Introduction to CUDA

Mark Greenstreet

CpSc 418 – Mar. 29, 2016

GPU Summary: architecture

- Lots of cores:
  - Up to 90 or more SIMD processors.
  - Each SIMD processors has 32 pipelines.
  - This is the nVidia architecture – other GPUs are similar.
- Deep, simple, execution pipelines
  - Optimized for floating point.
  - No bypassing: use multi-threading for performance.
  - Branches handled by predicated execution

    “When you come to a fork in the road, take it.”
    (Often attributed to Yogi Berra.)

- Limited on-chip memory.
  - 1 or 2 MBytes total. Big CPUs have 32-64MB of L3 cache.
  - The programmer manages data placement.

GPU Summary: Performance

- Today's processors are constrained by how much performance can you get using $\sim$ 200 watts.
    - Moving bits around takes lots of energy.
    - Performing operations as quickly as possible takes lots of energy.
    - $E \sim dt^{\alpha}$, where E is energy, d is distance, t is time per operation, and $1 < \alpha < 2$ depending on design details.
        - Corollary: $P \sim d^{\alpha+1}$. Power grows someplace between quadratically and cubically with clock period.
- How GPUs optimize performance/power
    - SIMD: instruction fetch and decode moves lots of bits. Amortize over many cores.
    - Simple pipelines: bypassing means moving bits quickly. GPUs omit bypasses.
    - High latency: avoid pipeline stages that must do a lot in a hurry.
    - Expose the memory hierarchy: let the programmer control moving data bits around.

GPU Summary: Economics

- GPUs are designed for the high-volume, consumer graphics market.
  - Amortize high design cost over a large number of units sold.
- This means GPUs aren't really optimized for scientific computing:
  - More on-chip memory would certainly help scientific computing, but not needed for graphics rendering.
  - Comparison: An nVidia GPU has about 2 MBytes of on-chip memory, an Intel Xeon can have 40MBytes or more.
  - Cache memory is about 60-70 transistors per byte.
  - A high-end nVidia GPU has 7 billion transistors, 1 or 2% for memory.
  - What if the chip were 30-40% memory?
    - ★ better for general purpose computing
    - ★ little pay-off for graphics
    - ★ smaller distinction with Intel CPUs
- Cheap is good
  - It's the economics of cheap-computing that drives Moore's Law and all the other exponential growth-rate trends that make computing a field of intense, ongoing innovation.
  - That keeps the field in transition – deal with it.

Programming GPUs: CUDA

Data Parallelism

- When you see a `for`-loop:
  - Is the loop-index used as an array index?
  - Are the iterations independent?
  - If so, you probably have data-parallel code.
- Data-Parallel problems:
  - Run well on GPUs because each element (or segment) of the array can be handled by a different thread.
  - Data parallel problems are good candidate for most parallel techniques because the available parallelism grows with the problem size.
  - Compare with “task parallelism” where the problem is divided into the same number of tasks regardless of its size.

Which of the following loops are data parallel?

```
for(int i = 0; i < N; i++)
   c[i] = a[i] + b[i].
```

```
dotprod = 0.0;
for(int i = 0; i < N; i++)
   dotprod += a[i]*b[i];
```

```
for(int i = 1; i < N; i++)
   a[i] = 0.5*(a[i-1] + a[i]);
```

```
for(int i = 1; i < N; i++)
   a[i] = sqrt(a[i-1] + a[i]);
```

```
for(int i = 0; i < M; i++) {
   for(int j = 0; i < N; j++) {
      sum = 0.0;
      for(int k = 0; i < L; k++)
         sum += a[i,k]*b[k,j];
      c[i,j] = sum;
}  }
```

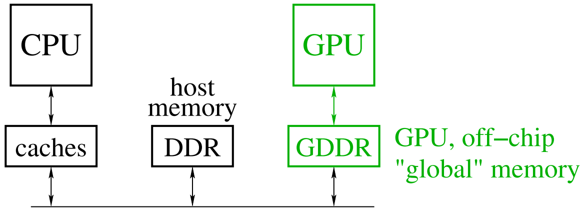
CUDA Program Structure

- A CUDA program consists of three kinds of functions:
  - Host functions:
    - callable from code running on the host, but not the GPU.
    - run on the host CPU;
    - In CUDA C, these look like normal functions – they can be preceeded by the `__host__`.
  - Device functions.
    - callable from code running on the GPU, but not the host.
    - run on the GPU;
    - In CUDA C, these are declared with a `__device__` qualifier.
  - Global functions
    - called by code running on the host CPU,
    - they execute on the GPU.
    - In CUDA C, these are declared with a `__global__` qualifier.

Structure of a simple CUDA program

- A `__global__` function to called by the host program to execute on the GPU.
  - There may be one or more `__device__` functions as well.
- One or more host functions, including `main` to run on the host CPU.
  - Allocate device memory.
  - Copy data from host memory to device memory.
  - "Launch" the device kernel by calling the `__global__` function.
  - Copy the result from device memory to host memory.

Execution Model: Memory



- Host memory: DRAM and the CPU's caches
  - Accessible to host CPU but not to GPU.
- Device memory: GDDR DRAM on the graphics card.
  - Accessible by GPU.
  - The host can initiate transfers between host memory and device memroy.
- The CUDA library includes functions to:
  - Allocate and free device memory.
  - Copy blocks between host and device memory.
  - BUT host code can't read or write the device memory directly.

More Memory

- GPUs support fairly large off-chip memory bandwidth: 200-400GB/s.
  - But this isn't fast enough to keep 1000 processors busy at 1Gflop/s each!
- The GPU has on-chip memory to help:
  - Shared memory: 16KBytes or 48KBytes.
  - Registers: 128Kbytes (256KBytes on more recent GPUs).
  - Note that we need to use each value from memory for 20 or more instructions or else the memory bandwidth will limit performance.
- GPUs also have L2 caches, around 1.5MByte in the most recent chips.
  - But I haven't found a good way to understand them from the textbook, or from other CUDA manuals.
  - The coherence/consistency guarantees seem to be pretty weak.

Example: `saxpy`

`saxpy` = “Scalar `a` times `x` plus `y`”.

- The device code.
- The host code.
- The running `saxpy`

`saxpy`: device code

```
__global__void saxpy(uint n, float a, float *x, float *y) {
   uint i = blockIdx.x*blockDim.x + threadIdx.x; // nvcc built-ins
   if(i < n)
      y[i] = a*x[i] + y[i];
}
```

- Each thread has `x` and `y` indices.
  - We'll just use `x` for this simple example.
- Note that we are creating one thread per vector element:
  - Exploits GPU hardware support for multithreading.
  - We need to keep in mind that there are a large, but limited number of threads available.

saxpy: host code (part 1 of 5)

```
int main(int argc, char **argv) {
   uint n = atoi(argv[1]);
   float *x, *y, *yy;
   float *dev_x, *dev_y;
   int size = n*sizeof(float);
   x = (float *)malloc(size);
   y = (float *)malloc(size);
   yy = (float *)malloc(size);
   for(int i = 0; i < n; i++) {
      x[i] = i;
      y[i] = i*i;
   }
   ...
}
```

- Declare variables for the arrays on the host and device.
- Allocate and initialize values in the host array.

saxpy: host code (part 2 of 5)

```
int main(void) {
   ...
   cudaMalloc((void**)(&dev_x), size);
   cudaMalloc((void**)(&dev_y), size);
   cudaMemcpy(dev_x, x, size, cudaMemcpyHostToDevice);
   cudaMemcpy(dev_y, y, size, cudaMemcpyHostToDevice);
   ...
}
```

- Allocate arrays on the device.
- Copy data from host to device.

`saxpy`: host code (part 3 of 5)

```
int main(void) {
    ...
    float a = 3.0;
    saxpy<<<ceil(n/256.0),256>>>(n, a, dev_x, dev_y);
    cudaMemcpy(yy, dev_y, size, cudaMemcpyDeviceToHost);
    ...
}
```

- Invoke the code on the GPU:
  - `add<<<ceil(n/256.0),256>>>(...)` says to create $\lceil/256\rceil$ blocks of threads.
  - Each block consists of 256 threads.
  - See slide 20 for an explanation of threads and blocks.
  - The pointers to the arrays (in device memory) and the values of `n` and `a` are passed to the threads.
- Copy the result back to the host.

saxpy: host code (part 4 of 5)

```
   ...
   for(int i = 0; i < n; i++) { // check the result
      if(yy[i] != a*x[i] + y[i]) {
         fprintf(stderr, "ERROR: i=%d, a[i]=%f, b[i]=%f, c[i]=%f\r
                 i, a[i], b[i], c[i]);
         exit(-1);
      }
   }
   printf("The results match!\n");
   ...
}
```

- Check the results.

saxpy: host code (part 5 of 5)

```
int main(void) {
   ...
   free(x);
   free(y);
   free(yy);
   cudaFree(dev_x);
   cudaFree(dev_y);
   exit(0);
}
```

- Clean up.
- We're done.

Launching Kernels

- Terminology
  - Data parallel code that runs on the GPU is called a **kernel**.
  - Invoking a GPU kernel is called **launching** the kernel.
- How to launch a kernel
  - The host CPUS invokes a `__global__` function.
  - The invocation needs to specify how many threads to create.
  - Example:
    - `add<<<ceil(n/256.0),256>>>(...)`
    - creates $\left\lceil \frac{n}{256} \right\rceil$ **blocks**
    - with 256 **threads** each.

Threads and Blocks

- The GPU hardware combines threads into **warps**
  - Warps are an aspect of the hardware.
  - All of the threads of warp execute together – this is the SIMD part.
  - The functionality of a program doesn't depend on the warp details.
  - But understanding warps is critical for getting good performance.
- Each warp has a "next instruction" pending execution.
  - If the dependencies for the next instruction are resolved, it can execute for all threads of the warp.
  - The hardware in each streaming multiprocessor dispatches an instruction each clock cycle if a ready instruction is available.
  - The GPU in `lin25` supports 32 such warps of 32 threads each in a "thread block."
- What if our application needs more threads?
  - Threads are grouped into "thread blocks".
  - Each thread block has up to 1024 threads (the HW limit).
  - The GPU can swap thread-blocks in and out of main memory
    - This is GPU system software that we don't see as user-level programmers.

Compiling and running

```
lin25$ nvcc saxpy.cu -o saxpy
lin25$ ./saxpy 1000
The results match!
```

But is it fast?

- For the `saxpy` example as written here, not really.
  - Execution time dominated by the memory copies.
- But, it shows the main pieces of a CUDA program.
- To get good performance:
  - We need to perform many operations for each value copied between memories.
  - We need to perform many operations in the GPU for each access to global memory.
  - We need enough threads to keep the GPU cores busy.
  - We need to watch out for thread divergence:
    - If different threads execute different paths on an if-then-else,
    - Then the else-threads stall while the then-threads execute, and vice-versa.
  - And many other constraints.
- GPUs are great if your problem matches the architecture.

Preview

March 3: Data Parallel Programming 1
Reading: Kirk & Hwu, Chapter 4.
March 7: Data Parallel Programming 2
Reading: Kirk & Hwu, Chapter 4.
March 9: The GPU Memory Model 1
Reading: Kirk & Hwu, Chapter 5.
March 11: The GPU Memory Model 2
Reading: Kirk & Hwu, Chapter 5.
March 14: GPU Performance 1
Reading: Kirk & Hwu, Chapter 6.
March 16: GPU Performance 2
Reading: Kirk & Hwu, Chapter 6.
March 18: Parallel Sorting
Reading: TBD.

But of course, we'll adjust this as we go.

Review

- What is SIMD parallelism?
- What is the difference between “shared memory” and “global memory” in CUDA programming.
- Think of a modification to the `saxpy` program and try it.
  - You’ll probably find you’re missing programming features for many things you’d like to try.
  - What do you need?
  - Stay tuned for upcoming lectures.