CUDA: Big Picture

Mark Greenstreet

CpSc 418 – November 28, 2018

- Data Parallel Computation
- Grids, Blocks, and Threads
- But, the hardware isn't hidden (very well? at all?)

Data Parallel Computation

Matrix Multiplication

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

- Data parallel: all iterations of the `i` and `j` loops are independent.
- But the compiler doesn't know that – it can't rule out pointer aliasing.
- So, we will use a special CUDA construct to tell the compiler that we "promise" that `c` overlaps neither `a` nor `b`.

Matrix Multiply in CUDA (ver. 1)

```
__global__ mmult_kernel(uint N, float *a, float *b, fl
  uint i = threadIdx.y; // my row
  uint j = threadIdx.x; // my column
  float sum = 0.0;
  for(uint k = 0; k < N; k++)
    sum += a[i*N + k] + b[k*N + j];
  c[i*N + j] = sum;
}
...
mmult(uint N, float *a, float *b, float *c) {
  dim3 dimBlock(N, N, 1); // dimensions in x, y, and z
  dim3 dimGrid(1, 1, 1); // what's a grid?
  mmult_kern<<<dimGrid, dimBlock>>>(N, a, b, c);
}
```

- Create one thread for each element of `c`.
- Compute all of the elements in parallel.

Grids, Blocks, and Threads

- On the previous slide, we launched a kernel with one block that had one thread per element of `C`.
  - Why? Because we took a software point-of-view and don't know how the hardware actually implements it.
  - But, CUDA has blocks and grids.
- A block is an array of threads.
  - Can be 1, 2, or 3 dimensional.
  - That's awesome. I used a 2-dimensional block to correspond with the rows and columns of my matrix.
  - A block can have at most 1024 threads?
  - Why? Because a block is implemented by the hardware, and there are limited resources.
  - But, CUDA has blocks and grids.
- A grid is an array of blocks.
  - grids are implemented by the CUDA runtime (software).
  - grids can be **huge**.

Matrix Multiply in CUDA (ver. 2)

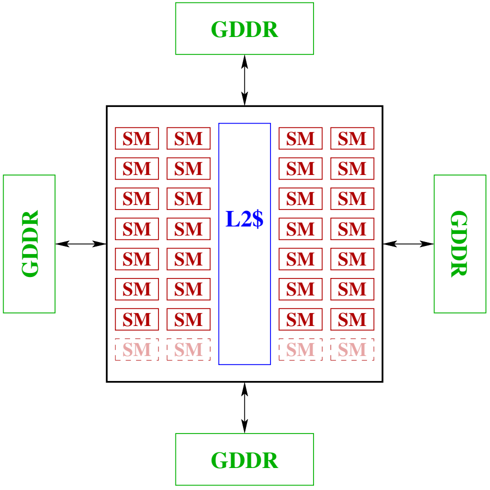
```
__global__ mmult_kernel(uint N, float *a, float *b, fl
  uint i = 32*blockIdx.y + threadIdx.y; // my row
  uint j = 32*blockIdx.x + threadIdx.x; // my column
  if((i < N) && (j < N)) {
    float sum = 0.0;
    for(uint k = 0; k < N; k++)
      sum += a[i*N + k] + b[k*N + j];
    c[i*N + j] = sum;
  }
  }
  ...
  mmult(uint N, float *a, float *b, float *c) {
    dim3 dimBlock(32, 32, 1); // dimensions in x, y, and z
    dim3 dimGrid(ceil(N/32.0), ceil(N/32.0), 1); //
    mmult_kern<<<dimGrid, dimBlock>>>(N, a, b, c);
  }
```

- We need `if((i < N) && (j < N))` because of rounding up

Grids, Blocks, and Threads (part 2)

- A grid is an array of blocks.
  - Grids can be huge.
  - No communication between threads in different blocks.
  - What if different threads read and write the same location in memory?
    - No guarantees on the order.
    - No guarantees on coherence (e.g. no promise of sequential consistency).
- A block is an array of threads
  - Block size limited by hardware: max 1024 threads per block.
  - Threads in a block can communicate using “shared memory”
  - What’s shared memory, and why should I care?
- This leads us to global memory, coalesced references, shared memory, bank conflicts, and thread synchronization.
  - Now we need to know about warps, and the underlying execution model.

GPU: Top-Level Architecture

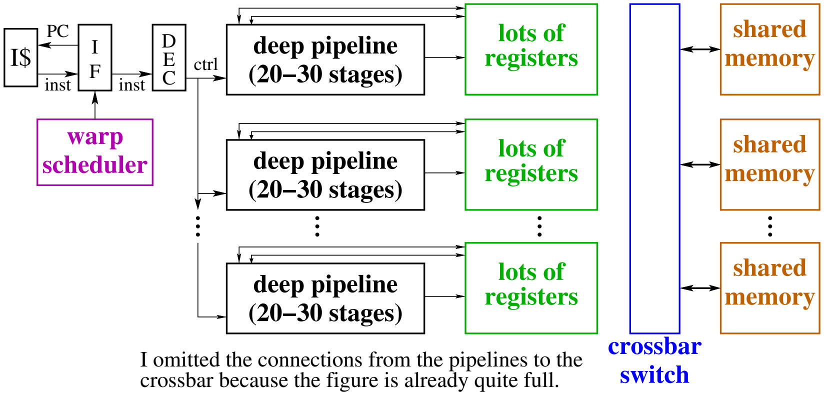


GTX 1080 Architecture
(high–end, "Pascal" generation)

GPUs in the the `linXX.ugrad.cs.ubc.ca` machines

- GeForce GTX 1060
- 9 SMs (10 on chip, 9 reported by the device):
  - 128 SPs/SM.
  - That's 1152 SPs on the chip.
  - Each SM can schedule 4 warps in a single cycle.
- $\sim 1.6\mathrm{GHz}$ clock frequency.
- 3 GBytes of GDDR5 memory, $\sim 192\mathrm{GBytes/sec.}$ memory bandwidth.

A Streaming Multiprocessor (SM)



I omitted the connections from the pipelines to the crossbar because the figure is already quite full.

- Each of the pipelines is an SP (streaming processor)
- Lots of deep pipelines.
- Lots of threads: when we encounter an architectural challenge:
  - Raising throughput is easy, lowering latency is hard.
  - Solve problems by increasing latency and adding threads.
  - Make the programmer deal with it.

Threads, Warps, SMs, Blocks, ...

- Each warp has a warp index: w.
- Each thread in a warp executes on a particular SP, s.
- We can compute a "raw thread index" (within the block):

$$rti \quad = \quad 32 * w + s$$

- The block has `threadIdx.x`, `threadIdx.y`, and `threadIdx.z`:
  - `threadIdx.x = ` rti `% blockDim.x`
  - `threadIdx.y = (`rti` / blockDim.x) % blockDim.y`
  - `threadIdx.r = (`rti` / (blockDim.x * blockDim.y)`
- Isn't that a lot of work to calculate those indices?
  - Done once when the block is dispatched.
  - Each thread stores its `threadIdx`, `blockIdx`, `blockDim`, and `gridDim` variables in dedicated registers.

The big thing about warps

- Programmers are supposed to pretend that they don't know about warps – warps are a hardware thing.
- Programmers must know about warps to get good performance:
  - Coalescing memory references.
  - Avoiding shared-memory bank conflicts.
  - Warp-level synchronization is fast.

Preview

November 30: Review
Dec 3: Final Exam review session (piazza poll)
Dec 4: Final Exam – 8:38am (ouch!)