AMS 148 Chapter 2: CPU vs GPU, and CUDA C/C++

Steven Reeves

1 CPU vs GPU

As the title of this class would indicate, we will be using **graphics processing units** (GPUs) to do parallel computing. As per the industry standard we will be performing these calculations on NVIDIA GPUs. In AMS 147 and previous computing classes, you will have most likely only been executing codes using the **central processing unit** (CPU).

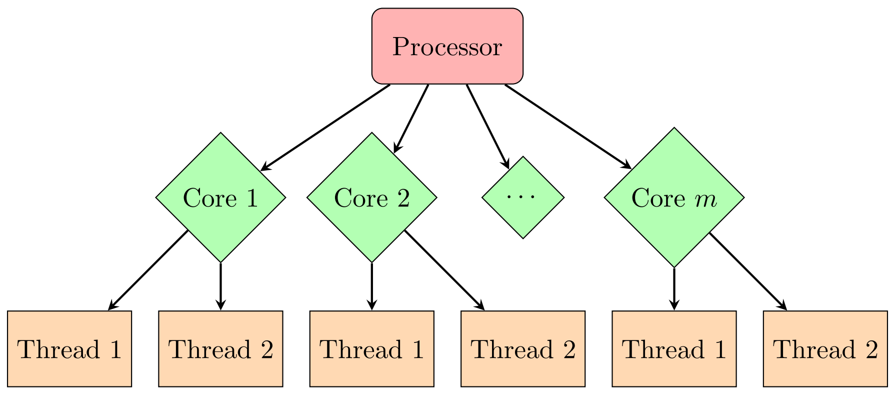
Modern CPUs having many cores are support hyper-threading, the appility to handle two or more processes with one core. The majority of even laptop level CPUs have two more cores within them. So the types of parallel processing discussed in the previous chapter are already being accessed and used within basic applications and the operating system.

In scientific applications both CPUs and GPUs are used for many large scale programs. Supercomputers like NERSC's Cori and Edison at Lawrence Berkeley Lab are massively parallel CPU based machines. However, systems like Oakridge National Laboratory's TITAN and the soon to come SUMMIT, are mostly GPU based in their calculations.

1.1 Central Processing Unit

The central processing unit has been the brain of computer systems since the early days of computing. This chip excutes the operating system, manages memory, and writes to disc space when it needs to. Further, when you launch a program the CPU has direct access to off chip memory. A modern processor will have multiple cores, and each core can handle a number of threads. Figure 1 illustrates the compute heirarchy for CPUs.

Figure 1: Compute Heirarchy in a CPU



A typical Intel CPU supports hyperthreading, where each core can support 2 threads. Some architectures allow for greater hyperthreading, i.e. the Intel Xeon Phi Knight's Landing Chip.

Notice that CPUs have on chip memory, and access to main memory. **Cache** is the "on chip" memory that allows the CPU to make very fast calculations. Generally, a program will have the CPU transfer data from main memory into cache. Data that is being reused should be in cache to offer the faster programs, and the for **vectorization**. Vectorization is a form of data parallelism, where an operation can be applied to the multiple entries in an array rather than one at a time. As CPUs have evolved, the ability for vectorization has increased. The Supercomputer Cori, at Lawrence Berkeley Labs, has the new intel ”Knights Landing” many core CPU. Knights Landing has 32 compute tiles, that contain 2 cores and 2 512-bit vector processing units each. So each core can handle 8 double precision floating point opertaions each cycle. Knights Landing is not the typical CPU. A more commercial CPU comes from the Haswell architecture can handle 4 double precision floating point operations per core, each cycle.

Figure 2 is a simplified diagram of a single-core CPU. A multicore CPU will have more arithmetic logic units (ALUs) and other componenets.

The control unit is an internal portion of the CPU that co-ordinates the instructions and data flow between the CPU and other components of the computer, such as main memory.

The arithmetic logic unit is the internal circuitry of a CPU that performs all the arithmetic and logical operations on a computer. The ALU recieves three types of inputs:

- Control signal from the Control Unit
- Data from memory for operation
- Status information from previous operations

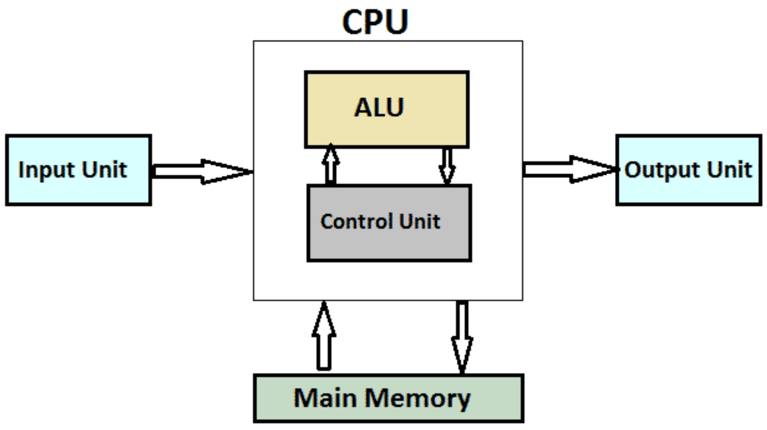


Figure 2: A basic diagram illustrating the function of a CPU, it4nextgen.com.

Processors in general, have a **clock speed**. This clock speed defines how fast the processor handle instructions. Typically today’s CPUs have clock speeds on the order of GigaHertz. That is, a CPU operates on the millions of thousands of cycles per second. Clock speed is limited by a number of factors on processors: the number of transistors, the energy supplied to the system, and the heat tolerance of the system. One major reason for the drive for parallel computing was that increasing the clockspeed of single-core processors was becoming too energy expensive. If we continued to increase clockspeed to reach the performance of the supercomputers today, that super-CPU would need a dedicated Nuclear Reactor. We can calculate the power necessary via the following formula:

$$P = CV^2f$$

where P is the dynamic power consumed by the CPU, C is the capacitance, V voltage, and f is the clock speed frequency.

There are two main ways to do parallelizm on CPUs, through the use of application programming interaces: Open Multi-Processing (OpenMP) or Message Passing Interface (MPI). Essentially, OpenMP assigns threads to the cores in a CPU, and each thread can ”see” the same memory. However, in MPI, memory is segmented into many processes, and data must be passed from one core or processor to another. The most effective CPU based high performance computing codes use a hybrid parallel programming model.

This model combines the use of MPI for nodes on a compute cluster and OpenMP threading for cores within nodes. If you wish to learn more on how to take advantage of CPU based systems and more about CPU parallelization, look into AMS 250: High Performance Computing.

1.2 Graphics Processing Units

Graphics Processing Units (GPUs) are like an extreme version of Figure 1. These types of processors are many-core, and are used as accelerators. That is compute intensive applications are offloaded to GPUs for effeciency.

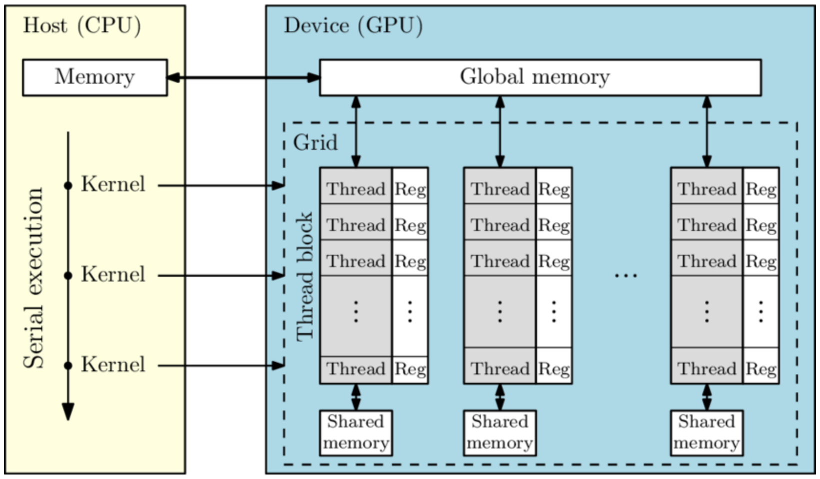
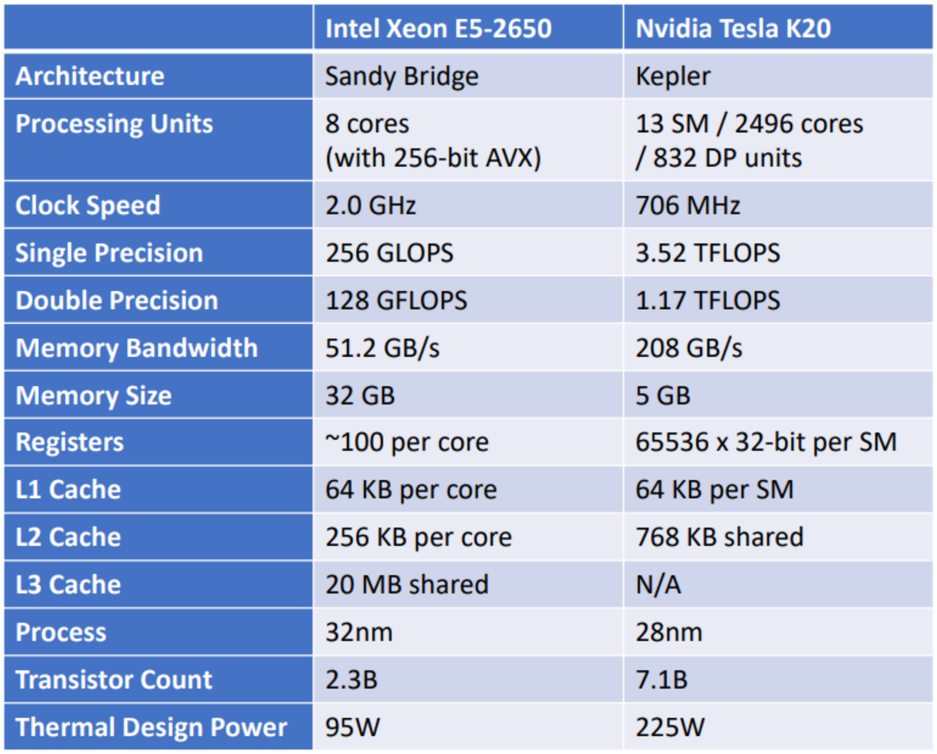


Figure 3: A basic diagram illustrating the function of an Nvidia GPU, An Li et al. 2016.

Figure 3 illustrates the relationship between an Nvidia GPU and the **host** machine. Within this course, we will refer to the CPU as the host, and GPUs as **devices**. An NVIDIA GPU has cores and **streaming multiprocessors** (SMs). These SMs can be thought of as foremen for a group of workers, in this case **CUDA cores**. Like the host system, the GPU has memory. A GPU will have memory of three speeds. **Global memory** is much like main memory on the host, and is the slowest. Any CUDA core can access data in global memory, but will be a significantly slower transaction than with **shared memory**. Shared memory can be thought of as L3 cache, it is memory that is local to an SM. Any CUDA core on that SM can access the shared memory. **Local memory** is the fastest, but is the smallest in space. Local memory is memory that can only be accessed by one CUDA Core, and is used for temporary variables to simplify algorithms. In the diagram, the local memory is refered to as the register, and the CUDA core is expressed as a thread. Later in the course we will disucss how to effectively use the three different on-GPU memory types.

Are GPUs faster than CPUs? For inherently sequential tasks, they aren't. However, due to the massively parallel nature of the architecture, data parallel tasks can be done in unison. Figure 4 is a table showing the difference between a real CPU and a GPU. A useful measurement of compute power is the FLOP (floating point operation per second), a combination of clockspeed, cores, and memory transfer rates. A FLOP is tells us As we can see in the table, the Intel chip has a greater clock speed than the Nvidia card. However, the Nvidia card has a greater performance in FLOPS than the CPU, by an order of magnitude. Something that is about as important as compute power is memory bandwidth, especially for problems involving *Big Data* or *Machine Learning*. We also see the GPU has a larger memory bandwidth, by a factor of 4.



	Intel Xeon E5-2650	Nvidia Tesla K20
Architecture	Sandy Bridge	Kepler
Processing Units	8 cores (with 256-bit AVX)	13 SM / 2496 cores / 832 DP units
Clock Speed	2.0 GHz	706 MHz
Single Precision	256 GLOPS	3.52 TFLOPS
Double Precision	128 GFLOPS	1.17 TFLOPS
Memory Bandwidth	51.2 GB/s	208 GB/s
Memory Size	32 GB	5 GB
Registers	~100 per core	65536 x 32-bit per SM
L1 Cache	64 KB per core	64 KB per SM
L2 Cache	256 KB per core	768 KB shared
L3 Cache	20 MB shared	N/A
Process	32nm	28nm
Transistor Count	2.3B	7.1B
Thermal Design Power	95W	225W

Figure 4: A table illustrating the performance difference between an Intel Xeon CPU and an Nvidia Tesla K20.

2 Compute Unified Device Architecture: CUDA

CUDA is the programming language provided by the NVIDIA corperation to do computational tasks on the GPU. In this course we will focus on CUDA C, with a splash of C++. There are other flavors of CUDA that programmers can harness, such as pyCUDA (CUDA based on python), and CUDA Fortran. CUDA C/C++ and pyCUDA have free to download compilers, however, CUDA Fortran currently only has the PGI compiler. This compiler is not free, however, the local compute cluster Hyades has this compiler. If you collaborate with someone who does astrophysical research, you can get an account on Hyades.

The CUDA C/C++ compiler is NVCC, this is the one we will use for the duration of this course. The Hummingbird cluster, which has a GPU compute node with four NVIDIA P100 GPUs, will be our main source for hardware. If you have an NVIDIA GPU you can also run your codes on your home machine.

2.1 Hummingbird

Hummingbird is a compute cluster that is owned by the University of California Santa Cruz. Every UCSC student has access to this compute server. Hummingbird contains a compute node with 2 NVIDIA P100 GPUs. The P100 is from the Pascal Architecture released in 2016, it is the latest NVIDIA GPU behind the Volta architecture released late last year. Each P100 has 3584 CUDA Cores, with 4.7 TFLOPs of double precision performance and 9.3 TFLOPs of single precision. Further, this processor has 18.7 TFLOPs of half precision performance, illustrating its machine learning proficiency. The P100 has 16 GB of high bandwidth memory, a 732 GB/s transfer rate. The node also has 96 GB of main memory to support the four GPUs.

To login to Hummingbird use the ssh protocol:

```
ssh <username>@hb.ucsc.edu
```

Now for this computer you must use your *blue* password. If you wish to use the Hummingbird cluster at home, you need to download and use the campus vpn, directions found here UCSC VPN.

2.1.1 SLURM

Unfortunately you cannot simply run your programs on the Hummingbird cluster, we must use a batch job script called SLURM. Here are some directions on how to use SLURM.

Script files contain the information that SLURM needs to run your job. In has directives about names of files, how many cpus to use, which queue to run the job on. Script files are text only that is dont use Word and any other word processor to create the files. You can create your script files directly in your home directory on the cluster using: vi, vim, nano etc. Or you can create them on your computer using MacIntosh :TextEdit and Windows: Notepad and then move the file to the cluster via sftp. Most peoples scripts will probably be straightforward, e.g. not parallel or using multiple nodes. There are example scripts located on the cluster in /hb/software/scripts

The following is an example of a script files content.

```
#!/bin/bash      ##Tells the OS which shell to run this script in
#SBATCH --job-name=myname       ##Assigns the name   m y n a m e   to the job
#SBATCH --partition=<partition> default is 128 2 4       ##Tells the job to run in the  <queue> = name of
    queue
#SBATCH --mail-user=cruzid@ucsc.edu     ##Email address to send job-related status (may not work)
#SBATCH --mail-type=ALL
##Specifies which events it should send email about(ALL, BEGIN, END, REQUEUE, FAIL)
#SBATCH --output=myname.out-%j  ##Sends the standard output to the named file
#SBATCH --error=myname.err-%j   ##Sends the standard error to the named file
#SBATCH --mem=1G        ## The amount of memory your program requires per node.
myprogram       ##Your program information goes on the following line
```

For running GPU codes we should have:

```
#!/bin/bash
#note - there can be no line spaces between #SBATCH directives.
#SBATCH --job-name=gpuTest
#SBATCH --output=gpuTest_%j.out
#SBATCH --error=gpuTest_%j.err
#SBATCH --ntasks=1
#SBATCH --cpus-per-task=1
#SBATCH --time=12:00:00
#SBATCH --mail-type=END,FAIL
#SBATCH --mail-user=<cruzid>@ucsc.edu
#SBATCH --partition=96x24gpu4
#SBATCH --gres=gpu:p100:x  ## x = 1,2, 3, 4

module load cuda/cuda-9.1

<Your program goes here>
```

Then to launch you batch script you use

```
sbatch <yourscript>
```

2.2 Getting CUDA on your own machine

If you have a CUDA enabled graphics card, i.e. any NVIDIA GPU, you can do GPGPU on your home machine. To do this follow the steps in the following link: CUDA download.

You may need to sign up for the CUDA developer program, its free and they send you interesting information regarding the state of GPU computing. CUDA is available for Windows, Mac OS, and just about every flavour of Linux.

If you are a Windows user I strongly encourage you to check out the Linux subsystem. Using a Linux shell will make it easier to compile, run and ssh into the various computers. Details and installation instructions: Linux Subsystem for Windows. You can install any flavour of Linux you wish, I find that new linux users have the easiest time with Ubuntu.

2.3 Writing your first CUDA program

When writing in CUDA C/C++, you devide your program in to two parts, host code and device code. The host will manage memory transfer and establishing data, where the device code will have the bulk of the computation. Note that GPU io is not currently supported (without great difficulty), so a "Hello World!" type program will not be our first CUDA one. Instead, let us revist our $c\mathbf{x} + \mathbf{y}$ algorithm from Chapter 1.

CUDA mirrors C/C++ in the initiallization, and main programs. We will call this algorithm SAXPY, for single-precision A*X Plus Y.

Listing 1: My First CUDA C program

```
#include <stdio.h>
#include <cuda.h>

__global__
void saxpy(int n, float a, float *x, float *y)
{
  int i = blockIdx.x*blockDim.x + threadIdx.x;
  if (i < n) y[i] = a*x[i] + y[i];
}

int main(void)
{
  int N = 256;
  float *x, *y, *d_x, *d_y;
  x = (float*)malloc(N*sizeof(float));
  y = (float*)malloc(N*sizeof(float));

  cudaMalloc(&d_x, N*sizeof(float));
  cudaMalloc(&d_y, N*sizeof(float));

  for (int i = 0; i < N; i++) {
    x[i] = 1.0f;
    y[i] = 2.0f;
  }

  cudaMemcpy(d_x, x, N*sizeof(float), cudaMemcpyHostToDevice);
  cudaMemcpy(d_y, y, N*sizeof(float), cudaMemcpyHostToDevice);

  // Perform SAXPY on 1M elements
  saxpy<<<(N+255)/256, 256>>>(N, 2.0f, d_x, d_y);

  cudaMemcpy(y, d_y, N*sizeof(float), cudaMemcpyDeviceToHost);
  cudaDeviceSynchronize();

  float maxError = 0.0f;
  for (int i = 0; i < N; i++)
    maxError = max(maxError, abs(y[i]-4.0f));
  printf("Max error: %f\n", maxError);

  cudaFree(d_x);
  cudaFree(d_y);
  free(x);
  free(y);
}
```

2.3.1 Host Code

The main function declares two pairs of arrays, one for the host and one for the device. As a convenction we denote device arrays with d_"array name". The pointers x and y point to the host arrays, allocates with malloc in the typical fashion. While, the d_x and d_y arrays are allocated with the **cudaMalloc** fuction from the CUDA runtime API. As illustrated before, the host and device have separate memory spaces, both of which can be managed from host code. However, some GPUs can support device code that allocates device memory.

Listing 2: Host Allocation

```
  float *x, *y, *d_x, *d_y;
  x = (float*)malloc(N*sizeof(float));
  y = (float*)malloc(N*sizeof(float));

  cudaMalloc(&d_x, N*sizeof(float));
  cudaMalloc(&d_y, N*sizeof(float));
```

Next the host initializes the data in x and y.

Listing 3: Host Initialization

```
   for (int i = 0; i < N; i++) {
    x[i] = 1.0f;
    y[i] = 2.0f;
  }
```

To initialize the device arrays we must transfer data from the host arrays, using **cudaMemcpy**. This function works like the standard C memcpy function, only that it takes a fourth argument. This fourth arugment dictates the direction of the memory transfer. For the transfer from the host to the device, we use **cudaMemcpyHostToDevice**.

Listing 4: Device Initialization

```
  cudaMemcpy(d_x, x, N*sizeof(float), cudaMemcpyHostToDevice);
  cudaMemcpy(d_y, y, N*sizeof(float), cudaMemcpyHostToDevice);
```

After completing the computation we will use

Listing 5: Result transfer

```
  cudaMemcpy(y, d_y, N*sizeof(float), cudaMemcpyDeviceToHost);
```

.

The function that the host calls to execute computation on the device is called a **kernel**. To launch the saxpy kernel use the statement:

Listing 6: Launching the Kernel

```
saxpy<<<(N+255)/256, 256>>>(N, 2.0, d_x, d_y);
```

. The triple chevrons dictate the **execution configuration**. That is how many device threads (or CUDA Cores) to execute the kernel in parallel. In CUDA there is a hierarchy of threads which mimics how threads processors are grouped on the GPU. The first argument in the execution configureation specifies the number of thread blocks in the grid, and the second specifies the number of threads in the thread block. We will explain this more in detail later.

After the computation we must deallocate memory.

Listing 7: Deallocating memory

```
cudaFree(d_x);
cudaFree(d_y);
free(x);
free(y);
```

Here we use the classic C free for the host arrays, and **cudaFree** for the device arrays.

2.3.2 Device Code

Before we mentioned that the kernel was where the GPU computation happens.

Listing 8: The saxpy kernel

```
__global__
void saxpy(int n, float a, float *x, float *y)
{
  int i = blockIdx.x*blockDim.x + threadIdx.x;
  if (i < n) y[i] = a*x[i] + y[i];
}
```

For device code there are declaration specifiers, for kernels we use the __global__ specifier. Global, lets the compiler know that the code in the function is device code, but can be called from the host code. There are other specifiers, notably __device__, which is a function that can be called within kernels.

In the serial case we loop over an index to do the operation on all entries. In CUDA we create a local index based on the ”location” based on the thread number in the block. Recall we specified the block dimension, and grid dimension in listing 6 while launching the kernel. The `blockIdx.x` term selects the block number within grid, `blockDim.x` is the number of blocks within the grid, and `threadIdx.x` is the local thread within `blockIdx.x`. You can think of this as a mailing address, for example UCSC's address is 1156 High St, Santa Cruz, CA. Here California will be our GPU, Santa Cruz is the block, and 1156 High St is the analogous thread id.

The next step is to perform the computation for threads that are within the range of the array. For performance reasons, most of the time the number of threads allocated will not be the same as the dimensions of the array.

All these components together make a CUDA C/C++ program.

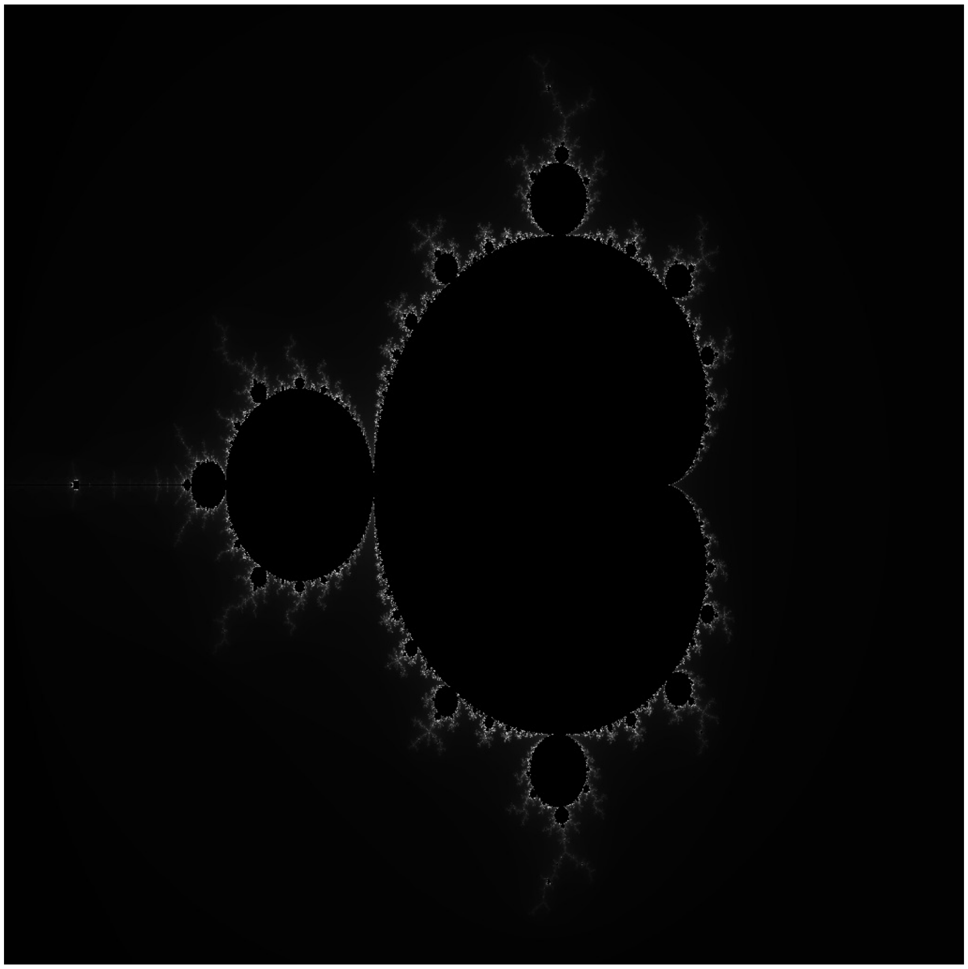


Figure 5: 1024x1024 Mandelbrot set generated with max_iter = 256

3 Parellel Mandelbrot Calculation

The Mandelbrot set is a fun example to demonstrate parallel computing. The Mandelbrot set has a well known fractal structure, and is interesting both mathematically and aesthetically because it has an infinitely recursive structure. You can zoom in to reveal swirls, sprials, snowflakes, and other interesting shapes if you are willing to do enough computation.

3.1 Serial Implementation

This program uses the escape time algorithm. For each pixel in the image, it starts with the x and y position, and then computes a recurrence relation until it exceeds a fixed value or runs for max iterations. At that point, a pixel is assigned a color according to the number of iterations completed. A serial implemention looks like this:

Listing 9: Serial Mandelbrot Rendering

```
void render ( char *out, const int width, const int height , const int max_iter)
{
        float x_origin, y_origin, xtemp, x, y;
        int iteration, index;
        for(int i = 0; i < width; i++)
        {
                for(int j = 0; j < height; j++)
                {
                        index = 3*width*j + i*3;
                        iteration = 0;
                        x = 0.0f;
                        y = 0.0f;
                        x_origin = ((float) i/width)*3.25f -2.0f;
                        y_origin = ((float) j/width)*2.5f - 1.25f;
                        while(x*x + y*y <= 4 && iteration < max_iter)
                        {
                                xtemp = x*x - y*y + x_origin;
                                y = 2*x*y + y_origin;
                                x = xtemp;
                                iteration++;
                        }
                        if(iteration==max_iter)
                        {
                                out[index] = 0;
                                out[index + 1] = 0;
                                out[index + 2] = 0;
                        }
                        else
                        {
                                out[index] = iteration;
                                out[index + 1] = iteration;
                                out[index + 2] = iteration;
                        }
                }
        }
}
```

The max iteration controls the amount of work done by the algorithm. As this increases the image becomes more detailed. However, the resolution is defined by width and height. Notice that each pixel is independent of each other. So this is a good candidate for the GPU. It is important to use a max iteration however, because each thread can do a different ammount of work. We do not want a few threads holding up the execution of the kernel. Since each pixel is independent, we just need to transform the function `render` into a CUDA Kernel.

3.2 CUDA implementation

3.2.1 CUDA Kernel

Let us start by transforming the above function into a CUDA Kernel. Here the for loops will turn to thread indexes.

Listing 10: The Mandelbrot Kernel

```
__global__ void render__global__ void render(char *out, const int width, const int height, const int
    max_iter) {
  int x_dim = blockIdx.x*blockDim.x + threadIdx.x;
  int y_dim = blockIdx.y*blockDim.y + threadIdx.y;
  int index = 3*width*y_dim + x_dim*3;
  float x_origin = ((float) x_dim/width)*3.25 - 2;
  float y_origin = ((float) y_dim/width)*2.5 - 1.25;

  float x = 0.0;
  float y = 0.0;

  int iteration = 0;
  while(x*x + y*y <= 4 && iteration < max_iter) {
    float xtemp = x*x - y*y + x_origin;
    y = 2*x*y + y_origin;
    x = xtemp;
    iteration++;
  }

  if(iteration == max_iter) {
    out[index] = 0;
    out[index + 1] = 0;
    out[index + 2] = 0;
  }
  else {
    out[index] = iteration;
    out[index + 1] = iteration;
    out[index + 2] = iteration;
  }
}
```

3.2.2 Host Code

Now that we have the CUDA kernel let us examine some host code to activate it. Here we will need a char pointer and the CUDA primitives.

```
void mandelbrot(int width, int height, int max_iter)
{
        // Multiply by 3 here, since we need red, green and blue for each pixel
  size_t buffer_size = sizeof(char) * width * height * 3;

        char *d_image; // device image
  cudaMalloc((void **) &d_image, buffer_size); //allocation

  char *host_image = (char *) malloc(buffer_size); // host image

  dim3 blockDim(16, 16, 1);
  dim3 gridDim(width / blockDim.x, height / blockDim.y, 1);
  render<<< gridDim, blockDim>>>(image, width, height, max_iter);

  cudaMemcpy(host_image, d_image, buffer_size, cudaMemcpyDeviceToHost);

  // Now write the file
  write_bmp("output.bmp", width, height, host_image);

        cudaFree(d_image);
  free(host_image);
}
```

Often it is cleaner to separate the CUDA allocation and kernel calls from the main program in a function. Here in `mandelbrot` the host defines the pointer to the character containing the Mandelbrot image. We have structures `dim3` that define the execution configuration, we will discuss this in more detail in the next chapter. Here `render` is our kernel and is executed with with the configuration perscribed. In this case we do not need to transfer data from the host to the device, since the device is generating the data. After the kernel executes we must transfer the data to the host, in order to great the bmp image using the data. In this code I use a bmp header. This code will be available on the course website.

4 Exercises

4.1 Acquire CUDA

Either access and set up an account on one of the compute servers hummingbird, or get the CUDA API on your home computer (if you have an Nvidia GPU).

4.2 Compare and Contrast

4.2.1 Saxpy

Write a serial CPU saxpy in C/C++ and compare run times for serial codes and GPU codes. Use the following sizes for the vectors: $N = 16, 128, 1024, 2048, 65536$.

4.2.2 Mandelbrot

Compare run times for the serial version and the CUDA version for the following resolutions:

Width	Height
1024	1024
2048	2048
4096	4096
8192	8192

using the max iteration as 512.

In the GPU case only, compare clarity of the 8192×8192 version with max iterations $256, 512, 1024, 2048$. You may have to zoom in to see the difference. The majority of the code will be provided, but you must add the timing rouintes. Make sure to include your results in the report.