix} \text{offs}_x \\ \text{offs}_y \end{pmatrix} = \begin{pmatrix} -2.05 \\ -1.3 \end{pmatrix} + w \begin{pmatrix} \text{rnd}_1 \\ \text{rnd}_2 \end{pmatrix}$$

The spacing of the raster, i.e. the grid interval, is given by

$$w = 2.6 \frac{1.01 + 0.01\ \text{rnd}_3}{NM}$$

where rnd_1, rnd_2 and rnd_3 are random numbers between 0 and 1. These random numbers are arbitrarily but consistently choosen for each raster.

Figures 3 and 4 illustrate a typical raster and the different grids. Each GPU grid covers the hole Mandelbrot set. So, the work load of the different GPU rasters is approximately balanced.

2. Parallelization and Performance Considerations

As mentioned in section II B 1 large rasters can not be efficiently handled by the GPU due to memory and

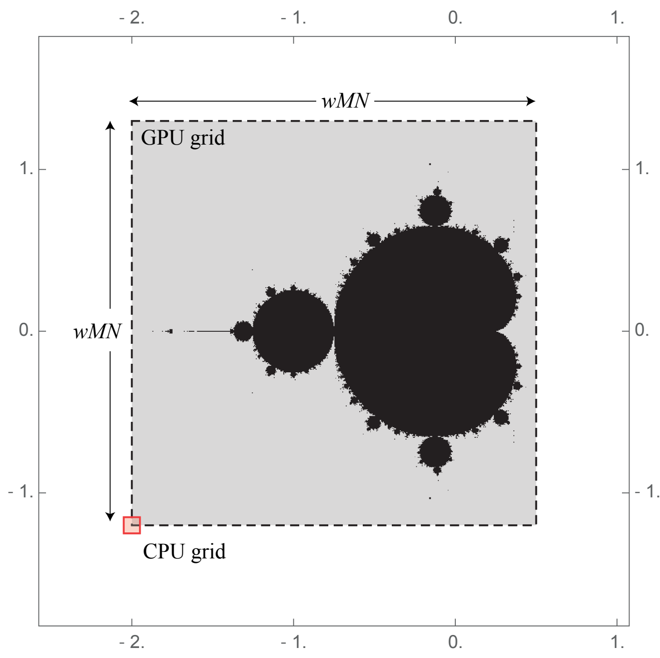


FIG. 3. The GPU grid is marked gray. The grid interval wM is too small to identify individual grid points. The CPU grid is marked red.

timing issues. In this approach the GPU grid size is set to $N = 2048$. Thus, the complete rasters of size (NM) has to be partitioned in $M \times M$ GPU grids.

The calculations of the individual $M \times M$ GPU grids can be parallelized. In Mathematica four worker kernels process the list of the GPU grids in parallel. Each worker kernel is bound to an individual GPU. There are two graphic cards with two GPUs each (refer to figure 1).

Since $N = 2048$, about 4 million pixels are simultaneously loaded into each GPU. By starting iterating on these points more and more points will diverge or converge. Some points stay chaotic, these points remain in the pixel pipeline (refer to section II B 3).

Each GPU contains 1600 stream processors (refer to figure 2). After a certain number of iterations the number of remaining pixels in the pipeline 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. 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 5 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 add_List procedure

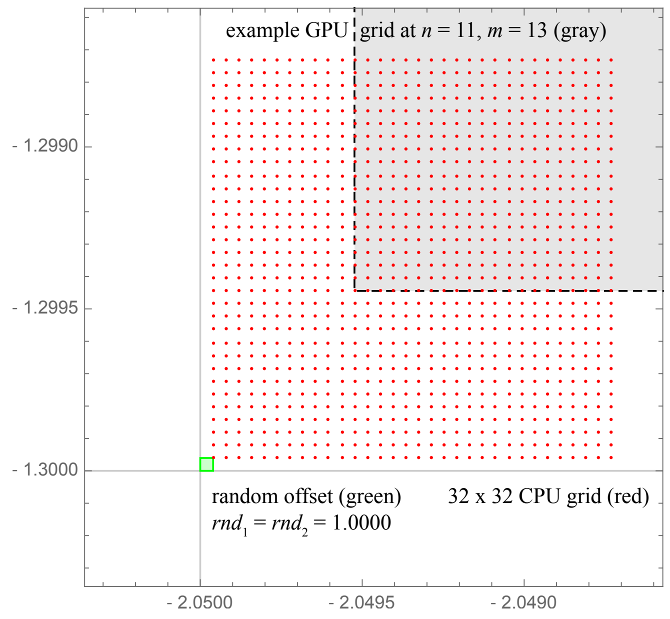


FIG. 4. The CPU grid for $\text{rnd}_1 = \text{rnd}_2 = 1$ and $\text{rnd}_3 = 0$ is marked red. An example GPU grid is given for $n = 11$ and $m = 13$. The random offset is marked green. The raster size is 65536×65536, since $65536 = 32N$ and $N = 2048$.

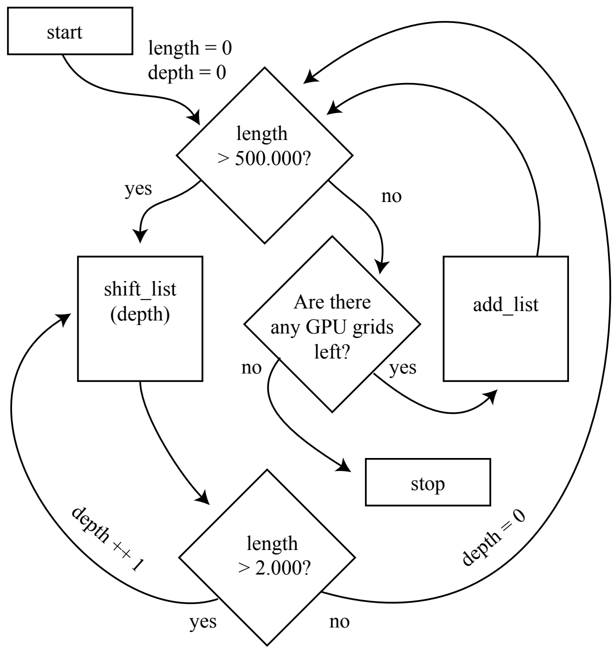


FIG. 5. Pipeline handling

(refer to algorithm 1) appends a list of pre selected points of a GPU grid to the pipeline. The details of the add_List procedure are outlined in the next section. The procedure add_List is applied until the length of the pipeline is greater than 500.000 points (refer to figure 5).

The shift_List procedure (refer to algorithm 2) tries to apply $2^{\text{depth}} \cdot 10\,000$ iterations on each pixel in the pipeline. Pixels belonging to the Mandelbrot set and diverging pixels are removed from the pipeline. After applying shift_List 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 long periods. When the length of the pipeline is smaller than 2000 pixels depth is set to zero and again add_List is applied to the pipeline (refer to figure 5).

Very high iteration depths can be reached by recycling the less than 2000 non member and non diverging, i.e. chaotic pixels in the next cycle of the algorithm. At larger rasters the parameter depth reaches values up to 16.

A drawback of this approach is that there is no control over the total number of iterations for each pixel, due to the recycling of chaotic pixels. Thus, the maximum iteration is not really a maximum, i.e. some pixels may reach even higher iterations. But it is a guarantied minimum number of iterations that is applied before a pixel is finally regarded as chaotic.

4. *The add_List and shift_List Procedures*

The add_List procedure adds preselected pixels to the pixel pipeline. Following figure 5 the add_List procedure is applied repeatedly to the pipeline until the length of the pipeline is greater than 500 000 pixels. Timings and statistical portions of diverging, converging or chaotic pixels are discussed in section III C.

The pseudocode of the add_List procedure reads as follows:

Algorithm 1 Outline of the add_List procedure

```
1: create_GPU_grid()
2: do_iterations(990)
3: test_circle_cardioid()
4: for $n = 1$, $n < 4$ do
5:     do_iterations($2^n * 990$)
6:     test_orbit($2^n * 1000$)
```

The specific definitions of the sub procedures are:

create_GPU_grid(): This sub routine creates an GPU grid with $2048^2 \approx 4$ million pixels following the procedure described in section II B 1.

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

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

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

The shift_List procedure removes diverging and converging pixels from the pixel pipeline. Following figure 5 the shift_List procedure is applied repeatedly to the pipeline until the length of the pipeline is smaller than 2000 pixels. Each time the parameter depth is increased by one. Depth reaches values up to 16. Timings and statistical portions of diverging, converging or chaotic pixels are discussed in section III C.

The pseudocode of the shift_List procedure reads as follows:

Algorithm 2 Outline of the shift_List procedure

```
1: for $n = 0$, $n < \text{depth} + 1$ do
2:     do_iterations($2^n * 9900$)
3:     test_orbit($2^n * 10000$)
```

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

5. *Post Processing of the Pipeline*

As mentioned in section II B 1 there are $M \times M$ GPU grids for a given raster of size $(NM)\times(NM)$. At a certain time all GPU grids are depleted and integrated into the four pipelines. Then the algorithm mentioned in section II B 3 ends with up to 4×2000 remaining pixels. A usual CPU based iteration algorithm with orbit detection is applied to these remaining 8000 pixels of a given raster. The chosen maximum iterations are listed in table II. On the one hand the maximum iterations are chosen to balance GPU and CPU load. On the other hand they are choosen to keep the estimation error of the area due to the remaining pixels smaller than the error due to the sampling of the 20 rasters. The total numbers of remaining pixels for all 20 rasters and the corresponding errors of the estimated area are listed in table II.

III. RESULTS

A. Current Area Estimates

The estimate from Munafo (2012) and the estimate calculated here can be read off table I. The estimate calculated here is based on the results listed in table II (refer to the last row). Table II is inspired from a similar table by Munafo[8].

20 rasters are calculated at each raster size given in table II. Details of the different rasters can be read off table II:

Raster size: The raster size is the number of raster points in both dimensions (quadratic rasters).

TABLE I. Comparison of current Mandelbrot area estimates

Estimate & error	Details
Förstemann (2016)	
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	1 677 721 600 000 calculated pixels
0.000 000 0256	691 255 sec. (8.00 days) CPU time
	2 427 066 pixel/sec

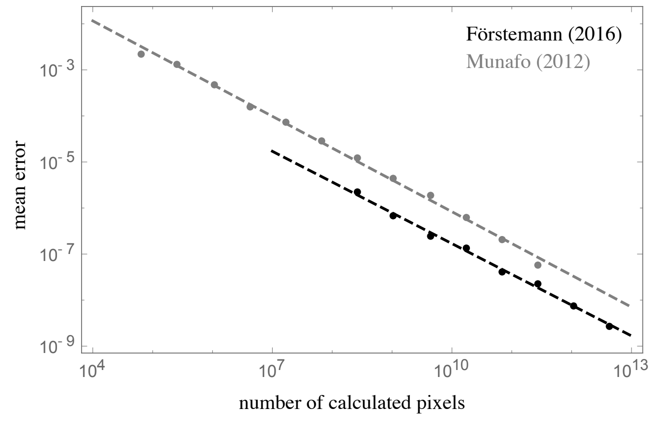


FIG. 6. A comparison of the mean error versus number of calculated pixels between this and Munafo's approach. The dashed lines correspond to regressions $y \sim x^{-0.68}$.

Max.iterations: The lower bound of the maximum iterations, i.e. the maximum iteration of the post processing of the pipeline (refer to II B 5).

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

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

CPU [sec.]: The total GPU/CPU time in seconds needed to calculate the 20 rasters

Av.pixel/s: The average pixel rate, i.e. number of calculated pixels per second on average

A comparison of the mean error versus number of calculated pixels between this and Munafo's approach is given in figure 6. The numbers are taken from table II and the corresponding table by Munafo mentioned above.

The achieved pixelrate hit 30 million pixels per second while the maximum iteration is larger than $8 \cdot 10^9$.

TABLE II. Parameters of grids calculated so far and numbers of remaining pixels for different grids after post processing the pipeline. Corresponding errors of the estimated area are given and can be compared with the error of the mean of the 20 rasters. For explanations of the table headings refer to sections III B and III A.

Raster size	Max.iterations	Area estimate	Error (mean)	Remain	Error (remain)	CPU [sec.]	Av.pixel/s
16 384	67 108 864	1.506 590 913 0	0.000 002 171 7	520	0.000 000 675 5	955	5 621 685
32 768	134 217 728	1.506 591 383 0	0.000 000 678 8	1 000	0.000 000 324 8	2 539	8 457 990
65 536	268 435 456	1.506 592 037 0	0.000 000 251 4	1 889	0.000 000 153 1	8 952	9 595 548
131 072	536 870 912	1.506 591 911 7	0.000 000 135 8	3 571	0.000 000 072 3	23 979	14 329 096
262 144	1 073 741 824	1.506 591 918 4	0.000 000 040 0	5 711	0.000 000 029 0	58 394	23 536 486
524 288	2 147 483 648	1.506 591 883 3	0.000 000 021 7	8 091	0.000 000 010 2	194 027	28 333 985
1 048 576	4 294 967 296	1.506 591 883 1	0.000 000 007 3	8 261	0.000 000 002 6	745 509	29 496 938
2 097 152	8 589 934 592	1.506 591 884 9	0.000 000 002 8	9 922	0.000 000 000 8	3 028 136	29 047 880

B. Errors Resulting from Chaotic Pixels

As mentioned above, the GPU based algorithm ends with up to 4×2000 remaining pixels per raster, thus a total of about 160 000 pixels for 20 rasters. The CPU based iteration algorithm applied to these remaining pixels. The number of the finally remaining chaotic pixels is given in table II. Since it is unknown wether these chaotic pixels belong to the Mandelbrot set or not, the corresponding area is regarded as an error. This error resulting from chaotic pixels should be sufficiently smaller than the error estimation based on the 20 rasters. Both errors are given in table II:

Error (mean): refer to section III A

remain: The total number of finally remaining pixels after post iteration of the 20 rasters

Error (remain): The error of the area based on the remaining pixels

Since most of the unknown pixels tend to be members of the Mandelbrot set they are regarded as members when calculating the area.

C. Timings and Interim Results

Each pipeline starts with the add_List procedure (refer to figure 5). The add_List procedure is outlined in algorithm 1). First of all a GPU grid is created containing 2048^2 pixels (refer to section II B 2). Typically these about 4 million pixels divide into the following parts: 22.25% belong to the Mandelbrot set, 77.69% diverge and 0.06% remain unknown. Thus, on average 2500 unknown pixels are appended to the pipeline. This procedure needs on average 100 milliseconds GPU time for iterations and about 200 milliseconds CPU time for rearranging the pipeline.

The shift_List procedure needs about 20 milliseconds GPU time on average. This timing is quite independent from the depth parameter because more and more iterations are applied to less and less pixels with increasing depth. For example, if shift_List is applied 10 times to the pipeline we have about 200 milliseconds GPU time. Finally rearranging the pipeline needs 100 milliseconds CPU time.

In total we have on average $200+10\cdot 20 = 400$ milliseconds GPU time and $100 + 100 = 200$ milliseconds CPU time per raster of size 2048^2. Thus, we have about 4 million pixels per 0.5 seconds for each of the four pipelines or in total a pixelrate of about 30 million pixels.

While the 400 milliseconds GPU time the CPU is idle. This time span is used for the post iteration on the CPU (refer to II B 5). The post iteration runs completely parallel to the four pipelines and thus the pixelrate is not influenced for larger rasters.

IV. DISCUSSION

The main advantage of this GPU based Monte Carlo approach is the gained pixelrate. It is about $40\times$ faster than a comparable single core approach. Since Munafo uses a quad core machine, we have a decent speedup of about factor 12.

But the main disadvantage of a typical Monte Carlo approach is still present: the calculated errors are only based on statistics and have no further theoretical significance. Although the errors are narrowed, they still can not be regarded as reliable upper and lower bounds of the Mandelbrot set.

Appendix A: Soucre code

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, 0}
    }, {n, 1, sizeSample}];

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

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) */
  int orbitMaxiter = 1000;  /* initial value maximum iteration depth with orbit test (empiric) */
  int maxDoubling = 3;     /* number of maximum iteration's doublings (empiric) */
  int doubling;
  double x0 = offx + step * xIndex;
  double y0 = offy + step * yIndex;
  double tmp, q, x = 0.0, y = 0.0;
  double ox, oy;
  bool card, circ;
  bool member = false;
  bool refuge = false;

  /* housekeeping: check index */
  if (xIndex < size && yIndex < size) {
    /* warm up iterations */
    for (iter = 0; (x*x+y*y <= 4.0) && (iter < initMaxiter); iter++) {
             tmp = x*x - y*y + x0;
```

```
              y = 2.0*x*y + y0;
              x = tmp;
          }
     refuge = (x*x+y*y > 4); /* Escape? */
     /* Circle and cardioide test */
     if (!refuge) {
       q = pow((x0 - 0.25), 2) + y0 * y0;
       card = q * (q + (x0 - 0.25)) < 0.25 * y0 * y0;
       circ = pow((x0 + 1.0), 2) + y0 * y0 < 0.0625;
       member = (card || circ);
     }
     /* Maxiter doubling loop */
     for (doubling = 1; !refuge && !member && doubling<maxDoubling; doubling++) {
       /* normal iteration, maxiter: 2^doubling * initMaxiter */
       for (iter = iter;
         (x*x+y*y <= 4.0) && (iter < pow(2.0f,doubling)*initMaxiter);
         iter++) {
         tmp = x*x - y*y + x0;
         y = 2.0*x*y + y0;
         x = tmp;
       }
       refuge = (x*x+y*y > 4); /* Escape? */
       /* iteration with orbit test, maxiter: 2^doubling * orbitMaxiter */
       if (!refuge) {
         ox = x; /* set real part for orbit test */
         oy = y; /* set imaginary part for orbit test */
         tmp = x*x - y*y + x0;
         y = 2.0*x*y + y0;
         x = tmp;
         for (iter = iter;
           (x*x+y*y <= 4.0) && (iter < pow(2.0f,doubling)*orbitMaxiter) &&
           ((ox != x) || (oy != y)); /* non orbit test */
           iter++) {
           tmp = x*x - y*y + x0;
           y = 2.0*x*y + y0;
           x = tmp;
         }
         refuge = (x*x+y*y > 4); /* Escape? */
         member = ((ox == x) && (oy == y)); /* Orbit? */
       }
     }
     if (!refuge && !member) {
       set[(yIndex + xIndex * size)] = 255; /* mark point as unknown */
     }
     if (member) {
       set[(yIndex + xIndex * size)] = 128; /* mark point as member */
     }
     if (refuge) {
       set[(yIndex + xIndex * size)] = 0; /* mark point as refugee */
     }
   }
  }
  ";
```

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.

```
sourceList = "
 #ifdef USING_DOUBLE_PRECISIONQ
 #pragma OPENCL EXTENSION cl_amd_fp64 : enable
 #endif /* USING_DOUBLE_PRECISIONQ */
  __kernel void mandelbrot_kernel(
   __global double * set, /* housekeeping: data pointer */
   mint initMaxiter,       /* maxiter without orbit test */
   mint orbitMaxiter,       /* maxiter with orbit test */
   mint size)               /* housekeeping: data size */
 {
  int xIndex = get_global_id(0);
  int iter;
  double tmp;
  double cx, cy, x, y, ox, oy;
  bool refuge = false;
  bool member = false;

  /* housekeeping: check index */
  if (xIndex < size) {
    cx = set[4*(xIndex)];     /* collecting real part of point coordinates */
    cy = set[4*(xIndex)+1];  /* collecting imaginary part of point coordinates */
    x = set[4*(xIndex)+2];   /* collecting real part of actual iteration value */
    y = set[4*(xIndex)+3];   /* collecting imaginary part of actual iteration value */
    /* normal iteration, maxiter: initMaxiter */
    for (iter = 0;
      (x*x+y*y <= 4.0) && (iter < initMaxiter);
      iter++) {
             tmp = x*x - y*y + cx;
             y = 2.0*x*y + cy;
             x = tmp;
        }
    refuge = (x*x+y*y > 4.0); /* Escape? */
    /* iteration with orbit test, maxiter: orbitMaxiter */
    if (!refuge) {
      ox = x; /* set real part for orbit test */
      oy = y; /* set imaginary part for orbit test */
      tmp = x*x - y*y + cx;
      y = 2.0*x*y + cy;
      x = tmp;
      for (iter = iter;
        (x*x+y*y <= 4.0) && (iter < orbitMaxiter) &&
        ((ox != x) || (oy != y)); /* non orbit test */
        iter++) {
        tmp = x*x - y*y + cx;
        y = 2.0*x*y + cy;
        x = tmp;
      }
      refuge = (x*x+y*y > 4); /* Escape? */
      member = ((ox == x) && (oy == y)); /* Orbit? */
    }
    if (refuge) {
      /* mark point as refugee */
      set[4*(xIndex)] = 0.0;
      set[4*(xIndex)+1] = 0.0;
      set[4*(xIndex)+2] = 0.0;
      set[4*(xIndex)+3] = 0.0;
    }
    if (member) {
      /* mark point as member */
      set[4*(xIndex)] = 1.0;
      set[4*(xIndex)+1] = 1.0;
      set[4*(xIndex)+2] = 1.0;
      set[4*(xIndex)+3] = 1.0;
    }
    if (!refuge && !member) {
      /* mark point as unknown */
      set[4*(xIndex)] = cx;
      set[4*(xIndex)+1] = cy;
      set[4*(xIndex)+2] = x;
      set[4*(xIndex)+3] = y;
    }
  }
 }
 ";
```

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.

```
Needs["OpenCLLink`"]
ParallelNeeds["OpenCLLink`"]

ParallelEvaluate[
 sourceBitmap="...";
 sourceList="...";

 (* wake up device *)
 mem=OpenCLMemoryAllocate[Integer,{1024,1024},"Platform"->1,"Device"->$KernelID];
 OpenCLMemoryUnload[mem];

 (* Load GPU functions *)
 clBitmap=OpenCLFunctionLoad[
  sourceBitmap,"mandelbrot_kernel", {{_Integer},"Double","Double","Double",_Integer},{16,16},"ShellOutputFunction"->Print,"ShellCommandFunction"->Print
 ];
 clList=OpenCLFunctionLoad[
  sourceList,"mandelbrot_kernel",{{"Double"},_Integer,_Integer,_Integer},64,"ShellOutputFunction"->Print,"ShellCommandFunction"->Print
  ];

 (* Helper function for addList, from point index to point coordinates *)
 gpuRaster[{{rndX_,rndY_,rndW_},cpuSize_,gpuSize_},idx_}]:=Block[
   {cpuOffs,cpuWidth,gpuOffsX,gpuOffsY,gpuWidth},
   If[idx<0||idx>cpuSize^2,Abort[]];
   cpuOffs={-2.05,-1.3}; (* left bottom *)
   cpuWidth=2.6*(1.01+0.01*rndW)/cpuSize; (* raster size with random variation *)
   gpuWidth=cpuWidth/gpuSize;
   {gpuOffsX,gpuOffsY}=cpuOffs-{rndX,rndY}*gpuWidth+{Quotient[idx,cpuSize],Mod[idx,cpuSize]}*cpuWidth;
   {gpuOffsX,gpuOffsY,gpuWidth,gpuSize}
 ];

 (* fill up work list *)
 addList[{{fullList_,doneList_,workList_},{{rndX_,rndY_,rndW_},{cpuSize_,gpuSize_}},time_,kernel_,depth_,storedepth_}]:=Block[
  {idx,ox,oy,step,size,mem,res,out,img,erg,lst,timer,timers},
  timers={0,0};timer=AbsoluteTime[];
  idx=Length[doneList]+Length[fullList];
  If[idx>cpuSize^2/4-1,Return[{{doneList,workList,timers},{{rndX,rndY,rndW},{cpuSize,gpuSize}},time+timers,kernel,depth,storedepth}]];
  {ox,oy,step,size}=gpuRaster[{{rndX,rndY,rndW},{cpuSize,gpuSize},4*idx+(kernel-1)}];
  timers=timers+{AbsoluteTime[]-timer,0};timer=AbsoluteTime[];
  mem=OpenCLMemoryAllocate[Integer,{size,size}];
  res=clBitmap[mem,ox,oy,step,size];
  out=OpenCLMemoryGet[First[res]];OpenCLMemoryUnload[mem];
  timers=timers+{0,AbsoluteTime[]-timer};timer=AbsoluteTime[];
  img=Image[out,"Byte"];
  erg=ImageLevels[img,3][[{1,2},2]];
  lst=ImageData[Binarize[img,0.9]];
  lst=ArrayRules[SparseArray[lst]][[;;-2,1]];
  lst=Map[{4*idx+(kernel-1),{ox,oy}+(#-1)*step,{0.0,0.0},0}&,lst];
  timers=timers+{AbsoluteTime[]-timer,0};timer=AbsoluteTime[];
  {{fullList,Append[doneList,{4*idx+(kernel-1),erg}],Join[workList,lst]} ,{{rndX,rndY,rndW},{cpuSize,gpuSize}},time+timers,kernel,depth,storedepth}
 ];

 (* Helper function for shiftList, extract positions *)
 positionHelper[{doneList_,workList_},bothList_,pattern_]:=Block[
   {extractedList},
  extractedList=Extract[workList,Position[bothList,pattern]];
  extractedList=extractedList[[All,{1,2,3}]];
  extractedList=Split[Sort[extractedList],#1[[1]]==#2[[1]]&];
  extractedList=Map[{#[[1,1]],Length[#]}&,extractedList];
  extractedList=Map[{Position[doneList[[All,1]],#[[1]]][[1,1]],#[[2]]}&,extractedList];
  extractedList
 ];

 (* die out work list *)
 shiftList[{{fullList_, doneList_, workList_},{{rndX_, rndY_, rndW_},{cpuSize_, gpuSize_}},time_, kernel_, depth_, storedepth_}]:=Block[
  {bothList, refugeList, memberList, unknownList, doneOut, workOut, fullOut, timer, timers},
  timers={0,0};timer=AbsoluteTime[];
  doneOut=doneList;
  bothList=Flatten[Map[{#[[2,1]],#[[2,2]],#[[3,1]],#[[3,2]]}&, workList]];
  timers=timers+{AbsoluteTime[]-timer,0};timer=AbsoluteTime[];
  bothList=clList[bothList,2^depth*9900,2^depth*10000,Length[bothList]/4];
  timers=timers+{0,AbsoluteTime[]-timer};timer=AbsoluteTime[];
  bothList=Partition[First[bothList],4];
  refugeList=positionHelper[{doneList, workList},bothList,{0.,0.,0.,0.}];
  Do[doneOut[[refugeList[[i,1]],2,1]]=doneList[[refugeList[[i,1]],2,1]]+refugeList[[i,2]],{i,Length[refugeList]}];
  memberList=positionHelper[{doneList, workList},bothList,{1.,1.,1.,1.}];
  Do[doneOut[[memberList[[i,1]],2,2]]=doneList[[memberList[[i,1]],2,2]]+memberList[[i,2]],{i,Length[memberList]}];
  unknownList=Position[bothList, Except[{0.,0.,0.,0.}|{1.,1.,1.,1.}],{1}];
  unknownList=Transpose[{Drop[Extract[workList, unknownList],1],Drop[Extract[bothList, unknownList],1]}];
  workOut=Map[{#[[1,1]],{#[[2,1]],#[[2,2]]},{#[[2,3]],#[[2,4]]},#[[1,4]]+depth}&,unknownList];
  fullOut=Join[fullList, Select[doneOut,Not[gpuSize^2-Total[#[[2]]]>0]&]];
  doneOut=Select[doneOut, gpuSize^2-Total[#[[2]]]>0&];
  timers=timers+{AbsoluteTime[]-timer,0};timer=AbsoluteTime[];
  {{fullOut, doneOut, workOut},{{rndX, rndY, rndW},{cpuSize, gpuSize}},time+timers, kernel, depth+1, storedepth}
 ];
];
```

5. Mathematica code for the CPU part

This Mathematica code represents the main loop.

```
path=NotebookDirectory[];
SetDirectory[path];
files=FileNames["in_*"];

(* Main loop *)
While[files!={},
 allTime=AbsoluteTime[];
 file=First[files];
 set=Get[file];
 upperLimit=500000;lowerLimit=2000;finalLimit=20; (* list dimensions *)
 DistributeDefinitions[set, upperLimit, lowerLimit, finalLimit, path, file];

 (* Begin parallel evaluation *)
 ParallelEvaluate[
  SetDirectory[path];

  (* init working list *)
  {{fullList, doneList, workList},{{rndX, rndY, rndW},{cpuSize, gpuSize}},timer, kernel, depth, storedepth}=set;
  set={{fullList, doneList, workList},{{rndX, rndY, rndW},{cpuSize, gpuSize}},{0,0},$KernelID,0, storedepth};

  (* CPU raster loop *)
  While[4*(Length[fullList]+Length[doneList])<cpuSize^2,
   (* fill up working list *)
   set=NestWhile[addList, set, Length[#[[1,1]]]+Length[#[[1,2]]]<#[[2,2,1]]^2/4&&Length[#[[1,3]]]<upperLimit&];
   (* die out working list *)
   set=NestWhile[shiftList, set, Length[#[[1,3]]]>lowerLimit&];
   (* update working list *)
   {{fullList, doneList, workList},{{rndX, rndY, rndW},{cpuSize, gpuSize}},timer, kernel, depth, storedepth}=set;
```

```

    (* print work list status *)
    Print[Grid[{{
      DateString[AbsoluteTime[],{"Day",":","Time"}],
      ToString[Round[(Length[fullList]+Length[doneList])/cpuSize^2*4*100,0.1]]<>"%",
      ToString[Length[fullList]],
      ToString[Length[doneList]],
      ToString[Length[workList]],
      ToString[Round[timer[[1]],0.1]]<>" s",
      ToString[Round[timer[[2]],0.1]]<>" s",
      ToString[depth]
    }},Frame->All,Alignment->Right,ItemSize->All,Background->Which[kernel==1,LightRed,kernel==2,LightGreen,kernel==3,LightBlue,kernel==4,LightGray]]];

    (* update working list with resetted times *)
    set={{fullList,doneList,workList},{{rndX,rndY,rndW},{cpuSize,gpuSize}},{0,0},kernel,0,depth};
   ];

   (* write work list to kernel file *)
   Put[set,StringReplace[file,{"in"->"out",".txt"->""}]<>"_"<>ToString[$KernelID]<>".txt"];
  ];
 (* End parallel evaluation *)

 (* print work list final status *)
 Print[Grid[{{
  DateString[AbsoluteTime[],{"Day",":","Time"}],
  file,ToString[IntegerPart[AbsoluteTime[]-allTime]]<>" s"
 }},Frame->All,Alignment->Right,ItemSize->All]];

 (* Read 4 kernel files *)
 tmp=Map[Get,Table[StringReplace[file,{"in"->"out",".txt"->""}]<>"_"<>ToString[i]<>".txt",{i,4}]];
 tmp=Transpose[tmp[[All,1]]];
 tmp=Apply[Join,tmp,1];
 cpu=tmp[[3,All,{2,3}]];
 cpu=Map[#[[1]]+I*#[[2]]&,cpu,{2}];
 summ=Total[tmp[[1,All,2]]]+Total[tmp[[2,All,2]]];

 (* Write result files: { time, {#member, #nonmember}, list of coords of unknown} *)
 Put[{AbsoluteTime[]-allTime,summ,cpu},StringReplace[file,{"in"->"out",".txt"->""}]<>"_5"<>".txt"];
 (* Delete input file, mark as done *)
 DeleteFile[file];
 files=FileNames["in_*"];
];
```

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.

```
path=NotebookDirectory[];
SetDirectory[path];

(* file1-4: kernel files *)
files=FileNames["out_0001_1.txt"];
While[files=={},Pause[10];files=FileNames["out_0001_1.txt"]];
data1=Get["out_0001_1.txt"][[2]];
cpuSize=data1[[2,1]] (* get only cpu grid size *)
level=Log[2,cpuSize] (* calc depth *)

(* compile start *)
cPostIter =Compile[{{zz, _Complex,1}},
 Module[{n = 0,nn=0,c=zz[[1]],z=zz[[2]],z0=0.+0. I,maxi0=2^15,maxi1=2^16,refuge=False,member=False,istep=0},
   While[istep<zz[[3]]+7&&!refuge&&!member,
    nn=0;
    While[nn<2^istep &&!refuge ,
     n=0;
     While[n < maxi0 &&!refuge ,
      z = z^2 + c;
      refuge=Re[z]^2 + Im[z]^2 > 4;
      n++
     ];
     nn++
    ];
    If[!refuge,
     z0=z;
     z=z^2+c;
     nn=0;
     While[nn <2^istep &&!refuge&&!member ,
      n=0;
      While[n < maxi1 &&!refuge&&!member ,
       z = z^2 + c;
       refuge=Re[z]^2 + Im[z]^2 > 4;
       member=(Re[z0]==Re[z])&&(Im[z0]==Im[z]);
       n++
      ];
      nn++
     ];
    ];
    istep++;
   ];
   Which[
    refuge,{0,istep,n,nn},
    member,{1,istep,n,nn},
    True,{zz[[1]],istep,n,nn}
   ]
 ],RuntimeAttributes->{Listable},Parallelization->True,CompilationTarget->"C",RuntimeOptions->"Speed"
];

(* Main loop *)
While[True,
 path=NotebookDirectory[];
 SetDirectory[path];
 files5=FileNames["out_*_5.txt"];
 files6=FileNames["out_*_6.txt"];
 files5=Map[StringReplace[#,"_5.txt"->""]&,files5];
 files6=Map[StringReplace[#,"_6.txt"->""]&,files6];
 files=Complement[files5,files6];
 If[files=={},Pause[10],
  allTime=AbsoluteTime[];
  file=First[files];
  data=Get[file<>"_5.txt"];
  tmp=Map[{#[[1]],#[[2]],level}&,data[[3]]];
  cpu=Chop[cPostIter[tmp]][[All,{1,2}]];
  Put[cpu,file<>"_6"<>".txt"];
  summ=data[[2]]+{Length[Select[cpu,#[[1]]==0.&]],Length[Select[cpu,#[[1]]==1.&]]};
  left=Length[cpu]-Length[Select[cpu,#[[1]]==0.&]]-Length[Select[cpu,#[[1]]==1.&]];
  rnd=Get[file<>"_1"<>".txt"][[2,1,3]];
  area=(summ[[2]]+left)/(Total[summ]+left)*(2.6*(1.01+0.01*rnd))^2;
  Put[{area,summ[[1]],summ[[2]],left,rnd,IntegerPart[AbsoluteTime[]-allTime]},file<>"_7"<>".txt"];
  Print[Grid[{{
    DateString[AbsoluteTime[],{"Day",":","Time"}],
    file,
    ToString[AccountingForm[area,{8,8},DigitBlock->3,NumberPadding->{"","0"}]],
    ToString[summ[[1]]],
    ToString[summ[[2]]],
    ToString[left],
    rnd,
    ToString[IntegerPart[AbsoluteTime[]-allTime]]<>" s"
  }},Frame->All,Alignment->Right,ItemSize->All]];
 ];
];
```

[1] `http://en.wikipedia.org/wiki/Mandelbrot_set`
[2] `http://www.mrob.com/pub/muency/pixelcounting.html`
[3] `http://oeis.org/A098403`
[4] `http://en.wikipedia.org/wiki/Monte_Carlo_method`
[5] `http://en.wikipedia.org/wiki/OpenCL`
[6] `http://forums.wolfram.com/mathgroup/archive/2011/Jun/msg00615.html`
[7] `http://www.foerstemann.name/labor/area/Mset_area.pdf`
[8] `http://www.mrob.com/pub/muency/pixelcounting.html`