CS 179 Lecture 5

GPU Memory Systems

Last time

- A block executes on a single streaming multiprocessor (SM).
- Threads execute in groups of 32 called warps.
- Want threads in a warp to do the same thing to avoid divergence.
- SMs hide latency by executing instructions for multiple warps at once.

Last time

- Minimize instruction dependencies to take advantage of instruction level parallelism (ILP)
- *Occupancy* allows us to reason about how well we are using hardware (but higher occupancy isn't always better)

Final notes on compute

- Integer instructions (especially / and %) slow
- GPUs have >4GB of RAM, so pointers are 64 bits :(
- All instructions have dependencies on previous writes to memory
- CPU hyper-threading is similar to what GPU does. 2 concurrent hyper threads processing 16 elements (AVX-512) on CPU rather than 64 concurrent warps processing 32 elements on SM

Clarification on dependencies

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

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

An instruction cannot start executing until

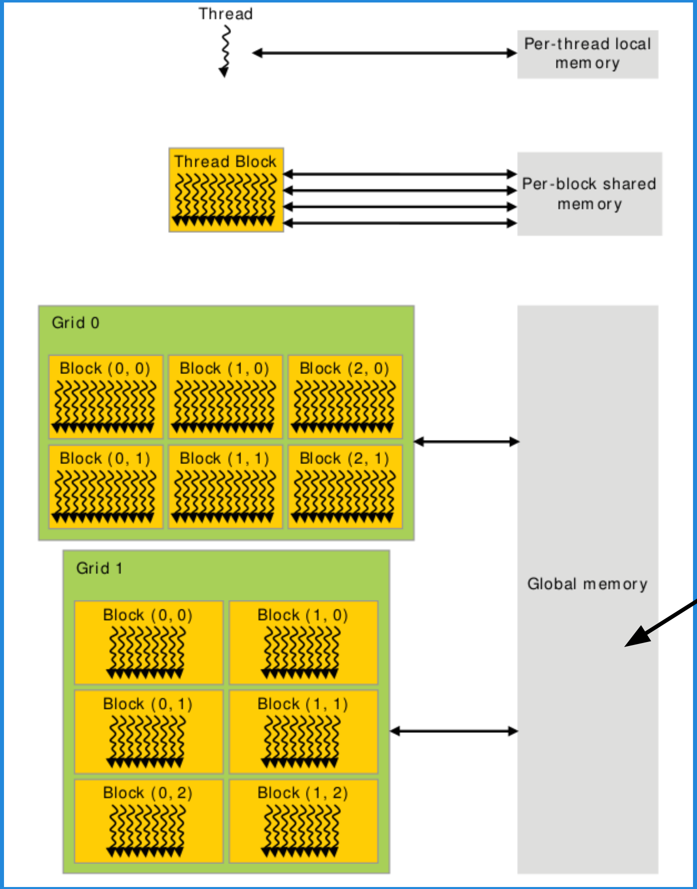
(1) all of its dependencies have finished executing

(2) all of the instructions before it have at least started executing (which mean dependencies for all previous instructions are met)

GPU Memory Breakdown

- Global memory & local memory
- Shared memory & L1 cache
- Registers
- Constant memory
- Texture memory & read-only cache (CC 3.5)
- L2 cache

Memory types implemented on same hardware are grouped.



constant memory

Memory Scope

Global Memory

Global memory is the “main” memory of the GPU. It has global scope and lifetime of the allocating program (or until `cudaFree` is called).

Global memory is similar to the heap in a C program.

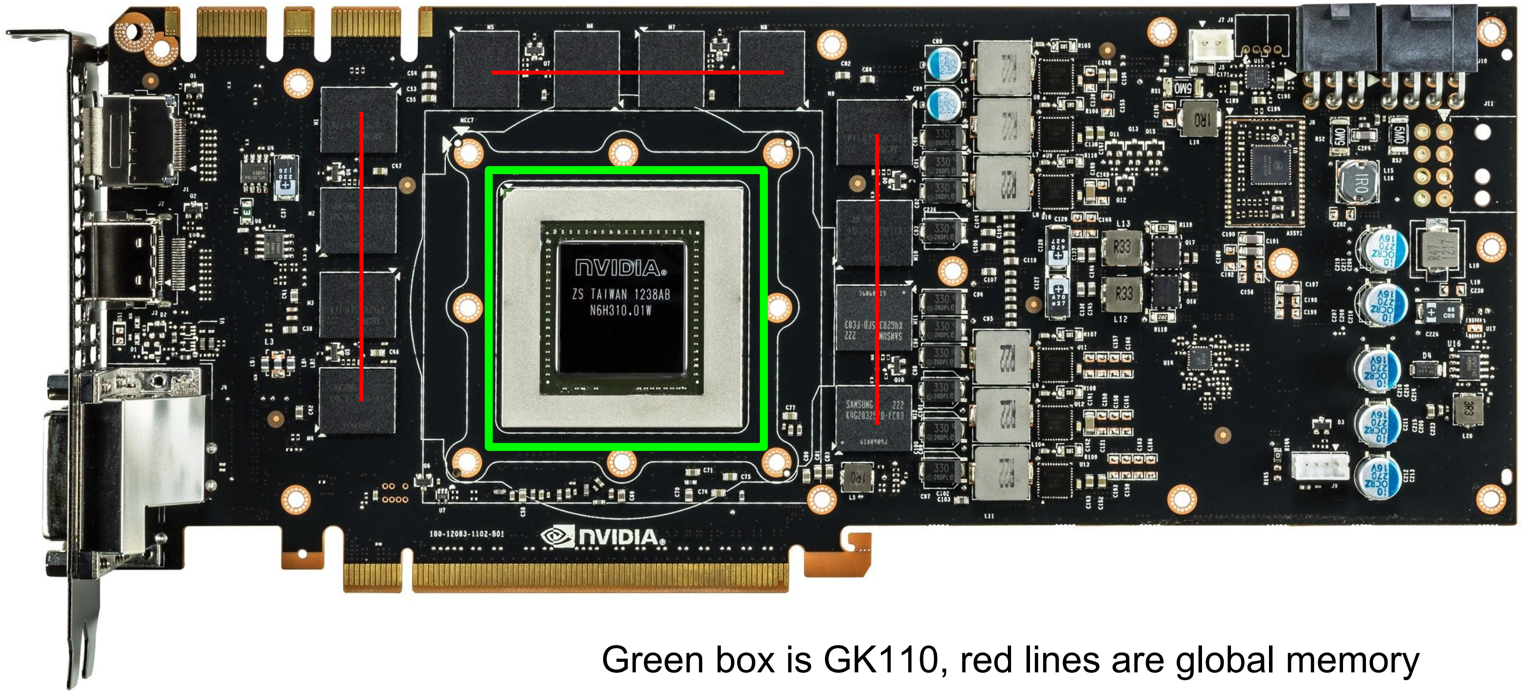
Global Memory syntax

Allocate with

```
cudaMalloc(void** devPtr, size_t size)
```

Free with

```
cudaMalloc(void* devPtr)
```



Green box is GK110, red lines are global memory

Nvidia GeForce GTX 780

Global Memory Hardware

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

The vast majority of memory on 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

Accessing global memory efficiently

IO often dominates computation runtime, and global memory IO is the slowest form of IO on GPU (except for accessing host memory).

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

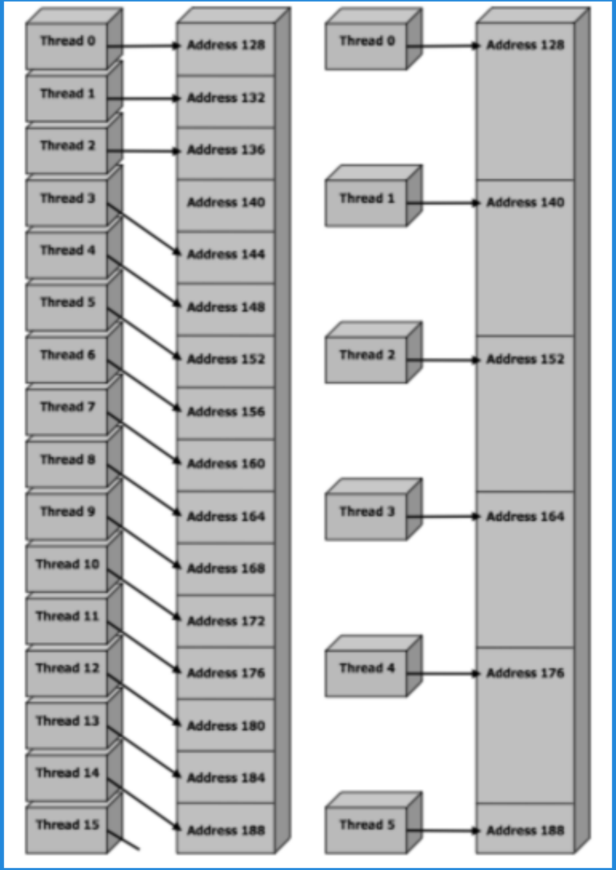
Access patterns that play nicely with GPU hardware are called *coalesced memory accesses*.

Memory Coalescing

Memory coalescing is a bit more complicated in reality (see Ch 5.2 of CUDA Handbook), but there’s 1 simple thing to remember that will lead to coalesced accesses:

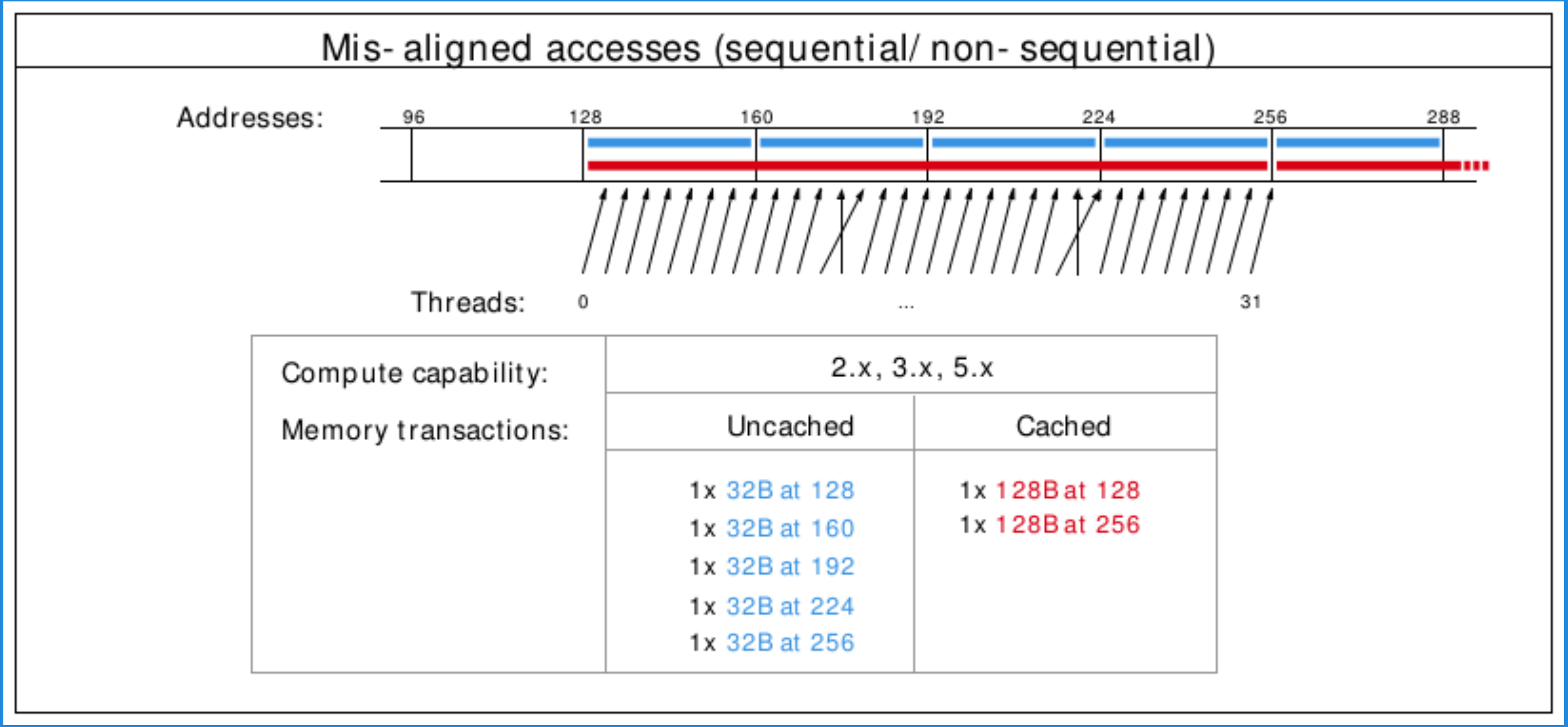
GPU cache lines are 128 bytes and are aligned. Try to make all memory accesses by warps touch the minimum number of cache lines (ideally 1 for 4 byte / warp accesses).

touches 2 cache lines



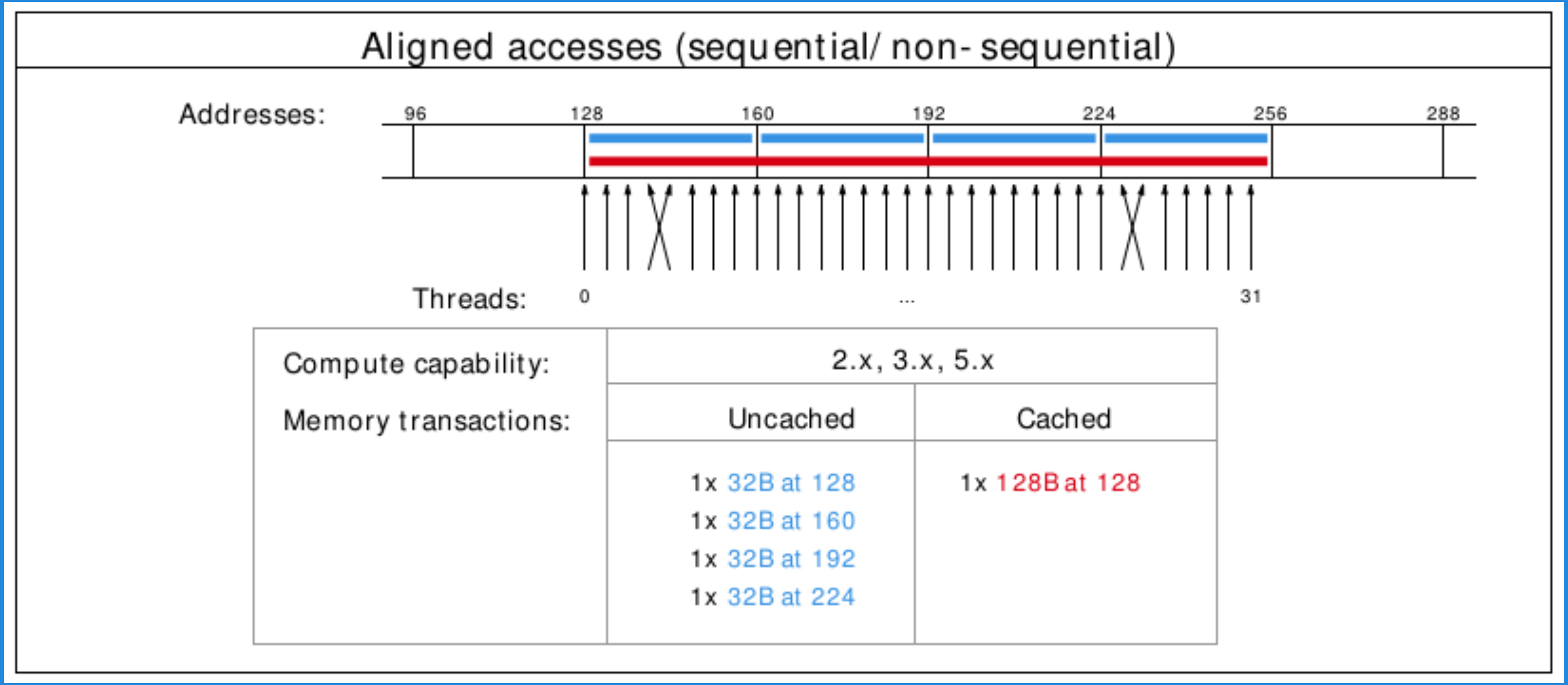
touches 3 cache lines

Two different non-coalesced accesses



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



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, ~5ns of latency
- Maximum size of 48KB, but user configurable
- 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];` declaration in kernel, nothing in host code

Shared memory dynamic allocation

Host:

```
kernel<<<grid, block, numBytesShMem>>>(arg);
```

Device (in kernel):

```
extern __shared__ float s[];
```

Some complexities with multiple dynamically sized variables, see this blog post

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 = FLOPs / IO

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 consists of 32 *banks* of width 4 bytes.

Element `i` is in bank `i % 32`.

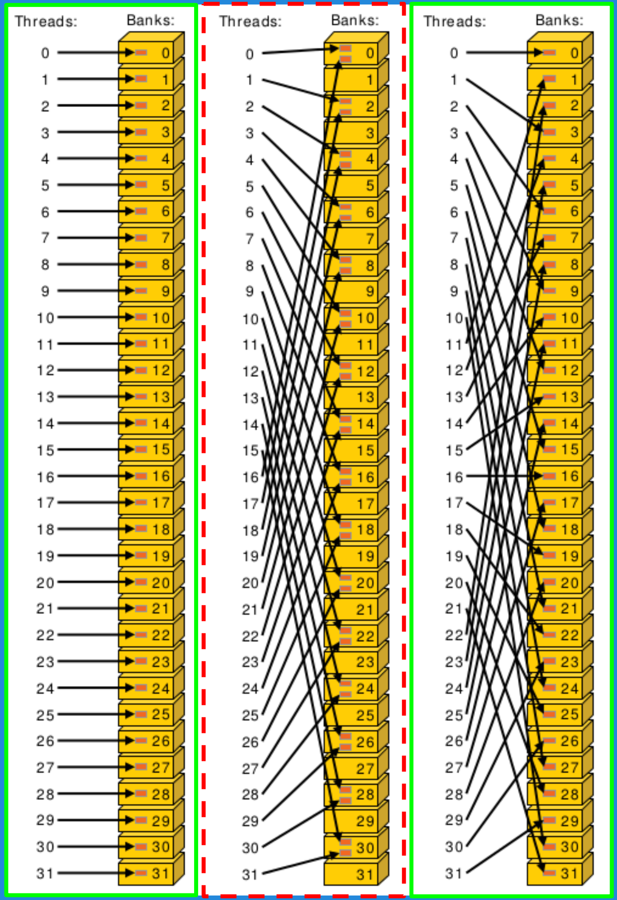
A *bank conflict* occurs when 2 threads in a warp access different elements in the same bank.

Bank conflicts cause serial memory accesses rather than parallel, are bad for performance.

Left: Conflict free with stride 1

Center: 2-way bank conflict due to stride 2

Right: Conflict free with stride 3

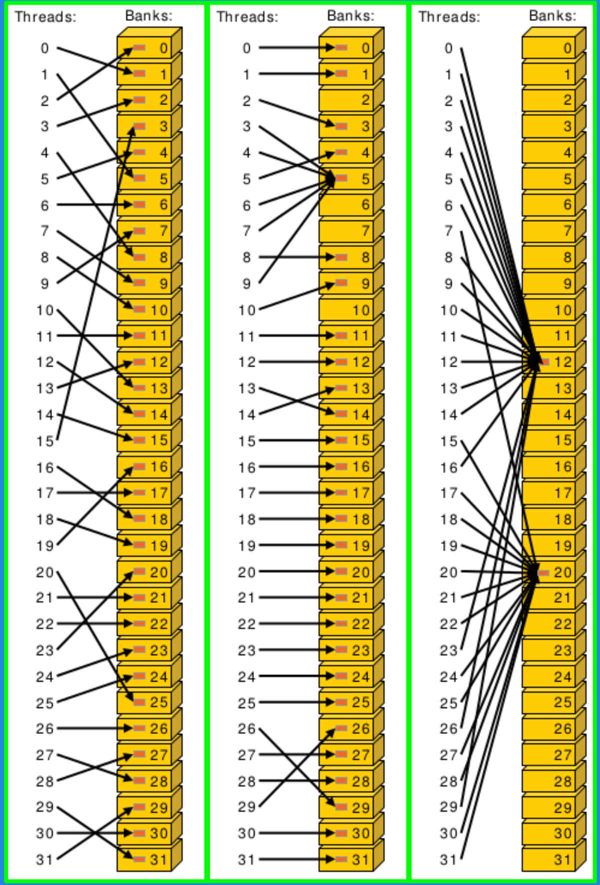


Bank conflict examples

Left: conflict-free

Center: conflict-free because same element accessed in bank 5

Right: conflict-free because same element accessed in banks 12 and 20. Broadcasting occurs.



More 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 :(

Can anyone think of a way to modify the data to have conflict-free access in the stride 32 case?

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`)

Bank conflicts and coalescing

Bank conflicts are the “noncoalesced access” equivalent for shared memory.

Note stride 1 accesses are both conflict-free and coalesced.

In the “load from global, store into shared, do quadratic computation on shared data” pattern, you sometimes have to choose between noncoalesced loads or bank conflicts on stores. Generally bank conflicts on stores will be faster, but it’s worth benchmarking. The important thing is that the shared memory loads in the “quadratic computation” part of the code are conflict-free (because there are more of these loads than either other operation).

Registers

Fastest “memory” possible, about 10x faster than shared memory

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

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

Recall coordinate addition from previous lecture.

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...

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

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

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

L1 Cache

- Fermi - caches local & global memory
- Kepler, Maxwell - only caches local memory
- same hardware as shared memory
- configurable size (16, 32, 48KB)
- each SM has its own L1 cache

L2 cache

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

Constant Memory

- Used for constants that cannot be compiled into program
- Constants must be set from host before running kernel.
- Constant memory is global memory with a special cache
- 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

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 for now, more on future set!

It’s a complex topic, you can learn everything you want to know about it from CUDA Handbook Ch 10

Read-Only Cache

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

Differs from constant memory because doesn’t require static indexing

Extra topic: vectorized IO

Besides vectorizing over the 32 threads in a warp, CUDA has instructions for each thread to do 64 or 128 bit loads/stores (rather than standard 32 bit transactions).

These transactions happen whenever an appropriately sized and aligned type is dereferenced. Alignment requirements are equal to type size, so a `double` must be 8 byte aligned, `float4` must be 16 byte aligned, etc.

Compute & IO Throughput

GeForce GTX Titan Black (GK110 based)

Compute throughput	5 TFLOPS (single precision)
Global memory bandwidth	336 GB/s (84 Gfloat/s)
Shared memory bandwidth	3.4 TB/s (853 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.

cuBLAS obtains about 4 TFLOPS on this GPU. Utilization is hard!