GPU Lecture Notes

BRIAN RAILING

1 HISTORY

Graphics processing units began as specialized hardware to support the generation of a video signal, and then provide a richer graphical experience, first supporting GUIs and then further developments mainly support computer games. These units had to model the diversity of textures and lighting that exists in the real-world. Initially, they approached this using a pre-defined parameter space, but with an infinite set of possible inputs, the option was instead made for programmers to provide specially written routines, called shaders, to express the desired effects.

Shaders were (and still are) programmable, and were simple mapping functions that took in components from the scene and generated output for the specific pixel. The graphics hardware was designed to apply these computations to all of the millions of pixels within a scene in a highly efficient manner. This piece of programming, then gives a way to execute arbitrary computations on the hardware. Thus researchers first developed a ‘programming environment’ where a simple scene is no longer graphical, but each pixel encodes computational inputs. The shaders were written to execute the necessary computations, so that the output ‘scene’ would contain the computational output rather than an image.

2 COMMON GPU ARCHITECTURE

As GPUs were designed to process graphical scenes, the thousands upon thousands of pixels in a scene are independent and thus the GPU will handle these independent data elements all in parallel. The individual execution components are simple compared to a CPU, but placing hundreds or thousands of components together provides significant performance.

By having a lot of parallel, independent work, GPU architects could revisit prior research and rely on parallelism to hid memory latency, rather than caches and prefetching. The idea is the precurssor to hyper-threading (see Chapter **??**). When an executing thread is blocked waiting on a long duration operation (such as, a divide taking 20 cycles or a load taking 100 cycles)[1], then the processor has two possible approaches. The first is taken by CPU cores, which try to find and execute future operations. The second approach is taken by GPU designs, where they instead switch to another thread and execute it.

Specifically with GPUs, the processor design will do this across tens to hundreds of threads on each core. If then, a memory operation takes 100 cycles and an addition instruction takes one, and the code alternates between the two instructions, the GPU core would need 50 independent threads to always make progress. Packing many cores onto the chip then enables the GPU to execute thousands of threads in parallel.

[1]Times are architecture dependent

```
const int Nx = 12;
const int Ny = 6;

dim3 threadsPerBlock(4, 3, 1);
dim3 numBlocks(Nx/threadsPerBlock.x,  Ny/threadsPerBlock.y, 1);

// assume A, B, C are allocated Nx x Ny float arrays

// this call will trigger execution of 72 CUDA threads:
// 6 thread blocks of 12 threads each
matrixAdd<<<numBlocks, threadsPerBlock>>> (A, B, C);
```

Fig. 1. Simple CUDA Example for Launching a Kernel

```
// kernel definition
__global__  void matrixAdd(float A[Ny][Nx], float B[Ny][Nx], float C[Ny][Nx])
{
  int i = blockIdx.x * blockDim.x+ threadIdx.x;
  int j = blockIdx.y * blockDim.y+ threadIdx.y;

  C[j][i] = A[j][i] + B[j][i]
}
```

Fig. 2. Simple CUDA Kernel

3 CUDA PROGRAMMING

With CUDA, we can write a single common program that contains both CPU as well as GPU code. Then at specified times, the program's execution can request that certain operations be offloaded to the GPU. The runtime can take care of passing the necessary code to the GPU, in preparation for launching the execution. CUDA launch sends a set of threads with the associated data to the GPU scheduler. This scheduler then manages splitting the threads into smaller units of blocks, which can contain 1 or more warps (see Section 3.1). The individual cores within the GPU receive the next thread block as resources are available (primarily threads and shared memory) splitting it into warps for instruction scheduling.

All of the code in Figure 1 is 'Host' code running on the CPU as part of a normal C/C++ program. It demonstrates two of the three general steps for launching a computation on a GPU. First, the code specifies how many threads it needs to run the computation (see Section 3.2). Next, the (see Section 3.3). Finally, GPU function is called (i.e., launched) with the special <<< and >>> notation used, and the function parameters supplied. The special notation provides the two 3-tuples of the threads to use in the computation.

As the GPU is a different architecture from the CPU, there is a clear separation of the host and the device (i.e., GPU) code. Host code is written as per any other C/C++ program. Device code is placed in separate file(s) and each function must be tagged with `__global__` to indicate that the function will be used to launch a CUDA kernel. Functions can also be tagged with `__device__`, which indicates that they are support routines used by CUDA kernels.

```
int index = blockIdx.x * blockDim.x + threadIdx.x
```

Fig. 3. Mapping a thread to a unique identifier

3.1 Warp

Warps are a unit of scheduling, in that groups of 32 threads share an instruction stream. Each cycle, the core selects one or more warps (depending on hardware design) that are ready to execute (i.e. not waiting on memory or other operations) and fetches the next instruction for that warp to execute. (See also Chapter **??** and Section 2)

The threads running in a warp are scheduled collectively to run on the execution resources. And so executing, every thread fetchs and executes the same instruction in each cycle, with only the thread IDs differing (see Section 3.2). As a consequence, the GPU architecture must use conditional execution and predication bits, as each instruction is executed by the processing units and the predication bits mask off those instructions whose results are not to be retained.

The GPU must fetch every instruction needed by any of the scheduled threads in that warp. If all of the threads in the warp are executing the same branches and conditionals, then the fetched instructions compute and retain their values. However, if some of the threads take different conditionals from others, then both paths must be fetched and executed. And each path will be masked by the predication bits, so that the useful work is less than the hardware could provide. This phenomena is known as *Branch divergence*. Sometimes the divergence cannot be avoided, but is at least a phenomena to be minimized.

3.2 Thread IDs

The data is mapped to a three dimensional grid, according to the specified execution configuration for the function call. Many computational kernels need fewer dimensions, in which case the unused dimensions are set to size 1.

Threads are launched in blocks, which are further composed of warps. Threads often need to compute the reverse mapping to determine their exact three dimensional identifier. For each dimension, the grid is split into a sequence of blocks, each containing the same number of threads for that dimension. Thus each thread receives two identifiers, `blockIdx` and `threadIdx`, which are global variables set by the CUDA runtime. The first identifies which block in the grid this thread belongs. The later identifies which thread within that block, so `threadIdx` ranges from `0` to `blockIdx - 1`. Each identifier is a 3-tuple: x, y, and z. For a given dimension, we can use the formulation in Figure 3 to uniquely identify the thread in one dimension. By this calculation, the two components (`blockIdx` and `threadIdx`) are together accessing a two dimensional array, as each dimension of the grid is split into blocks, each composed of threads.

Threads do not have to map 1:1 to the input data. While GPUs achieve the best performance through many thousands of individual work items, at a certain point it can instead be advantageous to combine redundant calculations so that the thread processes several elements of the input data.

3.3 CUDA Memory Model

The GPU is (generally) a different device in the system from the CPU, and it has its own dedicated memory space [2]. This memory can be specifically designed for the memory access characteristics of high performing GPU computations, such as highly concurrent, strided accesses, as well as historically containing the data for rendering graphical scenes.

More specifically there are three memories on the GPU: device, per-block shared, per-thread private.

[2]There are some systems that have Unified Memory, where the CPU and GPU share the DRAM.

```
__global__ void convolve(int N, float* input, float* output) {
   int index = blockIdx.x * blockDim.x + threadIdx.x;  // thread local variable

   float result = 0.0f;  // thread-local variable
   for (int i=0; i<3; i++)
     result += input[index + i];

   output[index] = result / 3.f;
}
```

Fig. 4. Convolution Example

Given that the GPU has separate memory from the CPU, we need to use separate functions to manage that memory. `cudaMalloc` and `cudaFree` are equivalents to the similarly named C functions; however, `cudaMalloc` populates the 'pointer' rather than providing it as a return value. Thus the signature is `cudaMalloc(*ptr, size)`.

`cudaMemcpy` is similar to the C routine by taking in two pointers and the count of bytes to copy; however, given that the pointers could each contain any address, this call cannot directly distinguish which address is device and which is host and therefore, while it knows a source and destination, these locations cannot solely be located within the multiple memory spaces without additional information. Thus, this information is passed as a separate parameter, either `cudaMemcpyHostToDevice` or `cudaMemcpyDeviceToHost`.

Per-thread private memory usage relates to the hardware provided registers. This is a limited resource that can constrain launches of thread blocks. Usually not an issue, but kernels that have significant unrolling of code can reach these limits. You can use the `--resource-usage` flag to nvcc to see the resources required by the compiled code. Registers are a hardware limited resource, where using too much can limit the number of thread blocks that can be launched to a multiprocessor. The compiler is far more aggressive in CUDA about placing small arrays and other variables into registers rather than leaving them in the per-thread local memory, which exhibits similar performance to global memory (i.e., it is like the stack in CPU execution).

3.4 Shared Memory

Shard memory is a smaller region available on each core that is faster to access, much like hardware memory caches; however, given that GPUs often stream across large datasets with little locality, this memory region does not automatically populate based on memory accesses. Instead, the shared memory is software-managed, such that the program specifies how much space it needs and explicitly performs the load and store operations involving this region.

Just as increasing the size of thread blocks beyond a certain point will result in compiler errors, so will the code fail to compile if there is too much shared memory statically requested. In both cases, if the requested resources exceed the limits for the targeted CUDA compute level, then the program would not be able to run.

3.4.1 Shared Memory Examples. To understand better how to use shared memory, we will go through several examples. In this first example, Figure 4 gives a simple CUDA routine that applies a convolution, averaging adjacent values, to a vector of floats. This routine is fairly efficient, except a certain key operation is repeated, which is the loading of each value from memory. Each CUDA thread has four memory operations, three loads and one store, where two of the three loads are also executed by each adjacent thread. These redundant loads are combined into a single memory operation by the GPU, but by the GPU's memory model they are not explicitly cached.

The GPU was assuming that memory accesses have minimal locality, so if we have temporal locality, the programmer must explicitly instruct the GPU to exploit it. This requires additional complexity, but will provide a performance benefit. As the threads execute in separate warps, there is no guarantee about ordering of each warp, so to ensure that the shared memory is populated before use, the function calls `__syncthreads()`, which blocks all warps in the threadblock as per a barrier. Then when each has completed its loading of the shared memory, all warps and threads can proceed to use the values now present. By critical path, (almost) all warps now issue one load to global memory, rather than the three before. The change is not free but against the shorter critical path, it is faster.

```
__global__ void convolve(int N, float* input, float* output) {
   __shared__ float support[THREADS_PER_BLK+2];        // per-block allocation
   int index = blockIdx.x * blockDim.x + threadIdx.x;  // thread local variable

   support[threadIdx.x] = input[index];
   if (threadIdx.x < 2) {
      support[THREADS_PER_BLK + threadIdx.x] = input[index+THREADS_PER_BLK];
   }

   __syncthreads();

   float result = 0.0f;  // thread-local variable
   for (int i=0; i<3; i++)
     result += support[threadIdx.x + i];

   output[index] = result / 3.f;
}
```

3.5 Scheduling

When a kernel is launched, the GPU is notified about the requested work and must now begin scheduling that work in discrete units until it is complete. Each kernel has specified thread dimensions, both in a total thread count as well as discretizing those threads into blocks. Each thread block is the dispatchable unit, sent to a specific core within the GPU. This unit has its associated resource requirements: number of threads, registers / thread local memory required per thread, and quantity of shared memory required. All resources are treated as requirements, and unlike much CPU programming, the system does not treat requests lazily and over promise the resource availablity.

The GPU is continually examining the state of each core, such that it can dispatch a new thread block to a core when there are sufficient resources available. A dispatch of work takes time to complete, time in which the core may not have enough load to saturate its compute units.

In optimizing a kernel, the programmer should maximize the *occupancy rate*, which is the measurement of the fraction of cycles in which a core is executing work. This requires a careful balancing act. Larger thread blocks would result in fewer blocks being dispatched, but the resource requirements are increased so fewer blocks might fit on a GPU core. The amount of work per thread can also vary, which can reduce the number of blocks required, but increasing work may require more resources. Fewer thread blocks can also reduce the opportunity to have blocks loaded on all cores.

3.6 Atomic Programming

When any thread in a warp invokes `__syncthreads()`, that warp will not continue executing until all warps in its threadblock have also invoked this function, thus serving as a barrier synchronization for the threads in the block. A common design practice is to have the threadblock populate shared memory with the necessary values and then wait for the other warps to complete this step using `__syncthreads()`. After the call returns, we have a guarantee that all threads have reached this barrier.

However, if the code design prevents some executions from reaching the `__syncthreads()`, such as placing the call within a conditional, then the call can never complete.

3.7 Memory Banking

Banking is a technique whereby multiple partitions of the address space can be accessed independently in the same cycle(s). On GPUs, there is the strong expectation that any computation will be split into warps of 32 threads, so every memory operation will entail accessing as many as 32 different locations in memory. As such, the address space is partitioned so that each successive 32-bit word maps to the next of the 32 different banks. Effectively, the specific bank is determined by the address: `bankID = (Address >> 2) % 32`, and not by thread ID or any other identifier.[3]

Each bank can access 1, 32-bit word at a time. Thus, if the warp's threads access data in 32 different banks. Or if more than one thread accesses the same location in one of the banks, then the memory can operate at full capacity.

On the other hand, if multiple threads access the same bank and are requesting different memory locations, then there is bank conflict and the bank must serialize the memory accesses. Similar to branch divergence, the threads in the warp must now stall some multiple of the bank access time, and the memory system is not operating at full capacity.

3.8 Compute Capability

GPUs keep the specifics of their design hidden to progrmamers; however, certain assumptions can be made about their provided capabilities and resources. Specifically, when a CUDA program is compiled, a flag can be passed indicating the minimimum capabilities expected of the hardware, such as shared memory, instructions, et cetera. A kernel may fail to compile if it expects more from hardware than that compute capability defines. And it can fail to launch if hardware does not support the desired capability level. However, higher levels provide greater performance and functionality.

4 TOOLS

4.1 nvprof

5 PROBLEMS

Assume a GPU with 4 cores running at 1GHz, each able to queue 64 warps of 32 threads each. Every cycle the core will fetch and execute 1 instruction. (Keep in mind this instruction is executed on an entire warp in that cycle.) The main workload (shown below) is run with a 1M-element input array, `X[]` (initialized randomly to values between `0.0f` and `1.0f`) and with an output array `Y[]` (also with 1M elements). Assume that the input and output arrays are resident in CUDA device memory (i.e. the GPU's memory) at the time of the kernel launch.

[3]Other memory systems also use banking and can have different mapping functions.

```
__global__ void foo(float* X, float* Y) {
   // get array index from CUDA block/thread id
   int idx = blockIdx.x * blockDim.x + threadIdx.x;

   float result = 0.f;
   float val = X[idx]; // memory load

   if (val > 0.5f) // 2 arithmetic cycles
      result = doA(val); // 200 arithmetic cycles
   else
      result = doB(val); // 200 arithmetic cycles

   Y[idx] = result; // memory store
}
```

(1) The performance of the workload is roughly *half* of the maximum, even after trying different sized thread blocks. Given your knowledge from 418, this should not be surprising. Please explain why this observation makes sense (given how GPUs would process the code above).

(2) In separate execution the input array is now sorted. How will this change the performance observed in the prior question.