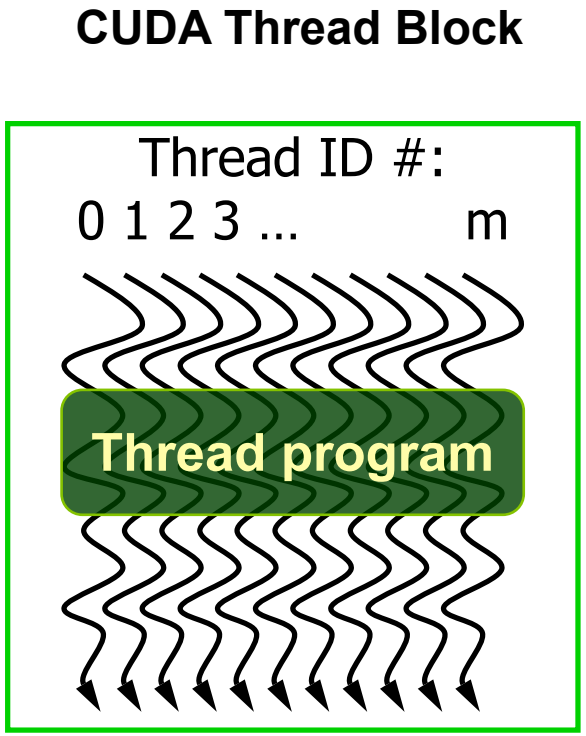
CUDA architecture and memory

CS 179: GPU Programming

Lecture originally written by Luke Durant, Tamas Szalay, Russell McClellan

Threads and warps

- Programmer runs a block with 1 to 512 threads
- All threads run same program (SPMD)
- Threads have unique thread IDs
- Threads scheduled as sets of “warps”
  - 32 threads per warp (not a CUDA standard)

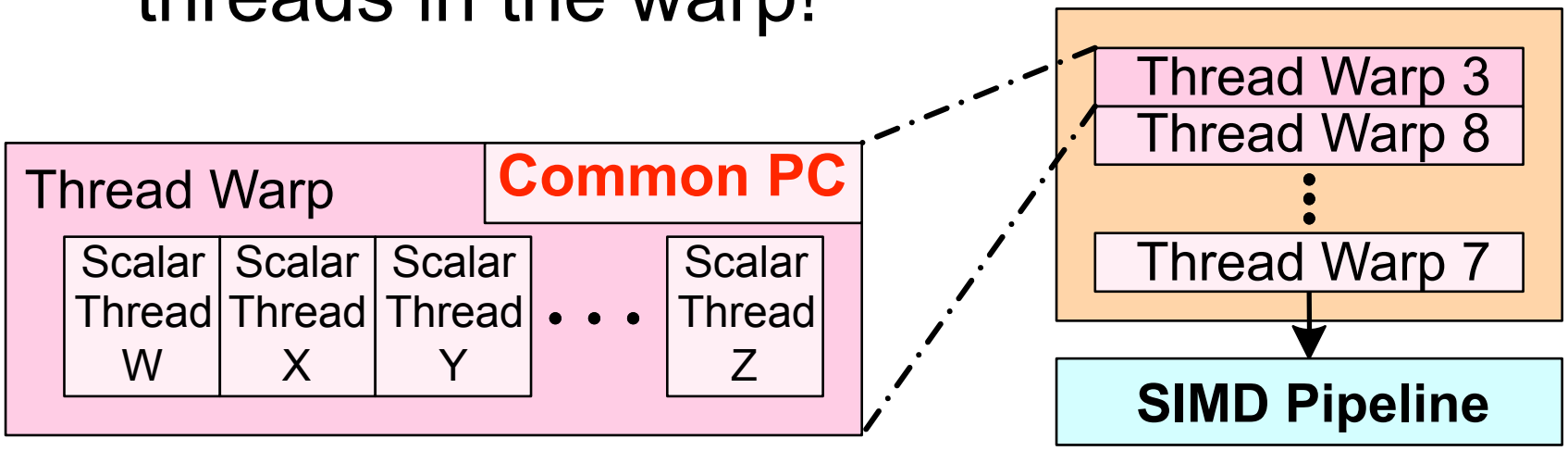


Multiple parallelism on GPU

- MIMD - Multiple Instruction, Multiple Data parallelism between MPs on GPU
  - Run potentially different blocks of code on each SM
- SIMD - Single Instruction, Multiple Data parallelism inside warps
  - Warp of 32 threads runs the same instruction on 32 different pieces of data

Warps

- 32 threads with sequential indices: [0..31], [32..63], ...
- SM executes same instruction on all threads in the warp!



Branch divergence

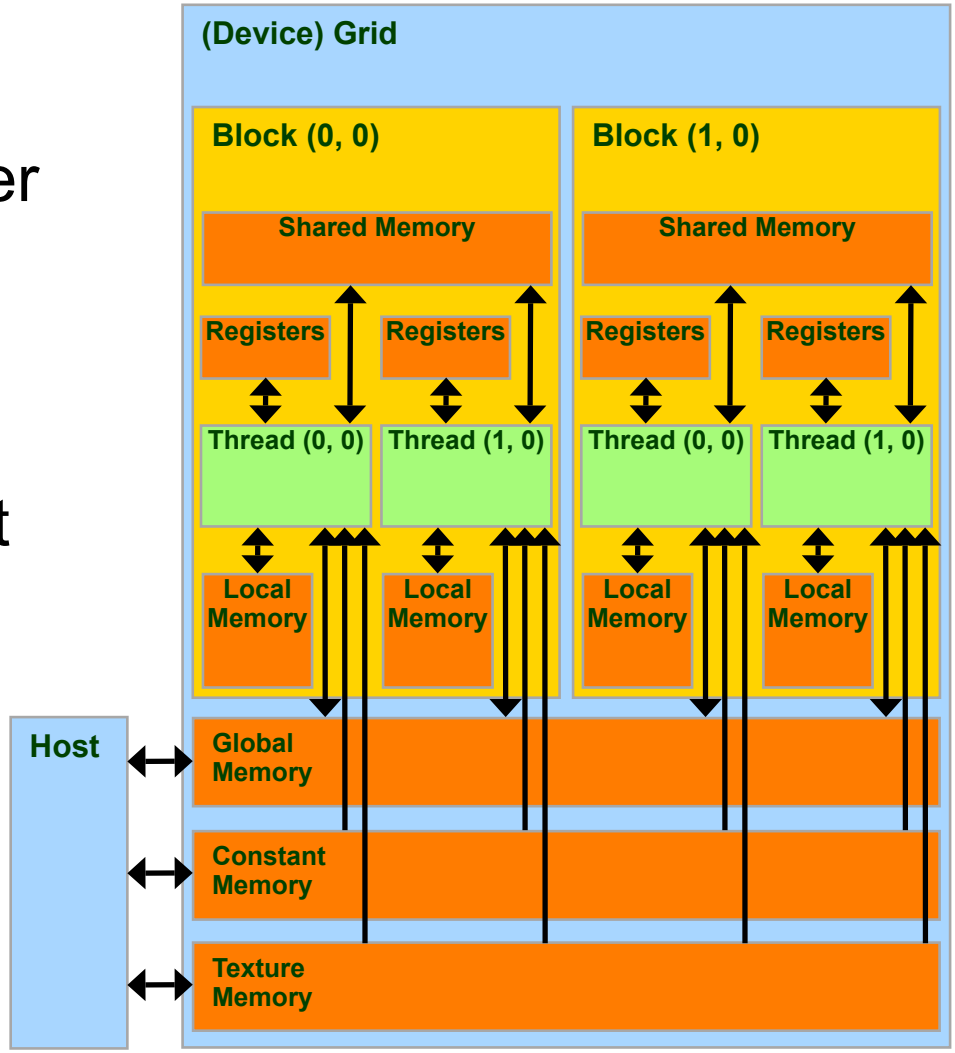
- Different threads in the same warp may follow different paths
  - `if (threadIdx.x < 2) a(); else b();`
- MP still executes the same instruction in all threads in the warp!
  - But tells some threads to do nothing
- SM then has to re-execute part of it
  - Run the other branch, tell the other set of threads to do nothing
  - Worst case: 32 threads run sequentially!

Avoiding branch divergence

- Sometimes you can't!
- Avoid if statements based on threadIdx
- Bad: if (threadIdx.x > 2)
- OK: if (threadIdx.x/WARP_SIZE > 2)
  - All threads in each warp still do the same thing
- More on this later...
  - Can sometimes use if/else without divergence

Memory architecture

- Threads can:
  - R/W per-thread Register
  - R/W per-thread Local
  - R/W per-block Shared
  - R/W per-grid Global
  - Read per-grid Constant
  - Read per-grid Texture
- Host can:
  - R/W/Alloc Global, Const, Texture
- Caching (automatic)

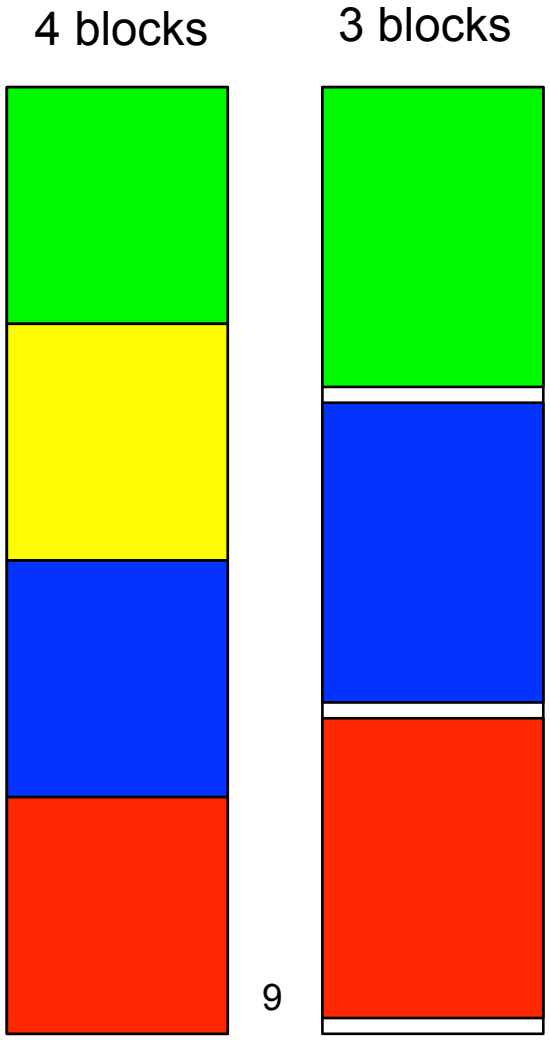


Memory: Registers

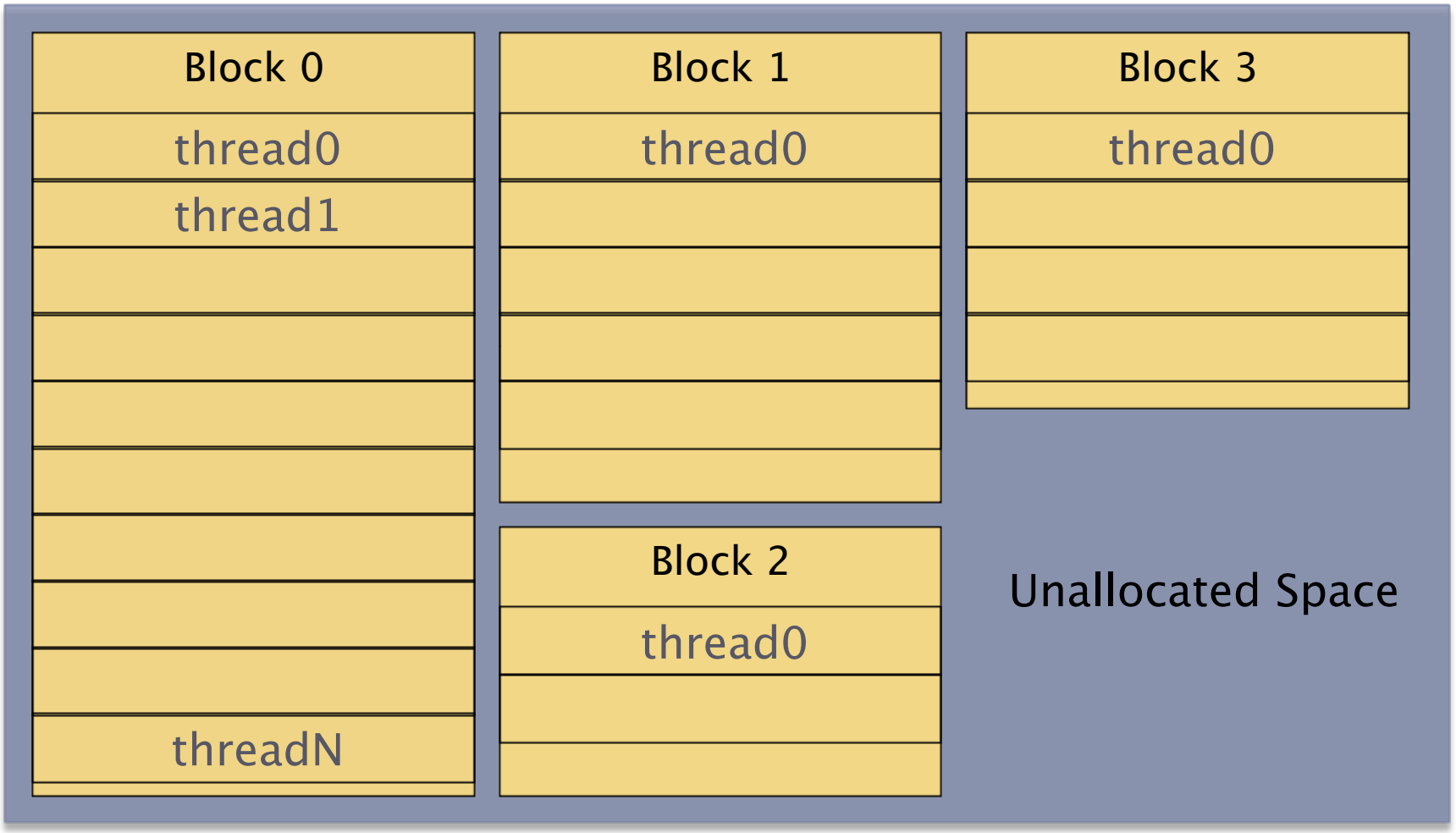
- One “register file” per multiprocessor
- Size of register file depends on multiprocessor
  - 32K of regs (8192 32-bit regs) per MP on G80, G92
  - 64K of regs (16384 32-bit regs) per MP on GTX 200
  - This is a hard limit! Can’t run many threads if each thread uses many registers!

Programmer's view of regs

- Registers dynamically partitioned across all blocks on SM
- Once assigned to a thread, a reg is not accessible to other threads in a block
- Each thread can only access regs assigned to it



Hardware's view of regs



Using registers

- Registers are read/write per-thread
  - Can’t access regs outside of each thread
  - Used for storing local variables in functions, etc.
  - No special syntax for doing this - just declare local variables as usual
- Physically stored in each MP
- Can’t index (no arrays)
- Obviously, can’t access from host code

Local memory

- Also read/write per-thread
- Can't read other threads' local memory
  - Declare a variable in local memory using the __local__ keyword
  - `__local__ float results[32];`
- Can index (this is where local arrays go)
- Much slower than register memory!
  - Don't use local arrays if you don't have to

Shared memory

- Read/write per-block
- All threads in a block share the same memory
- In general, pretty fast
  - Certain cases can hinder performance...

Using shared memory

- Similar to local memory:
  - `__shared__ float current_row[];`
- Only declare one variable as shared!
  - Multiple declarations of __shared__ variables will occupy same memory space!
  - `__shared__ float a[];`
  - `__shared__ float b[];`
  - `b[0] = 0.5f;`
  - `// now a[0] == 0.5f also!`

Global memory

- Read/write per-application
- Can share between blocks and grids
- Persistent across kernel executions
- Completely un-cached
- Really slow!

Using global memory

- Keyword: __device__
  - `__device__ float* matrix_data;`
- Can pass pointers to global memory into kernel, unlike other memory areas
- cudaMalloc (on host) gives you global memory
- cudaMallocPitch gives you global memory with more efficient pitch (talk about this later)

Using global memory

- cudaMemcpy
  - Copies from host to device global memory, or vice versa
- cudaMemcpy2D
  - Copies between arrays with different pitches
- Remember to specify which direction you're copying...
- Don't mix up your host/device pointers!

Using global memory

- cudaMemcpyToSymbol
  - Instead of pointer, takes symbol or char*
  - DeviceToDevice or HostToDevice
- cudaMemcpyFromSymbol
  - DeviceToHost only
- Use cudaGetSymbolAddress and cudaGetSymbolSize to find symbols

Constant memory

- Read-only from device
- Cached in each MP
- Cache can broadcast to every thread running - very efficient!

Using constant memory

- Keyword: __constant__
- Access from device code like normal variables
- Set values from host code with cudaMemcpyToSymbol
  - Can’t use pointers
  - Can’t dynamically allocate

Using constant memory

- cudaMemcpy[To/From]Symbol works with constant memory
- cudaGetSymbolAddress
  - Fails! Why?
  - All device pointers accessible by the host are in global memory space

Texture memory

- Read-only from device
- Complex 2D caching method
- Linear filtering/interpolation available, like GLSL's texture2D()

Memory coalescing

- Remember: global memory is slow!
- Often we want to access data indexed by the thread ID
- There is hardware support to copy large chunks of data at once
  - Using this to our advantage is called "memory coalescing"

Aligned accesses

- Threads can read 4, 8, or 16 bytes at a time from global memory
  - However, only if accesses are aligned!
- 4-byte reads must be aligned to 4-byte boundaries, 8-byte reads must be aligned to 8-byte boundaries, etc.
- So, it takes longer to read 16 bytes from 0xABCDE than from 0xABCD0

Aligned accesses

- Built-in types force correct alignment
  - Note that this means a float3 takes the same amount of space as a float4!
  - Arrays of float3 are NOT aligned!
- For structs, use the __align__(x) keyword (x = 4, 8, or 16)

Memory coalescing

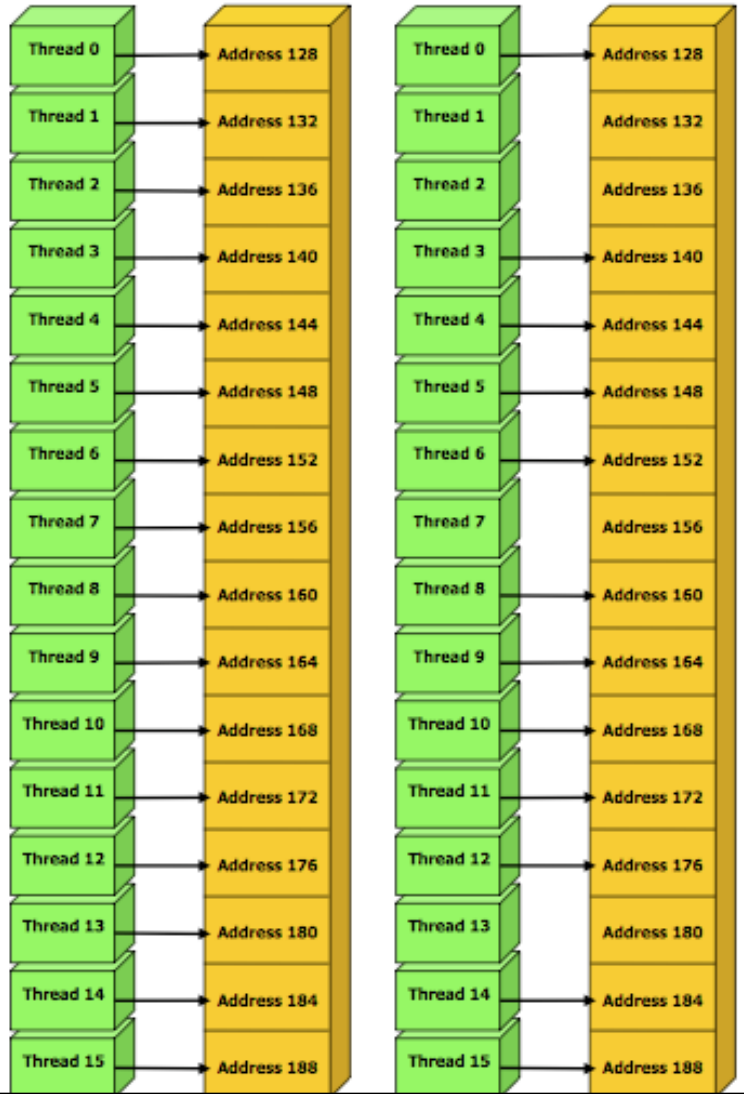
- Multiple threads making coalesced accesses are much faster
- Coalesced means:
  - Contiguous
  - In-order (thread 0 reads index 0, thread 1 reads index 1, etc.)
  - Aligned to a multiple of the total data size
    - If each thread reads a 4-byte float, thread 0 should access an address which is a multiple of 4*32

Memory coalescing

- Don’t worry about aligning the start of your memory blocks - cudaMalloc does this for you!
  - Remember to use cudaMalloc2D for 2-D arrays so that each row will be aligned as well

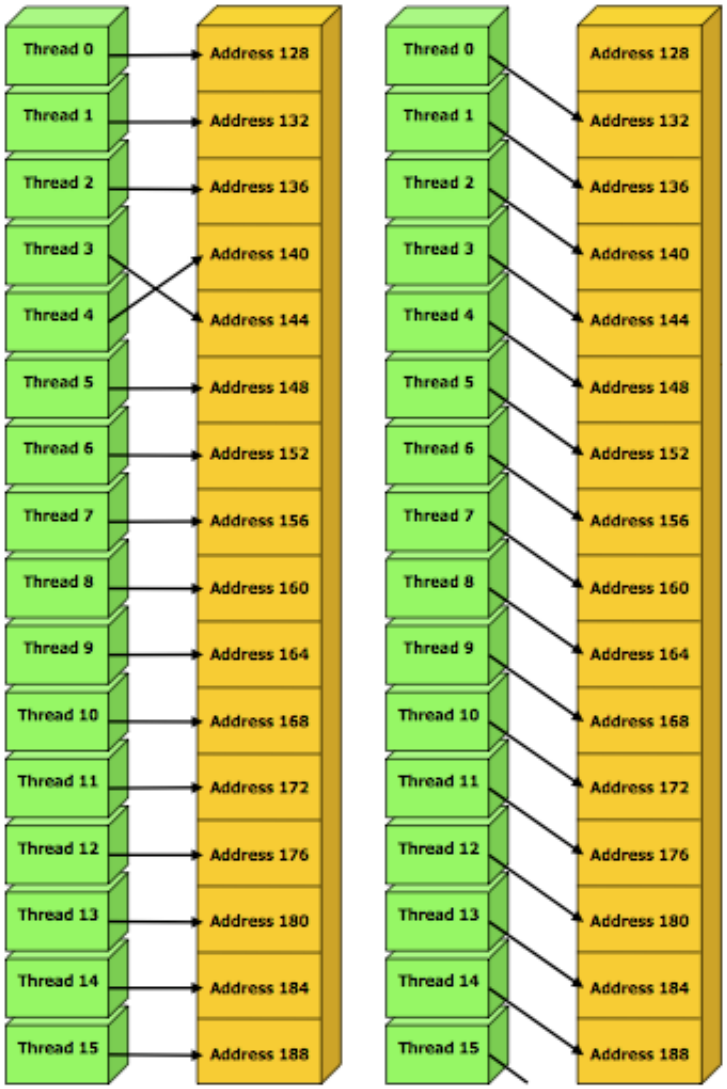
Memory coalescing

- Fast:



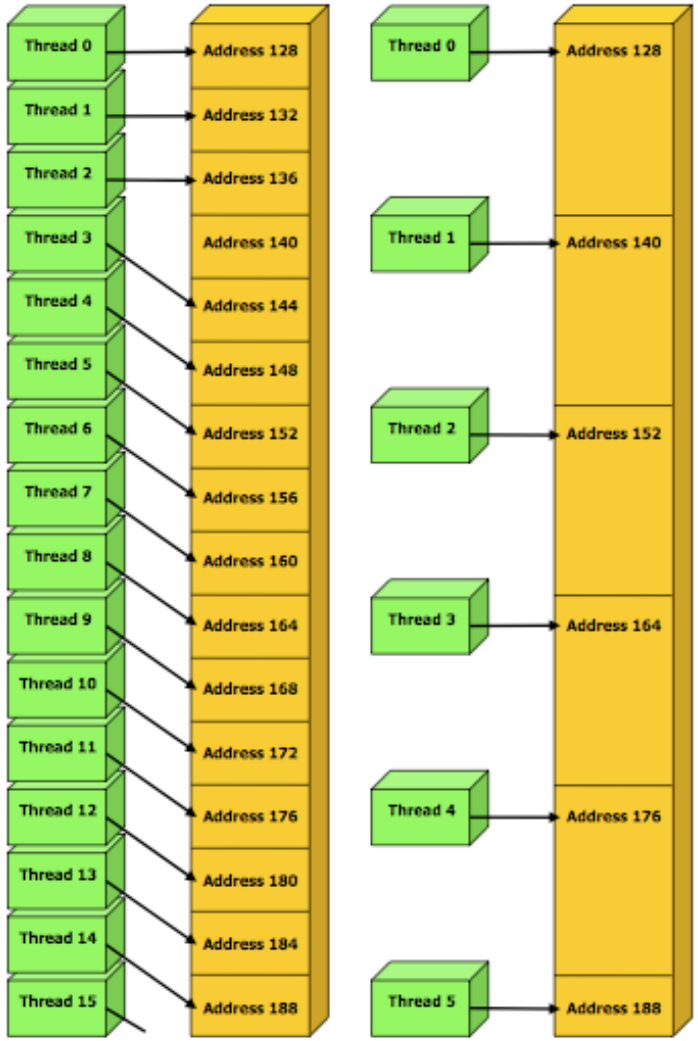
Memory coalescing

- Slow:



Memory coalescing

- Slow:



Coalescing on newer GPUs

- On newer cards, restrictions are slightly relaxed
- Sequentialness no longer mandatory - can have random access within an aligned chunk

Coalescing on newer GPUs

- GPU will try to get around misaligned accesses by reading a larger chunk
  - Ex. accessing 64B from 0x101 will cause a 128B read from 0x100
- Can't always do this - 256B is the maximum read/write size
- Also, has to be "sort of" aligned
  - No way to convert 64B from 0x078 to 1 aligned 128B access

Shared memory architecture

- Many threads need to access memory at the same time
  - Divide memory into banks, interleave them
- Each bank services one address per cycle
  - Can service as many accesses as there are banks
- Multiple simultaneous accesses to the same bank causes “bank conflict”
  - Accesses are serialized!

Shared memory banks

- Each bank services 32-bit words at addresses mod N * 4
  - If N = 16, bank 0 has 0x00, 0x40, 0x80, ...
  - bank 1 has 0x04, 0x44, 0x84, ...
- Unlike in global memory in older hardware, no need for threads to do sequential accesses - bank 1 has no special connection to thread 1

Bank conflicts

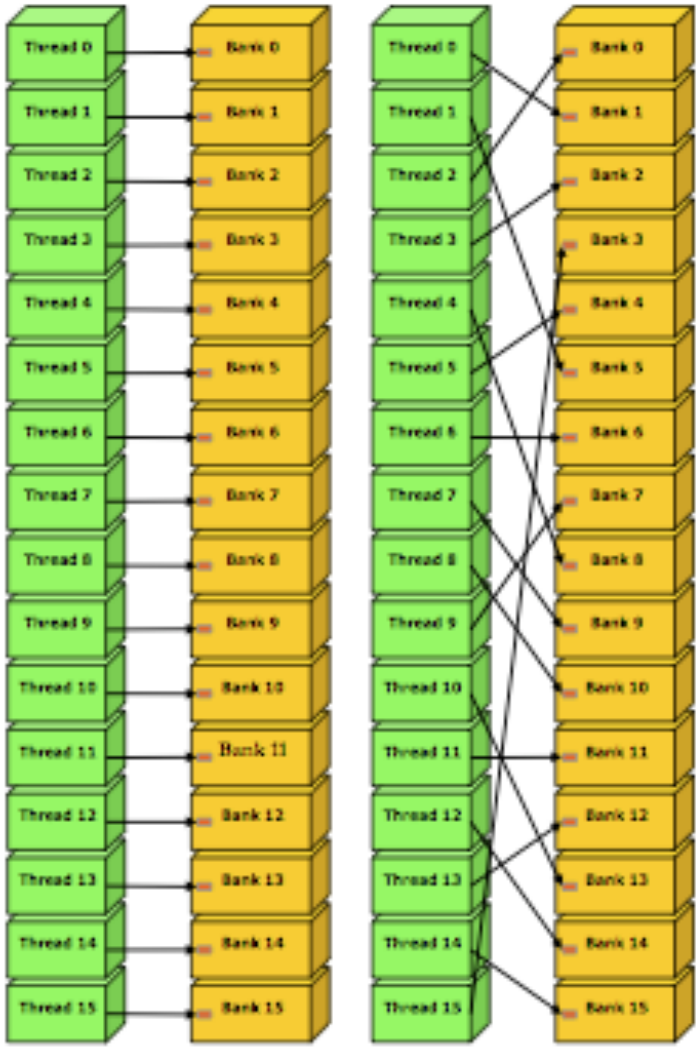
- Want to avoid multiple threads accessing the same bank
  - Don’t close-pack data - keep everything spread out by at least 4 bytes
  - Split data elements that are more than 4 bytes into multiple accesses
  - Watch out for structures with even strides

Shared memory bank conflicts

- Shared memory is as fast as regs since it's stored on-chip, if there are no bank conflicts
- Fast cases:
  - All threads in a warp access different banks
  - All threads in a warp access the same address
    - Broadcast: read the data once, give it to all threads
- Slow cases:
  - Bank conflict: multiple threads access the same bank at different addresses - must serialize accesses
  - Cost = max # of simultaneous accesses to a bank

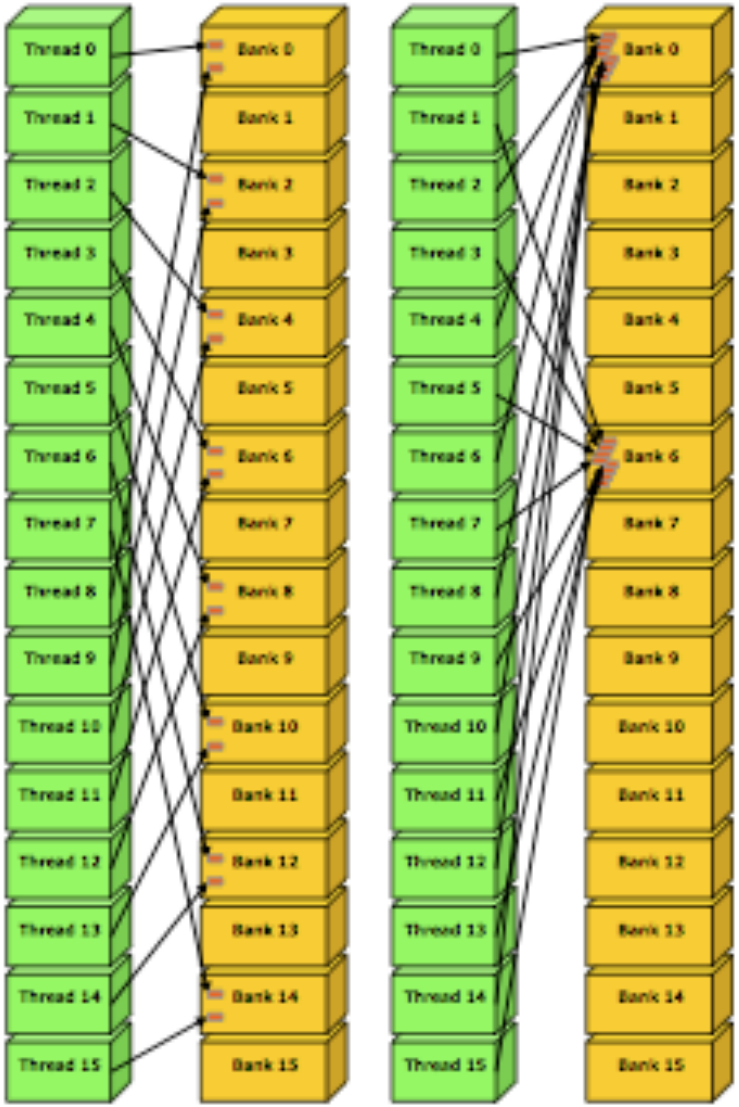
Shared memory banks

- Fast:



Shared memory banks

- Slow:

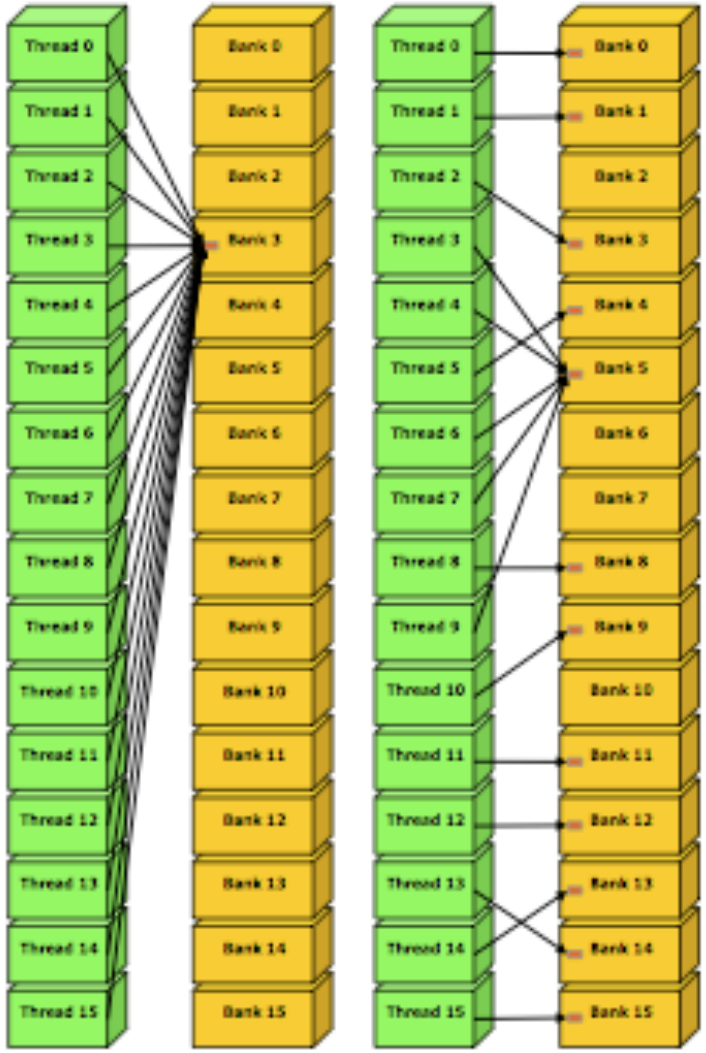


Shared memory banks

- One helpful feature for reading from memory banks: broadcast
  - If multiple threads read the same address at the same time, only one read is performed, and data is broadcast to all reading threads

Shared memory banks

- Fast (reading):



Constant/texture mem usage

- Both of these memories have 2 levels of caching
  - To achieve good usage, locality far more important than alignment
  - Textures use 2D locality rules
- In general writing optimized accesses to constants is what you'd expect from CPU coding

Using texture memory

- Can put textures in global memory
  - Take advantage of texture caching!
- Syntax similar to GLSL textures
- Step 1: make a texture ref
  - Use templates: `texture<Type> texRef;`
  - Type: float, float4, etc.
- Step 2: bind memory to the texref
  - `cudaBindTexture(0, texRef, devPtr, size);`

Using texture memory

- Step 3: get texref into your kernel
  - Pass it as a parameter into your kernel
  - Remember that texRef is a templated type...
- Step 4: read the texture!
  - `tex1Dfetch(texRef, x);`
  - x is an int!

Using floats instead...

- Can convert ints to normalized floats automatically when reading textures
- To do so, change the readmode:
  - `texRef<unsigned char, 1, cudaReadModeNormalizedFloat>`
  - Here, the 1 is for a 1D texture

One important caveat...

- If you use a texture in global memory, a kernel can write to it directly and read it using texture functions
  - This is a terrible idea!
  - Even without caching, it's a synchronization nightmare
- Nothing is done in hardware to keep texture cache coherent with global memory

Want to use real textures?

- Called “CUDA Arrays”
  - Stored in texture memory (not global memory); faster due to specialization
- Check the reference manual...