King Abdullah University of Science and Technology

KAUST

CS 380 - GPU and GPGPU Programming
Lecture 19: CUDA Memories, Pt. 3

Markus Hadwiger, KAUST

Reading Assignment #8 (until Oct 28)

Read (required):

- Programming Massively Parallel Processors book, 4th edition
  Chapter 10: Reduction
- Optimizing Parallel Reduction in CUDA, Mark Harris,

https://developer.download.nvidia.com/assets/cuda/files/reduction.pdf

Read (optional):

- Faster Parallel Reductions on Kepler, Justin Luitjens

https://devblogs.nvidia.com/parallelforall/faster-parallel-reductions-kepler/

Next Lectures

Lecture 20: Tue, Oct 22 (make-up lecture; 14:30 – 15:45)

Lecture 21: Thu, Oct 24

Lecture 22: Mon, Oct 28

Lecture 23: Tue, Oct 29 (make-up lecture; 14:30 – 15:45)

Lecture 24: Thu, Oct 31

CUDA Memory: Shared Memory

Memory and Cache Types

Global memory

- [Device] **L2 cache**
- [SM] **L1 cache** (shared mem carved out; *or* L1 shared with tex cache)
- [SM/TPC] **Texture cache** (separate, or shared with L1 cache)
- [SM] **Read-only data cache** (storage might be same as tex cache)

Shared memory

- [SM] Shareable only between threads in same thread block
  (Hopper/CC 9.x: also thread block clusters)

Constant memory: Constant (uniform) cache

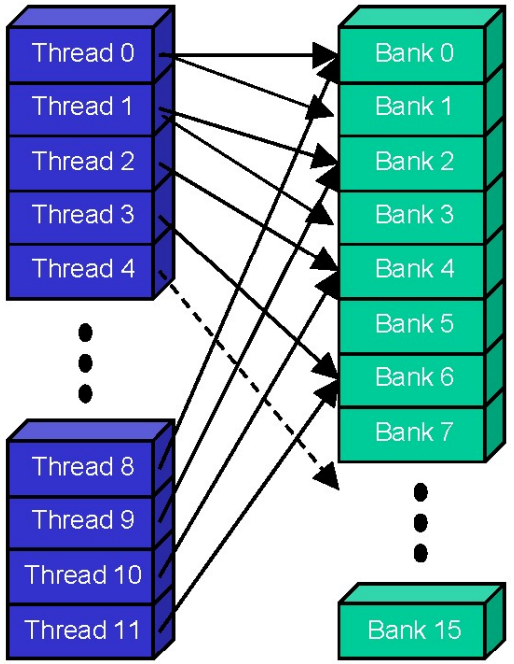
Unified memory programming: Device/host memory sharing

Common Array Bank Conflict Patterns 1D

- **Each thread loads 2 elements into shared mem:**
  - 2-way-interleaved loads result in 2-way bank conflicts:

```
int tid = threadIdx.x;
shared[2*tid] = global[2*tid];
shared[2*tid+1] = global[2*tid+1];
```

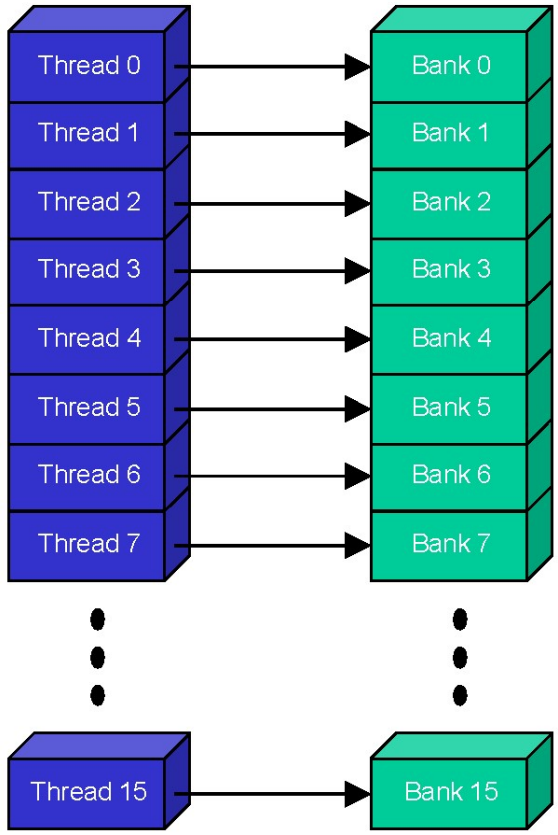
- **This makes sense for traditional CPU threads, locality in cache line usage and reduced sharing traffic.**
  - Not in shared memory usage where there is no cache line effects but banking effects



A Better Array Access Pattern

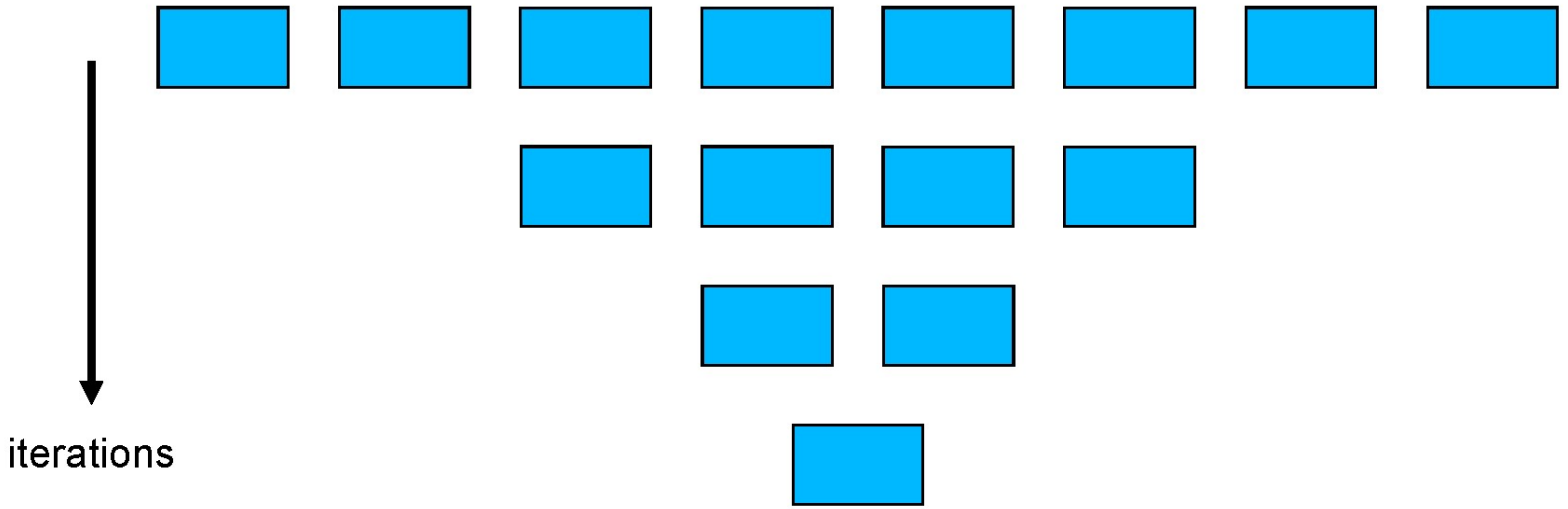
- **Each thread loads one element in every consecutive group of `blockDim` elements.**

```
shared[tid] = global[tid];
shared[tid + blockDim.x] =
  global[tid + blockDim.x];
```



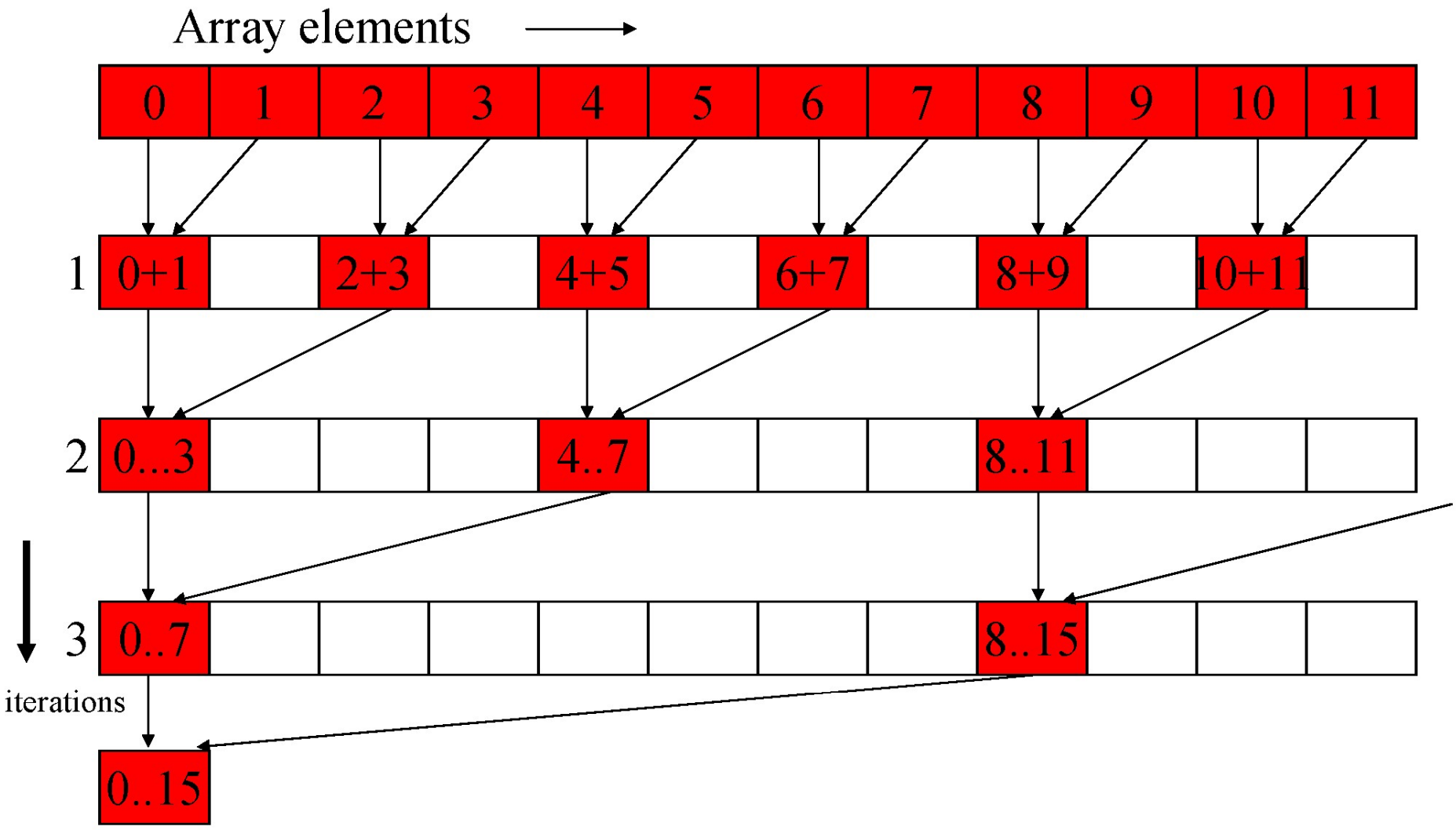
Typical Parallel Programming Pattern

- **log(n) steps**

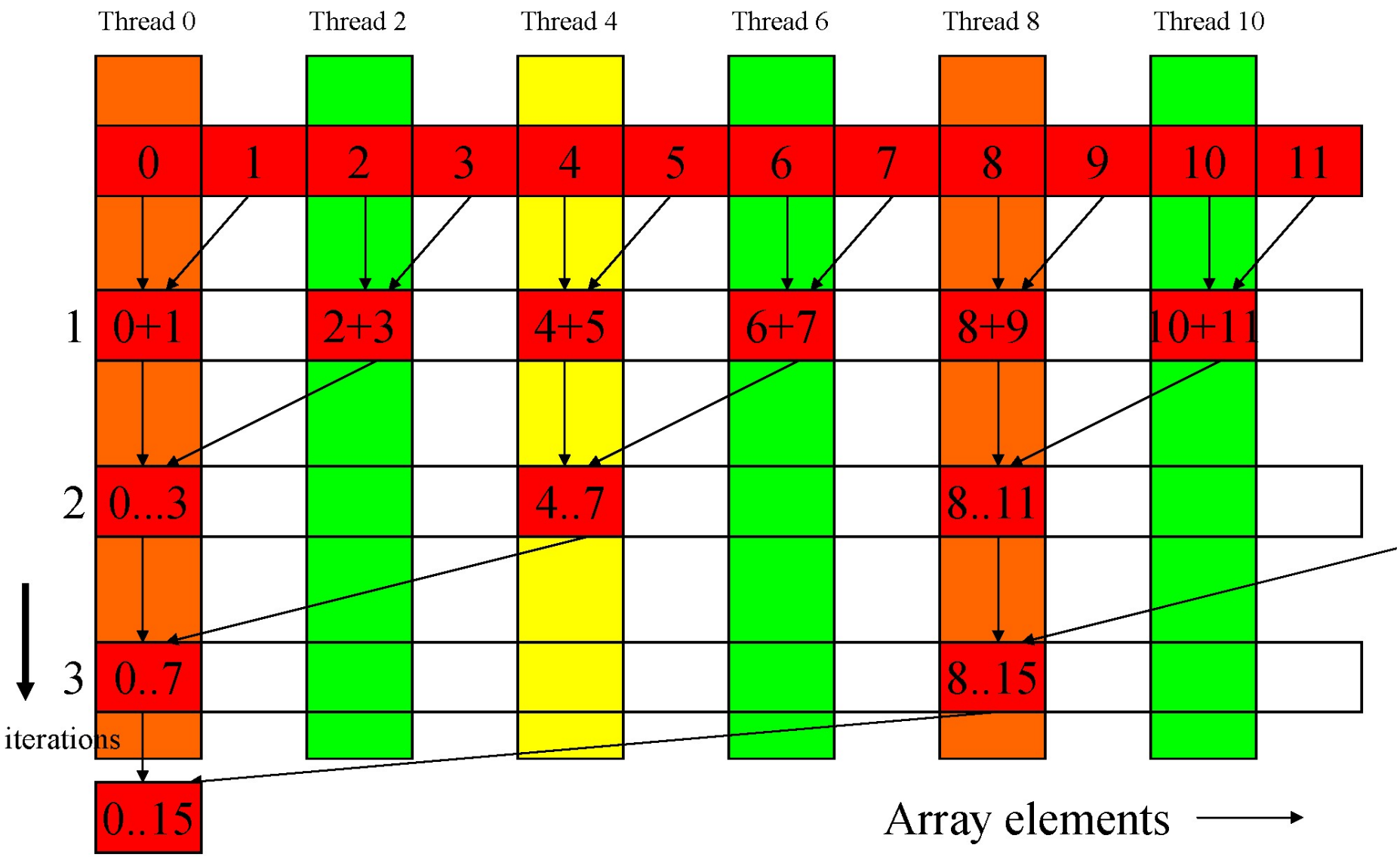


Helpful fact for counting nodes of full binary trees:
If there are N leaf nodes, there will be N-1 non-leaf nodes

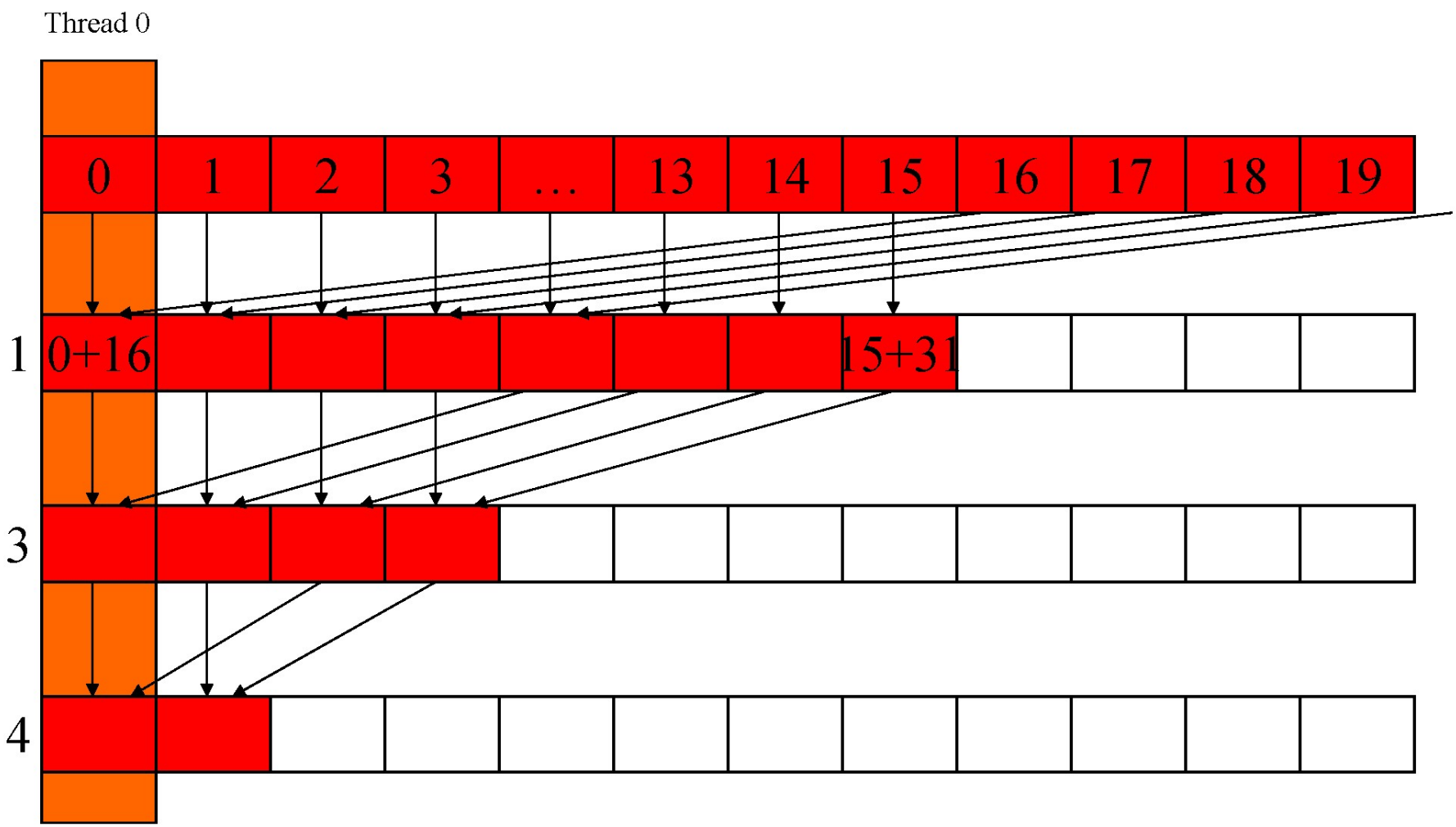
Vector Reduction



Vector Reduction with Branch Divergence

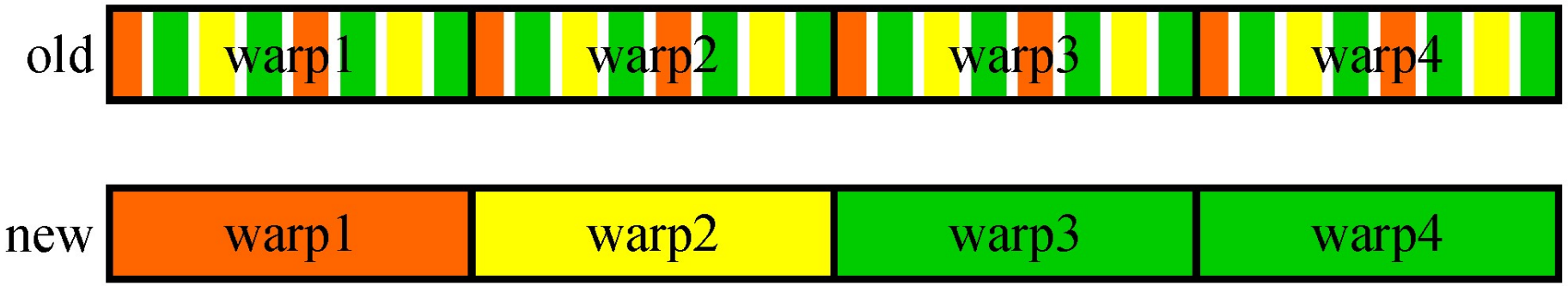


A better implementation



A better implementation

- **Only the last 5 iterations will have divergence**
- **Entire warps will be shut down as iterations progress**
  - For a 512-thread block, 4 iterations to shut down all but one warp in each block
  - Better resource utilization, will likely retire warps and thus blocks faster
- **Recall, no bank conflicts either**



CUDA Memory: Uniforms & Textures

Memory and Cache Types

Global memory

- [Device] **L2 cache**
- [SM] **L1 cache** (shared mem carved out; *or* L1 shared with tex cache)
- [SM/TPC] **Texture cache** (separate, or shared with L1 cache)
- [SM] **Read-only data cache** (storage might be same as tex cache)

Shared memory

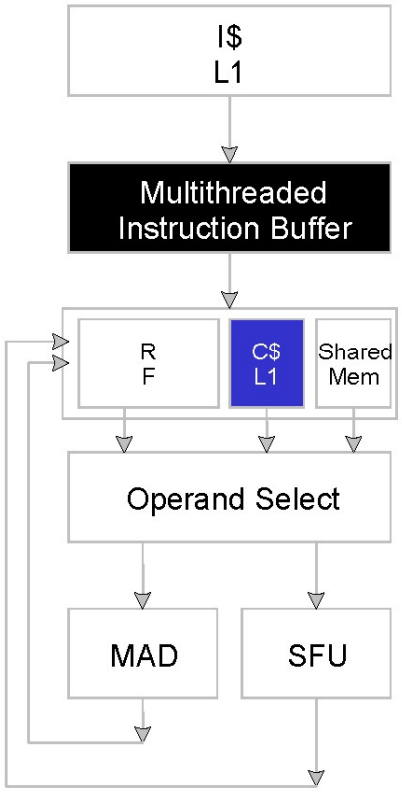
- [SM] Shareable only between threads in same thread block
  (Hopper/CC 9.x: also thread block clusters)

Constant memory: Constant (uniform) cache

Unified memory programming: Device/host memory sharing

Constants

- **Immediate address constants**
- **Indexed address constants**
- **Constants stored in DRAM, and cached on chip**
  - L1 per SM
- **A constant value can be broadcast to all threads in a Warp**
  - Extremely efficient way of accessing a value that is common for all threads in a block!




```
// specify as global variable
__device__ __constant__ float gpuGamma[2];
…
// copy gamma value to constant device memory
cudaMemcpyToSymbol(gpuGamma, &gamma, sizeof(float));
// access as global variable in kernel
res = gpuGamma[0] * threadIdx.x;
```

Memory and Cache Types

Global memory

- [Device] **L2 cache**
- [SM] **L1 cache** (shared mem carved out; *or* L1 shared with tex cache)
- [SM/TPC] **Texture cache** (separate, or shared with L1 cache)
- [SM] **Read-only data cache** (storage might be same as tex cache)

Shared memory

- [SM] Shareable only between threads in same thread block
  (Hopper/CC 9.x: also thread block clusters)

Constant memory: Constant (uniform) cache

Unified memory programming: Device/host memory sharing

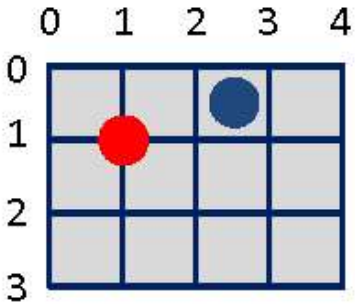
Texture Memory

- ***Cached***, potentially exhibiting higher bandwidth if there is locality in the texture fetches;
- They are not subject to the constraints on memory access patterns that global or constant memory reads must respect to get good performance
- The latency of addressing calculations is hidden better, possibly improving performance for applications that perform random accesses to the data
- No penalty when accessing float4
- Optional
  - 8-bit and 16-bit integer input data may be optionally converted to 32-bit floatingpoint
  - Packed data may be broadcast to separate variables in a single operation;
  - values in the range [0.0, 1.0] or [-1.0, 1.0]
  - texture filtering
  - address modes, e.g. wrapping / texture borders

Additional Texture Functionality

- **All of these are "free"**
  - Dedicated hardware
  - Must use CUDA texture objects
    - See CUDA Programming Guide for more details
    - Texture objects can interoperate graphics (OpenGL, DirectX)
- **Out-of-bounds index handling: clamp or wrap-around**
- **Optional interpolation**
  - Think: using fp indices for arrays
  - Linear, bilinear, trilinear
    - Interpolation weights are 9-bit
- **Optional format conversion**
  - {char, short, int, fp16} -> float

Examples of Texture Object Indexing



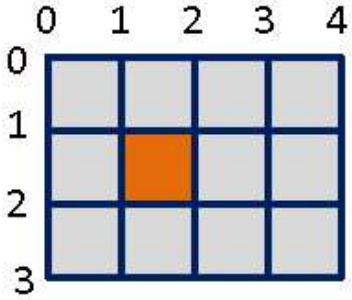
(2.5, 0.5)
(1.0, 1.0)

Integer indices fall between elements

Optional interpolation:

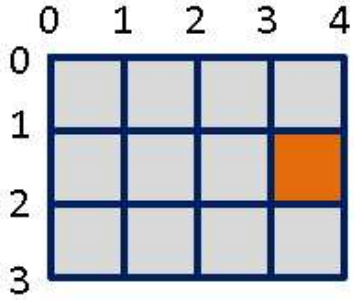
Weights are determined by coordinate distance

Index Wrap:



(5.5, 1.5)

Index Clamp:

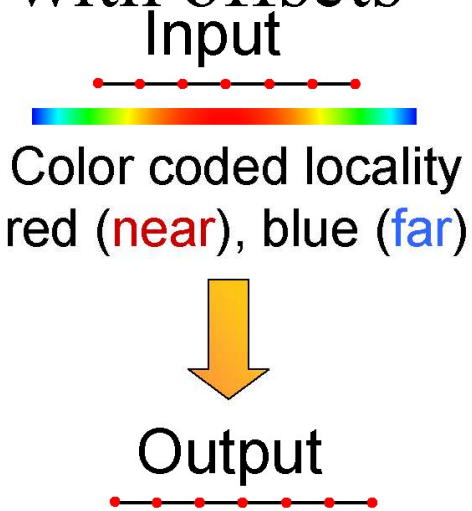


(5.5, 1.5)

Native Memory Layout – Data Locality

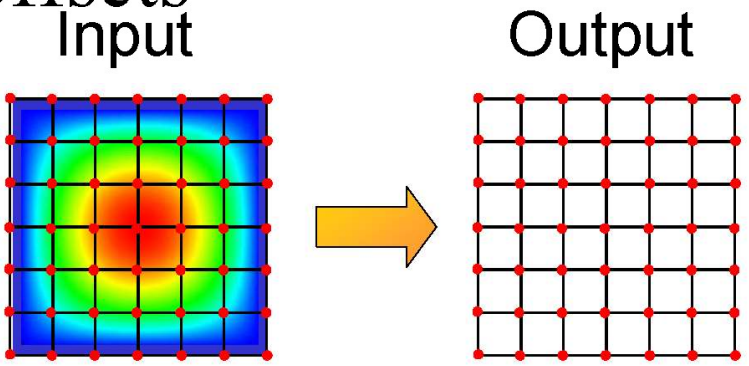
CPU

- 1D input
- 1D output
- Other dimensions with offsets



GPU

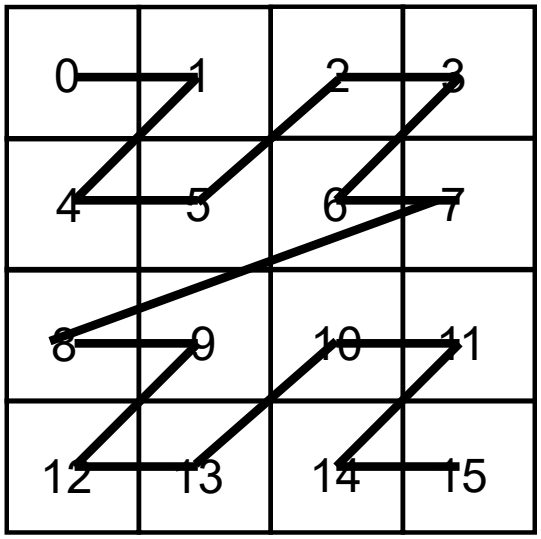
- 2D input
- 2D output
- Other dimensions with offsets



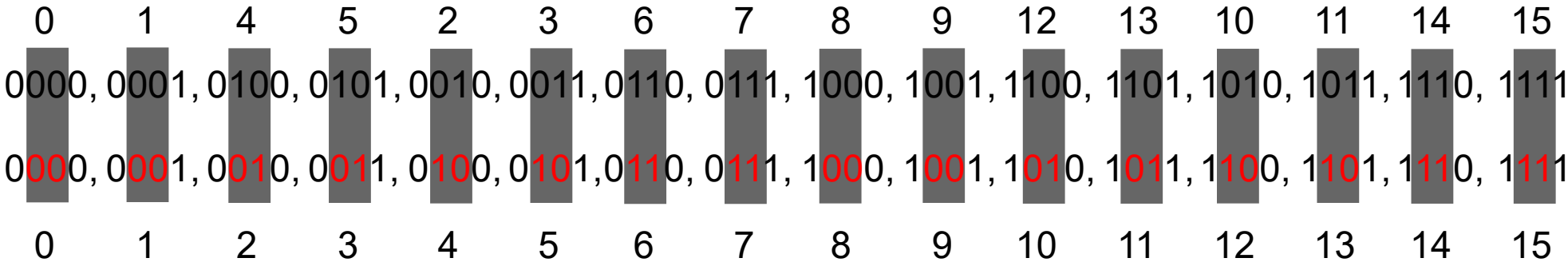
Space-Filling Curves: Morton Order (Z Order)

Map higher-dimensional space to 1D

- Z-order: Equivalent to quadtree (octree in 3D) depth-first traversal order



0000	0001	0010	0011
0100	0101	0110	0111
1000	1001	1010	1011
1100	1101	1110	1111



1D Access

- **Access to linear Cuda memory**

```
float4* pos; cudaMalloc( (void**)&pos, x*sizeof(float4) );
```

- **Texture reference**
  - type
  - access/filtering mode

```
// global texture reference
texture< float4, 1, cudaReadModeElementType> texPos;
```

- **Bind to linear array**

```
cudaBindTexture(0, texPos, pos, x*sizeof(float4)));
cudaUnbindTexture(texPos);
```

- **Within kernel**

```
float4 pa1 = tex1Dfetch( texPos, threadIdx.x);
```

- **Writing to a texture that is currently read by some threads is undefined!!!**

2D Access

- **Optimized for 2D / 3D locality**

```
texture< float4, 2, cudaReadModeElementType> texImg;
```

- **Requires binding to special *Array* memory – special memory layout**

```
cudaChannelFormatDesc floatTex =
cudaCreateChannelDesc<float4>();

float4* src;

cudaArray* img;

cudaMallocArray( &img, &floatTex, w, h);

cudaMemcpyToArray(img, 0, 0, src, w*h*sizeof(float4),
 cudaMemcpyHostToDevice);

cudaBindTextureToArray( texImg, img, floatTex) );

cudaUnbindTexture(texImg);
```

2D Access

- **Within kernel**

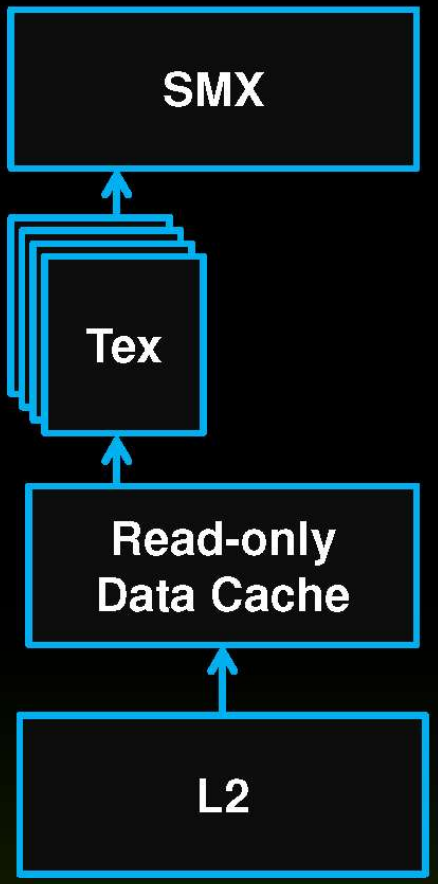
```
float4 r = tex2D( texImg, x +xoff, y+yoff);
```

- **Pros**
  - optimized for 2D locality (optimized memory layout / spacefilling curve)
- **Cons**
  - If the result of some kernel should be used as 2D texture `cudaMemcpyToArray` is required
  - You cannot write to a texture which is currently read from

- CUDA “surfaces” are writeable textures!

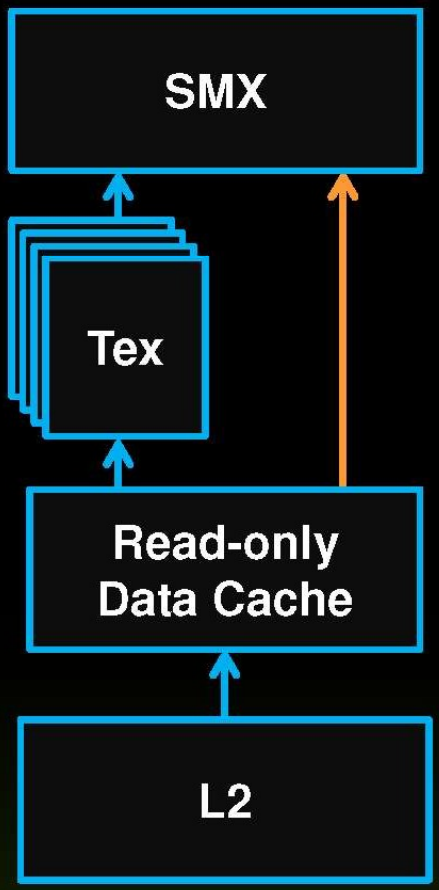
Texture performance

- Texture :
  - Provides hardware accelerated filtered sampling of data (1D, 2D, 3D)
  - Read-only data cache holds fetched samples
  - Backed up by the L2 cache
- SMX vs Fermi SM :
  - 4x filter ops per clock
  - 4x cache capacity



Texture Cache Unlocked

- Added a new path for compute
  - Avoids the texture unit
  - Allows a global address to be fetched and cached
  - Eliminates texture setup
- Why use it?
  - Separate pipeline from shared/L1
  - Highest miss bandwidth
  - Flexible, e.g. unaligned accesses
- Managed automatically by compiler
  - “const __restrict” indicates eligibility



CUDA Memory: Global Memory

- Memory coalescing
- Cached memory access (L2 / L1)

Memory and Cache Types

Global memory

- [Device] **L2 cache**
- [SM] **L1 cache** (shared mem carved out; *or* L1 shared with tex cache)
- [SM/TPC] **Texture cache** (separate, or shared with L1 cache)
- [SM] **Read-only data cache** (storage might be same as tex cache)

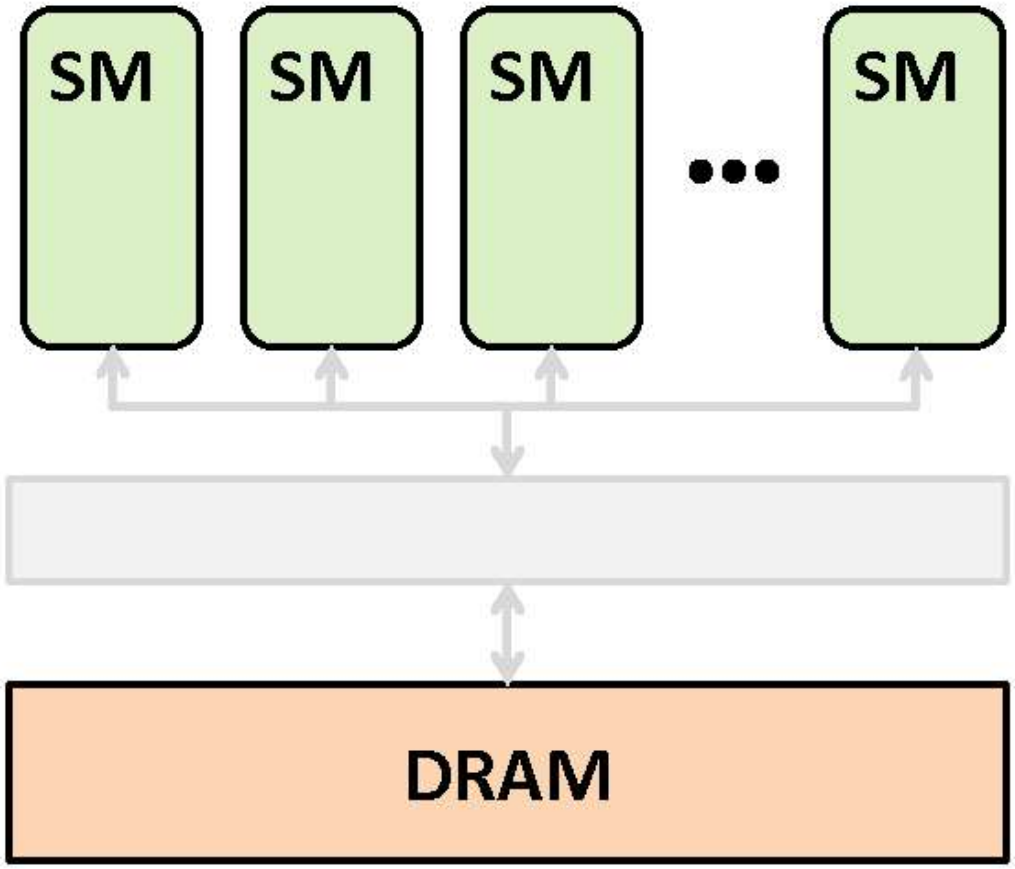
Shared memory

- [SM] Shareable only between threads in same thread block
  (Hopper/CC 9.x: also thread block clusters)

Constant memory: Constant (uniform) cache

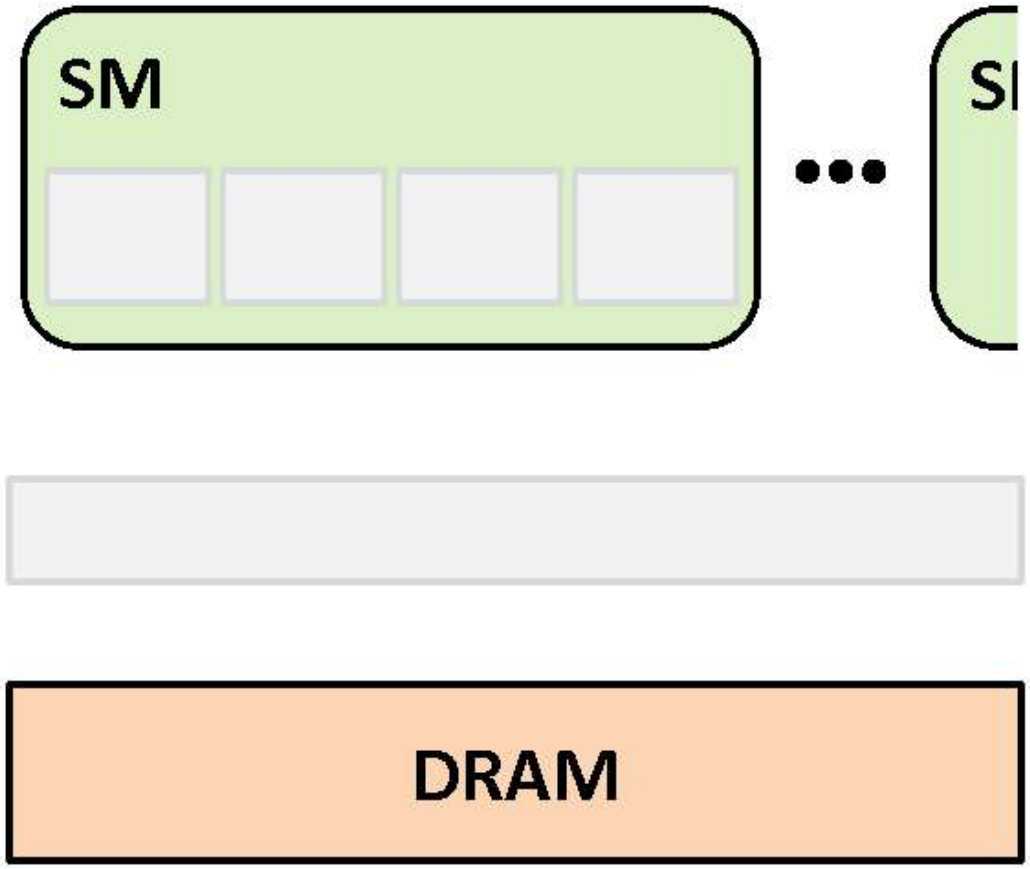
Unified memory programming: Device/host memory sharing

Maximize Byte Use



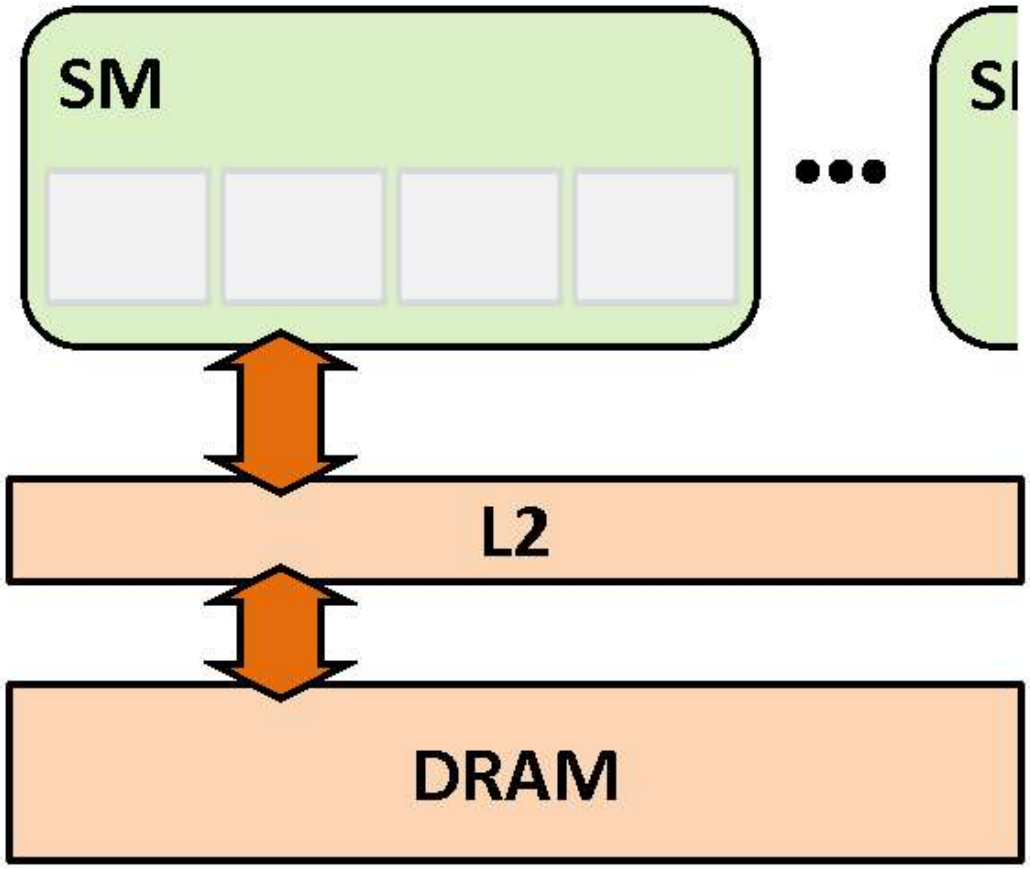
- **Two things to keep in mind:**
  - Memory accesses are per warp
  - Memory is accessed in discrete chunks
    - lines/segments
    - want to make sure that bytes that travel from DRAM to SMs get used
      - For that we should understand how memory system works
- **Note: not that different from CPUs**
  - x86 needs SSE/AVX memory instructions to maximize performance

GPU Memory System



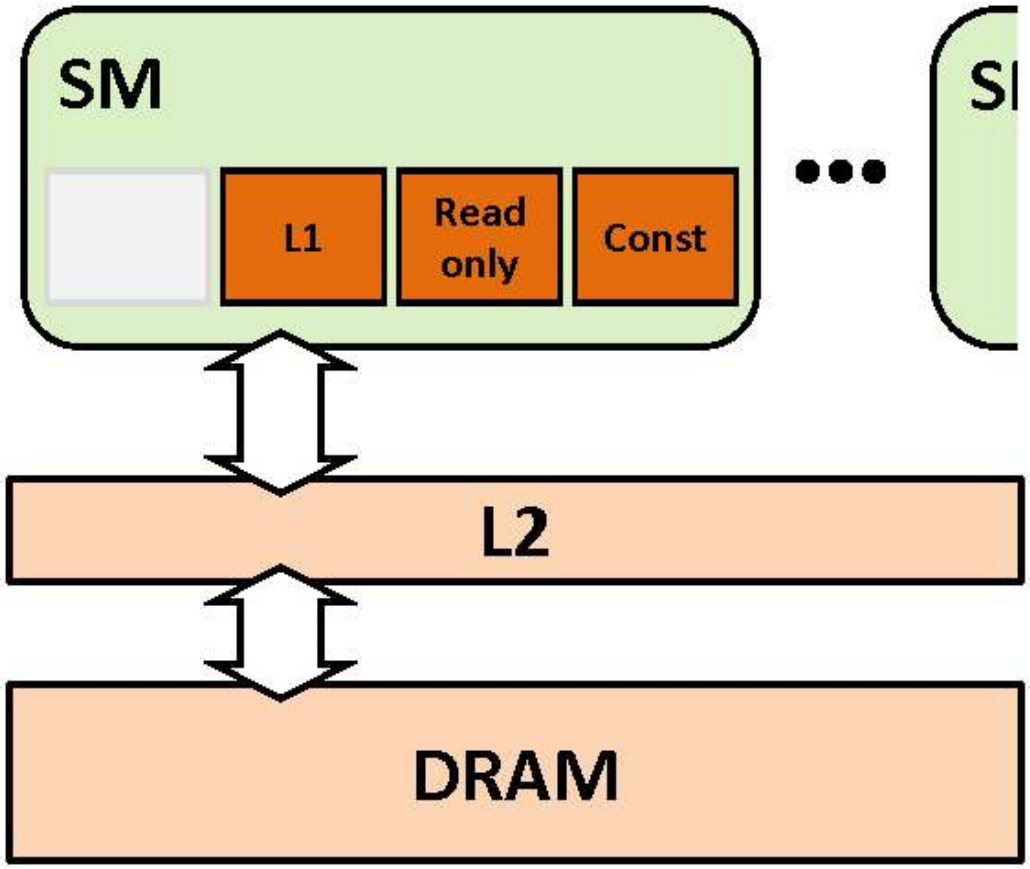
- **All data lives in DRAM**
  - Global memory
  - Local memory
  - Textures
  - Constants

GPU Memory System



- All DRAM accesses go through L2
- Including copies:
  - P2P
  - CPU-GPU

GPU Memory System



- Once in an SM, data goes into one of 3 caches/buffers
- Programmer's choice
  - ~~L1 is the "default"~~
  - Read-only, Const require explicit code

Access Path

- **L1 path**
  - Global memory
    - Memory allocated with cudaMalloc()
    - Mapped CPU memory, peer GPU memory
    - Globally-scoped arrays qualified with __global__
  - Local memory
    - allocation/access managed by compiler so we'll ignore
- **Read-only/TEX path**
  - Data in texture objects, CUDA arrays
  - CC 3.5 and higher:
    - Global memory accessed via intrinsics (or specially qualified kernel arguments)
- **Constant path**
  - Globally-scoped arrays qualified with __constant__

Access Via L1

- **Natively supported word sizes per thread:**
  - 1B, 2B, 4B, 8B, 16B
    - Addresses must be aligned on word-size boundary
  - Accessing types of other sizes will require multiple instructions
- **Accesses are processed per warp**
  - Threads in a warp provide 32 addresses
    - Fewer if some threads are inactive
  - HW converts addresses into memory transactions
    - Address pattern may require multiple transactions for an instruction
    - If ***N*** transactions are needed, there will be (***N-1***) replays of the instruction

Global Memory Access

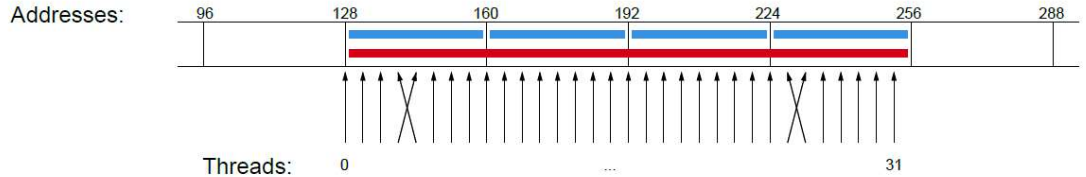
all recent
compute capabilities
(- 9.x)

Beware:

Uncached here means
not cached in L1

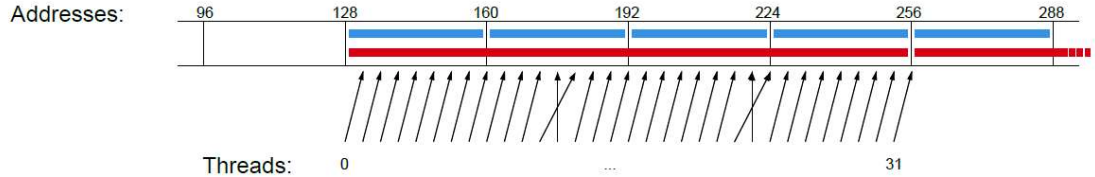
the L2 cache is
***always* used!**

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

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

Compute Capab. 3.x (Kepler, Part 1)

K.3.2. Global Memory

Global memory accesses for devices of compute capability 3.x are cached in L2 and for devices of compute capability 3.5 or 3.7, may also be cached in the read-only data cache described in the previous section; they are normally not cached in L1. Some devices of compute capability 3.5 and devices of compute capability 3.7 allow opt-in to caching of global memory accesses in L1 via the `-Xptxas -dlcm=ca` option to `nvcc`.

A cache line is 128 bytes and maps to a 128 byte aligned segment in device memory. Memory accesses that are cached in both L1 and L2 are serviced with 128-byte memory transactions, whereas memory accesses that are cached in L2 only are serviced with 32-byte memory transactions. Caching in L2 only can therefore reduce over-fetch, for example, in the case of scattered memory accesses.

If the size of the words accessed by each thread is more than 4 bytes, a memory request by a warp is first split into separate 128-byte memory requests that are issued independently:

- Two memory requests, one for each half-warp, if the size is 8 bytes,
- Four memory requests, one for each quarter-warp, if the size is 16 bytes.

Compute Capab. 3.x (Kepler, Part 2)

Each memory request is then broken down into cache line requests that are issued independently. A cache line request is serviced at the throughput of L1 or L2 cache in case of a cache hit, or at the throughput of device memory, otherwise.

Note that threads can access any words in any order, including the same words.

If a non-atomic instruction executed by a warp writes to the same location in global memory for more than one of the threads of the warp, only one thread performs a write and which thread does it is undefined.

Data that is read-only for the entire lifetime of the kernel can also be cached in the read-only data cache described in the previous section by reading it using the `__ldg()` function (see Read-Only Data Cache Load Function). When the compiler detects that the read-only condition is satisfied for some data, it will use `__ldg()` to read it. The compiler might not always be able to detect that the read-only condition is satisfied for some data. Marking pointers used for loading such data with both the `const` and `__restrict__` qualifiers increases the likelihood that the compiler will detect the read-only condition.

Figure 21 shows some examples of global memory accesses and corresponding memory transactions.

Compute Capab. 5.x (Maxwell, Part 1)

K.4.2. Global Memory

Global memory accesses are always cached in L2 and caching in L2 behaves in the same way as for devices of compute capability 3.x (see Global Memory).

Data that is read-only for the entire lifetime of the kernel can also be cached in the unified L1/texture cache described in the previous section by reading it using the `__ldg()` function (see Read-Only Data Cache Load Function). When the compiler detects that the read-only condition is satisfied for some data, it will use `__ldg()` to read it. The compiler might not always be able to detect that the read-only condition is satisfied for some data. Marking pointers used for loading such data with both the `const` and `__restrict__` qualifiers increases the likelihood that the compiler will detect the read-only condition.

Data that is not read-only for the entire lifetime of the kernel cannot be cached in the unified L1/texture cache for devices of compute capability 5.0. For devices of compute capability 5.2, it is, by default, not cached in the unified L1/texture cache, but caching may be enabled using the following mechanisms:

Compute Capab. 5.x (Maxwell, Part 2)

Data that is not read-only for the entire lifetime of the kernel cannot be cached in the unified L1/texture cache for devices of compute capability 5.0. For devices of compute capability 5.2, it is, by default, not cached in the unified L1/texture cache, but caching may be enabled using the following mechanisms:

- Perform the read using inline assembly with the appropriate modifier as described in the PTX reference manual;
- Compile with the `-Xptxas -dlcm=ca` compilation flag, in which case all reads are cached, except reads that are performed using inline assembly with a modifier that disables caching;
- Compile with the `-Xptxas -fscm=ca` compilation flag, in which case all reads are cached, including reads that are performed using inline assembly regardless of the modifier used.

When caching is enabled using one of the three mechanisms listed above, devices of compute capability 5.2 will cache global memory reads in the unified L1/texture cache for all kernel launches except for the kernel launches for which thread blocks consume too much of the SM's register file. These exceptions are reported by the profiler.

PTX State Spaces (1)

Memory type/access etc. organized using notion of *state spaces*

Table 6 State Spaces

Name	Description
`.reg`	Registers, fast.
`.sreg`	Special registers. Read-only; pre-defined; platform-specific.
`.const`	Shared, read-only memory.
`.global`	Global memory, shared by all threads.
`.local`	Local memory, private to each thread.
`.param`	Kernel parameters, defined per-grid; or Function or local parameters, defined per-thread.
`.shared`	Addressable memory shared between threads in 1 CTA.
`.tex`	Global texture memory (deprecated).

PTX State Spaces (2)

Table 7 Properties of State Spaces

Name	Addressable	Initializable	Access	Sharing
`.reg`	No	No	R/W	per-thread
`.sreg`	No	No	RO	per-CTA
`.const`	Yes	Yes[1]	RO	per-grid
`.global`	Yes	Yes[1]	R/W	Context
`.local`	Yes	No	R/W	per-thread
`.param` (as input to kernel)	Yes[2]	No	RO	per-grid
`.param` (used in functions)	Restricted[3]	No	R/W	per-thread
`.shared`	Yes	No	R/W	per-CTA
`.tex`	No[4]	Yes, via driver	RO	Context

Notes:

[1] Variables in `.const` and `.global` state spaces are initialized to zero by default.

[2] Accessible only via the `ld.param` instruction. Address may be taken via `mov` instruction.

[3] Accessible via `ld.param` and `st.param` instructions. Device function input and return parameters may have their address taken via `mov`; the parameter is then located on the stack frame and its address is in the `.local` state space.

[4] Accessible only via the `tex` instruction.

PTX Cache Operators

Table 27 Cache Operators for Memory Load Instructions

Operator	Meaning
`.ca`	Cache at all levels, likely to be accessed again. The default load instruction cache operation is ld.ca, which allocates cache lines in all levels (L1 and L2) with normal eviction policy. Global data is coherent at the L2 level, but multiple L1 caches are not coherent for global data. If one thread stores to global memory via one L1 cache, and a second thread loads that address via a second L1 cache with `ld.ca`, the second thread may get stale L1 cache data, rather than the data stored by the first thread. The driver must invalidate global L1 cache lines between dependent grids of parallel threads. Stores by the first grid program are then correctly fetched by the second grid program issuing default `ld.ca` loads cached in L1.
`.cg`	Cache at global level (cache in L2 and below, not L1). Use `ld.cg` to cache loads only globally, bypassing the L1 cache, and cache only in the L2 cache.
`.cs`	Cache streaming, likely to be accessed once. The `ld.cs` load cached streaming operation allocates global lines with evict-first policy in L1 and L2 to limit cache pollution by temporary streaming data that may be accessed once or twice. When `ld.cs` is applied to a Local window address, it performs the `ld.lu` operation.
`.lu`	Last use. The compiler/programmer may use `ld.lu` when restoring spilled registers and popping function stack frames to avoid needless write-backs of lines that will not be used again. The `ld.lu` instruction performs a load cached streaming operation (`ld.cs`) on global addresses.
`.cv`	Don't cache and fetch again (consider cached system memory lines stale, fetch again). The ld.cv load operation applied to a global System Memory address invalidates (discards) a matching L2 line and re-fetches the line on each new load.

SASS LD/ST Instructions

Architecture-dep.

Kepler:

Compute Load/Store Instructions	
LDC	Load from Constant
LD	Load from Memory
LDG	Non-coherent Global Memory Load
LDL	Load from Local Memory
LDS	Load from Shared Memory
LDSLK	Load from Shared Memory and Lock
ST	Store to Memory
STL	Store to Local Memory
STS	Store to Shared Memory
STSCUL	Store to Shared Memory Conditionally and Unlock
ATOM	Atomic Memory Operation
RED	Atomic Memory Reduction Operation
CCTL	Cache Control
CCTLL	Cache Control (Local)
MEMBAR	Memory Barrier

(see also LDG.CI etc.)

Compute Capab. 6.x (Pascal)

K.5.2. Global Memory

Global memory behaves the same way as in devices of compute capability 5.x (See Global Memory).

Compute Capab. 7.x (Volta/Turing)

K.6.3. Global Memory

Global memory behaves the same way as in devices of compute capability 5.x (See Global Memory).

Compute Capab. 8.x (Ampere/Ada)

K.7.2. Global Memory

Global memory behaves the same way as for devices of compute capability 5.x (See Global Memory).

Compute Capab. 9.x (Hopper)

K.8.2. Global Memory

Global memory behaves the same way as for devices of compute capability 5.x (See Global Memory).

Thank you.