TDT4200 --- Fall 2024
PS 4 – CUDA Intro: Mandelbrot Fractals

Author: Anne C. Elster. Due date: Oct 21 by 10pm. Grading: Pass/Fail.

This problem set / exercise give you an opportunity to write a CUDA program based PS1b which had a nice graphical interface.

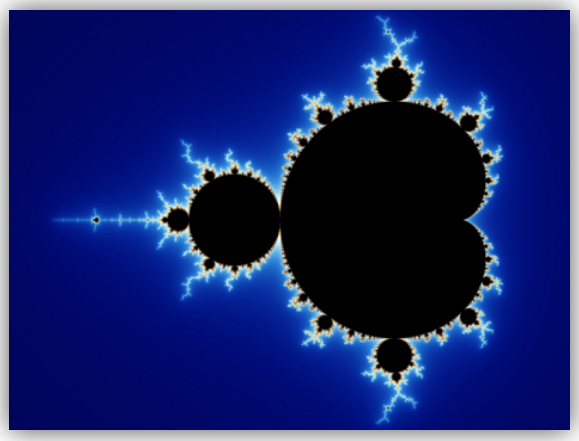
This Problem Set Pass/Fail, and as such need to be passed to get PS6 graded and to take the final and pass the course. See also information under Course info on BlackBoard.on how you log into Snotra.

This assignment, like the others, should be done individually and handed in by the posted deadline, preferably by 10pm on Monday Oct 21. If you cannot make the deadline, contact Prof. Elster.

Introduction

The Mandelbrot set is one of the most well-known mathematical visualizations methods used to generate fractal images – images that have "infinite resolution", i.e. for each step it gets more detailed. It is described by the quadratic polynomial $z_{n+1} = z_n^2 + c$, where c is a part of the Mandelbrot set if the absolute value of z_n remains bounded.

To achieve great level of detail (LOD), a lot of computations are required. However, **by utilizing the power of the GPU, amazingly detailed fractal images can be generated on-the-fly. We will be using the GPU for this in PS5.**



3783OT_01_01.png

Figure 1. Fractal image of a Mandelbrot set from Wiki
http://en.wikipedia.org/wiki/Mandelbrot_set

Another animated fractal generation is available here:

http://en.wikipedia.org/wiki/File:Animation_of_the_growth_of_the_Mandelbrot_set_as_you_iterate _towards_infinity.gif

More info on generalization and implementation:

http://en.wikipedia.org/wiki/Mandelbrot_set#Generalizations

Setting up benchmark basic C program (CPU) – mandel-cpu.c

In this section, we in PS1b described how we developed the Mandelbrot CPU-based C code from scratch. We wrote this code as a benchmark to run against the CUDA code you will be developing.

The main difference between the codes provided by the CUDA SDK and the ones we develop here, is that the code we develop here has fewer options and uses a more classical dive-and-conquer parallelization technique. This makes the codes easier to understand when explaining the main differences between CPU C codes and GPU CUDA codes. It also makes them more efficient for the cases they handle.

The codes we present here were developed by Ruben Spaans as a programming assignment Dr. Elster asked him to developed for her parallel computing course. The codes are based on the Mandelbrot algorithm described on the Wiki http://en.wikipedia.org/wiki/Mandelbrot_set.

C Implementation – see file provided for PS1b

Please also see the PS1b description for useful hints!

Compiling and running the C code we provided for PS1b with, for instance, gcc , will generate a Mandelbrot image very similar to Figure 1. Unlike the SDK sample, the color settings and window size are here hard coded.

WHAT TO DO:

This section outlines how to write a basic skeleton GPU program for calculating the Mandelbrot set based on the algorithm we implemented for the CPU in Section 3.

The skeleton will indicate the 5 classic subtasks for CUDA codes that you need to provide to complete the CUDA program and run it:

1. Create the kernel that performs the calculations.
2. Set up device memory.
3. Execute the kernel on the device.
4. Transfer the result from the device to the host.
5. Free the device memory.

You are to use the skeleton provided and fill out the missing code so that it runs correctly using CUDA.

Like in the serial version, first set the size of the image in x and y. In addition, we need to define BLOCKX and BLOCKY which will be used to specify number of threads used in each direction. These will be used in Task 1. MAXITER set the maximum number of iterations we wish our algorithm should use.

```
/* Problem size */
#define XSIZE 2560
#define YSIZE 2048

/* Divide the problem into blocks of BLOCKX x BLOCKY threads */
#define BLOCKY 32
#define BLOCKX 32
```

For CUDA 3.0 and higher, you can use 32 instead of 8 or 16 for older GPUs. In fact, this is the dimension of the thread block used in the SDK Mandelbrot sample, i.e. it checks for the GPU version (compute capability) and uses 32x32 (1024) for newer GPUs.

Device properties

To reports the GPU device parameters so that they can be used by your program, use:

```
cudaDeviceProp p;
      cudaSetDevice(0);
      cudaGetDeviceProperties (&p, 0);
      printf("Device compute capability: %d.%d\n", p.major,
p.minor);
```

As in the CPU version, set up the rage of the y-axis values so that we can preserve the aspect ratio of our Mandelbrot image.

```
/* Calculate the range in the y-axis such that we preserve the
      aspect ratio */
      step=(xright-xleft)/XSIZE;
      yupper=ycenter+(step*YSIZE)/2;
      ylower=ycenter-(step*YSIZE)
```

Below is the complete listing of mandel_skeleton.cu – the skeleton code for the Mandelbrot programming assignment you will be using for this assignment.

Our code includes both the C CPU (host code), e.g. the host_calculate function is the same as the "calculate" function in the previous PS1b assignment, as well as the skeleton for the 5 subtask that will implement how to do the same calculations in parallel on the GPU in CUDA. These 5 subtasks will be show in the following skeleton.

```
//  mandel_skeleton.cu  by Ruben Spaans and Anne C. Elster  BSD v2 license
//
#include<stdio.h>
#include<stdlib.h>
#include<math.h>
#include<sys/time.h>

/* Problem size */
#define XSIZE 2560
#define YSIZE 2048

/* Divide the problem into blocks of BLOCKX x BLOCKY threads */
#define BLOCKY 32
#define BLOCKX 32
#define MAXITER 255 // you may want to increase this

double xleft=-2.01;
double xright=1;
double yupper,ylower;
double ycenter=1e-6;
double step;
```

```
int host_pixel[XSIZE*YSIZE];
int device_pixel[XSIZE*YSIZE];

typedefstruct {
   double real,imag;
} my_complex_t;

#define PIXEL(i,j) ((i)+(j)*XSIZE)

// ********** SUBTASK1: Create kernel device_calculate *****************/

//Insert code here
// Hint: Use _global_ for the kernal function to be executed on the GPU.
//        Also set up a single grid with a 2D thread block

// ********** SUBTASK1 END ***********************************************/

void host_calculate() {
   for(int j=0;j<YSIZE;j++) {
         for(inti=0;i<XSIZE;i++) {
               /* Calculate the number of iterations until divergence for
each pixel.
                  If divergence never happens, return MAXITER */
               my_complex_tc,z,temp;
               intiter=0;
               c.real = (xleft + step*i);
               c.imag = (yupper - step*j);
               z = c;
               while(z.real*z.real + z.imag*z.imag<4.0) {
                     temp.real = z.real*z.real - z.imag*z.imag + c.real;
                     temp.imag = 2.0*z.real*z.imag + c.imag;
                     z = temp;
                     if(++iter==MAXITER) break;
               }
               host_pixel[PIXEL(i,j)]=iter;
         }
   }
}

typedefunsignedchar uchar;

// save 24-bits bmp file, buffer must be in bmp format: upside-down
void savebmp(char *name,uchar *buffer,intx,int y) {
   FILE *f=fopen(name,"wb");
   if(!f) {
         printf("Error writing image to disk.\n");
         return;
   }
   unsignedint size=x*y*3+54;
   uchar
header[54]={'B','M',size&255,(size>>8)&255,(size>>16)&255,size>>24,0,

   0,0,0,54,0,0,0,40,0,0,0,x&255,x>>8,0,0,y&255,y>>8,0,0,1,0,24,0,0,0,0,0,0
,
         0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0};
   fwrite(header,1,54,f);
   fwrite(buffer,1,x*y*3,f);
   fclose(f);
```

```
}

// given iteration number, set a color
void fancycolour(uchar *p,intiter) {
   if(iter==MAXITER);
   elseif(iter<8) { p[0]=128+iter*16; p[1]=p[2]=0; }
   elseif(iter<24) { p[0]=255; p[1]=p[2]=(iter-8)*16; }
   elseif(iter<160) { p[0]=p[1]=255-(iter-24)*2; p[2]=255; }
   else { p[0]=p[1]=(iter-160)*2; p[2]=255-(iter-160)*2; }
}

// Get system time to microsecond precision
// ostensibly, similar to MPI_Wtime),
//  returns time in seconds

double walltime ( void ) {
   staticstruct timeval t;
   gettimeofday ( &t, NULL );
   return ( t.tv_sec + 1e-6 * t.tv_usec );
}

int main(intargc,char **argv) {
   if(argc==1) {
         puts("Usage: MANDEL n");
         puts("n decides whether image should be written to disk (1=yes,
0=no)");
         return0;
   }
   double start;
   double hosttime=0;
   double devicetime=0;
   double memtime=0;

   cudaDeviceProp p;
   cudaSetDevice(0);
   cudaGetDeviceProperties (&p, 0);
   printf("Device compute capability: %d.%d\n", p.major, p.minor);

   /* Calculate the range in the y-axis such that we preserve the
   aspect ratio */
   step=(xright-xleft)/XSIZE;
   yupper=ycenter+(step*YSIZE)/2;
   ylower=ycenter-(step*YSIZE)/2;

   /* Host calculates image */
   start=walltime();
   host_calculate();
   hosttime+=walltime()-start;



   //********** SUBTASK2: Set up device memory ************************/

   // Insert code here

   /********** SUBTASK2 END ********************************************/
```

```
    start=walltime();

    //********** SUBTASK3: Execute the kernel on the device ************/

    //Insert code here

    //********** SUBTASK3 END ***************************************/

    devicetime+=walltime()-start;

    start=walltime();

    //***** SUBTASK4: Transfer the result from device to device_pixel[][]*/

    //Insert code here

    //********** SUBTASK4 END ***************************************/

    memtime+=walltime()-start;

    /****** SUBTASK5: Free the device memory also ************************/

    //Insert code here

    /********** SUBTASK5 END ***************************************/

    int errors=0;
    /* check if result is correct */
    for(int i=0;i<XSIZE;i++) {
          for(int j=0;j<YSIZE;j++) {
                int diff=host_pixel[PIXEL(i,j)]-device_pixel[PIXEL(i,j)];
                if(diff<0) diff=-diff;
                /* allow +-1 difference */
                if(diff>1) {
                      if(errors<10) printf("Error on pixel %d %d: expected
%d, found %d\n",

    i,j,host_pixel[PIXEL(i,j)],device_pixel[PIXEL(i,j)]);
                      elseif(errors==10) puts("...");
                      errors++;
                }
          }
    }
    if(errors > 0) printf("Found %d errors.\n",errors);
    else puts("Device calculations are correct.");

    printf("\n");
    printf("Host time:          %7.3f ms\n",hosttime*1e3);
    printf("Device calculation: %7.3f ms\n",devicetime*1e3);
    printf("Copy result:        %7.3f ms\n",memtime*1e3);

    if(strtol(argv[1],NULL,10)!=0) {
          /* create nice image from iteration counts. take care to create it
upside
                down (bmp format) */
          unsignedchar *buffer=(unsignedchar *)calloc(XSIZE*YSIZE*3,1);
          for(int i=0;i<XSIZE;i++) {
                for(int j=0;j<YSIZE;j++) {
```

```
                        int p=((YSIZE-j-1)*XSIZE+i)*3;
                        fancycolour(buffer+p,device_pixel[PIXEL(i,j)]);
                    }
                }
                /* write image to disk */
                savebmp("mandel1.bmp",buffer,XSIZE,YSIZE);
        }
        return 0;
}
```

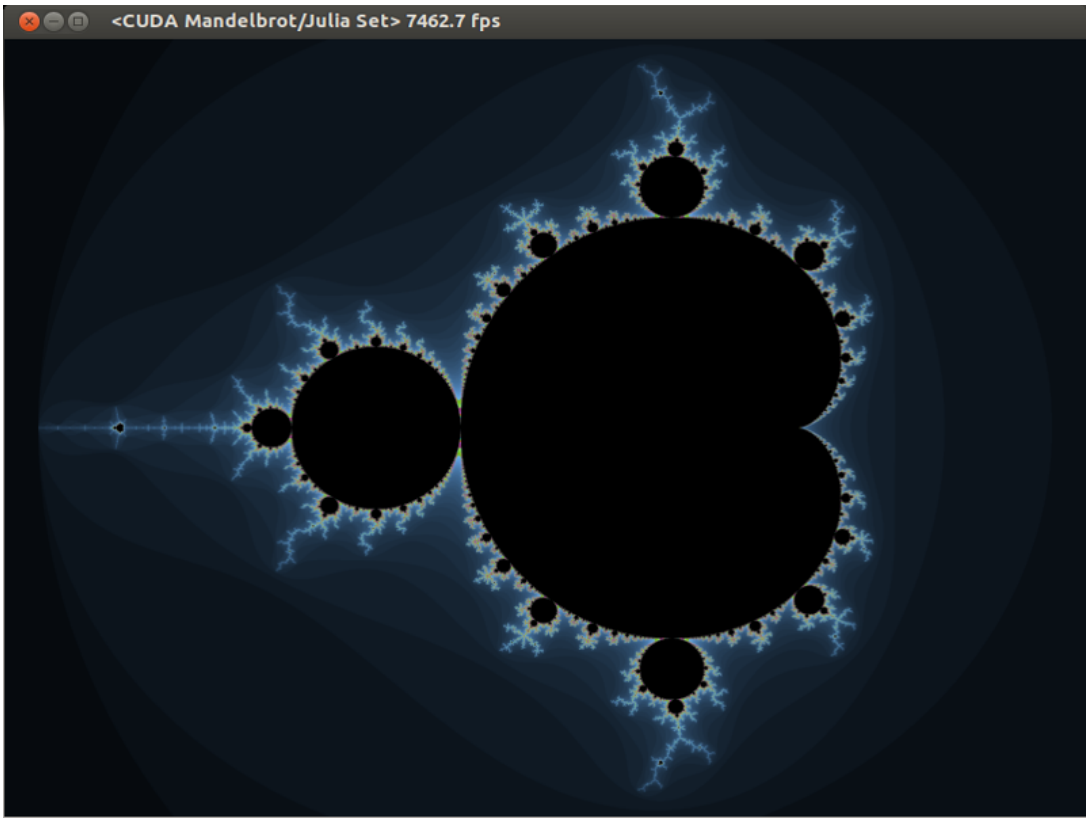


Figure 2 Mandelbrot set [output]

TO HAND IN: Write & Submit the CUDA version

- Provide a CUDA code with the 5 steps as indicated above
- Run your code and check if there is a speed-up.

Questions to be answered in submitted PDF file:

1) Did you get a speed-up? Why or why not?
2) Which GPU did you use? (Info from system).
3) Explain the differences between SIMD and SPMD. Which one is CUDA?