Chapter 4

Introduction to GPU Programming with CUDA

Even if you don't play video games, you can be grateful to the game players, as their numbers have given rise to a class of highly powerful parallel processing devices—**graphics processing units** (GPUs). Yes, you program right on the video card in your computer, even though your program may have nothing to do with graphics.

4.1 Overview

The video game market is so lucrative that the industry has developed ever-faster GPUs, in order to handle ever-faster and ever-more visually detailed video games. These actually are parallel processing hardware devices, so around 2003 some people began to wonder if one might use them for parallel processing of nongraphics applications.

Originally this was cumbersome. One needed to figure out clever ways of mapping one's application to some kind of graphics problem. Though some high-level interfaces were developed to automate this transformation, effective coding required some understanding of graphics principles.

But the current generation of GPUs have separated out the graphics operations, and now consist of multiprocessor elements that run under the familar threads model. Thus they are easily programmable. Granted, effective coding still requires an intimate knowledge of the hardwre, but at least it's (more or less) familar hardware, not requiring knowledge of graphics.

Moreover, unlike a multicore machine, with the ability to run just a few threads at one time, e.g. four threads on a quad core machine, GPUs can run *hundreds or thousands* of threads at once. There are various

restrictions that come with this, but you can see that there is fantastic potential for speed here.

NVIDIA has developed the CUDA language as a vehicle for programming on their GPUs. It's basically just a slight extension of C, and has become very popular. More recently, the OpenCL language has been developed by Apple, AMD and others (including NVIDIA). It too is a slight extension of C, and it aims to provide a uniform interface that works with multicore machines in addition to GPUs.

Our discussion here focuses on CUDA and NVIDIA GPUs.

A CUDA program consists of code to be run on the **host**, i.e. the computer as a whole, and code to run on the **device**, i.e. the GPU. A function that is called by the host to execute on the device is called a **kernel**.

4.2 Hardware Structure

Scorecards, get your scorecards here! You can't tell the players without a scorecard—classic cry of vendors at baseball games

Know thy enemy—Sun Tzu, *The Art of War*

The enormous computational potential of GPUs cannot be unlocked without an intimate understanding of the hardware. This of course is a fundamental truism in the parallel processing world, but it is acutely important for GPU programming. This section presents an overview of the hardware, but true mastery of the GPU software genre requires delving into further details.

4.2.1 Processing Units

A GPU consists of a large set of **streaming multiprocessors** (SMs); you might say it's a multi-multiprocessor. Each SM consists of a number of **streaming processors** (SPs). It is important to understand the motivation for this hierarchy: Two threads located in different SMs cannot synchronize with each other. Though this sounds like a negative at first, it is actually a great advantage, as the independence of threads in separate SMs means that the hardware can run faster. So, if the CUDA application programmer can write his/her algorithm so as to have certain independent chunks, those chunks can be assigned to different SMs (we'll see how, shortly), then that's a "win."

The threads running within an SM *can* synchronize with each other, but there is further hierarchy: The threads within an SM are subdivided by the hardware into groups called **warps**. The key point is that *all the threads in a warp run the code in lockstep*. During the machine instruction fetch cycle, the same instruction will be fetched for all of the threads. Then in the execution cycle, each thread will either execute that particular instruction or execute nothing. This is the classical **single instruction, multiple data** (SIMD) pattern used in some early special-purpose computers such as the ILLIAC; here it is called **single instruction, multiple thread** (SIMT).

This trait of thread execution has major implications for performance. Consider what happens with if/then/else code. If some threads in a warp take the "then" branch and others go in the "else" direction, they cannot operate in lockstep. That means that some threads must wait while others execute. This renders the code at that point serial rather than parallel, a situation called **thread divergence**. As one CUDA Web tutorial points out, this is a "performance killer."

4.2.2 Memory Structure

Yet another key hierarchy—memory structure. Here's how it works:

- Registers: Each thread is allocated it's own set of registers. They are much more numerous than in a CPU, say in the hundreds, and access to them is very fast.
- Shared memory: All the threads in an SM share this memory, and use it to communicate, just as is the case with threads in CPUs. Access is said to be as fast as to registers!
- Global memory: This is shared by all the threads in an entire application, and is persistent across calls. It is very slow.
- Local memory: This is actually part of global memory, but is an area within that memory that is allocated by the compiler for a given thread. As such, it is slow, and accessible only by that thread.

4.2.3 Thread Management

Each SM runs the threads on a timesharing basis, just like an operating system (OS). This timesharing is implemented in the hardware, though, not in software as in the OS case.

With an OS, if a thread reaches an input/output operation, the OS suspends the thread while I/O is pending, and runs some other thread instead, so as to avoid wasting CPU cycles during the long period of time needed for the I/O. With an SM, the analogous situation is a long memory operation, so global memory; if a a warp of threads needs to access global memory (including local memory), the SM will schedule some other warp while the memory access is pending.

The hardware support for threads is extremely good. A context switch takes very little time.

4.3 Software Structure

We'll start with a running example, then go into the details and generalizations.

4.3.1 Sample Program

Here's a sample program. And I've kept the sample simple: It just finds the sums of all the rows of a matrix.

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <cuda.h>
4
5  // CUDA example:  finds row sums of the integer matrix m, placing them
6  // in the array rs
7
8  // finds one element of rs, determined by x, of the n x n matrix m
9  // coordinate of this thread; assume grid consists of 1 block, with
10 // threads in n x 1 x 1 arrangement; matrices stored as 1-dimenstional,
11 // row-major order
12 __global__ void find1elt(int *dm, int *drs, int n)
13 {
14    int rownum = threadIdx.x;  // this thread will handle row # rownum
15    int sum=0;
16    for (int k = 0; k < n; k++)
17       sum += dm[rownum*n+k];
18    drs[rownum] = sum;
19 }
20
21 int main(int argc, char **argv)
22 {
23     // the size of the matrix is specified on the command line
24     int n = atoi(argv[1]);
25     int *hm, // host matrix
26         *dm, // device matrix
27         *hrs, // host rowsums
28         *drs; // device rowsums
29     int msize = n * n * sizeof(int);
30     hm = (int *) malloc(msize);
31     // as a test, fill matrix with consecutive integers
32     int t = 0,i,j;
33     for (i = 0; i < n; i++) {
34        for (j = 0; j < n; j++) {
35           hm[i*n+j] = t++;
36        }
37     }
38     cudaMalloc((void **)&dm,msize);
39     cudaMemcpy(dm,hm,msize,cudaMemcpyHostToDevice);
40     int rssize = n * sizeof(int);
41     hrs = (int *) malloc(rssize);
42     cudaMalloc((void **)&drs,rssize);
43     cudaMemcpy(drs,hrs,rssize,cudaMemcpyHostToDevice);
44     dim3 dimGrid(1,1);
45     dim3 dimBlock(n,1,1);
46     find1elt<<<dimGrid,dimBlock>>>(dm,drs,n);
47     cudaThreadSynchronize();
48     cudaMemcpy(hrs,drs,rssize,cudaMemcpyDeviceToHost);
49     for(int i=0; i<n; i++)  {
50        printf("%d\n",hrs[i]);
51     }
52     free(hm);
```

```
53      cudaFree(dm);
54      free(hrs);
55      cudaFree(drs);
56  }
```

Well, this is mostly C, with a bit of CUDA added here and there. Here's how the program works:

- Our **main()** runs on the host.
- Kernel functions are identified by **__global__ void**, are called by the host, and serve as entries to the device. We have one such function here.
- When a kernel is called, each thread runs it. Each thread receives the same arguments, though different threads may act differently based on programmer use of **threadIdx**.
- One calls **cudaMalloc()** to allocate space on the device's memory.
- Data is transferred to and from the host and device via **cudaMemcpy()**.
- Kernels return values via their arguments. However, the *call* to the kernel returns immediately. For that reason, the code above has a barrier call, to avoid copying the results back to the host from the device before they're ready:

  ```
  cudaThreadSynchronize();
  ```

  There is also a barrier available for the threads themselves, needed in many applications. The call is

  ```
  __syncthreads();
  ```

I've written the program so that each thread will handle one row of the matrix; thread i will find the sum in row i. Since I've chosen to store the matrix in one-dimensional form, and since the matrix is of size n x n, the loop

```
for (int k = 0; k < n; k++)
   sum += dm[rownum*n+k];
```

will indeed traverse the n elements of row number **rownum**, and compute their sum. That sum is then placed in the proper element of the output array:

```
drs[rownum] = sum;
```

4.3.2 Threads Hierarchy

Like the hardware, threads in CUDA software follow a hierarchy:

- The entirety of threads for an application is called a **grid**.
- A grid consists of one or more **blocks** of threads.
- Each block has its own ID within the grid, consisting of an "x coordinate" and a "y coordinate."
- Likewise each thread has x, y and z coordinates within whichever block it belongs to.
- Just as an ordinary CPU thread needs to be able to sense its ID, e.g. by calling **omp_get_thread_num()** in OpenMP, CUDA threads need to do the same. A CUDA thread can access its block ID via the built-in variables **blockIdx.x** and **blockIdx.y**, and can access its thread ID within its block via **threadIdx.x** and **threadIdx.y**.
- The programmer specifies the grid size (the numbers of rows and columns of blocks within a grid) and the block size (numbers of rows, columns and layers of threads within a block). In the example above, this was done by the code

  ```
  dim3 dimGrid(1,1);
  dim3 dimBlock(n,1,1);
  find1elt<<<dimGrid,dimBlock>>>(dm,drs,n);
  ```

  Here the grid is specified to consist of just one (1×1) block, and each block consists of just n $(n \times 1 \times 1)$ threads.

  That last line is of course the call to the kernel. As you can see, CUDA extends C syntax to allow specifying the grid and block sizes. CUDA will store this information in **blockDim** and **threadDim**.
- All threads in a block must run in the same SM, though more than one block might be on the same SM.
- The "coordinates" of a block within the grid, and of a thread within a block, are merely abstractions. **They do not correspond to any physical arrangment.**

The motivation for the two-dimensional block arrangment is to make coding conceptually simpler for the programmer if he/she is working an application that is two-dimensional in nature.

For example, in a matrix application one's parallel algorithm might be based on partitioning the matrix into rectangular submatrices, as we'll do in Section 7.3. In a small example there, the matrix

$$A = \begin{pmatrix} 1 & 5 & 12 \\ 0 & 3 & 6 \\ 4 & 8 & 2 \end{pmatrix} \tag{4.1}$$

is partitioned as

$$A = \begin{pmatrix} A_{00} & A_{01} \\ A_{10} & A_{11} \end{pmatrix}, \tag{4.2}$$

where

$$A_{00} = \begin{pmatrix} 1 & 5 \\ 0 & 3 \end{pmatrix}, \tag{4.3}$$

$$A_{01} = \begin{pmatrix} 12 \\ 6 \end{pmatrix}, \tag{4.4}$$

$$A_{10} = \begin{pmatrix} 4 & 8 \end{pmatrix} \tag{4.5}$$

and

$$A_{11} = \begin{pmatrix} 2 \end{pmatrix}. \tag{4.6}$$

We might then have one block of threads handle A_{00}, another block handle A_{01} and so on. CUDA's two-dimensional ID system for blocks makes life easier for programmers in such situations. Indeed, the submatrices in such partitioning are often called *blocks* in the parallel algorithms literature.

4.3.3 Memory Placement

When you copy data from the host to the device, it goes into the latter's global memory. As noted, this memory is slow. If it is to be repeatedly read, copy it to shared memory, stored in a variable whose declaration is marked **__shared__**.

One also uses such variables for communication between threads in the same block.

4.3.4 What's NOT There

We're not in Kansas anymore, Toto–character Dorothy Gale in *The Wizard of Oz*

It looks like C, it feels like C, and for the most part, it *is* C. But in many ways, it's quite different from what you're used to:

- You don't have access to the C library, e.g. **printf()** (the library consists of host machine language, after all). There are special versions of math functions, however, e.g. **__sin()**.
- No recursion.
- No stack. Functions are essentially inlined.

4.4 Synchronization

As noted earlier, a barrier for the threads in a block is available by calling **__syncthreads()**. Note carefully that if one thread writes a variable to shared memory and another then reads that variable, one must call this function in order to get the latest value.

CUDA and GPU hardware allow for certain atomic operations, such as atomic fetch-and-add, in functions such as **atomicExch()** and **atomicAdd()**. However, classic locks are NOT advised; they can easily lead to deadlock, due to thread divergence problems.

Remember, threads across blocks cannot sync with each other.

4.5 Performance Issues

Our brief introduction here barely scratches the surface. There are many considerations and tricks, especially in terms of memory access. The interested reader is referred to the many resources on the Web.

4.6 CUBLAS

CUDA includes some parallel linear algebra routines callable from straight C code. In other words, you can get the benefit of GPU in linear algebra contexts without using CUDA!

See `http://www.gsic.titech.ac.jp/~ccwww/tebiki/tesla_e/tesla5_e.html` for an example and compilation instructions.

4.7 Hardware Requirements, Installation, Compilation, Debugging

You do need what is currently (March 2010) a high-end NVIDIA video card. There is a list at `http://www.nvidia.com/object/cuda_gpus.html`. If you have a Linux system, run **lspci** to determine what kind you have.

Download the CUDA toolkit from NVIDIA. Just plug "CUDA download" into a Web search engine to find the site. Install as directed.

You'll need to set your search and library paths to include the CUDA **bin** and library directories.

To compile **x.cu**, type

```
$ nvcc -g -G x.cu -I/your_CUDA_include_path
```

The **-g -G** options are for setting up debugging.

Run the code as you normally would.

To determine the limits, e.g. maximum number of threads, for your device, use code like this:

```
cudaDeviceProp Props;
cudaGetDeviceProperties(Props,0);
```

The 0 is for device 0, assuming you only have one device. The return value of **cudaGetDeviceProperties()**. I recommend printing it from within GDB to see the values. One of the fields gives clock speed, which is typically slower than that of the host.

For debugging, CUDA includes a special version of GDB, **cuda-gdb**.