Optimizing CUDA for GPU Architecture

CSInParallel Project

August 13, 2014

CONTENTS

nVIDIA GPU cards use an advanced architecture to efficiently execute massively parallel programs written in CUDA C. This module explains how to take advantage of this architecture to provide maximum speedup for your CUDA applications using a Mandelbrot set generator as an example. It is intended to be a resource for instructors wishing to create lectures, though it can also be presented to students as is.

Feel free to use and modify the code and images in this module in your lectures.

created by Jeffrey Lyman 2014 JLyman@macalester.edu

CHAPTER

ONE

CUDA ARCHITECTURE

CPUs are designed to process as many sequential instructions as quickly as possible. While most CPUs support threading, creating a thread is usually an expensive operation and high-end CPUs can usually make efficient use of no more than about 12 concurrent threads.

GPUs on the other hand are designed to process a small number of parallel instructions on large sets of data as quickly as possible. For instance, calculating 1 million polygons and determining which to draw on the screen and where. To do this they rely on many slower processors and inexpensive threads.

1.1 Physical Architecture

CUDA-capable GPU cards are composed of one or more **Streaming Multiprocessors** (SMs), which are an abstraction of the underlying hardware. Each SM has a set of **Streaming Processors** (SPs), also called CUDA cores, which share a cache of shared memory that is faster than the GPU's global memory but that can only be accessed by the threads running on the SPs the that SM. These streaming processors are the "cores" that execute instructions.

The numbers of SPs/cores in an SM and the number of SMs depend on your device: see the *Finding your Device Specifications* section below for details. It is important to realize, however, that regardless of GPU model, there are many more CUDA cores in a GPU than in a typical multicore CPU: hundreds or thousands more. For example, the Kepler Streaming Multiprocessor design, dubbed SMX, contains 192 single-precision CUDA cores, 64 double-precision units, 32 special function units, and 32 load/store units. (See the Kepler Architecture Whitepaper for a description and diagram.)

CUDA cores are grouped together to perform instructions in a what nVIDIA has termed a **warp** of threads. Warp simply means a group of threads that are scheduled together to execute the same instructions in lockstep. All CUDA cards to date use a warp size of 32. Each SM has at least one warp scheduler, which is responsible for executing 32 threads. Depending on the model of GPU, the cores may be double or quadruple pumped so that they execute one instruction on two or four threads in as many clock cycles. For instance, Tesla devices use a group of 8 quadpumped cores to execute a single warp. If there are less than 32 threads scheduled in the warp, it will still take as long to execute the instructions.

The CUDA programmer is responsible for ensuring that the threads are being assigned efficiently for code that is designed to run on the GPU. The assignment of threads is done virtually in the code using what is sometimes referred to as a 'tiling' scheme of blocks of threads that form a grid. Programmers define a **kernel function** that will be executed on the CUDA card using a particular tiling scheme.

1.2 Virtual Architecture

When programming in CUDA C we work with blocks of threads and grids of blocks. What is the relationship between this virtual architecture and the CUDA card's physical architecture?

When kernels are launched, each block in a grid is assigned to a Streaming Multiprocessor. This allows threads in a block to use `__shared__` memory. If a block doesn't use the full resources of the SM then multiple blocks may be assigned at once. If all of the SMs are busy then the extra blocks will have to wait until a SM becomes free.

Once a block is assigned to an SM, it's threads are split into warps by the warp scheduler and executed on the CUDA cores. Since the same instructions are executed on each thread in the warp simultaneously it's generally a bad idea to have conditionals in kernel code. This type of code is sometimes called *divergent*: when some threads in a warp are unable to execute the same instruction as other threads in a warp, those threads are diverged and do no work.

Because a warp's context (it's registers, program counter etc.) stays on chip for the life of the warp, there is no additional cost to switching between warps vs executing the next step of a given warp. This allows the GPU to switch to hide some of it's memory latency by switching to a new warp while it waits for a costly read.

1.3 CUDA Memory

CUDA on chip memory is divided into several different regions

- **Registers act the same way that registers on CPUs do, each** thread has it's own set of registers.
- **Local Memory local variables used by each thread. They are** not accessible by other threads even though they use the same L1 and L2 cache as global memory.
- **Shared Memory is accessible by all threads in a block. It** must be declared using the `__shared__` modifier. It has a higher bandwidth and lower latency than global memory. However, if multiple threads request the same address, the requests are processed serially, which slows down the application.
- **Constant Memory is read-accessible by all threads and must** be declared with the `__const__` modifier. In newer devices there is a separate read only constant cache.
- **Global Memory is accessible by all threads. It's the** slowest device memory, but on new cards, it is cached. Memory is pulled in 32, 64, or 128 byte memory transactions. Warps executing global memory accesses attempt to pull all the data from global memory simultaneously therefore it's advantageous to use block sizes that are multiples of 32. If multidimensional arrays are used, it's also advantageous to have the bounds padded so that they are multiples of 32
- **Texture/Surface Memory is read-accesible by all threads, but** unlike Constant Memory, it is optimized for 2D spacial locality, and cache hits pull in surrounding values in both x and y directions.

1.4 Finding your Device Specifications

nVIDIA provides a program with the installation of the CUDA developer toolkit that prints out the specifications of your device. To run it on a unix machine, execute this command:

```
/usr/local/cuda/samples/1_Utilities/deviceQuery/deviceQuery
```

If that doesn't work you probably need to build the samples:

```
cd /usr/local/cuda/samples/1_Utilities/deviceQuery
sudo make
./deviceQuery
```

Look for the number of Multiprocessors on your device, the number of CUDA cores per SM, and the warp size.

The CUDA Toolkit with the samples is also available for Windows using Visual studio. See the excellent and thorough Getting Started Guide for Windows provided by nVIDIA for more information. However, some of the code described in the next section uses X11 calls for its graphical display, which will not easily run in Windows. You will need a package like Cygwin/X.

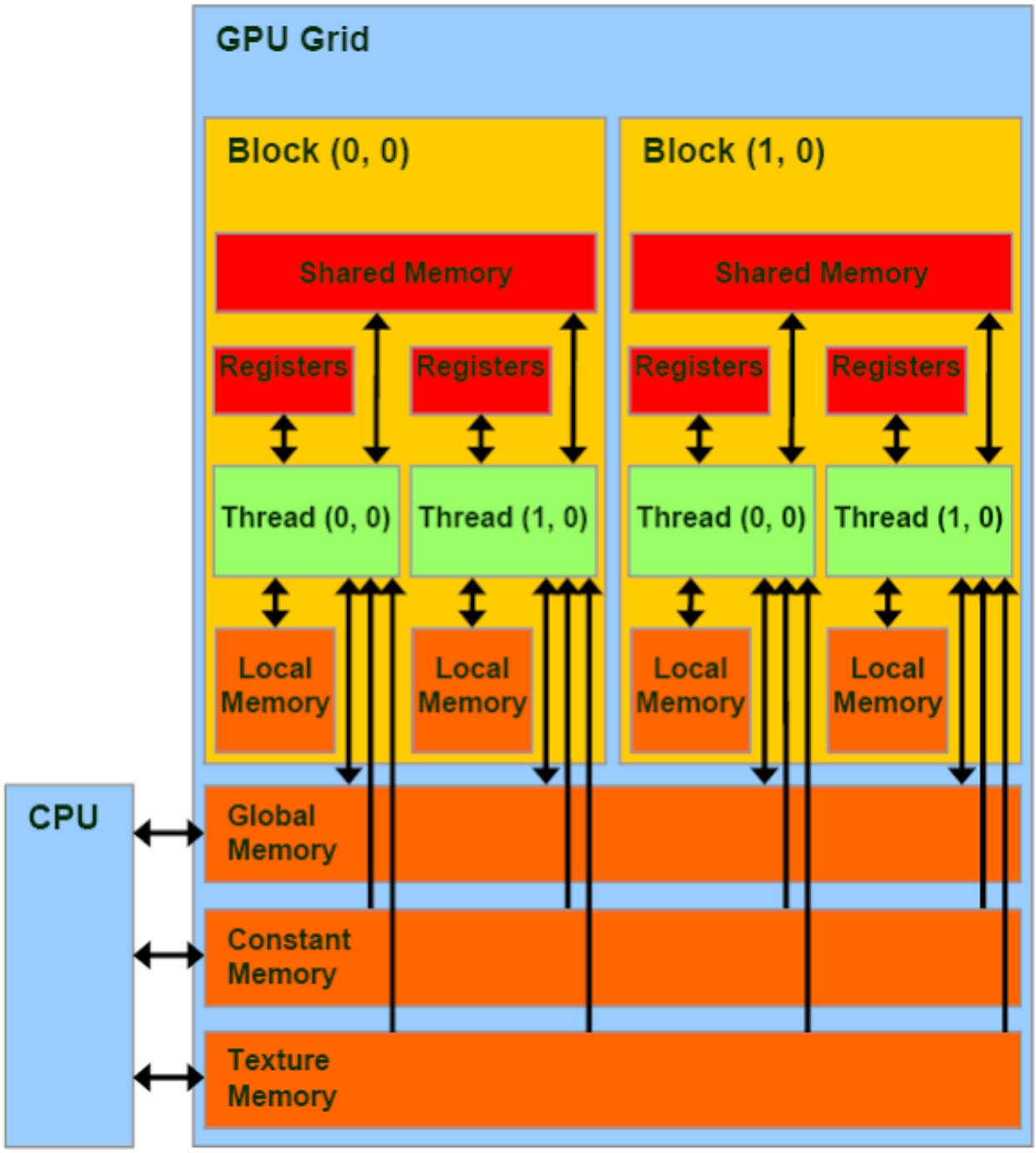


Figure 1.1: *CUDA Memory Hierarchy Image courtesy of NVIDIA*

CHAPTER

TWO

MANDELBROT TEST CODE

Choosing a good number of blocks and threads per block is an important part of CUDA Programming. To illustrate this, we will take a look at a program that generates images of the Mandelbrot set. To run the programs you will need a CUDA capable machine as well as the appropriate XOrg developer package (X11 is likely installed on your linux machine and needs to be installed on a Mac). Download `mandelbrot.cu` and the `Makefile` and run `make all` This will generate 3 programs:

Mandelbrot is a Mandelbrot set viewer designed for demonstrations . It allows you to zoom in and out and move around the Mandelbrot set. The controls are w for up, s for down, a for left, d for right, e to zoom in, q to zoom out and x to exit.

The executable named **benchmark** runs the computation without displaying anything and prints out the time it took before exiting.

XBenchmark is a hybrid that prints out the computation time and allows you to move around. This is useful because the computation time is dependent on your position within the Mandelbrot set.

Each of the programs takes between 0 and 4 commandline arguments

1. the number of blocks used by the kernel
2. the number of threads per block
3. the image size in pixels, the image is always square
4. the image depth (explained later)

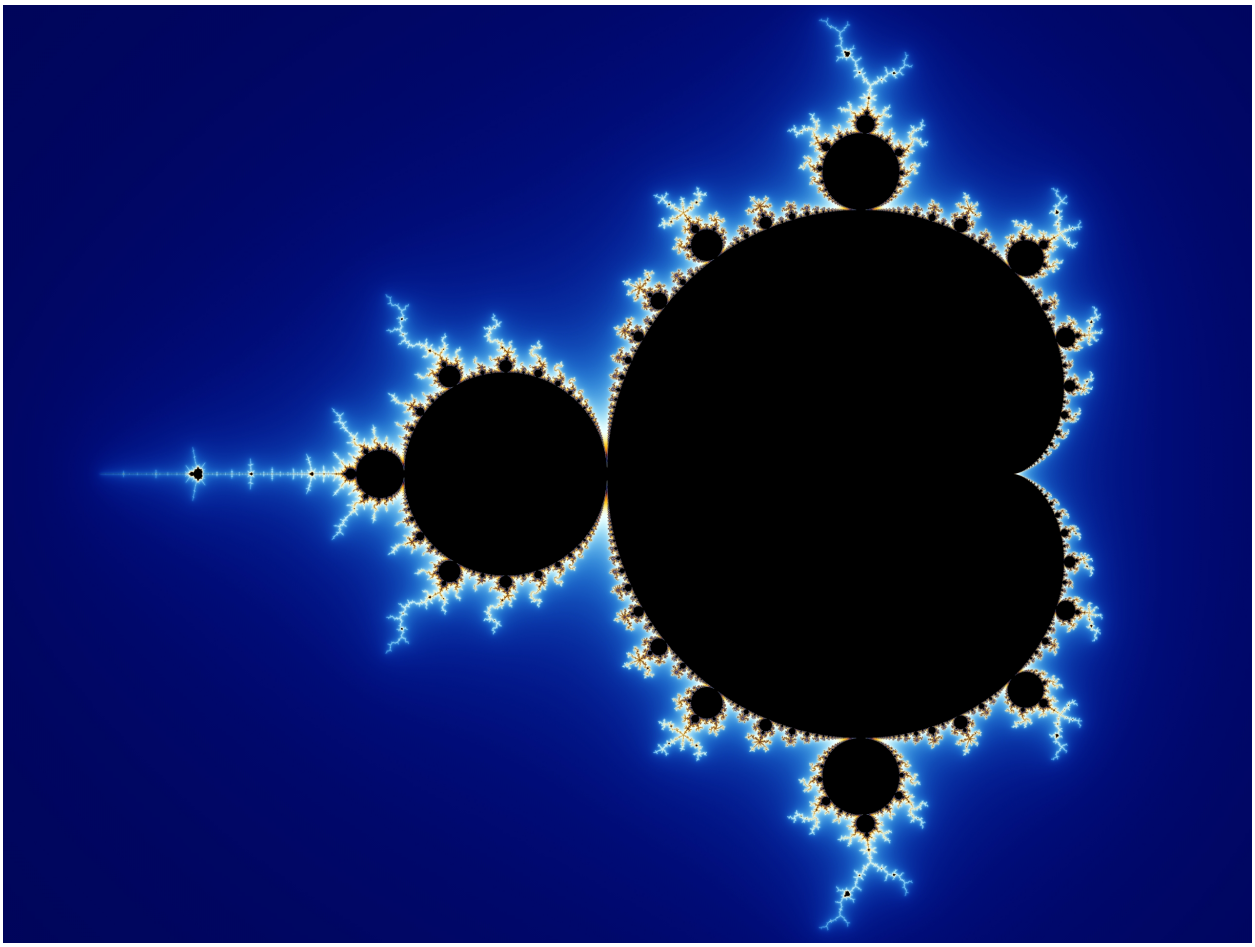
2.1 What is the Mandelbrot set?

The Mandelbrot set is defined as the set of all complex numbers C such that the formula $Z_{n+1} = Z_n^2 + C$ tends towards infinity. If we plot the real values of C on the X axis and the imaginary values of C on the Y axis we get a two dimensional fractal shape, such as this one created from running this code.

2.2 Coding the Mandelbrot set

The to determine whether a value is in or out of the Mandelbrot set we loop through the formula $Z_{n+1} = Z_n^2 + C$ a certain number of times (this is the image depth from earlier) and during each iteration, check if the magnitude of Z is greater than 2; if so, we return false. However we want our Mandelbrot image to look pretty, so instead we'll return the iteration in which it went out of bounds and and then interpret that number as a color. If it completes the loop without going out of bounds we'll assign it the color black

After some algebraic manipulation to reduce the number of floating point multiplications, our code looks like this:



```
__device__ uint32_t mandel_double(double cr, double ci, int max_iter) {
    double zr = 0;
    double zi = 0;
    double zrsqr = 0;
    double zisqr = 0;

    uint32_t i;

    for (i = 0; i < max_iter; i++){
                zi = zr * zi;
                zi += zi;
                zi += ci;
                zr = zrsqr - zisqr + cr;
                zrsqr = zr * zr;
                zisqr = zi * zi;

    //the fewer iterations it takes to diverge, the farther from the set
                if (zrsqr + zisqr > 4.0) break;
    }

    return i;
}
```

Some notes about this CUDA code

- A function that is designed to be run on the GPU is designated with the special keyword *__device__*.
- The type uint32_t is an unsigned 32-bit integer declared in stdint.h.
- The variable max_iter is defaulted to be 100, and can be changed with the image depth command line argument.

But wait didn't we say in the last chapter that conditionals should be avoided? Yes, when a thread returns early, it's just dead weight in the warp, however due to the nature of the Mandelbrot set it is very likely that some warps have threads that all terminate before reaching `max_iter` so in some cases it will give us a slight speed up. If the warp contains a point within the Mandelbrot set, we won't get any speed up from breaking.

We also need a kernel that will divide the pixels between the threads and run `mandel_double` on each of them Our code is as follows where `dim` is the image dimension, `counts` is the list representing our image, and `step` represents the distance between the points represented by the pixels:

```
__global__ void mandel_kernel(uint32_t *counts, double xmin, double ymin,
             double step, int max_iter, int dim, uint32_t *colors) {
    int pix_per_thread = dim * dim / (gridDim.x * blockDim.x);
    int tId = blockDim.x * blockIdx.x + threadIdx.x;
    int offset = pix_per_thread * tId;
    for (int i = offset; i < offset + pix_per_thread; i++){
        int x = i % dim;
        int y = i / dim;
        double cr = xmin + x * step;
        double ci = ymin + y * step;
        counts[y * dim + x]  = colors[mandel_double(cr, ci, max_iter)];
    }
    if (gridDim.x * blockDim.x * pix_per_thread < dim * dim
            && tId < (dim * dim) - (blockDim.x * gridDim.x)){
        int i = blockDim.x * gridDim.x * pix_per_thread + tId;
        int x = i % dim;
        int y = i / dim;
        double cr = xmin + x * step;
```

```
        double ci = ymin + y * step;
        counts[y * dim + x]  = colors[mandel_double(cr, ci, max_iter)];
    }

}
```

Some notes about this CUDA code

- The keyword *__global__* designates the kernel function.
- We execute the kernel function on the GPU device from main() like this, where *n* is the number of blocks of threads and 'm' is the number of threads per block:

```
mandel_kernel<<<n, m>>>(dev_counts, xmin , ymin, step, max_iter, dim, colors);
```

- In this case, the 'tiling' of the blocks of threads into a grid is a one-dimensional array of n blocks.
- Each thread calculates a particular value in the set based on its thread id (tId in the above code), which can be calculated using a data structure called *blockDim*, along with ones called *blockIdx* and *threadIdx*. The value blockDim.x gives us the total number of threads per block. The blockIdx.x value gives us the index of the block in which a particular thread running this code is located. Lastly, the threadIdx.x value is the index of this thread in its block. Thus, a thread running this code can uniquely identify itself with the computation *blockDim.x * blockIdx.x + threadIdx.x*.
- We use blockDim.x when calculating the thread id above so that we could change the number of blocks, n, and the number of threads per block, m, programatically with command-line arguments and not have to change the kernel function.

In order to compensate for block and grid dimensions that do not easily divide the picture we make the first threads pick up the 'slack.' This is also the reason why we are not using 2 dimensional grids and blocks.

Warning: Always try to make your threads do the same amount of work. Scheduling extra work for some threads is inefficient since the other threads in the warp will have to wait for them to finish anyway. This code is purposefully messy so that it runs for any problem size.

That's the meat of the program, feel free to explore the it on your own, most of the rest of the program is dedicated to displaying the data generated by these 2 functions.

In the next section, we will discuss how to choose the number of blocks and the number of threads per block in order to take maximum advantage of the GPU hardware.

CHAPTER

THREE

CHOOSING THE RIGHT DIMENSIONS

Compute Capability

The compute capability of a CUDA card designates what features are available. The Wikipedia CUDA page provides an overview of various cards and their compute capability, along with the features available with that compute capability.

One of the most important elements of CUDA programming is choosing the right grid and block dimensions for the problem size. Early CUDA cards, up through compute capability 1.3, had a maximum of 512 threads per block and 65535 blocks in a single 1-dimensional grid (recall we set up a 1-D grid in this code). In later cards, these values increased to 1024 threads per block and 2^{31} - 1 blocks in a grid.

It's not always clear which dimensions to choose so we created an experiment to answer the following question:

What effect do the grid and block dimensions have on execution times?

To answer this question, we wrote a `script` to run our Mandelbrot code for every grid size between 1 and 512 blocks and every number of threads per block between 1 and 512 which produced 262,144 data points. We chose these ranges because our Mandelbrot set picture is 512x512, so each thread will calculate the value of at least one pixel at the largest value of each.

The device we ran the tests on was a Jetson TK1 which is a Kepler class card that has one Streaming Multiprocessor with 192 CUDA cores. To ensure that our code was the only thing running on the GPU, we first disabled the X server.

3.1 Results

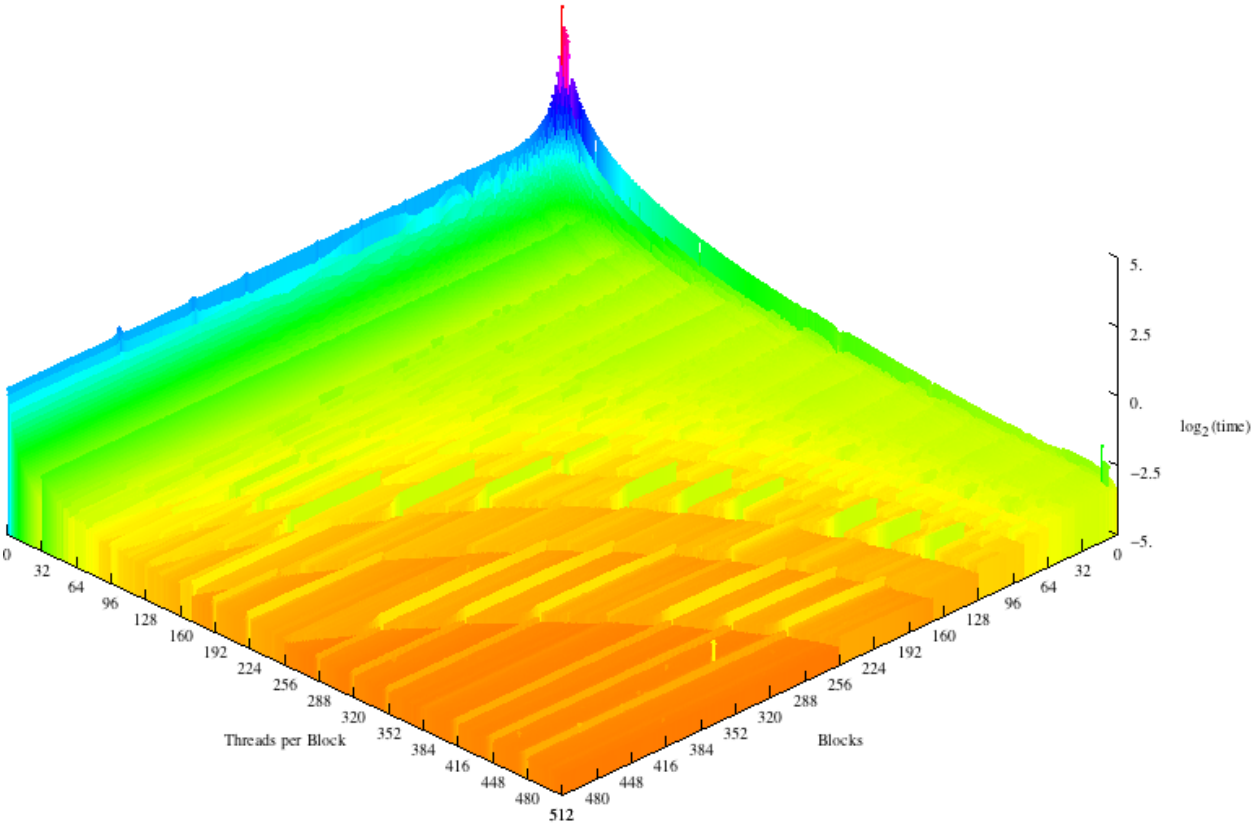
This is a 3D graph of our `results` where the z axis is the $\log_2$(time) we took the log so that all results fit neatly on the graph.

There are a number of interesting things to note about this graph:

- Trials with one block and many threads are faster than trials with many blocks of one thread each.
- There are horizontal lines indicating a spike in execution time after every 32 threads per block
- 512 threads per 512 blocks was the fastest execution time
- There are convex lines running through the middle of the graph

Each of these observations relates directly to CUDA's architecture or the specifics of the code.

Many threads in 1 block is always faster than many blocks of one thread because of the way threads are put into warps. The Jetson can execute 4 warps simultaneously. This means that when the number of threads/block is one only 4



threads can run concurrently but when the number of blocks is one and there are many threads per block, the threads can be evenly divided into warps so that up to 128 are being run simultaneously.

Warp size also explains the horizontal lines every 32 threads per block. When block are are evenly divisible into warps of 32, each block uses the full resources of the CUDA cores on which it is run, but when there are (32 * x) + 1 threads, a whole new warp must be scheduled for a single thread which wastes 31 cycles cycles per block.

512x512 is the fastest execution time even though the GPU can't run that many threads at a time. This is because it is inexpensive to create threads on a CUDA card and having one pixel per thread allows the GPU to most efficiently schedule warps as the CUDA cores become free. Additionally, since accessing the color data takes time, the GPU can help us out by calculating other warps while waiting for the read to finish.

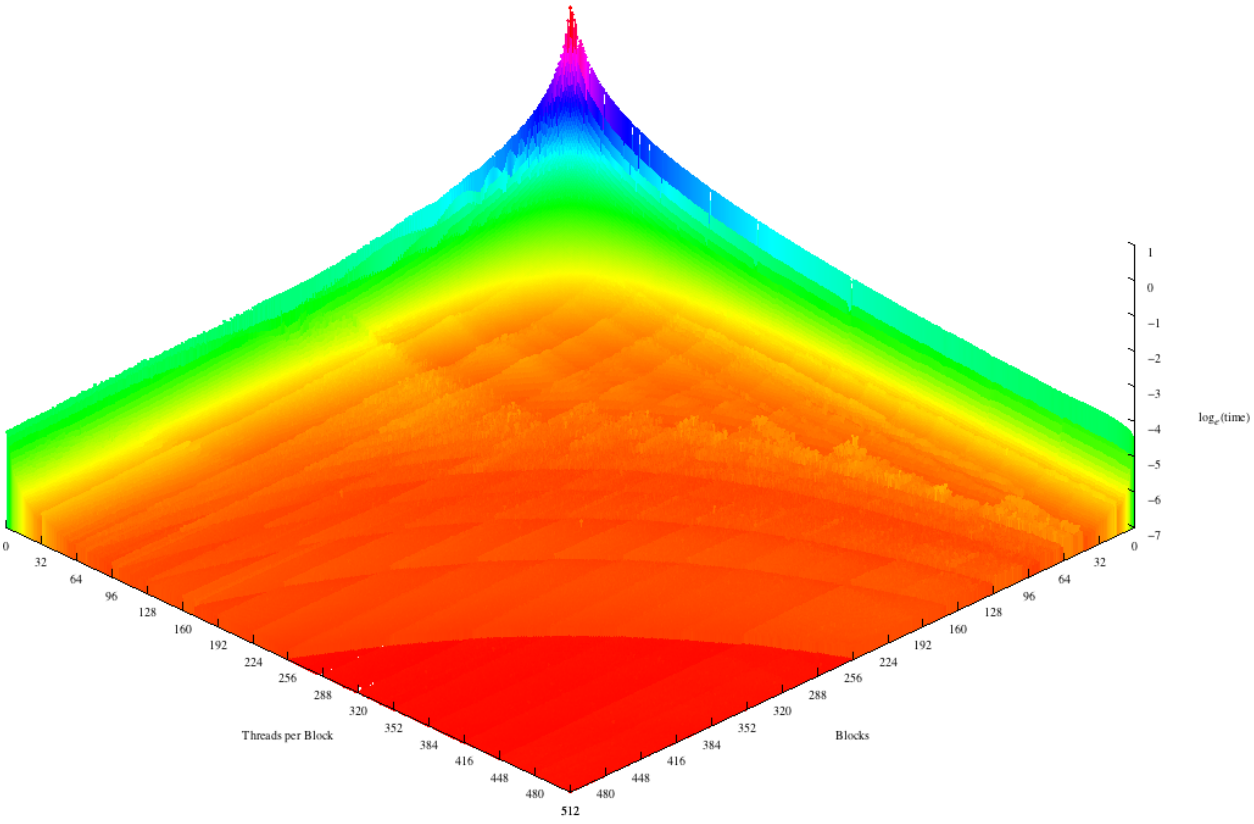
The convex lines appear for a few different reasons. The first has to do with our code. When the picture is evenly divisible by the total number of threads and blocks, each thread performs the same amount of work so the warps aren't bogged down by the threads that calculate the extra pixels. Second, when block and grid dimensions are about roughly equal, the block and warp schedulers share the work of dividing the threads.

3.2 Adding More Streaming Multiprocessors

We executed our code again on a GeForce GTX 480 card that has 15 SMs with 32 CUDA cores each.

This graph also features horizontal lines at multiples of 32 corresponding to the warp size, concave lines, and a top execution speed at 512x512. However there are 2 important differences.

First, one block of many threads and many blocks with one thread each take about the same amount of time to execute. Because this card uses the Fermi architecture, each SM can run two warps concurrently, this means that 64 threads



can be running at any given time. While still not as fast as using one block, many blocks is significantly faster with multiple SMs.

The second difference is a series of valleys running perpendicular to the warp lines about every 15 blocks. These valleys come from the way blocks are distributed between the SMs. When the block size is a multiple of the number of SMs, each processor will do the about same amount of work. However, as the number of blocks increases this difference becomes less and less important because the blocks don't all take the same amount of time to execute and so it's possible for three blocks to execute on one SM in the time it takes for another to execute 2.

3.3 CUDA best practices

From these results we can draw up a list of best practices:

1. Try to make the number of threads per block a multiple of 32.
2. Keep the number of threads per block and the number of blocks as close to equal as you can without violating the first tip.
3. Keep the amount of work each thread does constant, it's inefficient to have one thread perform calculations for two pixels while the rest only calculate one.
4. When in doubt use more threads not less, creating threads is inexpensive.
5. In general avoid having threads that do extra work or have conditionals.
6. Try to have a block size that is a multiple of the number of SMs on your device, this is less important than the other tips.