CS 179: GPU Computing

LECTURE 4: GPU MEMORY SYSTEMS

Last time

- Each block is assigned to and executed on a single streaming multiprocessor (SM).
- Threads execute in groups of 32 called warps.
  - Threads in a single warp can only run 1 set of instructions at once
  - Can cause divergence if threads are told to perform different tasks

NVIDIA Architecture Names

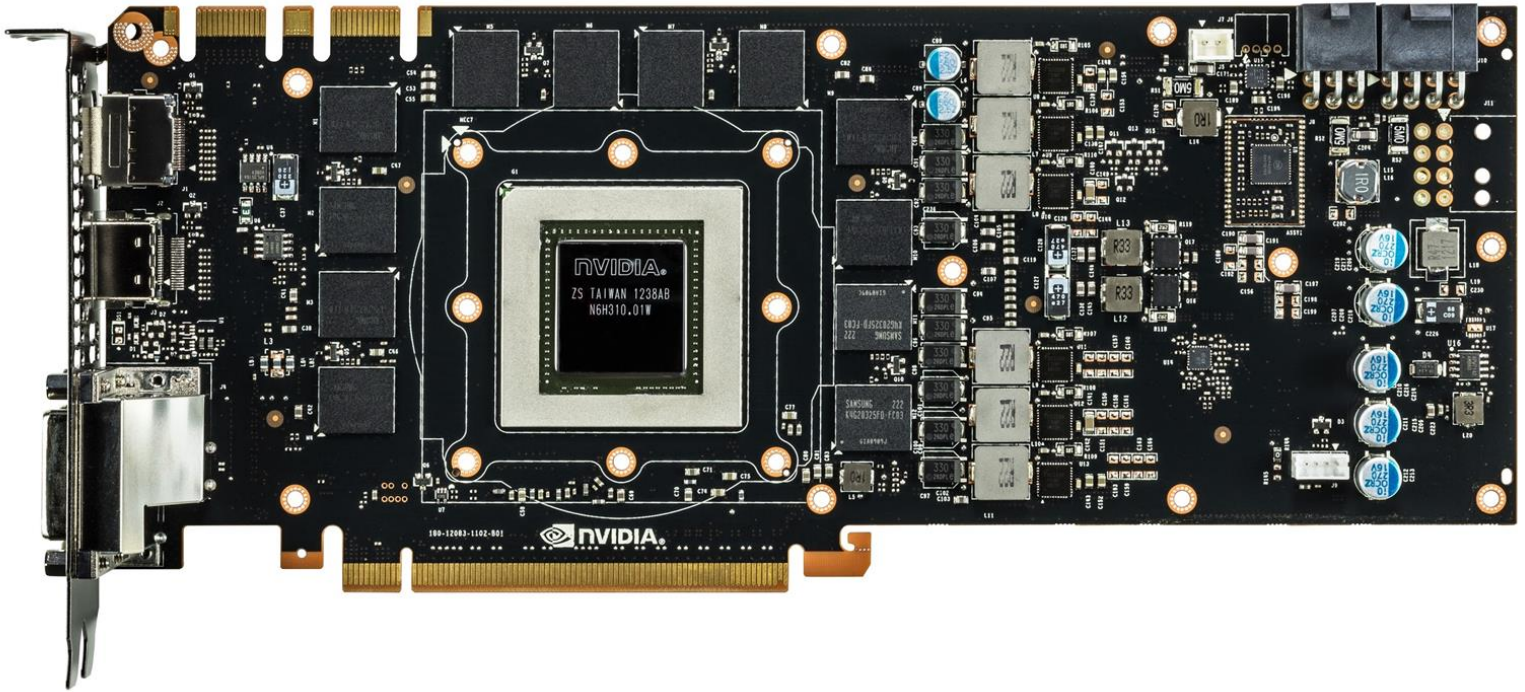
Architecture Family Names

- Tesla (2006) -> Fermi (2010) -> Kepler (2012) -> Maxwell (2014) -> …
- Engineering Names: GT218 -> GF110 -> GK104 -> GM206 -> ...

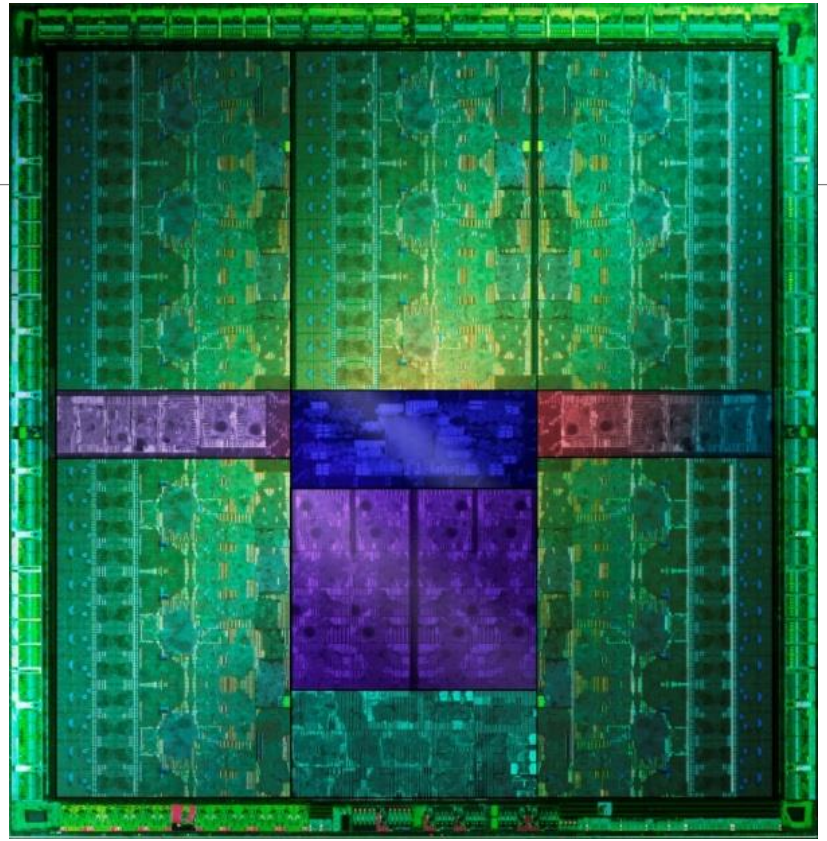
Each new Architecture comes with new capabilities

- Compute Capability (CC) tells you the set of capabilities included
- 1.x -> 2.x -> 3.x -> 4.x -> …

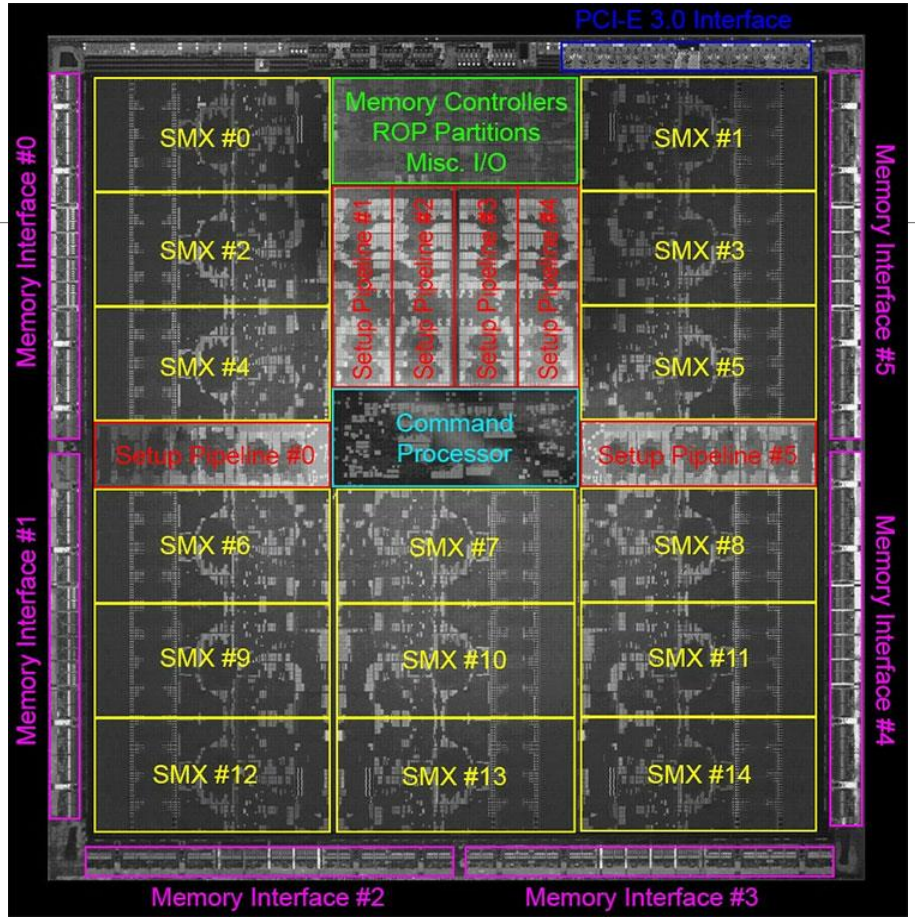
For more detail: https://developer.nvidia.com/cuda-gpus



Nvidia GeForce GTX 780



GK110


PCI-E 3.0 Interface
Memory Interface #0
Memory Interface #5
Memory Interface #1
Memory Interface #4
Memory Interface #2
Memory Interface #3
SMX #0
SMX #1
SMX #2
SMX #3
SMX #4
SMX #5
SMX #6
SMX #7
SMX #8
SMX #9
SMX #10
SMX #11
SMX #12
SMX #13
SMX #14
Memory Controllers
ROP Partitions
Misc. I/O
Setup Pipeline #1
Setup Pipeline #2
Setup Pipeline #3
Setup Pipeline #4
Command
Processor
Setup Pipeline #0
Setup Pipeline #5

GK110

Latency & Throughput

Latency is the delay caused by the physical speed of the hardware

CPU = low latency, low throughput

- CPU clock = 3 GHz (3 clocks/ns)
- CPU main memory latency: ~100+ ns
- CPU arithmetic instruction latency: ~1+ ns

GPU = high latency, high throughput

- GPU clock = 1 GHz (1 clock/ns)
- GPU main memory latency: ~300+ ns
- GPU arithmetic instruction latency: ~10+ ns

Above numbers were for Kepler GPUs (e.g. GTX 700 series)

For Fermi, latencies tend to be double that of Kepler GPUs

Compute & IO Throughput

GeForce GTX Titan Black (GK110 based)

Compute throughput	5 TFLOPS (single precision)
Global memory bandwidth	336 GB/s (84 Gfloat/s)

- GPU is very IO limited! IO is very often the throughput bottleneck, so its important to be smart about IO.
- If you want to get beyond ~900 GFLOPS, need to do multiple FLOPs per shared memory load.

Cache

A **cache** is a chunk of memory that sits in between a larger pool of memory and the processor
- Often times implemented at hardware level
- Has much faster access speed than the larger pool of memory

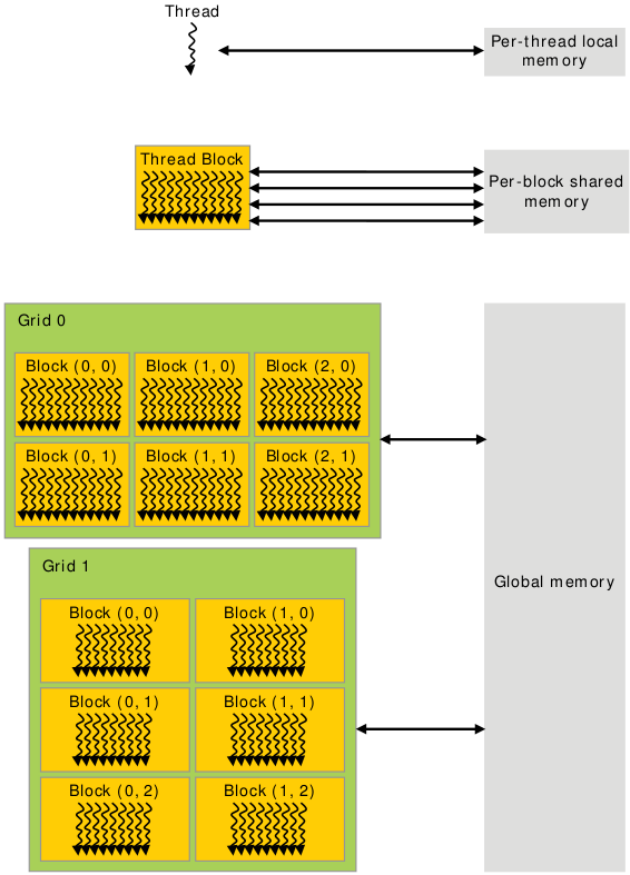
When memory is requested, extra memory near the requested memory is read into a cache
- Amount read is cache and memory pool specific
  - Regions of memory that will always be cached together are called **cache lines**
- This makes future accesses likely to be found in the cache
  - Such accesses are called **cache hits** and allow much faster access
  - If an access is not found in the cache, it's called a **cache miss** (and there is obviously no performance gain)

GPU Memory Breakdown

- Registers
- Local memory
- Global memory
- Shared memory
- L1/L2/L3 cache
- Constant memory
- Texture memory
- Read-only cache (CC 3.5+)

Part 1

- Registers
- Local memory
- Global memory
- Shared memory



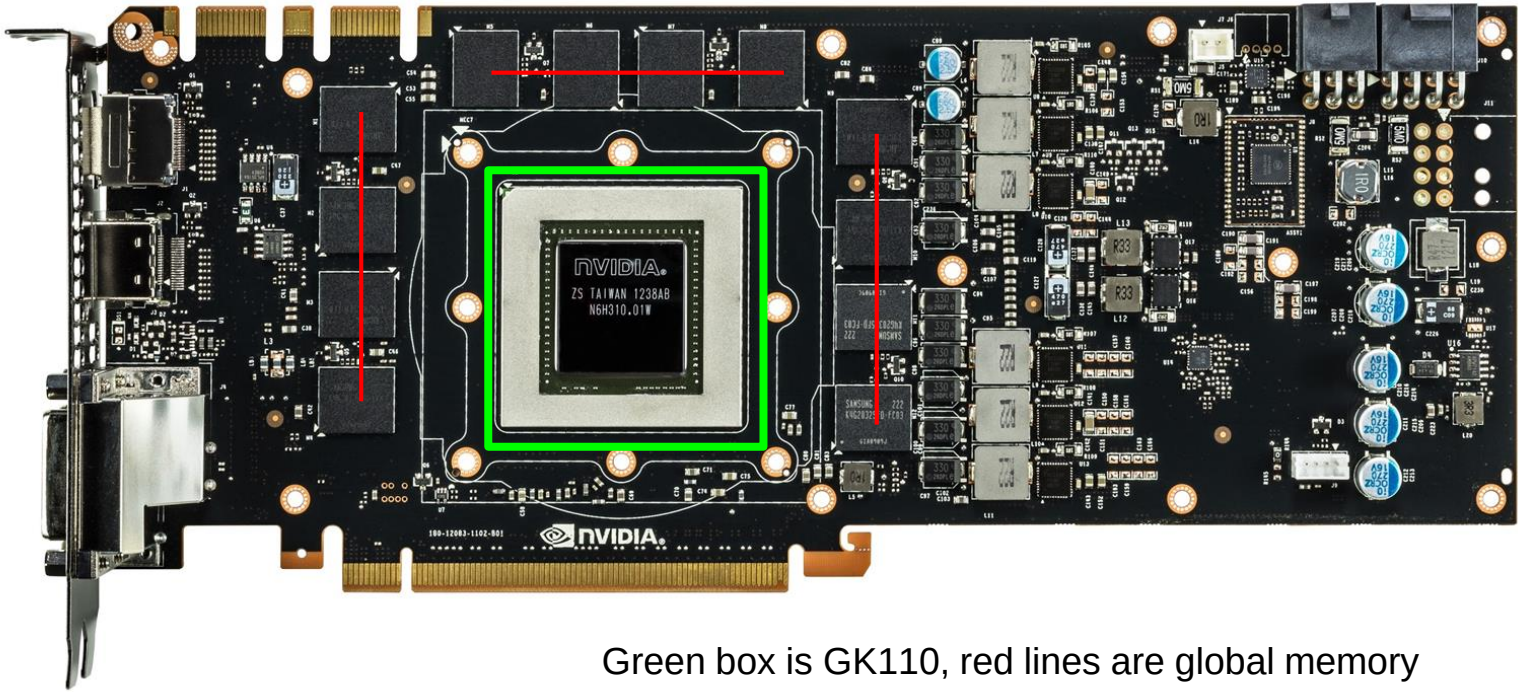
Memory Scope

Global Memory

Global memory is separate hardware from the GPU core (containing SM's, caches, etc).

- The vast majority of memory on a GPU is global memory
- If data doesn't fit into global memory, you are going to have process it in chunks that do fit in global memory.
- GPUs have .5 - 24GB of global memory, with most now having ~2GB.

Global memory latency is ~300ns on Kepler and ~600ns on Fermi



Green box is GK110, red lines are global memory

Nvidia GeForce GTX 780

Accessing global memory efficiently

Global memory IO is the slowest form of IO on GPU

- except for accessing host memory (duh...)

Because of this, we want to access global memory as little as possible

Access patterns that play nicely with GPU hardware are called **coalesced memory accesses.**

Memory Coalescing

Memory accesses are done in large groups setup as **Memory Transactions**

- Done per warp
- Fully utilizes the way IO is setup at the hardware level

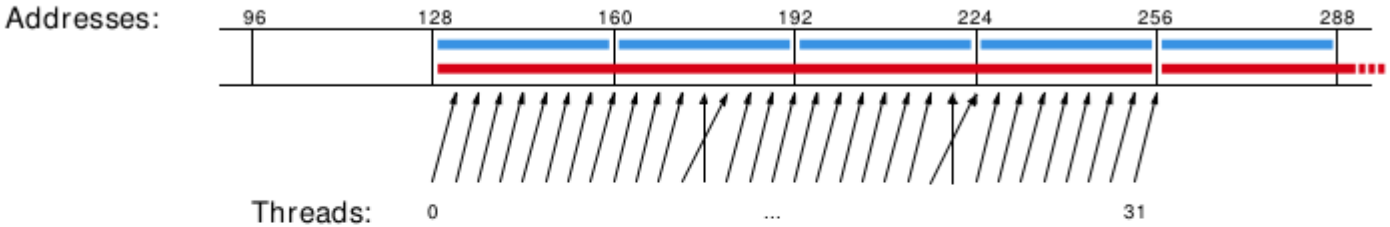
Coalesced memory accesses minimize the number of cache lines read in through these memory transactions

- GPU cache lines are 128 bytes and are aligned

Memory coalescing is much more complicated in reality

- See Ch 5 of the textbook for more detail if you're interested but it's not required (will be emailed out to the class late tonight.)

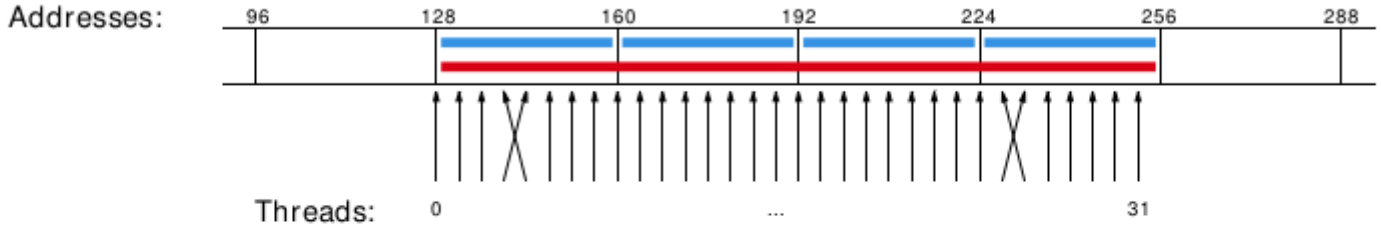
Mis-aligned accesses (sequential/ non-sequential)



Compute capability:	2.x, 3.x, 5.x	
Memory transactions:	Uncached	Cached
	1x 32B at 128 1x 32B at 160 1x 32B at 192 1x 32B at 224 1x 32B at 256	1x 128B at 128 1x 128B at 256

Misalignment can cause non-coalesced access

Aligned accesses (sequential/ non- sequential)



Compute capability:	2.x, 3.x, 5.x	
Memory transactions:	Uncached	Cached
	1x 32B at 128 1x 32B at 160 1x 32B at 192 1x 32B at 224	1x 128B at 128

A coalesced access!

Shared Memory

- Very fast memory located in the SM
- Same hardware as L1 cache (will discuss later)
  - ~5ns of latency
- Maximum size of ~48KB (varies per GPU)
- Scope of shared memory is the block

Remember
SM = streaming multiprocessor
SM ≠ shared memory

Shared memory syntax

Can allocate shared memory statically (size known at compile time) or dynamically (size not known until runtime)

Static allocation syntax:

- `__shared__ float data[1024];`
- Declared in the kernel, nothing in host code

Dynamic allocation syntax

- Host:
  - `kernel<<<grid_dim, block_dim, numBytesShMem>>>(args);`
- Device (in kernel):
  - `extern __shared__ float s[];`
- For multiple dynamically sized variables, see this blog post
  - Little bit more complicated and there are easy alternatives

A shared memory application

Task: Compute byte frequency counts
Input: array of bytes of length n
Output: 256 element array of integers containing number of occurrences of each byte

Naive: build output in global memory, n global stores
Smart: build output in shared memory, copy to global memory at end, 256 global stores

Computational Intensity

Computational intensity is a representation of how many operations must be done on a single data point (FLOPs / IO)

- Vaguely similar to the big O notation in concept and usage
- e.x.
  - Matrix multiplication: $n^3 / n^2 = n$
  - n-body simulation: $n^2 / n = n$

If computational intensity is > 1, then same data used in more than 1 computation

- Do as few global loads and as many shared loads as possible

A common pattern in kernels

(1) copy from global memory to shared memory
(2) `__syncthreads()`
(3) perform computation, incrementally storing output in shared memory, `__syncthreads()` as necessary
(4) copy output from shared memory to output array in global memory

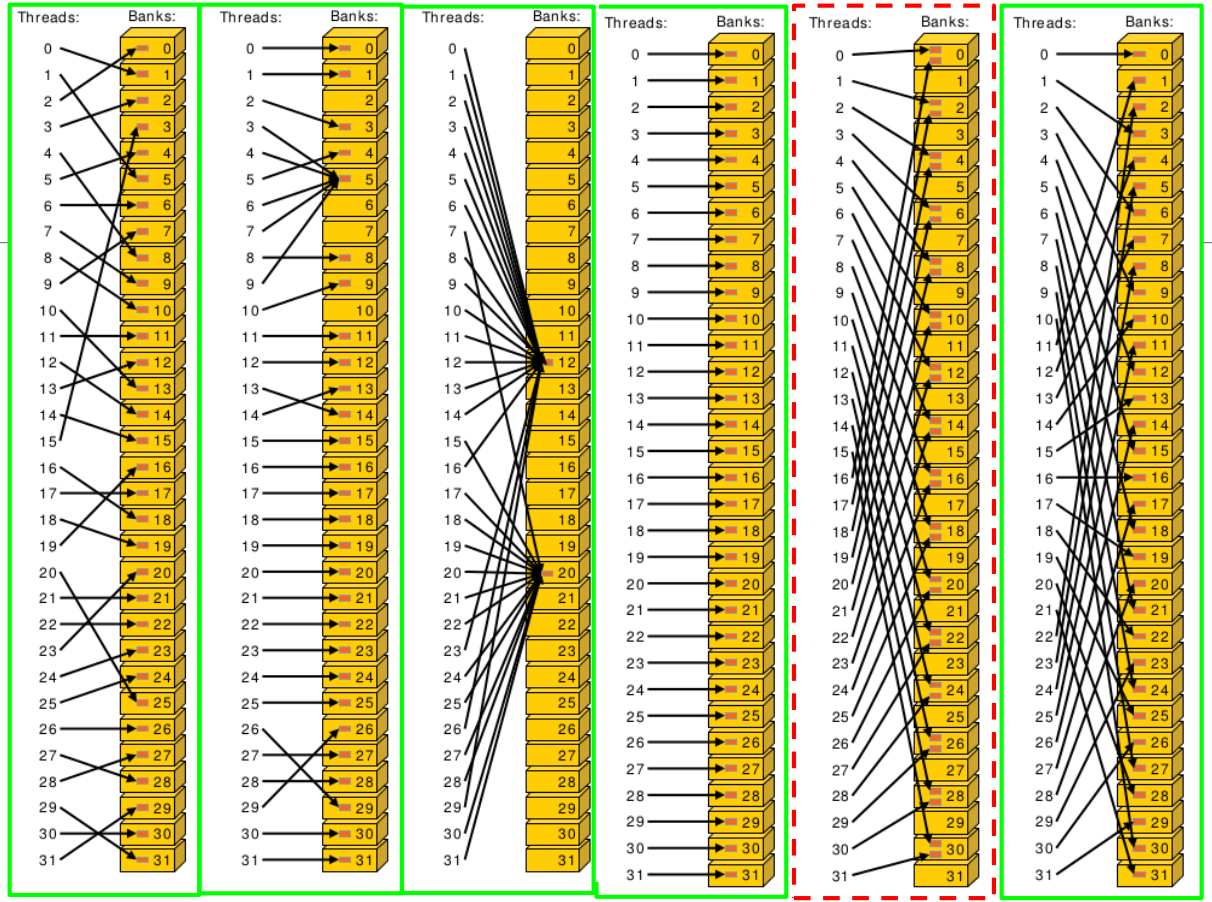
Bank Conflicts

Shared memory is setup as 32 **banks**

- If you divide the shared memory into 4 byte-long elements, element `i lies` in bank `i % 32.`

A **bank conflict** occurs when 2 threads in a warp access different elements in the same bank.

- Bank conflicts cause serial memory accesses rather than parallel
  - Serial *anything* in GPU programming = bad for performance



Bank conflict examples

Bank conflicts and strides

Stride 1 ⇒ 32 x 1-way “bank conflicts” (so conflict-free)
Stride 2 ⇒ 16 x 2-way bank conflicts
Stride 3 ⇒ 32 x 1-way “bank conflicts” (so conflict-free)
Stride 4 ⇒ 8 x 4-way bank conflicts
…
Stride 32 ⇒ 1 x 32-way bank conflict :(

Padding to avoid bank conflicts

To fix the stride 32 case, we'll waste a byte on padding and make the stride 33 :)

Don't store any data in slots 32, 65, 98, ...
Now we have
thread 0 ⇒ index 0 (bank 0)
thread 1 ⇒ index 33 (bank 1)
thread `i` ⇒ index `33 * i` (bank `i`)

Registers

A **Register** is a piece of memory used directly by the processor
- Fastest “memory” possible, about 10x faster than shared memory
- There are tens of thousands of registers in each SM
  - Generally works out to a maximum of 32 or 64 32-bit registers per thread

Most stack variables declared in kernels are stored in registers
- example: `float x; (duh...)`

Statically indexed arrays stored on the stack are sometimes put in registers

Local Memory

Local memory is everything on the stack that can't fit in registers

The scope of local memory is just the thread.

Local memory is stored in global memory

- much slower than registers

Register spilling example

When we have enough registers, this code does 4 loads from local memory and 0 stores.

Now assume we only have 3 free registers before any of this code is executed (but don't worry about `z0` and `z1`)

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

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

Register spilling example

starting with only 3 free registers...

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

cannot load `y[1]` until we free a register. store `x1` to make space.

Now we need to load `x1` again.

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

Register spilling cost:
1 extra load
1 extra store
2 extra pairs of consecutive dependent instructions

Questions?

- Global memory
- Local memory
- Shared memory
- Registers

Part 2

- L1/L2/L3 cache
- Constant memory
- Texture memory
- read-only cache (CC 3.5)

L1 Cache

- Fermi - caches local & global memory
- Kepler, Maxwell - only caches local memory
- same hardware as shared memory
- Nvidia used to allow a configurable size (16, 32, 48KB), but dropped that in recent generations
- each SM has its own L1 cache

L2 cache

- caches all global & local memory accesses
- ~1MB in size
- shared by all SM’s

L3 Cache

- Another level of cache above L2 cache
- Slightly slower (increased latency) than L2 cache but also larger.

Constant Memory

Constant memory is global memory with a special cache

- Used for constants that cannot be compiled into program
- Constants must be set from host before running kernel.

~64KB for user, ~64KB for compiler

- kernel arguments are passed through constant memory

Constant Cache

8KB cache on each SM specially designed to broadcast a single memory address to all threads in a warp (called static indexing)

- Can also load any statically indexed data through constant cache using “load uniform” (LDU) instruction
- Go to http://www.nvidia.com/object/sc10_cuda_tutorial.html for more details

Constant memory syntax

In global scope (outside of kernel, at top level of program):

```
__constant__ int foo[1024];
```

In host code:

```
cudaMemcpyToSymbol(foo, h_src, sizeof(int) * 1024);
```

Texture Memory

Complicated and only marginally useful for general purpose computation
Useful characteristics:

- 2D or 3D data locality for caching purposes through “CUDA arrays”. Goes into special texture cache.
- fast interpolation on 1D, 2D, or 3D array
- converting integers to “unitized” floating point numbers

Use cases:

(1) Read input data through texture cache and CUDA array to take advantage of spatial caching. This is the most common use case.
(2) Take advantage of numerical texture capabilities.
(3) Interaction with OpenGL and general computer graphics

Texture Memory

And that’s all we’re going to say on texture memory.
It’s a complex topic, you can learn everything you want to know about it from the textbook

Read-Only Cache (CC 3.5+)

Many CUDA programs don't use textures, but we should take advantage of the texture cache hardware.

CC ≥ 3.5 makes it much easier to use texture cache.

- Many `const restrict` variables will automatically load through texture cache (also called read-only cache).
- Can also force loading through cache with `__ldg` intrinsic function

Differs from constant memory because doesn't require static indexing

Questions?

- Registers
- Global memory
- Local memory
- Shared memory
- L1/L2/L3 cache
- Constant memory
- Texture memory
- Read-only cache (CC 3.5)

Next time….

Next lecture will be the last lecture focused on basic GPU knowledge
Few more optimization techniques
Synchronization
Will discuss GPU specific algorithms starting next week