

# CS 179 Lecture 4

GPU Compute Architecture

# This is my first lecture ever

Tell me if I'm not speaking loud enough, going too fast/slow, etc.

Also feel free to give me lecture feedback over email or at office hours.

# Today

```
kernel<<<gridSize, blockSize>>>(arg1, arg2);
```

What does this actually do?

How does the GPU execute this?

Reference: CUDA Handbook, Ch 7-8

# Compilation, runtime, drivers, etc

Interesting...

but much less useful than architecture

Maybe we'll cover these later in term!

# Streaming Multiprocessors

a streaming multiprocessor (SM)  $\approx$  a CPU

SMs have:

- registers (1000's!)
- caches
- warp schedulers
- execution cores

GPUs have 1-20 SMs

# Nvidia Architecture Names

Family names:

Tesla (2006) → Fermi (2010) → Kepler (2012) → Maxwell (2014)

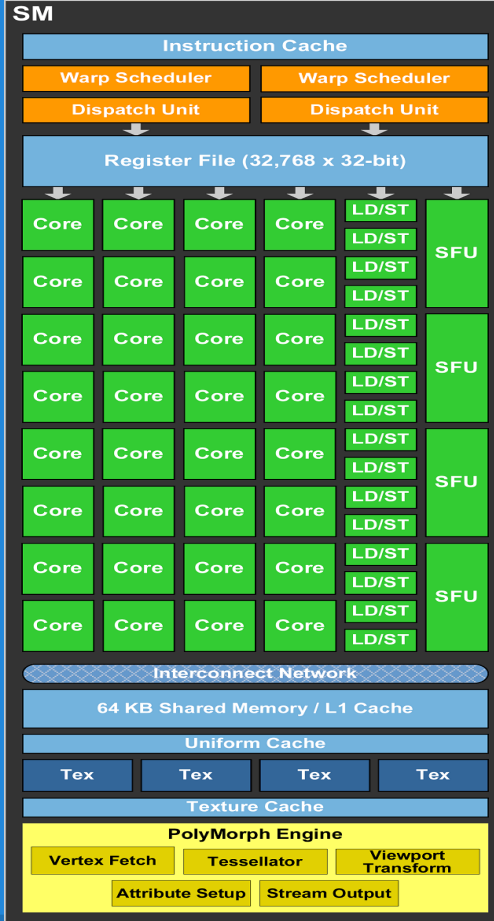
Engineering names:

GT218 → GF110 → GK104 → GM206

Compute capabilities (CC):

1.x → 2.x → 3.x → 5.x

GPU → CC table [here](#)



GF110 SM

# Warps

warp = group of threads that always execute same instructions simultaneously

Single instruction multiple thread (SIMT) programming model, similar to SIMD (D=data)

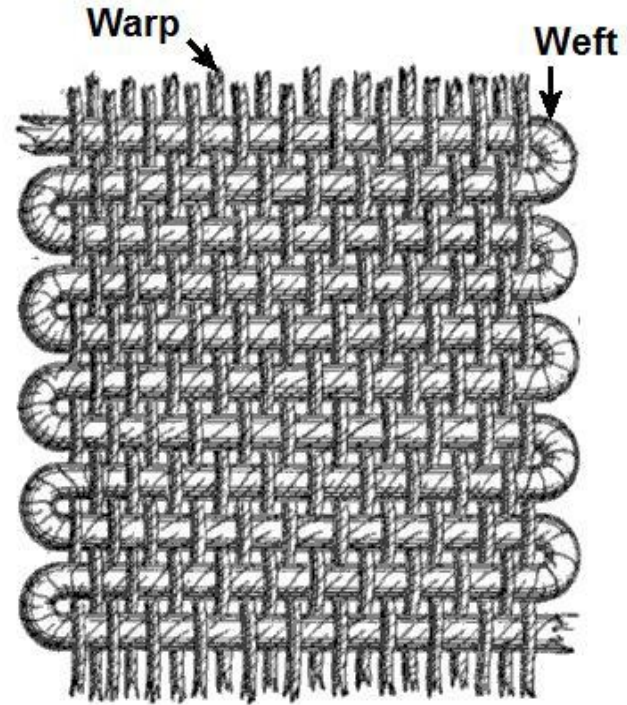
Warp size has always been 32 threads.



# Why warps?

Warps require less hardware than having each thread act independently

OpenCL equivalent is cooperative thread array (CTA)



# Warp Divergence

```
int idx = threadIdx.x + blockSize.x * (threadIdx.y + blockSize.y * threadIdx.z);
bool branch = (idx % 32 < 16);
if (branch) {
    foo();
} else {
    bar();
}
```

Different branches taken within a warp.

How can GPU handle this?


# Warp Divergence

```
int idx = threadIdx.x + blockSize.x * (threadIdx.y + blockSize.y * threadIdx.z);
bool branch = (idx % 32 < 16);
if (branch) {
    foo(); ←
} else {
    bar();
}
```

First the 16 threads with  
branch == 1 execute foo()  
while other 16 threads do  
nothing

# Warp Divergence

```
int idx = threadIdx.x + blockSize.x * (threadIdx.y + blockSize.y * threadIdx.z);
bool branch = (idx % 32 < 16);
if (branch) {
    foo();
} else {
    bar();
}
```



Then the other 16 threads execute `bar()` while the first 16 threads do nothing

# Warp Divergence

This clearly isn't good parallelism!

This phenomena is called “divergence”, and we want to avoid it!

Example was 2-way divergence, could have up to 32-way divergence :(

# A key idea of GPU programming

Have threads in the same warp do very similar work!

Limits divergence, helps with memory coalescing and avoiding shared memory bank conflicts (more on that next time)

# Latency & Throughput

CPU = low latency, low throughput

GPU = high latency, high throughput

CPU clock = 3 GHz (3 clocks/ns)

GPU clock = 1 GHz (1 clock/ns)

# Instruction & Memory Latency

CPU main memory latency: ~100ns ↗

GPU main memory latency: ~300ns ↗

CPU arithmetic instruction latency: ~1ns ↗

GPU arithmetic instruction latency: ~10ns ↗

GPU numbers are for Kepler; Fermi is ~2x this



# Hide the latency!

- Another key to GPU programming is hiding latency.
- While one warp waits on IO or an arithmetic instruction to finish executing, have another warp start executing an instruction!
- More warps  $\Rightarrow$  hide more latency



Chairlifts hide latency!

# SM's execute warps

Warps execute on a single SM.

SM's warp schedulers find a warp that is ready to execute and issues it instructions.

ready to execute  $\Leftrightarrow$  next instruction doesn't depend on currently executing instructions.

# Instruction Dependencies

acc += x0;

acc += x1;

acc += x2;

acc += x3;

...

All instructions write to same register...

Next instruction can't begin executing until previous finishes

# Instruction Dependencies

```
acc0 += x0;
```

```
acc1 += x1;
```

```
acc0 += x2;
```

```
acc1 += x3;
```

```
...
```

```
acc = acc0 + acc1;
```

Adds to `acc0` and `acc1` are independent and can run in parallel!

This is called instruction level parallelism (ILP)

# Adding coordinates (naive)

```
z0 = x[0] + y[0];  
z1 = x[1] + y[1];
```

COMPILATION



```
x0 = x[0];  
y0 = y[0];  
z0 = x0 + y0;
```

```
x1 = x[1];  
y1 = y[1];  
z1 = x1 + y1;
```

How is this “assembly” sub-optimal?

# Adding coordinates (ILP)

```
z0 = x[0] + y[0];  
z1 = x[1] + y[1];
```

BETTER COMPILATION

```
x0 = x[0];  
y0 = y[0];  
x1 = x[1];  
y1 = y[1];
```

```
z0 = x0 + y0;  
z1 = x1 + y1;
```



Instruction ordering impacts performance

# Warp Schedulers

Warp schedulers find a warp that is ready to execute its next instruction and available execution cores and then start execution.

GK110: 4 warp schedulers, 2 dispatchers each. Starts instructions in up to 4 warps each clock, and starts up to 2 instructions in each warp.



# Grid, blocks?

Grid has nothing to do with execution.

Each block executes on a single SM (but a SM can execute multiple blocks concurrently)

Shape of threads in block has nothing to do with execution.

# GK110 (Kepler) numbers

- max threads / SM = 2048 (64 warps)
- max threads / block = 1024 (32 warps)
- 32 bit registers / SM = 64k
- max shared memory / SM = 48KB

The number of blocks that run concurrently on a SM depends on the resource requirements of the block!

# Occupancy

occupancy = warps per SM / max warps per SM

max warps / SM depends only on GPU

warps / SM depends on warps / block,  
registers / block, shared memory / block.

# GK110 Occupancy picture

## **100% occupancy**

- 2 blocks of 1024 threads
- 32 registers/thread
- 24KB of shared memory / block

## **50% occupancy**

- 1 block of 1024 threads
- 64 registers/thread
- 48KB of shared memory / block

# Next time

We've looked at compute architecture...

Memory IO also has a huge effect on performance

Learn about memory systems on Wednesday