Timing Cuda Operations

CSInParallel Project

July 21, 2014

CONTENTS

This module was created by Joel Adams of Calvin College and extended and adapted for CSInParallel by Jeffrey Lyman in 2014 (JLyman@macalester.edu)

The purpose of this document is to teach students the basics of CUDA programming and to give them an understanding of when it is appropriate to offload work to the GPU.

Through completion of Vector Addition, multipliction, square root, and squaring programs, students will gain an understanding of when the overhead of creating threads and copying memory is worth the speedup of GPU coding.

Prerequisites

- Some knowledge of C coding and using makefiles.
- An ability to create directories and use the command line in unix.
- Access to a computer with a reasonably capable GPU card.

Contents

This activity contains three parts, linked below. First there is a short introduction to setting up code in CUDA to run on a GPU. Then you will try running vector addition code on your GPU machine. Lastly, you will experiment with various types of operations and large sizes of arrays to determine when it is worthwhile to use a GPU for general-purpose computing.

CHAPTER

ONE

PROGRAMMING IN CUDA

1.1 Introduction

Modern GPUs are composed of many cores capable of computing hundreds of threads simultaneously. In this module we will use the CUDA platform from nVIDIA to give our code access to the massive computing power of these cards.

1.2 What is CUDA?

CUDA is a free proprietary platform designed by NVIDIA specifically for NVIDIA devices. It allows us to transfer memory between the GPU and CPU and run code on both processors. CUDA code is written in an extended version of C and CUDA files have the prefix `.cu` They are compiled by NVIDIA's *nvcc* compiler.

1.3 Challenges

CUDA programming is fundamentally different from regular programming because the code is run on two seperate processors, the host CPU and the device GPU. This makes coding more difficult because

- The GPU and CPU don't share memory
- The GPU code can't be run on the CPU and visa versa

Let's look at how CUDA works around these limitations.

1.4 Memory management in CUDA

1.4.1 CUDA 6 Unified Memory

Warning: This section is for CUDA 6 only. The following methods won't work on all devices. You must use Windows or Linux and have a device with compute capability >= 3.0.
Linux: To find the compute capability of your device, run
`/usr/local/cuda/samples/1_Utilities/deviceQuery/deviceQuery`
and look at the CUDA capablity line. If that command doesn't work you may have to build the code:

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

GPUs have their own dedicated RAM that is seperate from the RAM the CPU can use. If we want both the CPU and GPU to be able to access a value we must tell our program to allocate unified memory.

Note: There is no physical unified memory, but instead CUDA unified memory

---

is simply a convenient abstraction for programmers. Underneath, the system moves data from host to device and back, and the code running on the CPU accesses the CPU memory, while the code running on the GPU accesses the GPU memory.

We do this with the `cudaMallocManaged()` function. It works nearly identically to malloc but we have to pass in an address to a pointer as well as the size because cudaMallocManaged returns a cudaError_t instead of a pointer.

Ex: `cudaMallocManaged((void**)&ptr, SIZE * sizeof(int))`

1.4.2 Using cudaMalloc and cudaMemcpy

Some older devices don't support unified memory. In addition, it can be advantageous to manage the memory on the CPU and the GPU in your programs. To accomplish this, you use `cudaMalloc()` and `cudaMemcpy()` to allocate and transfer memory. `cudaMalloc()` is very similar to `cudaMallocManaged()` and takes the same arguments. However, the CPU code will not be able to access memory allocated this way.

As shown in the following code segment, to transfer memory between the devices we use `cudaMemcpy()`, which takes a pointer to the destination, a pointer to the source, a size, and a fourth value representing the direction of the data flow. This last value should be `cudaMemcpyDeviceToHost` or `cudaMemcpyHostToDevice` Once you're done with memory allocated using either method you can free it by calling `cudaFree()` on the pointer.

```
1  int *ptr, *dev_ptr;
2  initialize_ptr(ptr);     //creating the block of SIZE data values of type int
3                           // in memory on CPU, pointed to by ptr (not shown)
4  cudaMalloc((void**)&ptr, SIZE * sizeof(int));
5  cudaMemcpy(dev_ptr, ptr, SIZE * sizeof(int), cudaMemcpyHostToDevice);
6
7          ...Perform GPU Operations ...
8
9  cudaMemcpy(ptr, dev_ptr, SIZE * sizeof(int), cudaMemcpyDeviceToHost);
10 cudaFree(dev_ptr);
11 free(ptr);
```

It's usually a good idea to use `cudaMalloc()` and `cudaMemcpy()` instead of `cudaMallocManaged()` unless you need to do deep copies on nested structs like linked lists. You will see why later on in the lab.

1.5 Host Code vs. Device Code

Because CPU code and GPU code use different instruction sets, we must tell the compiler whether our functions will run on the CPU or the GPU. We do this with three new modifiers.

1. `__global__` functions run on the GPU and can be called anywhere in the program. These functions are called kernels because they contain the information threads used to create threads.
2. `__device__` functions run on the GPU and can only be called by `__global__` and other `__device__` methods. They tend to be helper methods called by threads.
3. `__host__` functions are run on the CPU and can only be called by other `__host__` methods.

If you don't add one of these modifiers to a function definition the compiler assumes it's a `__host__` function. It's also possible for a function to be both `__host__` and `__device__` this is useful because it allows you to test GPU functions on the CPU.

1.6 Threads

CUDA splits it's treads into three dimensional blocks which are arranged into a two dimensional grid. Threads in the same block all have access to a local shared memory which is faster than the GPU's global memory.

CUDA provides a handy type, `dim3` to keep track of these dimensions. You can declare dimensions like this: `dim3 myDimensions(1,2,3);`, signifying the ranges on each dimension. Both blocks and grids use this type even though grids are 2D. To use a `dim3` as a grid dimension, leave out the last argument or set it to one. Each device has it's own limit on the dimensions of blocks. Run

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

to find the limits for your device.

1.7 Kernels

CUDA threads are created by functions called kernels which must be `__global__`. Kernels are launched with an extra set of parameters enclosed by <<< and >>> the first argument is a `dim3` representing the grid dimensions and the second is another `dim3` representing the block dimensions. You can also use `int`s instead of `dim3`s, this will create a Nx1x1 grid. After a kernel is launched, it creates the number of threads specified and runs each of them. CUDA automatically waits for the devices to finish before you can access memory using `cudaMemcpy()` although if you're using unified memory with `cudaMallocManaged()` you will need to call `cudaDeviceSynchronize()` to force the CPU to wait for the GPU.

```
dim3 numBlocks(8,8);
dim3 threadsPerBlock(8,8,8);
myKernel<<<numBlocks, threadsPerBlock>>>(args);
myKernel<<<16,64>>>(args);
```

Kernels have access to 4 variables that give information about a thread's location in the grid

1. `threadIdx.[xyz]` represents a thread's index along the given dimension.
2. `blockIdx.[xy]` represents a thread's block's index along the given dimension.

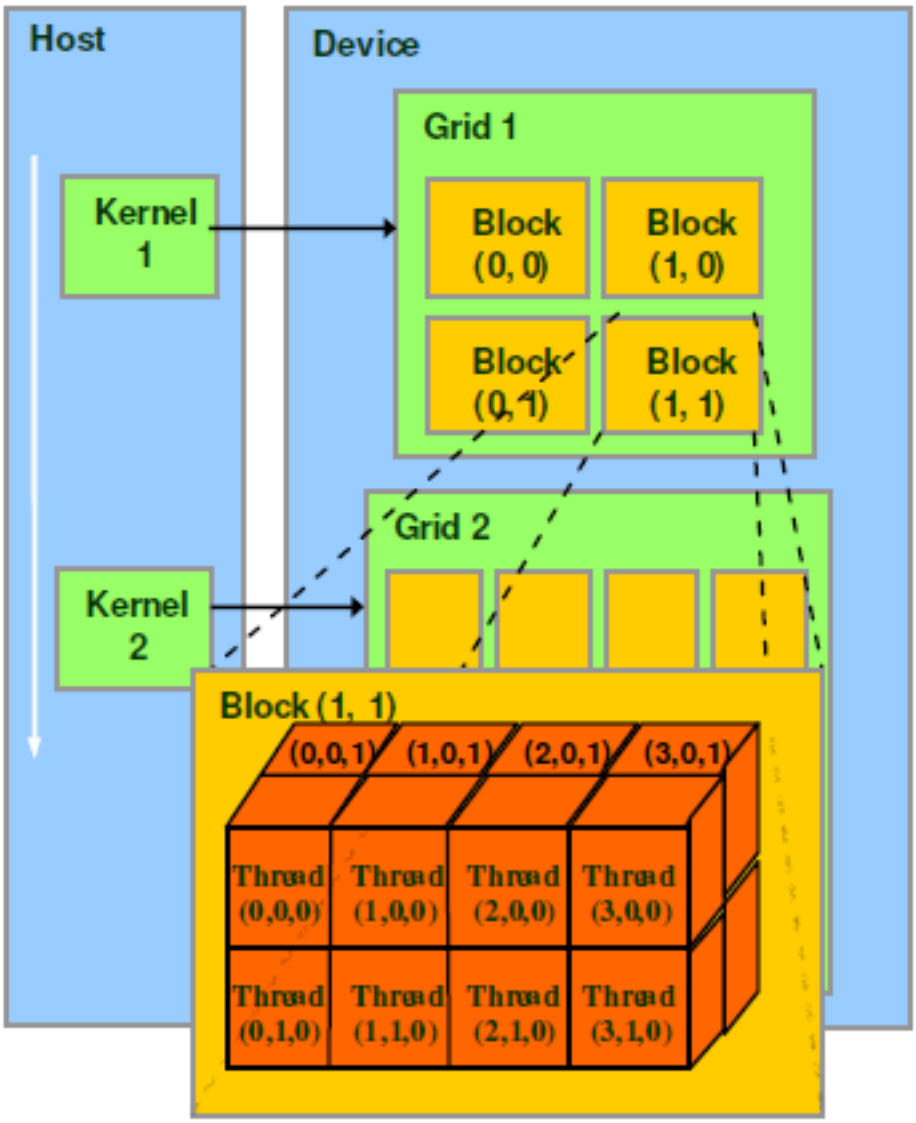

Figure 1.1: Image from North Carolina State University

3. `blockDim.[xyz]` represents the number of threads per block in the given direction.
4. `gridDim.[xy]` represents the number of blocks in the given direction.

By using these variables we can create a unique id for each thread indexed from 0 to N where N is the total number of threads. For a one dimensional grid and a one dimesional block this formula is `blockIdx.x * blockDim.x + threadIdx.x`

1.8 Compiling

CUDA code is compiled with NVIDIA's own compiler nvcc. You can still use makefiles like you do with regular c. To make sure your code is taking full advantage of your device's capabilities use the flag `-gencode arch=compute_XX,code=sm_XX` you can find the correct values of the Xs by running

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

and using at the output of the CUDA capability line without the period.

CHAPTER

TWO

VECTOR ADDITION LAB

Warning: The Unified Memory parts of this lab may not work on your machine. Run `/usr/local/cuda/samples/1_Utilities/deviceQuery/deviceQuery` and check that your device's CUDA capability is >= 3.0 If the command gives you an error, you may need to compile the samples like so

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

2.1 Research Questions

- For what size problem is the CUDA computation faster than the sequential computation?
- What effect does using Unified Memory have on the speed of our program?

2.2 Getting started

Download and extract `vectorAdd.tar.gz` and extract it with `tar -xzvf vectorAdd.tar.gz` Open vectorAdd.cu and vectorAdd6.cu and familiarize yourself with the code. Compile and run the programs.

Note: don't compile vectorAdd6.cu if your machine is incompatable

Run the programs and see what happens.

2.3 Exercise

2.3.1 Part One: cudaMemcpy

Using omp_get_wtime() modify vectorAdd.cu so that it reports

1. The time required by the CUDA computation specifically
   (a) The time spent allocating A, B, and C
   (b) The time spent copying A and B from the Host to the device

(c) The time spent computing the sum of A and B into C

(d) The time spent copying C from the device to the host

(e) The total time of the CUDA computation (i.e., the sum of a-d)

2. The time required by the sequential computation

None of these times should include any I/O so make sure you comment out the printf() statements.

Use the Makefile to build your modified version of the program. When it compiles successfully run it as follows:

```
./vectorAdd
```

The program's default array size is 50,000 elements

In a spreadsheet record and label your times in a column labeld 50,000. Which is faster, the CUDA version or the CPU version?

Repeat this problem with a larger array. Run it again with 500,000 elements.

```
./vectorAdd 500000
```

Record your results. Repeat the process again wih 5,000,000 elements, 50,000,000 and 500,000,000 elements. How do these times compare? Were you able to run all of them succesfully? If not why?

Create a line chart, with one line for the sequential code's times and one line for the CUDA code's total times. Your X-axis should be labeled with 50,000 500,000 5,000,000 and 50,000,000 your Y-axis should be the time.

Then create a "stacked" barchart of the CUDA times with the same X and Y axes as your first chart.. For each X-axis value, this chart should "stack" the CUDA computation's

1. allocation time
2. host-to-device transfer time
3. computation time
4. device-to-host transfer time

What observations can you make about the CUDA vs the sequential computations? How much time does the CUDA computation spend transferring data compared to computing? What is the answer to our first research question?

2.3.2 Part Two: Unified Memory

Note: skip this section if your device is not compatable with Unified Memory.

Using omp_get_wtime() modify vectorAdd6.cu so that it reports

1. The time spent allocating A, B, and C
2. The time spent computing the sum of A and B into C
3. The total time of the CUDA computation (i.e., the sum of a and b)

Again, none of these times should include any I/O so make sure you comment out the printf() statements.

Run your program using

```
./vectorAdd6
```

Record your results using 50,000 500,000 5,000,000 and 50,000,000 elements. How do these times compare?

Add this new data to the line chart and stacked bar charts from part one. How does using unified memory compare to using `cudaMemcpy`? What is the bottleneck for the `cudaMemcpy` version? What about the unified memory version?

CHAPTER

THREE

MORE EXERCISES

Note: If your device is incompatable with CUDA 6, don't edit or build the files ending with a 6

3.1 Exercise II: Vector Multiplication

Let's try the same research questions, but using a more expensive operation like multiplication.

In your vectorAdd directory, run

```
make clean
```

to remove the binary. Then run

```
cd ..
cp -r vectorAdd vectorMult
```

to create a copy of your vectorAdd folder named vectorMult. Inside there, rename vectorAdd.cu and vectorAdd6.cu vectorMult.cu and vectorMult6.cu and modify the Makefile to build vectorMult and vectorMult 6 instead of vectorAdd and vectorAdd6. Then edit vectorMult.cu and vectorMult6.cu so that they store the product of A[i] times B[i] in C[i] instead of the sum. Note that for both programs you will need to change:

- The CUDA version.
- The verification test for the CUDA version.
- The sequential version.
- The verification test for the sequential version.

Then build vectorMult and vectorMult6 and run them using 50,000 500,000 5,000,000 and 50,000,000 element arrays. Record the timings for each of these in your spreadsheet, and create charts to help us visualize the results like you did with the vectorAdd programs.

What are the answers to our research questions? How do your results compare to those of Exercise I?

3.2 Exercise III: Vector Square Root

Let's try the same research questions, but this time using an even more expensive operation AND reducing the amount of data we're transferring. Calculating a square root is a more expensive operation than multiplication, so let's try that.

As in Exercise II, clean and make a copy of your vectorMult folder named vectorRoot. Inside it, rename vectorMult.cu and vectorMult6.cu vectorRoot.cu and vectorRoot6.cu respectively. Modify the Makefile to build vectorRoot and vectorRoot6.

Then edit vectorRoot.cu and vectorRoot6.cu so they compute the square root of A[i] in C[i].

Then build vectorRoot and vectorRoot6 and run them using 50,000 500,000 5,000,000 and 50,000,000 element arrays. As before, record the timings for each of these in your spreadsheet, and create charts to help us visualize the results.

What are the answers to our research questions? How do these results compare to those of Exercises I and II?

3.3 Exercise IV: Vector Square

Let's try the same research questions one more time. This time, we will use a less expensive operation than square root, but keep the amount of data we're transferring the same.

As in Part III, clean and make a copy of your vectorRoot folder named vectorSquare. Inside it, rename vectorRoot.cu and vectorRoot6.cu vectorSquare.cu and vectorSquare6.cu respectively. Modify the Makefile to build vectorSquare and vectorSquare6.

Then edit vectorSquare.cu and vectorSquare6.cu so they compute the square of A[i] in C[i].

Then build vectorSquare and vectorSquare6 and run them using 50,000 500,000 5,000,000 and 50,000,000 array elements. As before, record the timings for each of these in your spreadsheet, and create charts to help us visualize the results.

What are the answers to our research questions? How do your results compare to those of the previous parts?