GPGPU Computing at Colby College

Spring 2016
Stephanie Taylor

At Colby, we have four graphics processing units (GPUs) dedicated to general purpose computing. They are hosted on four machines running Linux. To get to them, you first log into bombur.cs.colby.edu. From there, you log onto n1, n2, n3 or n4. These hosts share the same file system (much likes nodes n1, n2, n3, etc. on NSCC). The graphics processors are NVIDIA Tesla M2090 GPUs, each of which has 16 multi-processors. Each multi-processor can run up to 1536 threads concurrently.

To use the GPUs for non-graphics processing (i.e. general purpose GPU (GPGPU) computing), we will use CUDA, which is the set-up supplied by NVIDIA. Specifically, we will use CUDA C, which supports GPGPU programming with a minimal set of extensions to the C language. That means that most of our code will look like C (and will, in fact, be C), but that there are a few extra key words and symbols that enable us to direct the GPU appropriately.

Setup

To run the compiler for CUDA C, we need to add /usr/local/cuda/bin to our path. Log on to bombur.cs.colby.edu and add the following line of code to ~/.cshrc:

```
set path = (/usr/local/bin /usr/local/cuda/bin $path)
```

Advice for Reading Programming Guide

The NVIDIA CUDA C Programming Guide provides a great deal of information about CUDA C and how to use it. It is helpful and you should read it, but not all of it. My goal with this hand-out is to summarize the key aspects of this set-up that apply to us at Colby. I want to provide a context within which you can place the more complete, but rather large, programming guide.

In general, we write a program whose main function runs on the host computer (i.e. it runs on the CPU). That main program spawns "kernel" functions that run on the GPU, sort of like the way our main Pthreads programs create threads that run thread functions. The code is compiled and linked by NVIDIA's compiler, which is called nvcc.

What to read:

Ch. 1 and 2 (introduction and programming model)

Ch. 3

3.1 (compilation with NVCC): just 3.1.1.1

3.2 (CUDA C): intro, 3.2.1, 3.2.2

Ch. 4 4.1 (SIMT Architecture) and 4.2 (Hardware Multithreading)

Ch. 5 (Performance Guidelines)

What to ignore:

We are using CUDA C, which employs language extensions, and a few functions from the Runtime API. You can ignore the Driver API.

You can ignore everything about texture and surface memory as well as page-locked host memory.

You can ignore everything about built-in vector types. We will use float arrays.

Terminology

device: the GPU. Device code runs on the GPU. In CUDA C, only a subset of C++ is supported for device code.

host: the computer hosting the GPU. Host code runs on a CPU. Note that the host and device have separate memory spaces. The full set of C++ features is supported for host code.

Software Concepts

kernel: a function that is executed by each thread

thread: thread of execution (similar to a POSIX thread). Limited in number. We can execute up to 1024 threads per block. The number of threads is specified when we issue the call to execute a kernel (we essentially say "execute kernel foo on M blocks with N threads per block". Note that each thread has local memory, shared memory with all threads in its block, and shared global memory. Also, each thread has access to constant and texture memory spaces, which are read-only.

threadID: each thread is given a unique ID. the variable threadIdx contains it. You can use 1D, 2D, or 3D coordinates to specify the id. Note: If you are running more than one block of threads, then you will also need to use the blockId to determine which part of the data you are supposed to work on.

block: a grouping of threads (sometimes called a thread block). One block executes on one core of a GPU. If you have more blocks than cores, then some blocks wait until a processor is available. The number of thread blocks you use is determined both by the amount of data being processed and the number of cores in your GPU. Note that blocks must be independent (it must be acceptable to execute them at the same time or in any order). (Note also that it is possible for more than one block to be executed on a core if there is enough register space.)

grid: a grouping of thread blocks. When you execute a kernel, you specify how many blocks and how many threads per block. In order to get the indexing right, you must have a mental model of how the blocks and threads are laid out. This is that mental model. It is either a 1D or a 2D layout.

blockID: each block in a grid is given a unique ID. This is accessed through the built-in variable blockIdx.

Hardware Concepts

computeCapability: term used in the documentation to distinguish between the capabilities of older and newer GPUs. Ours is 2.1.

warp: Each multiprocessor places threads (in the same block) into groups of 32. Each group is called a warp. A warp executes one common instruction at a time, so full efficiency is realized when all 32 threads of a warp agree on their execution path. If threads of a warp diverge via a data-dependent conditional branch, the warp serially executes each branch path taken, disabling threads that are not on that path, and when all paths complete, the threads converge back to the same execution path.

SM: Streaming Multi-processor. A GPU is an array of SM's.

SIMT: Single Instruction, Multiple-Thread. The SIMT architecture is akin to SIMD (Single Instruction, Multiple Data) vector organizations in that a single instruction controls multiple processing elements. A key difference is that SIMD vector organizations expose the SIMD width to the software, whereas SIMT instructions specify the execution and branching behavior of a single thread. In contrast with SIMD vector machines, SIMT enables programmers to write thread-level parallel code for independent, scalar threads, as well as data-parallel code for coordinated threads. For the purposes of correctness, the programmer can essentially ignore the SIMT behavior; however, substantial performance improvements can be realized by taking care that the code seldom requires threads in a warp to diverge. In practice, this is analogous to the role of cache lines in traditional code: Cache line size can be safely ignored when designing for correctness but must be considered in the code structure when designing for peak performance. Vector architectures, on the other hand, require the software to coalesce loads into vectors and manage divergence manually.

Memory

Host memory is on the CPU. Device memory is on the GPU. cudaMalloc allocates space on the device.

Each thread has private local memory. Each thread block has shared memory visible to all threads of the block and with the same lifetime as the block. All threads have access to the same global memory. Which one of these is the "device memory" referred to in the previous paragraph? global.

In the add_vec example, the data is created on the CPU, then copied to the GPU's global memory. Each thread then operates on the global memory. In the mat_mult example, the threads load parts of the matrices into shared memory, and operate on those, because it is faster.

I believe the "private local memory" is also called the "register". There are limited numbers of 32-bit registers available per block. There is something the Programming Guide calls "local" memory, but that is just spillover memory for variables that don't fit in the registers. We shouldn't need to worry about that.

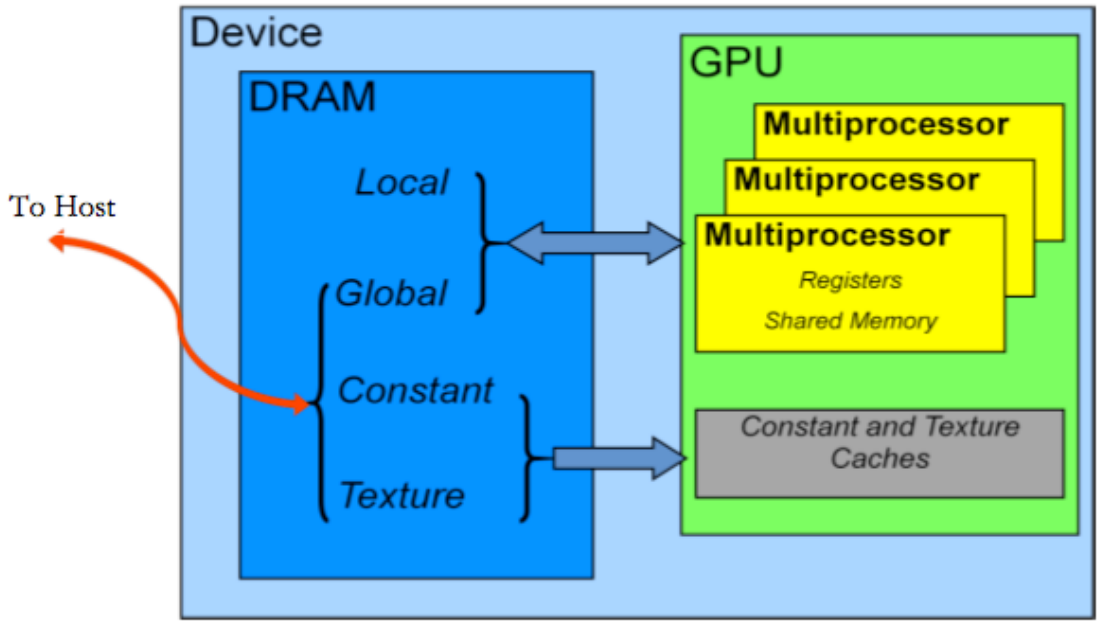
The Programming Guide lists these types of memory: global (where things are cudaMalloc'ed from the host or malloced on the device), local (spillover space for registers – this is slow), shared (shared among threads in a block – this is fast), constant memory (), and texture and surface memory.

In CPU code (i.e. code that does not have __device__ or __global__ in front of it), we use cudaMalloc to allocate space in global device memory. If we are in a device function, we use malloc.

The arguments passed to a kernel are stored in shared memory (so, in the add_vec example, the pointer is in shared memory, and the data is in global memory).

Applications should strive to minimize the transfer of data between host and device.

Below is a figure I snagged from the CUDA Best Practices Guide. It shows the various memory spaces on the device.



Reference

C Language Extensions

For functions

- __device__, executed on device, callable from device only
- __global__, this is a kernel (executed on device, callable from host only)
- __host__, executed on host, callable on host only
  Note that __device__ and __host__ qualifiers can be used together.

For variables

- __device__ (by itself) specifies the variable is a global variable (on the device). must be declared at the file scope and has the lifetime of the application. is accessible by both host and device code, I think. but host code can't dereference a pointer to memory on the device.
- __constant__ (by itself or with __device__) specifies the variable lives in constant memory. must be declared at the file scope and has the lifetime of the application. has implied static storage. must be assigned by the host. is accessible by both host and device code, I think.
- __shared__ (by itself or with __device__) specifies the variable lives in the shared memory space of a thread block. must be declared in device code (I expect it is normally declared in the kernel function). cannot have assignment as part of the declaration. is accessible by all threads in a block. has lifetime of the block. We don't use malloc for this type of memory – it should be declared as static memory.
- automatic variables (don't have any qualifiers) are put into registers, or local memory if there isn't enough space
- cudaMemCpy() using cudaMemCpyHostToDevice copies to the device's global memory (slow memory)
- cudaMemCpyToSymbol copies to the device's constant memory (fast). Use this in host code to write values to __constant__ variables. In device code, it is read-only.

Built-in variables (these can be accessed within the device code)

- blockIdx (type uint3): contains the block index for the given thread. If the blocks are laid out in a 1D array, then all we need is blockIdx.x to learn which block the thread is part of.
- threadIdx (type uint3): contains the thread index (within the block) for the given thread. If the threads are laid out in a 1D array, then all we need is threadIdx.x to learn which block the thread is part of.

CUDA Functions (from the Runtime API)

Host (i.e. functions called from host code)

- cudaError_t cudaDeviceReset(void): Explicitly cleans up all runtime-related resources associated with the calling host thread. Any subsequent API call reinitializes the runtime. Returns cudaSuccess
- cudaError_t cudaDeviceSynchronize (void): Blocks until the device has completed all preceding requested tasks. cudaDeviceSynchronize() returns an error if one of the preceding tasks has failed. If the cudaDeviceScheduleBlockingSync flag was set for this device, the host thread will block until the device has finished its work. Returns cudaSuccess
- cudaError_t cudaFree (void * devPtr) : Frees the memory space pointed to by devPtr, which must have been returned by a previous call to cudaMalloc() or cudaMallocPitch(). Otherwise, or if cudaFree(devPtr) has already been called before, an error is returned. If devPtr is 0, no operation is performed. cudaFree() returns cudaErrorInvalidDevicePointer in case of failure.
- cudaError_t cudaMalloc (void ** devPtr, size_t size): Allocates size bytes of linear memory on the device and returns in *devPtr a pointer to the allocated memory. The allocated memory is suitably aligned for any kind of variable. The memory is not cleared. cudaMalloc() returns cudaErrorMemoryAllocation in case of failure.

- cudaError_t cudaMemcpy (void * dst, const void * src, size_t count, enum cudaMemcpyKind kind): Copies count bytes from the memory area pointed to by src to the memory area pointed to by dst, where kind is one of cudaMemcpyHostToHost, cudaMemcpyHostToDevice, cudaMemcpyDeviceToHost, or cudaMemcpyDeviceToDevice, and specifies the direction of the copy. The memory areas may not overlap. Calling cudaMemcpy() with dst and src pointers that do not match the direction of the copy results in an undefined behavior.

Device (i.e. functions called from within a kernel (or within functions called by the kernel))

- __syncthreads(): sync the threads within the block
- printf: a special version that allows you to write to the screen from within device code. After the kernel has finished, you must call cudaDeviceSynchronize() on the host if you want to see the output. (Note also that you must tell the compiler that your device has compute capability of at least 2.0 if you want to use printf on the device. We do so with the -gencode arch=compute_20,code=sm_20 flags).
- malloc: a special version. This allocates global memory on the device.
- free: a special version. This frees global memory on the device.