

A Brief Introduction to Using CUDA at USF

Parallel and Distributed Computing

March 29, 2011

1 GPUs and GPGPU Programming

A Graphics Processing Unit or GPU is a processor designed to accelerate the rendering of images on computer screens. By the early 2000's GPUs had become so powerful that some programmers thought that it might be worthwhile to use GPUs to solve problems other than image rendering. Hence the growth of General Purpose GPU programming or GPGPU.

Initially GPUs had to be programmed using graphics languages such as OpenGL or DirectX. However, in 2007 Nvidia Corporation and AMD, the two main producers of GPUs, each introduced an API for GPGPU programming. In late 2008, an industry group introduced a standard for GPGPU programming called OpenCL.

AMD is now focused on supporting OpenCL, and the development environment for Nvidia's proprietary API is generally considered to be more advanced than the development environments for OpenCL.

So for Programming Assignment 4, we'll be using Nvidia's API. It's called CUDA, which is an acronym for "Compute Unified Device Architecture." It provides extensions to C and C++ that are designed to allow programmers to write programs that can make use of both conventional CPUs and Nvidia GPUs.

2 Nvidia GPUs

The details of GPU architectures vary considerably, and, in the literature the components are frequently described using terms taken from the graphics pipeline (e.g., rasterizer, shader). However, for our purposes Nvidia GPUs generally consist of two or more "multiprocessors" (*many* more in Nvidia's high-end GPUs) and eight or more "cores" per multiprocessor¹. The cores are ALU's, and the cores within a single multiprocessor operate in SIMD fashion: each core executes the same instruction on its pieces of data (or is idle). However, threads running on different multiprocessors can execute different instructions.

Each multiprocessor has a relatively small amount of memory that's shared among the cores (16 Kbytes on the systems we'll be using). There is also a "global" memory that's shared by all the multiprocessors. When data is transferred between the CPU and the GPU, by default it goes to/comes from the global memory. There is also a small block of constant

¹Even these terms are peculiar to Nvidia. Beware that others manufacturers may use different terminology.

memory that's shared by the multiprocessors. This memory can only be set by the CPU, and accessing it *can* be very fast. (But beware!)

The key to exploiting hundreds of cores is a thread scheduling system that has very little overhead involved in switching contexts. Thus, if one thread is blocked (e.g., waiting for a memory load), there can be almost no delay in starting another thread on the same core.

3 CUDA

CUDA source files are ordinarily stored in files with a “.cu” suffix. The source code itself looks very similar to a C/C++ programs. In fact, most C/ program (and many C++ programs) can be compiled by the CUDA compiler and run on a CPU.

So CUDA programs start execution in a main function that runs on the CPU or *host*, and they execute C/C++ statements in the same way that C/C++ programs do.

The most important difference between CUDA and C/C++ has to do with “kernels.” Kernels are CUDA functions that are started by the host, but run on the GPU or *device*. They are typically quite short.

4 An Example

4.1 Source Code

As an example, consider the following program, which adds two vectors:

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

/* Kernel for vector addition */
__global__ void Vec_add(float x[], float y[], float z[], int n) {
    /* First block gets first threads_per_block components.      */
    /* Second block gets next threads_per_block components, etc. */
    int i = blockDim.x * blockIdx.x + threadIdx.x;

    /* block_count*threads_per_block may be >= n */
    if (i < n) z[i] = x[i] + y[i];
} /* Vec_add */

/* Host code */
int main(int argc, char* argv[]) {
    int n, i;
    float *h_x, *h_y, *h_z;
    float *d_x, *d_y, *d_z;
    int threads_per_block;
    int block_count;
    size_t size;
```

```

if (argc != 2) {
    fprintf(stderr, "usage: %s <vector order>\n", argv[0]);
    exit(0);
}
n = strtol(argv[1], NULL, 10);
size = n*sizeof(float);

/* Allocate input vectors in host memory */
h_x = (float*) malloc(size);
h_y = (float*) malloc(size);
h_z = (float*) malloc(size);

/* Initialize input vectors */
for (i = 0; i < n; i++) {
    h_x[i] = i+1;
    h_y[i] = n-i;
}

/* Allocate vectors in device memory */
cudaMalloc(&d_x, size);
cudaMalloc(&d_y, size);
cudaMalloc(&d_z, size);

/* Copy vectors from host memory to device memory */
cudaMemcpy(d_x, h_x, size, cudaMemcpyHostToDevice);
cudaMemcpy(d_y, h_y, size, cudaMemcpyHostToDevice);

/* Define block size */
threads_per_block = 256;

/* Define grid size. The calculation is a kludge to get */
/* Ceiling(n/threads_per_block). This insures we have at */
/* least one thread per vector component. */
block_count = (n + threads_per_block - 1)/threads_per_block;

/* Invoke kernel using block_count blocks, each of which */
/* contains threads_per_block threads */
Vec_add<<<block_count, threads_per_block>>>(d_x, d_y, d_z, n);

/* Copy result from device memory to host memory */
/* h_z contains the result in host memory */
cudaMemcpy(h_z, d_z, size, cudaMemcpyDeviceToHost);

printf("The sum is: \n");

```

```

for (i = 0; i < n; i++)
    printf("%.2f ", h_z[i]);
printf("\n");

/* Free device memory */
cudaFree(d_x);
cudaFree(d_y);
cudaFree(d_z);

/* Free host memory */
free(h_x);
free(h_y);
free(h_z);

return 0;
} /* main */

```

4.2 Setup

In the program, the host starts executing the `main` function: it declares some variables, allocates storage for them and initializes them.

In the current version of CUDA, host and device don't share memory (at least not without some work on the programmer's part). So in the host code we typically need to declare pointers to any shared variables that we want to use on the device. We can also allocate storage *in device memory* for these variables by using `cudaMalloc` instead of `malloc`.

Then we need to initialize these variables, which we do in this example by copying the contents of the corresponding host variables using `cudaMemcpy`.

By analogy with the hardware dichotomy between multiprocessors and cores, CUDA has a dichotomy between *blocks of threads* and *threads*. So any time we start a CUDA kernel, we need to specify the number of blocks and the number of threads in each block.

In this example, we're using 256 threads/block, and we can insure that there's at least one thread for each element of the vectors by using `Ceiling(n/threads_per_block)` blocks. A quick and dirty way to get this is the integer division

$$\frac{n + \text{threads_per_block} - 1}{\text{threads_per_block}}$$

4.3 Calling the Kernel

After the setup, we call the kernel. Kernels look like ordinary `void C/C++` functions, except that their header begins with `__global__`. This tells the compiler that the function will be started by the host, but executed by the device. The call to the kernel (in `main`) is, however, somewhat different from an ordinary function call: between the name of the kernel and the argument list we've put

```
<<<block_count, threads_per_block>>>
```

This is telling the run-time system how many threads to start. It's also telling it that the threads should be grouped into `block_count` *blocks*, each of which consists of `threads_per_block` threads.

From a practical standpoint, threads within a block can be synchronized by calling a very fast function, `__syncthreads()`, while threads in different blocks can only be synchronized by returning from a kernel and starting a new kernel. However, there's a very finite limit on the number of threads in a block (currently 512 on our systems). So for all but the smallest problems, you need to use multiple blocks.

Arguments to kernels are passed by value: so the values of the pointers and `n` are passed to the `Vec_add` kernel. Beware, however, addresses on the host aren't directly accessible on the device, and vice-versa. So attempting to simulate pass-by-reference by passing a pointer won't work. For example, suppose we try to run the program

```
__global__ void Kernel(int *x_p) {
    *x_p = 8;
}

int main(void) {
    int x = 7;

    Kernel<<<1,1>>>(&x);
    printf("x = %d\n", x);
}
```

This will probably simply print "x = 7", but it could crash on the device, when the code attempts to dereference `x_p`. In other words, you can pass scalars (including pointers) into kernels, but you can't return anything. So after executing the kernel, we need to explicitly copy the result vector from the device to the host. In our example, we do this in the final call to `cudaMemcpy`.

4.4 Quitting

To finish up we print the results and free storage. Note that storage on the device should be freed with `cudaFree` — not `free`.

4.5 The Kernel

The kernel itself, `Vec_add`, is quite simple — which is fairly typical. When it's called, the run-time system starts

```
block_count*threads_per_block
```

threads. Each thread gets a copy of the arguments, as we discussed. A kernel uses SPMD: each thread executes the same code, but different actions can be taken by branching or operating on different data.

In our example, each thread should carry out a single addition. In order for a thread to get the subscript it should use, it uses three built-in variables: `blockDim`, `blockIdx`, and

`threadIdx`: these are initialized when the host calls the kernel. A thread's rank within a block is given by `threadIdx`. A block's rank within a collection of blocks or *grid* is given by `blockIdx`, and the size of a block is given by `blockDim`. Hence we can assign a unique index (across all threads) with the computation

```
int i = blockDim.x * blockIdx.x + threadIdx.x;
```

If n isn't evenly divisible by 256, there will be some threads for which $i \geq n$. Hence, before actually carrying out the addition, we test whether $i < n$.

5 Threads, Blocks, Grids

The threads within a block, and the blocks within a grid can be multidimensional. Blocks can have three dimensions. To get the y - and z -indexes, you look at `threadIdx.y` and `threadIdx.z`, respectively. To get the dimensions of a block, you look at `blockDim.x`, `blockDim.y`, and `blockDim.z`, and to get the indexes of the blocks you look at the fields of `blockIdx`. Finally you can get the dimensions of a grid by examining the fields of `gridDim`.

CUDA imposes limits on the numbers of threads and blocks. As we noted earlier, on our systems the size of a block to 512 threads. In addition the x - and y - dimensions must be ≤ 512 , while the z - dimension must be ≤ 64 . Grid dimensions must be $\leq 65536 \times 65536 \times 1$. So in our systems grids can only be two dimensional.

6 I/O

An extremely important point to remember is that none of the Nvidia GPUs that we'll be using supports I/O. So if you want to see a value that was computed by the GPU, you'll either have to use the debugger, or you'll have to copy the value from the device to the host.

In the subdirectory `C/src/simplePrintf` of the CUDA SDK (see below) there is an example implementation of a function that is somewhat analogous to the C function `printf`. I've found that it works best if the source code file `cuPrintf.cu` is included into a `.cu` source file:

```
#include "cuPrintf.cu"
```

7 Compiling and Running the Example

All of the Linux computers in 530 and all of the Macs in the 536 labs have Nvidia cards and CUDA installed. To use CUDA you'll need to check on some environment variable settings. On both machines you'll need `/usr/local/cuda/bin` in your path. To check this, type

```
$ echo $PATH
```

(As usual, the "\$" is the shell prompt. You shouldn't type it.) If `/usr/local/cuda/bin` isn't in the path, you can put it there by typing

```
$ export PATH=/usr/local/cuda/bin:$PATH
```

A better idea is to put it permanently in your path by adding this line (not including the “\$” sign) to your `.bash_profile` file.

For the Linux computers in 530 you’ll need to make sure that `/usr/local/cuda/lib64` is in your `LD_LIBRARY_PATH`. This can be done once by typing

```
$ export LD_LIBRARY_PATH=/usr/local/cuda/lib64:$LD_LIBRARY_PATH
```

or, more permanently, by adding the line to your `.bash_profile`. For the Macs in 536 you’ll need to add `/usr/local/cuda/lib` to your `DYLD_LIBRARY_PATH`. This can also be done from the command line or by modifying your `.bash_profile`.

You can get information on the facilities provided by the GPU by running the `deviceQuery` program from the command-line. On the Linux PC’s it’s in

```
/usr/local/NVIDIA_GPU_Computing_SDK/C/bin/linux/release/
```

(If you log on, and this directory isn’t available, try rebooting.) On the Macs it’s in

```
/Developer/GPU Computing/C/bin/darwin/release/
```

The example program we discussed earlier can be downloaded from http://cs.usfca.edu/peter/code/vec_add.cu. There are also *many* example programs in the CUDA SDK. SDK example source code is installed in the directory

```
/usr/local/NVIDIA_GPU_Computing_SDK/C/src
```

on the Linux computers. On the Macs it’s installed in

```
/Developer/GPU Computing/C/src
```

In order to compile a CUDA source file, use `nvcc`. For example,

```
$ nvcc -o vec_add vec_add.cu
```

To run the program, just type the name of the executable. The `vec_add` program takes a command line argument. So it could be started with something like

```
$ ./vec_add 10
The sum is:
11.00 11.00 11.00 11.00 11.00 11.00 11.00 11.00 11.00 11.00
```

Some CUDA programs running under Linux need Pthreads linked in (`-lpthread`). Programs that use features available only in more advanced implementations of CUDA (compute capability \geq 1.0) may need the architecture specified. For example, if you’re using any of the atomic functions, you’ll need `-arch=sm_XX` specified. Here `XX` should be “12” on the Linux machines, “11” on hrn53601 and hrn53602, and “13” on hrn53603 and hrn53604.

8 Debugging

Since CUDA is relatively new, the development tools are not as sophisticated as they are for ordinary C programs. The Linux computers in 530 do support a debugger that is a modified version of the GNU debugger, `gdb`. It's called `cuda-gdb`. The version of CUDA installed on the Macs doesn't have a debugger.

To use `cuda-gdb` you should compile your program with both the `-g` option and the `-G` option. The `-g` option builds a symbol table for code running on the host and the `-G` option builds a symbol table for code running on the device. See http://developer.download.nvidia.com/compute/cuda/3_2_prod/toolkit/docs/cuda-gdb.pdf for detailed information on using `cuda-gdb`.

If you're using a system that doesn't have a debugger, you'll have to resort to printing partial results. However, since the Nvidia cards installed in our labs can't write to `stdout` or `stderr`, you'll need to do the printing from the host. This means that you won't be able to see output from the GPU as it's being generated, and if your program is crashing, you'll probably need to set an artificial return from the kernel and print the partial results that were generated. Once the preliminary results are correct, the artificial return can be advanced.

Before copying results from the device to the host, it may be a good idea to insure that the kernel has actually stopped running by calling

```
cudaThreadSynchronize();
```

As we mentioned above, the output can either be coded by hand using `cudaMemcpy` and `printf/fprintf` or you can use `cuPrintf` from the CUDA SDK.

9 Additional Information

There's a *lot* of information available on CUDA. We already mentioned the guide to `cuda-gdb`, which can be downloaded from http://developer.download.nvidia.com/compute/cuda/3_2_prod/toolkit/docs/cuda-gdb.pdf. A CUDA reference manual is available at http://developer.download.nvidia.com/compute/cuda/3_2_prod/toolkit/docs/CUDA_Toolkit_Reference_Manual.pdf, and a programming guide is available at http://developer.download.nvidia.com/compute/cuda/3_2_prod/toolkit/docs/CUDA_C_Programming_Guide.pdf. More generally, Nvidia's GPU computing page <http://developer.nvidia.com/object/gpucomputing.html> has links to a lot of information on CUDA.