An Introduction to the Nvidia CUDA Architecture

Mike McDonald (mcdonamp@rose-hulman.edu)
CSSE/MA 335 Spring 2012-2013

May 20, 2013

Contents

1 Introduction to CUDA **2**
1.1 GPU Computing 2
1.2 CUDA vs MPI 2

2 Getting Started with CUDA **4**
2.1 Tools and Setup 4
2.2 Your first CUDA program 5

3 CUDA 335 **7**
3.1 Numerical Integration 7
3.2 Odd Even Transposition Sort 7

4 Summary **8**

5 Code Appendix **9**
5.1 Numerical Integration Code 9
5.2 Odd/Even Transposition 11

1 Introduction to CUDA

CUDA, or Compute Unified Device Architecture, is a parallel computing platform developed by Nvidia and first released in 2006 for use on their Graphics Processing Units (GPUs). CUDA is a shared memory architecture that makes efficient use of the large number of processors available on modern GPUs to effectively parallelize programs and increase speed of execution.

1.1 GPU Computing

GPU computing, often refered to as GPGPU (General Purpose computing on Graphics Processing Units), is a fairly recent development even though GPUs have been around since the late 1990's for use as graphics accelerators in both computers and video game consoles. GPU computing is similar to CPU computing, with several notable exceptions. While CPUs usually only have one or two, maybe four or eight, processors, GPU's have hundreds if not thousands of cores (for instance, the Nvidia Tesla C1060, of which Rose owns several, has 240 cores [1]). Functionally, a GPU core is a stripped down CPU processor, essentially just an Arithmetic Logic Unit (ALU) with the required framework to allow it to operate independently and in parallel with it's fellow cores. Additionally, each core is running at a slower clock rate, typically in the hundreds of MHz, instead of the GHz range, like a CPU. The main idea behind GPU computing is to execute small segments of code, called the kernel, on different threads of the cores of a GPU concurrently, which creates a speedup through increasing throughput on the GPU, rather than by increasing clock speed on the host processor.

1.2 CUDA vs MPI

The same idea of increasing throughput by using multiple processors was first done on separate CPU cores, each with their own memory and not necessarily on the same physical die. This is known as distributed memory computing, which is what we learned about in CSSE/MA 335. The MPI libraries we used allowed us to communicate between different CPU's and transfer data to and from the processors, in order to perform our parallel tasks. The typical workflow looked something like this:

1. Distribute data to each processor
2. Operate on given chunk of data locally
3. Communicate between processors if necessary
4. Gather data on the head processor
5. Reduce data if necessary

These steps are fundamentally the same using CUDA, the major difference being the method of communication. GPUs are shared memory devices, instead of distributed memory devices, which means that instead of using sends and receives of data from different nodes in a cluster, GPUs can simply read and write to different locations in memory, which increases the speedup since this operation is significantly faster than MPI sends and receives, and is not governed by network infrastructure or load (both of which we experienced issues with when using the MPI cluster). The following graphic illustrates the CUDA workflow:

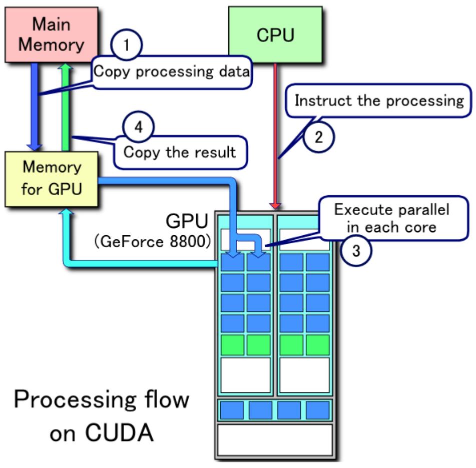

Figure 1: CUDA Processing Flow [2]

CUDA was specifically designed to be integrated into existing programs, in order to speed up computations that were already being carried out in serial, and adding CUDA enhancements to existing programs is not terribly difficult, as will be shown later in this project. Implementing a program in parallel using MPI can often result in major rewrites of code sections, and is not easily written in to existing code. CUDA programs can be compiled using the nvcc compiler, and then linked in to programs that were linked using gcc or g++ (say running GUIs or accessing databases), while this can't be done when compiling a program with mpic++.

2 Getting Started with CUDA

The learning curve for CUDA is not as steep as it is for MPI, since the shared memory architecture is more familiar to users than the message passing distributing architecture. Additionally, the errors associated with message passing have been eliminated (though new errors have been introduced by the potential to change locations in memory or have different processors try to access the same location in memory). CUDA is designed for many different architectures, and is primarily designed for C/C++ and FORTRAN; however, there are wrappers that allow use from other languages such as Python, Java, and MATLAB. We will be using the C/C++ development environment, since that is similar to what we used in class. Let us first go over the tools necessary to use CUDA in this environment, how they are set up, and then create our first CUDA program.

2.1 Tools and Setup

The first tool you will need in order to use CUDA is an Nvidia graphics card with CUDA support. Unlike OpenCL, another popular (and open source) parallel computing platform, CUDA requires the use of an Nvidia GPU. Many computers with Nvidia graphics cards (even laptops) have this support, but for serious computing, as we would like to do, we will use one of the Rose-Hulman dedicated servers, in our case, Clive. Clive (`clive.csse.rose-hulman.edu`) is equipped with two Nvidia Tesla C1060 GPUs, which we will be making use of in this project. In order to compile and run CUDA programs, one must have a valid version of gcc and a valid version of nvcc installed. One can check versions by typing:

```
mcdonamp@clive:~$ gcc --version
gcc (Ubuntu/Linaro 4.6.3-1ubuntu5) 4.6.3
Copyright (C) 2011 Free Software Foundation, Inc.
This is free software; see the source for copying conditions.  There is NO
warranty; not even for MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.
```

And:

```
mcdonamp@clive:~$ nvcc --version
nvcc: NVIDIA (R) Cuda compiler driver
Copyright (c) 2005-2012 NVIDIA Corporation
Built on Thu_Apr__5_00:24:31_PDT_2012
Cuda compilation tools, release 4.2, V0.2.1221
```

Once valid copies of gcc and nvcc are installed, we can continue and begin to program using CUDA.

2.2 Your first CUDA program

Just like when we first began to use MPI, we must get comfortable with some boilerplate code for CUDA. The basic CUDA program looks like this:

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

__global__ void kernel(optional params){
    //perform computations on each GPU core in parallel
}

int main(optional params){
    //create and allocate arrays for the device and the host

    //send data off to each processor to compute

    //gather data from each processor

    //deallocate memory
}
```

Note that these comments mirror the steps in Figure 1 almost exactly.

The nvcc compiler recognizes keywords such as `__global__`, and it is necessary that all files that have these keywords have the extension `.cu` or `.cuh`, to be compiled using nvcc. The nvcc compiler uses an LLVM, which optimizes compiled code (from potentially any language) and outputs machine specific assembly code, which is then run on the CPU (which controls the GPU appropriately).

The kernel, or the part executed on the GPU, can be thought of as the body of a loop. While running serial iterative algorithms, we are often running for loops in a 1D or 2D fashion, in order to manipulate data. This is where a majority of our time is spent, and it is why serial algorithms take so long to run on a single CPU. When using CUDA, these loops can be distributed across other processors, leading to speedup.

Our first CUDA program will be much simpler, since it will put all of that behind the scenes. Just as we did the first time we used MPI, we will print a greeting from both the host processor and all of the cores, to prove that they are actually communicating properly. The code, compilation command, and output are all listed on the next page. The astute reader will notice that the file cuPrintf.cu and cuPrintf.cuh are necessary to run this program, and can be downloaded from the Nvidia developer zone (after an application process) or from a variety of other online sources. Copies of this code will be included with this project, in order to run this project.

```
#include <stdio.h>
#include <stdlib.h>
#include "cuPrintf.cu"

__global__ void kernel(void){
    //Print a greeting from the GPU core
    cuPrintf("Hello from GPU processor %d thread %d\n", blockIdx.x, threadIdx.x);
}

int main(int argc, char** argv){
    //How many blocks and how many threads per block will we use?
    int blocks = 2;
    int threadsPerBlock = 2;

    //Say Hi from the CPU
    printf("Hello from the CPU\n");

    //Initialize printing from GPU cores
    cudaPrintfInit();

    //Instruct each GPU core to run its kernel section
    kernel<<<blocks,threadsPerBlock>>>();

    //Display the greetings gathered from the GPU cores
    cudaPrintfDisplay();

    //End the CUDA printing
    cudaPrintfEnd();

    //Exit from the calling program
    return 0;
}
```

And is compiled as follows:

```
mcdonamp@clive:~$ nvcc hello_cuda.cu -o hello_cuda
```

Running the program produces the following output:

```
mcdonamp@clive:~$ ./hello_cuda
Hello from the CPU
Hello from GPU processor 0 thread 0
Hello from GPU processor 0 thread 1
Hello from GPU processor 1 thread 0
Hello from GPU processor 1 thread 1
```

3 CUDA 335

If the previous example felt too simple, as if the GPU cores didn't actually do any work, we can now use real world examples to further explain GPU computing, as well as prove the performance increases we claim. We will work two examples seen from material we learned in class: numerical integration via the trapezoidal rule as well as odd-even transposition sort.

3.1 Numerical Integration

CUDA is very well suited to perform numerical integration of functions using large numbers of points. This example uses the composite trapezoidal rule to perform integration on a function (set before compilation) by a given number of threads (set before compilation) on an interval with a number of subintervals (both given as command line arguments). Note that we could have used another method, but the trapezoidal method was chosen as it is very simple, and since we are focusing on CUDA, rather than on numerical integration methods, we want to make the example as simple to understand as possible.

There are several important things to notice about this code, which is located in the Code Appendix, in section 4.1. First off, notice how the three functions at the top are preceded by `__device__` keyword. This is necessary any time a function is called from within the kernel. Without this, the nvcc compiler will throw and error and the code will fail to compile. Another thing to note about device functions: they cannot make recursive calls. This is a feature that isn't currently supported in nvcc (presumably since parallel computers should be able to run iterative code fast enough to use non-recursive solutions). Secondly, notice that floats are used, instead of doubles. On our GPU (Tesla C1060), double precision floating point numbers are not supported, so we must use single precision. This leads to slightly less precision in the long run.

3.2 Odd Even Transposition Sort

CUDA is also excellent at sorting algorithms, due to it's shared memory model architecture. We can perform an odd-even transposition sort very easily, with much less overhead than when using MPI, especially since we won't have the same communication overhead. We can also perform the odd and even stages in parallel, which increases the speedup drastically over the MPI version, since we couldn't do two way communication in parallel.

CUDA would also be useful for doing Bitonic sort, especially since it can easily morph its parallel architecture in software to form a 2D sorting array (or even a 3D array, if we created a 3D sort). Creating a different network style using distributed memory, on an architecture like MPI (like we have on the cluster), would require physically rewiring it, while using CUDA, it can be done without any physical changes to the hardware, which is a huge advantage to CUDA.

4 Summary

In summary, CUDA is an incredibly powerful, yet relatively inexpensive (when compared to cluster computing), tool that the average consumer (albeit, one with some knowledge of parallel programming paradigms and programming experience) can use to his or her advantage to create very efficient and very fast implementations of common operations, such as numerical integration or sorting. Additionally, CUDA is very adept at performing 2D and 3D computations, which is perfect for vector and matrix operations, and even higher dimension matrices. This project was intended to give a feeling for how the CUDA architecture works, what CUDA programming is like, and why the executed code runs so much faster than on the MPI cluster. CUDA is a simple jump from current C/C++ knowledge to adding parallelism to many applications, and it's sheer speed (it sorted 1000 integers by odd even sort almost instantly, while MPI had to think for quite some time in order to do the same) allows large computations to occur in the same time that smaller computations would occur on CPU based machines. Overall, GPU computing has a promising future in scientific and mathematic computing, and will only continue to improve in both speed and ease of use over the next several years.

5 Code Appendix

5.1 Numerical Integration Code

```
#include <stdio.h>
#include <math.h>
#include <stdlib.h>

//Function we are integrating
__device__ float myFunction(float x){
  return pow(x,4);
}

//Trapezoidal rule calculation
__device__ float trapezoidal(float a, float b){
return (b-a)*((myFunction(a)+myFunction(b))/2);
}

//Composite trap rule calculation
__device__ float composite_trapezoidal(float a, float b, int n){
float h=(b-a)/ ((float) n);
float total=0;
int i;
for (i=0;i<n;i++){
total=total+trapezoidal(a+i*h,a+(i+1)*h);
}
return total;
}

//This section runs on the GPUs
__global__ void kernel(float* arr, float A, float B, float P, int N){
//Who am I?
int id = blockIdx.x * blockDim.x + threadIdx.x;

//calculate interval bounds
float intervalWidth = (B-A)/(P);
float intervalStart = A+(intervalWidth)*(id);
float intervalEnd = intervalStart+intervalWidth;

//calculate the partial sum of this interval
arr[id] = composite_trapezoidal(intervalStart,intervalEnd,N);
}

int main(int argc, char** argv){
//Process input from command line
if (argc<3){
```

```
printf("Please enter a,b,N\n");
return 1;
}

float A=atof(argv[1]);
float B=atof(argv[2]);
int N=atoi(argv[3]);

printf("Integrating x^4 from %.3f to %.3f with %d points\n", A, B, N);

//How many threads will we use and how much data is in each thread?
int elements = 256;
int bytes = elements * sizeof(float);

//Create pointers to host and device arrays
float *hostArray = 0;
float *deviceArray = 0;

//Create arrays on host and device
hostArray = (float*) malloc(bytes);
cudaMalloc((void**)&deviceArray, bytes);

int blockSize = 128;
int gridSize = elements / blockSize;

//Instruct each GPU core to run its kernel section
kernel<<<gridSize,blockSize>>>(deviceArray, A, B, 256.0, N);

//Gather all the partial sums
cudaMemcpy(hostArray, deviceArray, bytes, cudaMemcpyDeviceToHost);

//Reduce the partial sums to a single integral
float sum = 0;
for(int i=0; i < elements; ++i){
sum += hostArray[i];
}

printf("Integrating x^4 from %.3f to %.3f with %d points is: %.3f\n", A, B, N, sum);

//Deallocate the two arrays
free(hostArray);
cudaFree(deviceArray);

//Exit from the calling program
return 0;
}
```

5.2 Odd/Even Transposition

```
#include <iostream>
#include <stdio.h>
#include <math.h>
#include <stdlib.h>
#include<fstream>
#include<sstream>
#include<vector>

using namespace std;

void cdf_intreader(char* fname,int** A, int* n){
    ifstream F(fname);
    stringstream buf;
    buf<< F.rdbuf();
    string S(buf.str());

    int lastidx=-1;
    int nextidx=string::npos+1;
    int nread=0;
    string toinsert;
    vector<int> Av;

    int i=0;
    while (nextidx!=string::npos){
        nextidx=S.find(',',lastidx+1);
        if (nextidx!=string::npos){
            toinsert=S.substr(lastidx+1,nextidx-lastidx-1);
            lastidx=nextidx;
        }else{
            toinsert=S.substr(lastidx+1,S.length()-lastidx-1);
        }
        Av.push_back(atoi(toinsert.c_str()));
        i++;
        }
    *n=Av.size();
    *A=new int[Av.size()];
    for (i=0;i<Av.size();i++){
    (*A)[i]=Av[i];
    }
}

void cdf_write_seq(char* fname, int* seq, int n){
    ofstream F(fname);
    for (int i=0;i<n-1;i++){
        F<<seq[i]<<",";
    }
    F<<seq[n-1]<<endl;
}

//This section runs on the GPUs
__global__ void kernel(int* arr, int length){
    //What is my ID?
    int id = blockIdx.x * blockDim.x + threadIdx.x;

    //If we're going to access something outside the array, exit
    if(id >= length-1) return;

    //Odd/even transpose elements in the list, in parallel
    for(int j = 0; j < length; j++){
        int temp;
        //If I'm going to perform a swap this round, swap!
        if((j % 2 == 0 && id % 2 == 0) || (j % 2 != 0 && id % 2 != 0)){
            if(arr[id] > arr[id+1]){
                temp = arr[id];
                arr[id] = arr[id+1];
                arr[id+1] = temp;
            }
        }
    }
}

//Main program
int main(int argc, char** argv){
    //Process input files
    if(argc < 3){
        cerr << "Please provide input and output filenames\n";
        abort();
    }
    char* in_file=argv[1];
    char* out_file=argv[2];

    int* A; //list of integers to be sorted
    int n; //size of the list of integers

    //Bring data in from txt file
    cdf_intreader(in_file,&A,&n);

    //Print out initial data if small enough
    if(n < 20){
        cout << "Input list is: ";
        for(int i = 0; i < n-1; i++){
            cout << A[i] << ",";
        }
        cout << A[n-1] << endl;
    }

    //How much data is in each thread?
    int bytes = n * sizeof(int);

    //Create pointer to device array
    int *deviceArray;

    //Create the array on the GPU and copy data to it's memory
    cudaMalloc((void**)&deviceArray, bytes);
    cudaMemcpy(deviceArray, A, bytes, cudaMemcpyHostToDevice);

    //How many threads and blocks per thread will we have?
    int threads = n/2;

    //Launch kernel on the GPU
    kernel<<<n/threads+1,threads>>>(deviceArray,n);

    //Gather data back from processors
    cudaMemcpy(A, deviceArray, bytes, cudaMemcpyDeviceToHost);

    //Print output
    if(n < 20){
        cout << "Sorted list is: ";
        for(int i = 0; i < n-1; i++){
            cout << A[i] << ",";
        }
        cout << A[n-1] << endl;
    }

    //Write to output file
    cdf_write_seq(out_file,A,n);

    //Deallocate the two arrays
    cudaFree(deviceArray);

    //Exit from the calling program
    return 0;
}
```

References

[1] Tesla C1060 Datasheet. [Online]. Available: http://www.nvidia.com/docs/IO/43395/NV_DS_Tesla_C1060_US_Jan10_lores_r1.pdf

[2] CUDA Processing Flow. [Online]. Available: http://en.wikipedia.org/wiki/File:CUDA_processing_flow_(En).PNG