King Abdullah University of Science and Technology

KAUST

CS 380 - GPU and GPGPU Programming
Lecture 20: CUDA Memories, Pt. 4; GPU Reduction

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: 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)

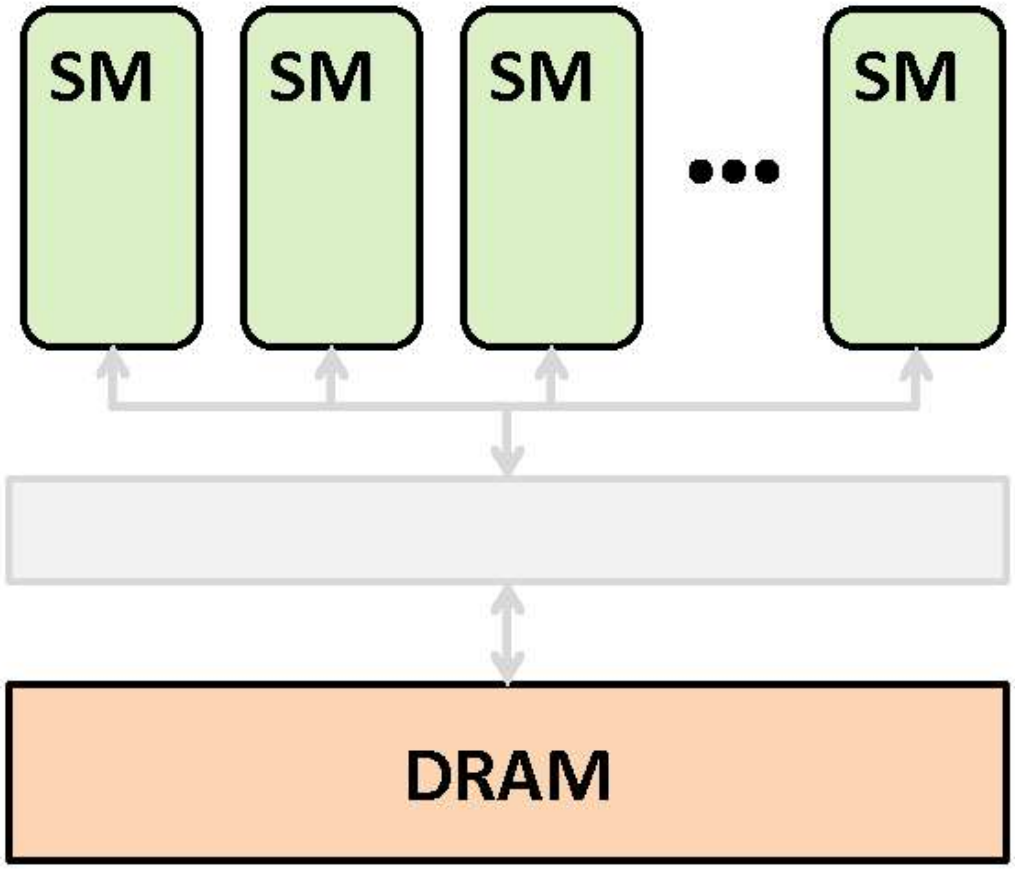
Shared memory

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

Constant memory: Constant (uniform) cache

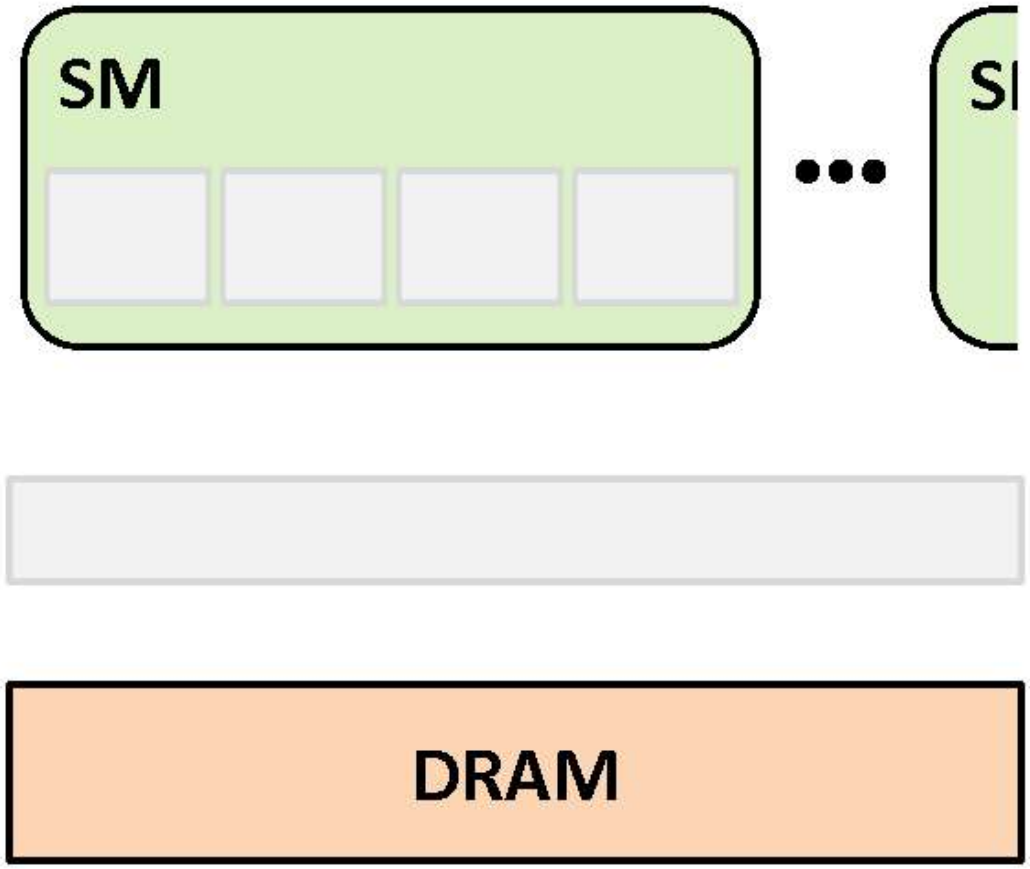
Unified memory programming: Device/host memory sharing

Maximize Byte Use



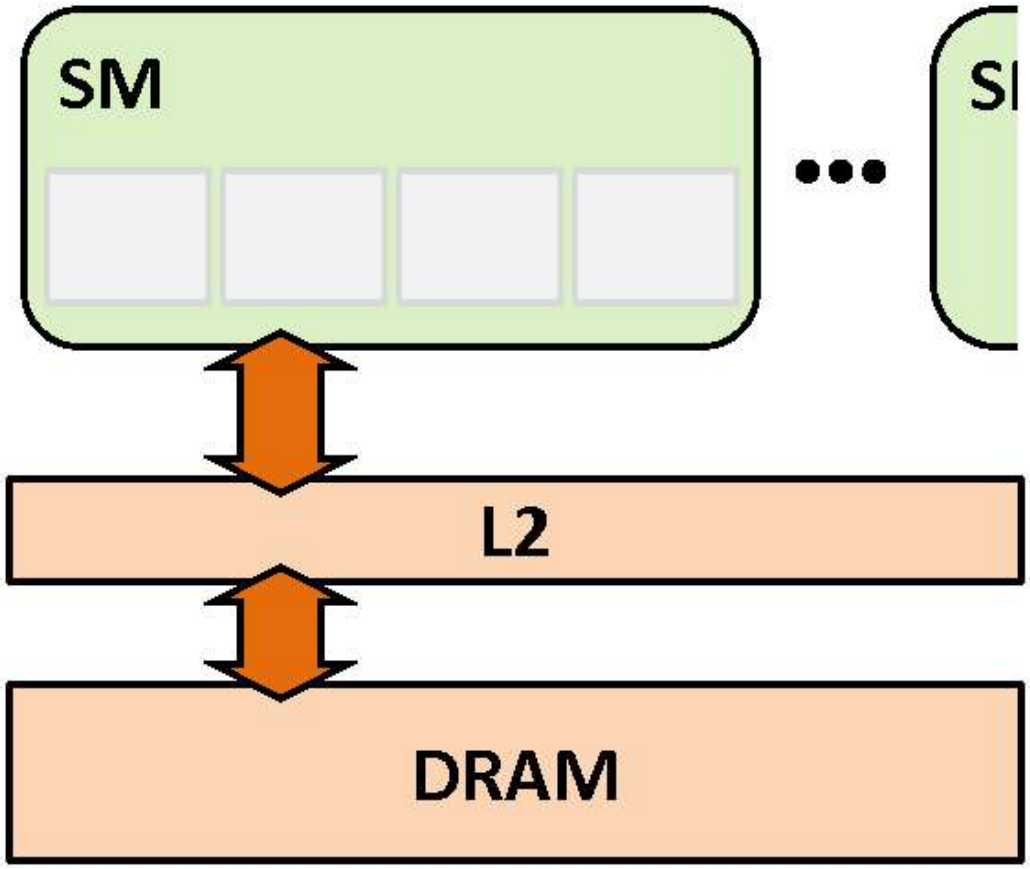
- **Two things to keep in mind:**
  - Memory accesses are per warp
  - Memory is accessed in discrete chunks
    - lines/segments
    - want to make sure that bytes that travel from DRAM to SMs get used
      - For that we should understand how memory system works
- **Note: not that different from CPUs**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

GPU Memory System



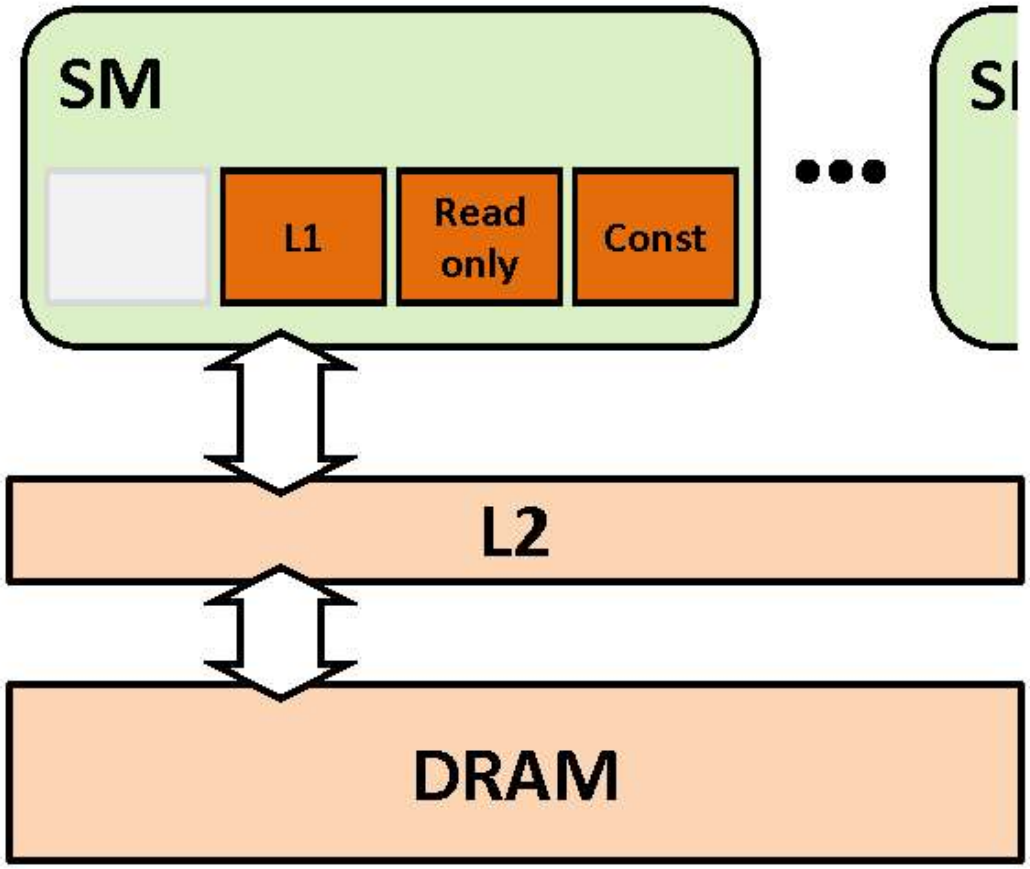
- **All data lives in DRAM**
  - Global memory
  - Local memory
  - Textures
  - Constants

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

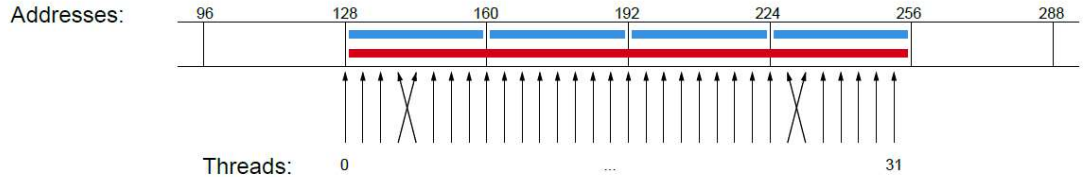
all recent
compute capabilities
(- 9.x)

Beware:

Uncached here means
not cached in L1

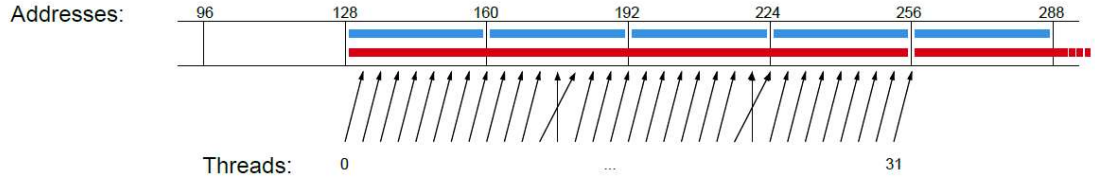
the L2 cache is
***always* used!**

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).

Vectorized Memory Access

See **https://devblogs.nvidia.com/cuda-pro-tip-increase-performance-with-vectorized-memory-access/**

```
__global__ void device_copy_vector2_kernel(int* d_in, int* d_out, int N) {
  int idx = blockIdx.x * blockDim.x + threadIdx.x;
  for (int i = idx; i < N/2; i += blockDim.x * gridDim.x) {
    reinterpret_cast<int2*>(d_out)[i] = reinterpret_cast<int2*>(d_in)[i];
  }

  // in only one thread, process final element (if there is one)
  if (idx==N/2 && N%2==1)
    d_out[N-1] = d_in[N-1];
}

void device_copy_vector2(int* d_in, int* d_out, int n) {
  threads = 128;
  blocks = min((N/2 + threads-1) / threads, MAX_BLOCKS);

  device_copy_vector2_kernel<<<blocks, threads>>>(d_in, d_out, N);
}
```

```
/*0088*/                   IMAD R10.CC, R3, R5, c[0x0][0x140]
/*0090*/                   IMAD.HI.X R11, R3, R5, c[0x0][0x144]
/*0098*/                   IMAD R8.CC, R3, R5, c[0x0][0x148]
/*00a0*/                   LD.E.64 R6, [R10]
/*00a8*/                   IMAD.HI.X R9, R3, R5, c[0x0][0x14c]
/*00c8*/                   ST.E.64 [R8], R6
```

SASS

LD.E.64, LD.E.128,
ST.E.64, ST.E.128

Vectorized Memory Access

See **https://devblogs.nvidia.com/cuda-pro-tip-increase-performance-with-vectorized-memory-access/**

```
__global__ void device_copy_vector4_kernel(int* d_in, int* d_out, int N) {
  int idx = blockIdx.x * blockDim.x + threadIdx.x;
  for(int i = idx; i < N/4; i += blockDim.x * gridDim.x) {
    reinterpret_cast<int4*>(d_out)[i] = reinterpret_cast<int4*>(d_in)[i];
  }

  // in only one thread, process final elements (if there are any)
  int remainder = N%4;
  if (idx==N/4 && remainder!=0) {
    while(remainder) {
      int idx = N - remainder--;
      d_out[idx] = d_in[idx];
    }
  }
}

void device_copy_vector4(int* d_in, int* d_out, int N) {
  int threads = 128;
  int blocks = min((N/4 + threads-1) / threads, MAX_BLOCKS);

  device_copy_vector4_kernel<<<blocks, threads>>>(d_in, d_out, N);
}
```

```
/*0090*/                   IMAD R10.CC, R3, R13, c[0x0][0x140]
/*0098*/                   IMAD.HI.X R11, R3, R13, c[0x0][0x144]
/*00a0*/                   IMAD R8.CC, R3, R13, c[0x0][0x148]
/*00a8*/                   LD.E.128 R4, [R10]
/*00b0*/                   IMAD.HI.X R9, R3, R13, c[0x0][0x14c]
/*00d0*/                   ST.E.128 [R8], R4
```

SASS

LD.E.64, LD.E.128,
ST.E.64, ST.E.128

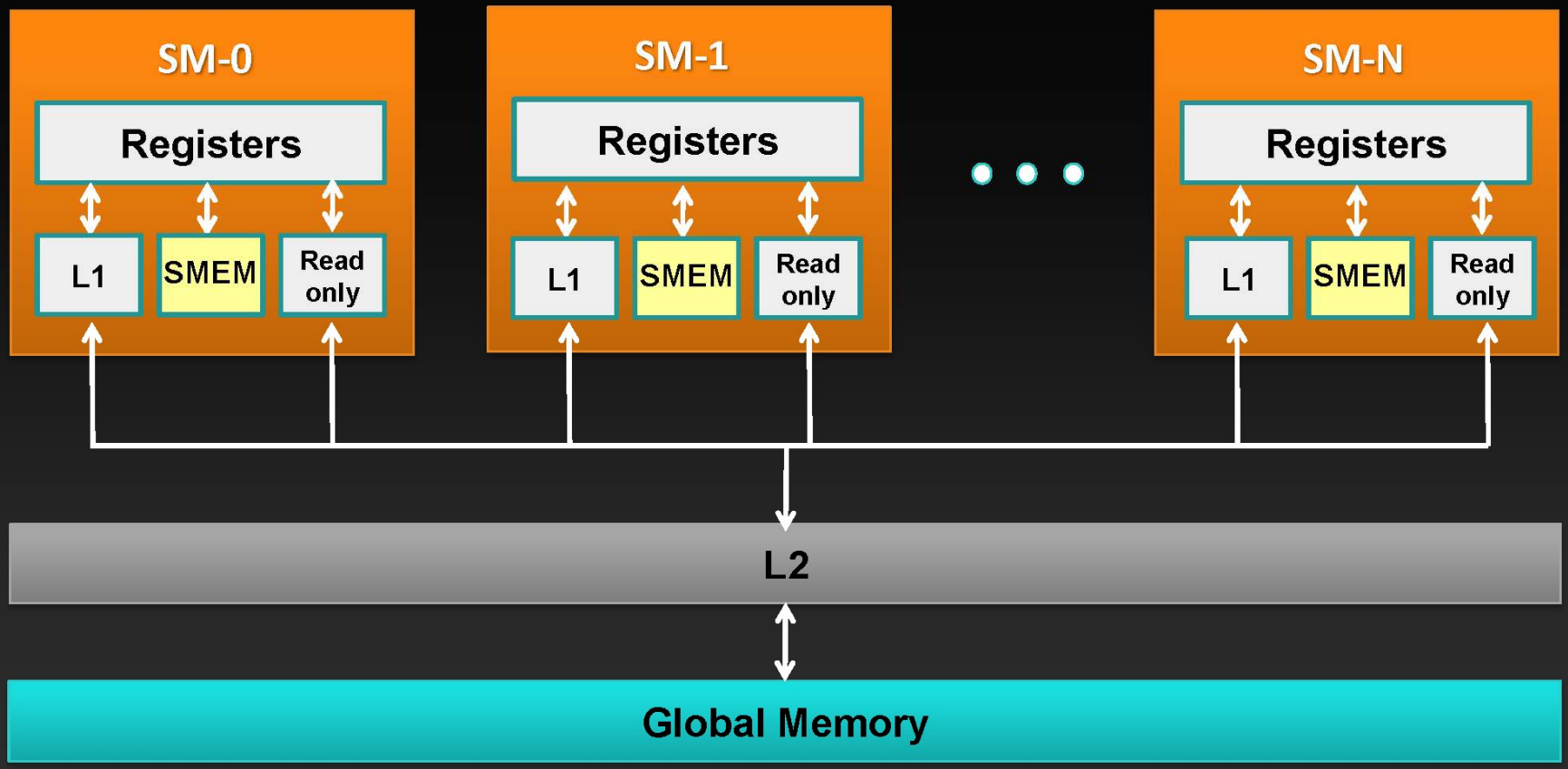
GMEM Writes

- **Not cached in the SM**
  - Invalidate the line in L1, go to L2
- **Access is at 32 B segment granularity**
- **Transaction to memory: 1, 2, or 4 segments**
  - Only the required segments will be sent
- **If multiple threads in a warp write to the same address**
  - One of the threads will "win"
  - Which one is not defined

OPTIMIZE

Kernel Optimizations: *Global Memory Throughput*

Kepler Memory Hierarchy



Load Operation

- Memory operations are issued per warp (32 threads)
  - Just like all other instructions
- Operation:
  - Threads in a warp provide memory addresses
  - Determine which lines/segments are needed
  - Request the needed lines/segments

Memory Throughput Analysis

- Two perspectives on the throughput:
  - Application's point of view:
    - count only bytes requested by application
  - HW point of view:
    - count all bytes moved by hardware
- The two views can be different:
  - Memory is accessed at 32 byte granularity
    - Scattered/offset pattern: application doesn't use all the hw transaction bytes
  - Broadcast: the same small transaction serves many threads in a warp
- Two aspects to inspect for performance impact:
  - Address pattern
  - Number of concurrent accesses in flight

Global Memory Operation

- Memory operations are executed per warp
  - 32 threads in a warp provide memory addresses
  - Hardware determines into which lines those addresses fall
    - Memory transaction granularity is 32 bytes
    - There are benefits to a warp accessing a contiguous aligned region of 128 or 256 bytes
- Access word size
  - Natively supported sizes (per thread): 1, 2, 4, 8, 16 bytes
    - Assumes that each thread's address is aligned on the word size boundary
  - If you are accessing a data type that's of non-native size, compiler will generate several load or store instructions with native sizes

Access Patterns vs. Memory Throughput

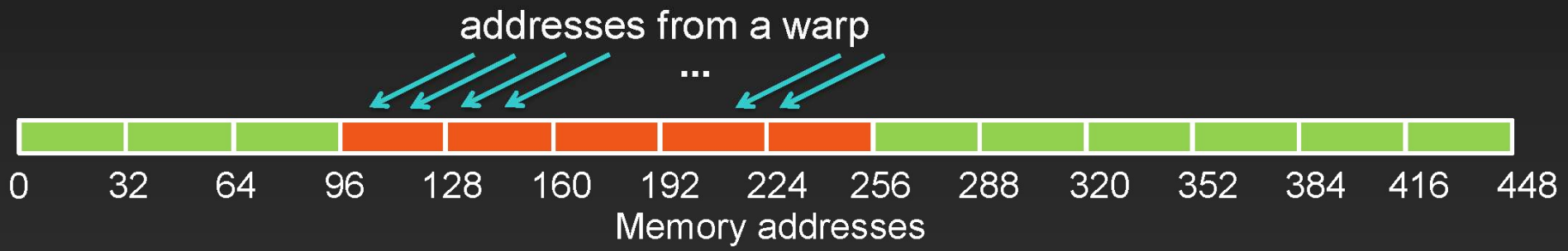
- Scenario:
  - Warp requests 32 aligned, consecutive 4-byte words
- Addresses fall within 4 segments
  - Warp needs 128 bytes
  - 128 bytes move across the bus
  - Bus utilization: 100%

Access Patterns vs. Memory Throughput

- Scenario:
  - Warp requests 32 aligned, permuted 4-byte words
- Addresses fall within 4 segments
  - Warp needs 128 bytes
  - 128 bytes move across the bus
  - Bus utilization: 100%

Access Patterns vs. Memory Throughput

- Scenario:
  - Warp requests 32 misaligned, consecutive 4-byte words
- Addresses fall within at most 5 segments
  - Warp needs 128 bytes
  - At most 160 bytes move across the bus
  - Bus utilization: at least 80%
    - Some misaligned patterns will fall within 4 segments, so 100% utilization

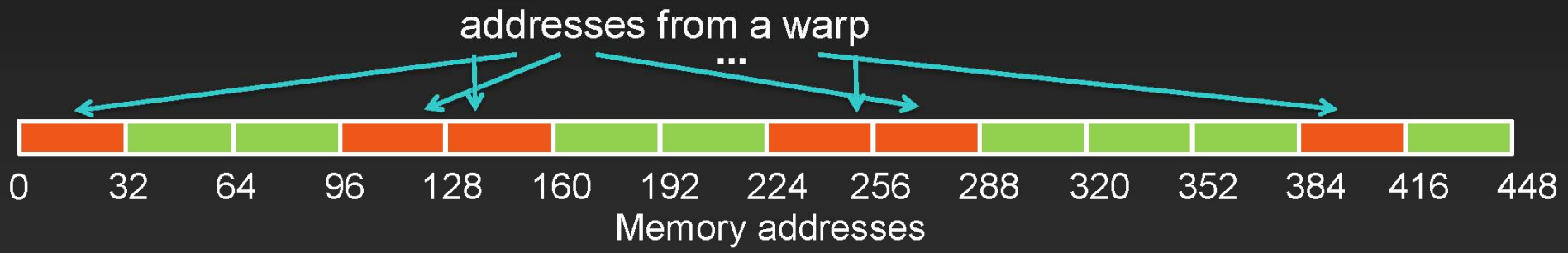


Access Patterns vs. Memory Throughput

- Scenario:
  - All threads in a warp request the same 4-byte word
- Addresses fall within a single segment
  - Warp needs 4 bytes
  - 32 bytes move across the bus
  - Bus utilization: 12.5%

Access Patterns vs. Memory Throughput

- Scenario:
  - Warp requests 32 scattered 4-byte words
- Addresses fall within *N* segments
  - Warp needs 128 bytes
  - *N**32 bytes move across the bus
  - Bus utilization: 128 / (*N**32)



Structures of Non-Native Size

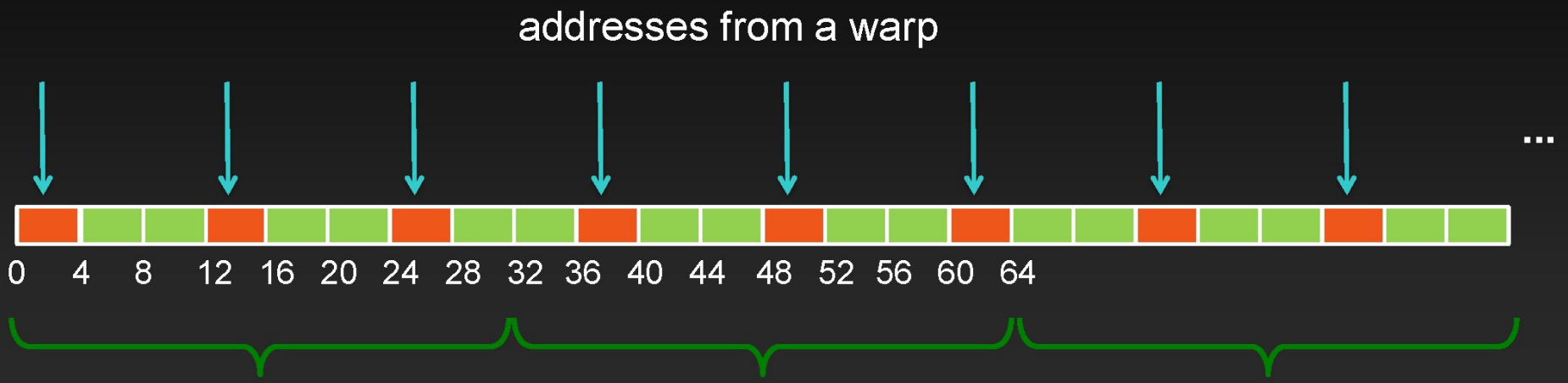
- Say we are reading a 12-byte structure per thread

```
struct Position
{
  float x, y, z;
};
...
__global__ void kernel( Position *data, ... )
{
  int idx = blockIdx.x * blockDim.x + threadIdx.x;
  Position temp = data[idx];
  ...
}
```

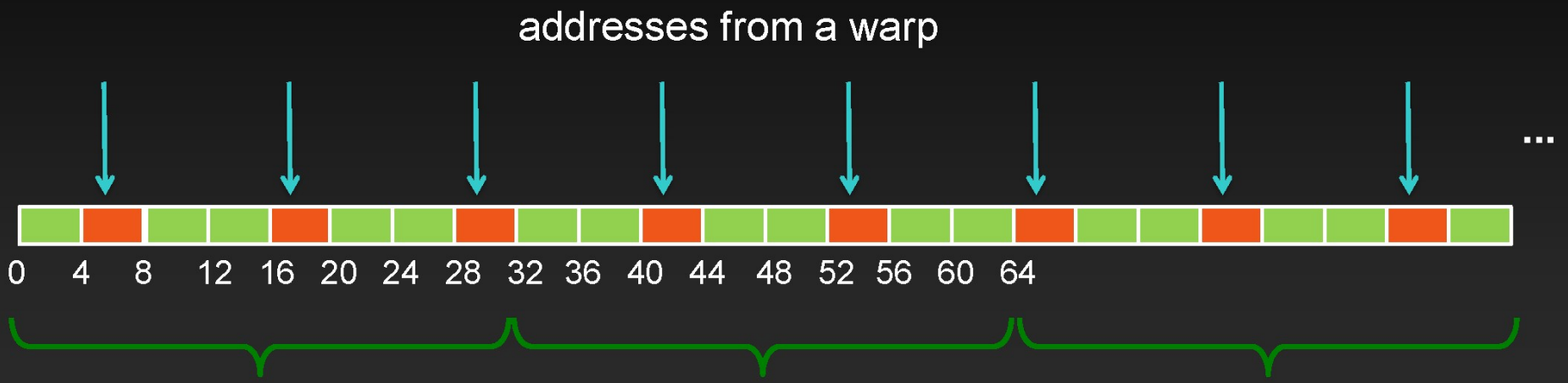
Structure of Non-Native Size

- Compiler converts `temp = data[idx]` into 3 loads:
  - Each loads 4 bytes
  - Can't do an 8 and a 4 byte load: 12 bytes per element means that every other element wouldn't align the 8-byte load on 8-byte boundary
- Addresses per warp for each of the loads:
  - Successive threads read 4 bytes at 12-byte stride

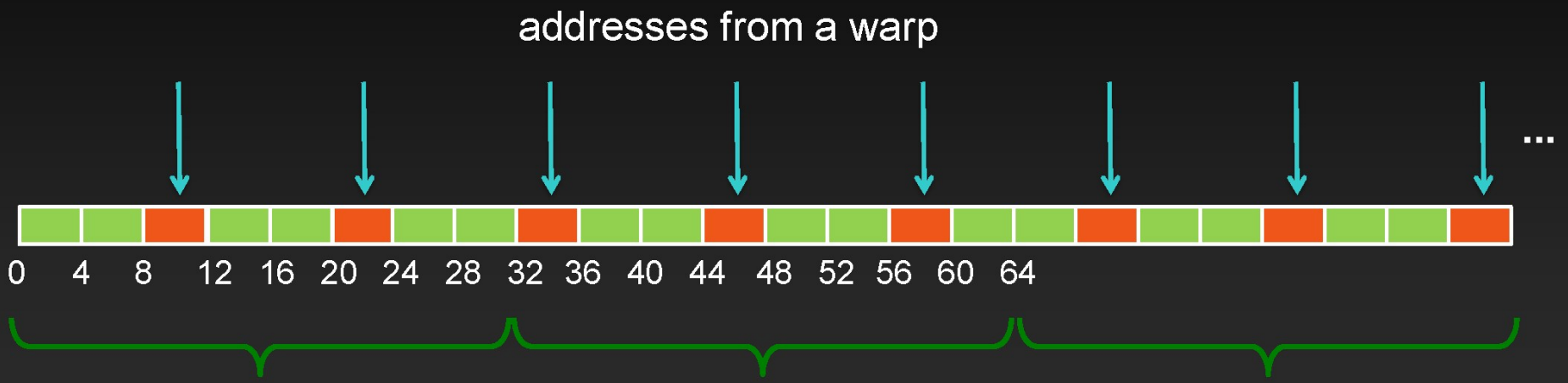
First Load Instruction



Second Load Instruction



Third Load Instruction

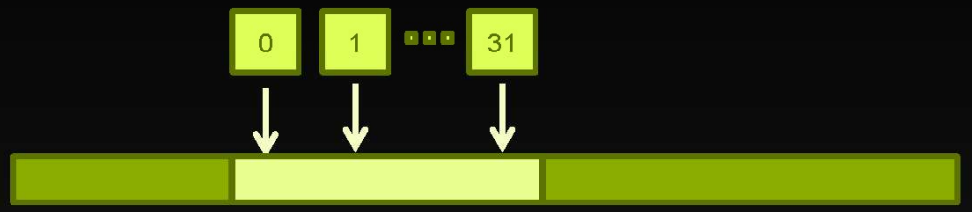


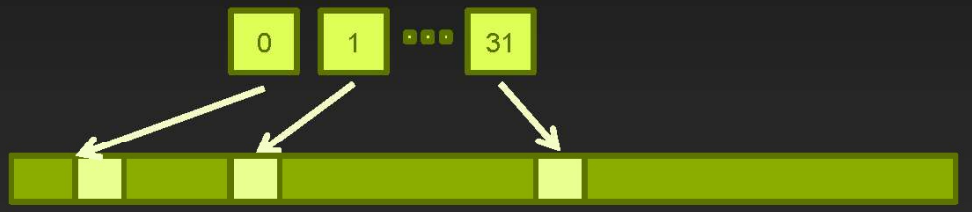
Performance and Solutions

- Because of the address pattern, we end up moving 3x more bytes than application requests
  - We waste a lot of bandwidth, leaving performance on the table
- Potential solutions:
  - Change data layout from array of structures to structure of arrays
    - In this case: 3 separate arrays of floats
    - The most reliable approach (also ideal for both CPUs and GPUs)
  - Use loads via read-only cache
    - As long as lines survive in the cache, performance will be nearly optimal
  - Stage loads via shared memory

Global Memory Access Patterns

- SoA vs AoS:
  - Good: point.x[i]
  - Not so good: point[i].x
- Strided array access:
  - ~OK: x[i] = a[i+1] – a[i]
  - Slower: x[i] = a[64*i] – a[i]
- Random array access:
  - Slower: a[rand(i)]



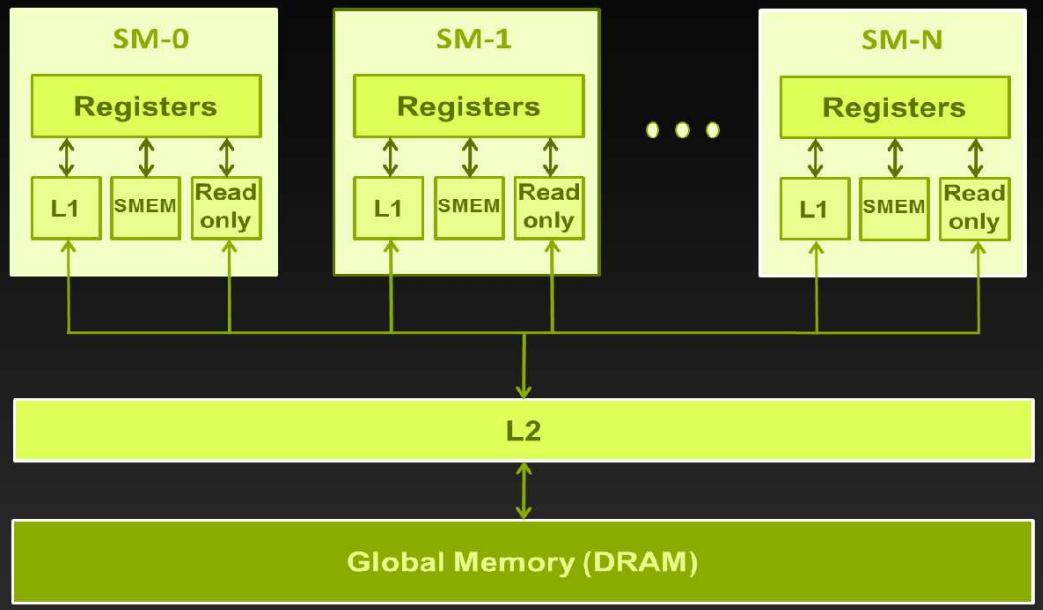


Summary: GMEM Optimization

- Strive for perfect address coalescing per warp
  - Align starting address (may require padding)
  - A warp will ideally access within a contiguous region
  - Avoid scattered address patterns or patterns with large strides between threads
- Analyze and optimize address patterns:
  - Use profiling tools (included with CUDA toolkit download)
  - Compare the transactions per request to the ideal ratio
  - Choose appropriate data layout (prefer SoA)
  - If needed, try read-only loads, staging accesses via SMEM

A note about caches

- L1 and L2 caches
  - Ignore in software design
  - Thousands of concurrent threads – cache blocking difficult at best
- Read-only Data Cache
  - Shared with texture pipeline
  - Useful for uncoalesced reads
  - Handled by compiler when `const __restrict__` is used, or use `__ldg()` primitive



Read-only Data Cache

- Go through the read-only cache
  - Not coherent with writes
  - Thus, addresses must not be written by the same kernel
- Two ways to enable:
  - Decorating pointer arguments as hints to compiler:
    - Pointer of interest: const __restrict__
    - All other pointer arguments: __restrict__
      - Conveys to compiler that no aliasing will occur
  - Using __ldg() intrinsic
    - Requires no pointer decoration

Read-only Data Cache

- Go through the read-only cache
  - Not coherent with writes
  - Thus, addresses must not be written by the same kernel
- Two ways to enable:
  - Decorating pointer argumen
    - Pointer of interest: const
    - All other pointer argumen
      - Conveys to compiler t
  - Using __ldg() intrinsic
    - Requires no pointer deco

```
__global__ void kernel(
                int* __restrict__ output,
          const int* __restrict__ input )
{
      ...
      output[idx] = input[idx];
}
```

Read-only Data Cache

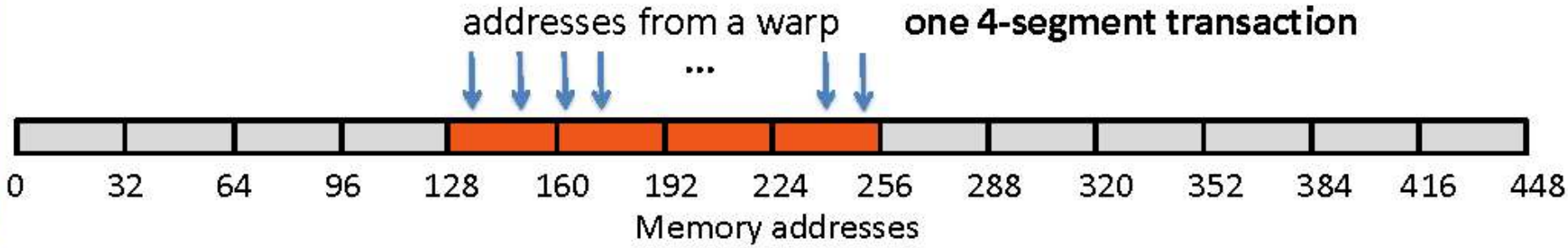
- Go through the read-only cache
  - Not coherent with writes
  - Thus, addresses must not be written by the same kernel
- Two ways to enable:
  - Decorating pointer argumen
    - Pointer of interest: const
    - All other pointer argument
      - Conveys to compiler t
  - Using __ldg() intrinsic
    - Requires no pointer deco

```
__global__ void kernel( int *output,
                        int *input )
{
    ...
    output[idx] = __ldg( &input[idx] );
}
```

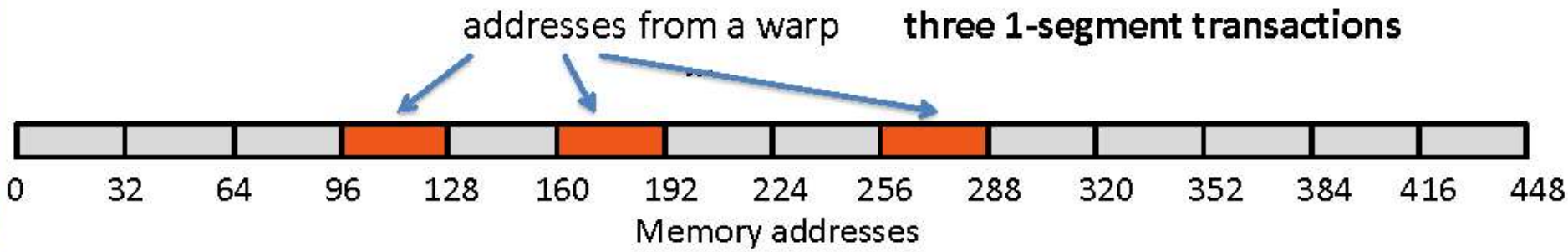
Blocking for L1, Read-only, L2 Caches

- Short answer: DON'T
- GPU caches are not intended for the same use as CPU caches
  - Smaller size (especially per thread), so not aimed at temporal reuse
  - Intended to smooth out some access patterns, help with spilled registers, etc.
- Usually not worth trying to cache-block like you would on CPU
  - 100s to 1,000s of run-time scheduled threads competing for the cache
  - If it is possible to block for L1 then it's possible block for SMEM
    - Same size
    - Same or higher bandwidth
    - Guaranteed locality: hw will not evict behind your back

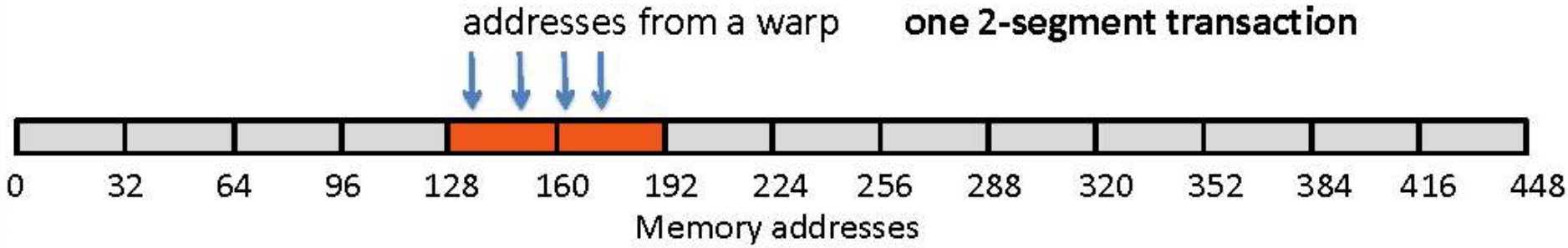
Some Store Pattern Examples



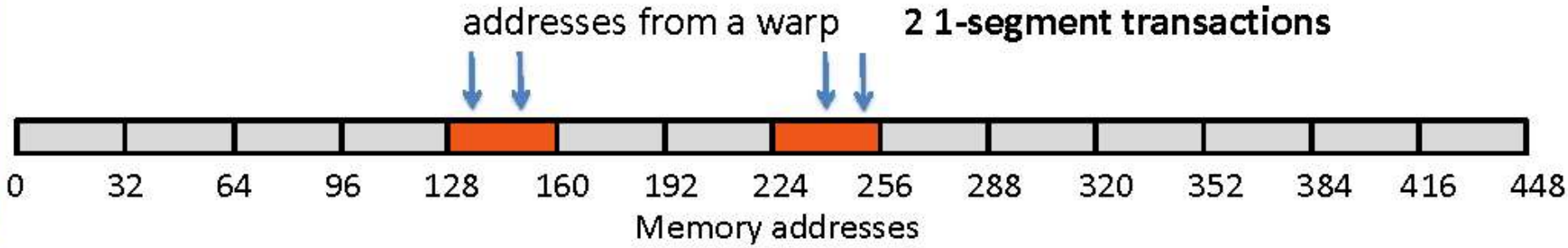
Some Store Pattern Examples



Some Store Pattern Examples



Some Store Pattern Examples

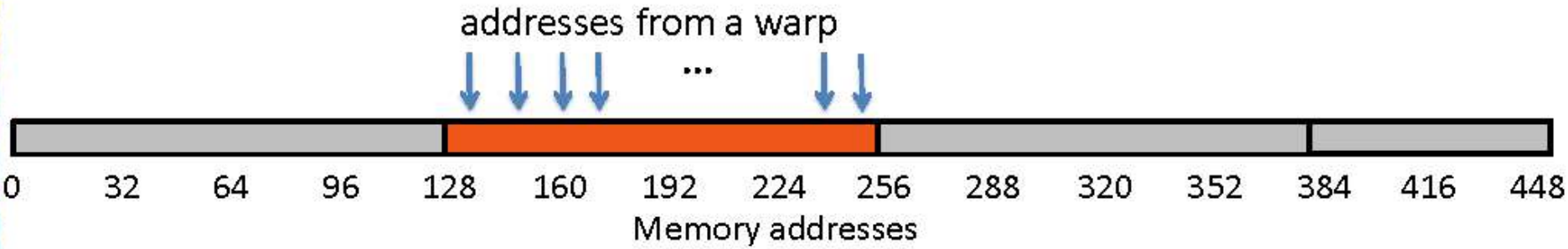


GMEM Reads

- **Attempt to hit in L1 depends on programmer choice and compute capability**
- **HW ability to hit in L1:**
  - CC 1.x: no L1
  - CC 2.x: can hit in L1
  - CC 3.0, 3.5: cannot hit in L1
    - L1 is used to cache LMEM (register spills, etc.), buffer reads
- **Read instruction types**
  - Caching:
    - Compiler option: **-Xptxas -dlcm=ca**
    - On L1 miss go to L2, on L2 miss go to DRAM
    - Transaction: 128 B line
  - Non-caching:
    - Compiler option: **-Xptxas -dlcm=cg**
    - Go directly to L2 (invalidate line in L1), on L2 miss go to DRAM
    - Transaction: 1, 2, 4 segments, segment = 32 B (same as for writes)

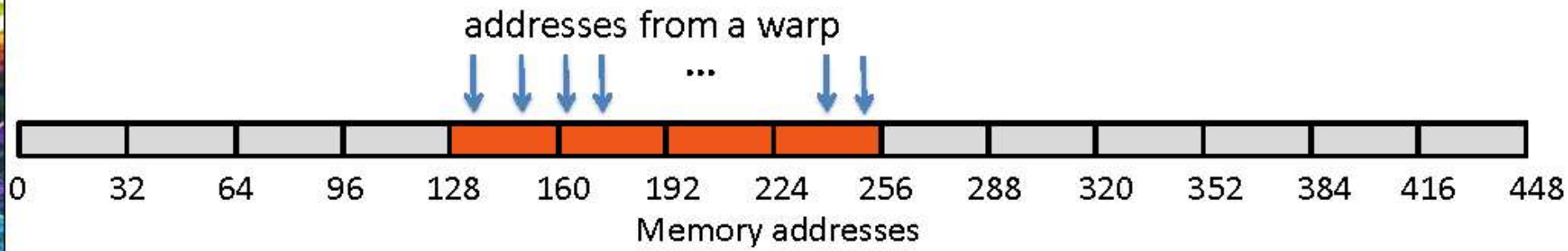
Caching Load

- **Scenario:**
  - Warp requests 32 aligned, consecutive 4-byte words
- **Addresses fall within 1 cache-line**
  - No replays
  - Bus utilization: 100%
    - Warp needs 128 bytes
    - 128 bytes move across the bus on a miss



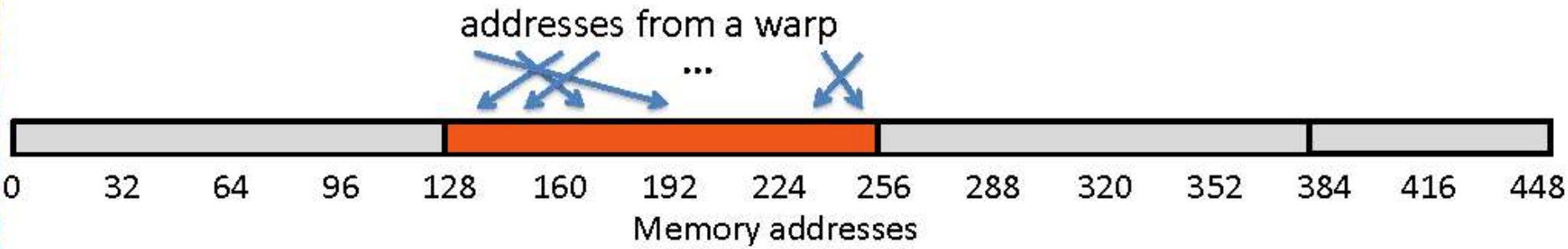
Non-caching Load

- **Scenario:**
  - Warp requests 32 aligned, consecutive 4-byte words
- **Addresses fall within 4 segments**
  - No replays
  - Bus utilization: 100%
    - Warp needs 128 bytes
    - 128 bytes move across the bus on a miss



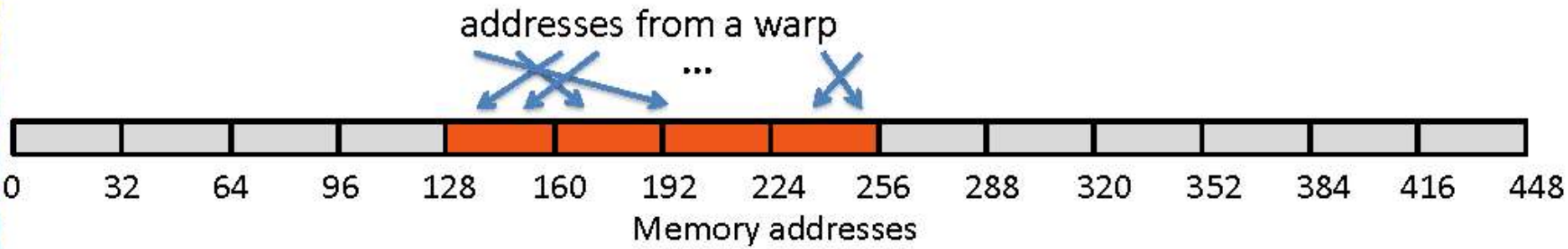
Caching Load

- **Scenario:**
  - Warp requests 32 aligned, permuted 4-byte words
- **Addresses fall within 1 cache-line**
  - No replays
  - Bus utilization: 100%
    - Warp needs 128 bytes
    - 128 bytes move across the bus on a miss



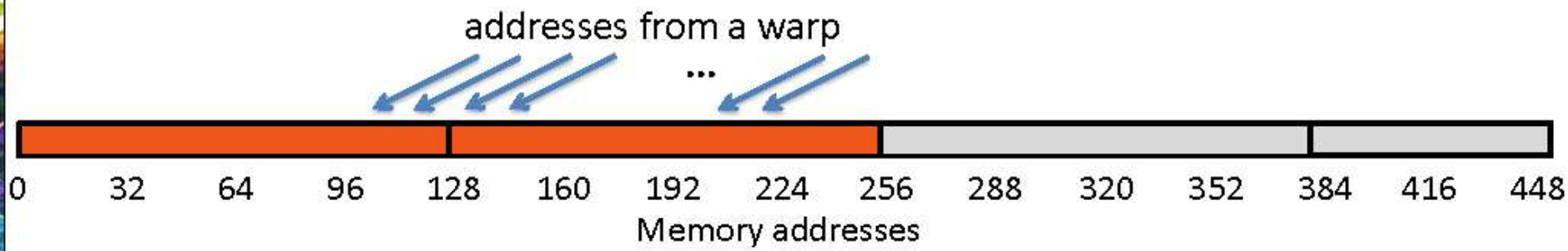
Non-caching Load

- **Scenario:**
  - Warp requests 32 aligned, permuted 4-byte words
- **Addresses fall within 4 segments**
  - No replays
  - Bus utilization: 100%
    - Warp needs 128 bytes
    - 128 bytes move across the bus on a miss



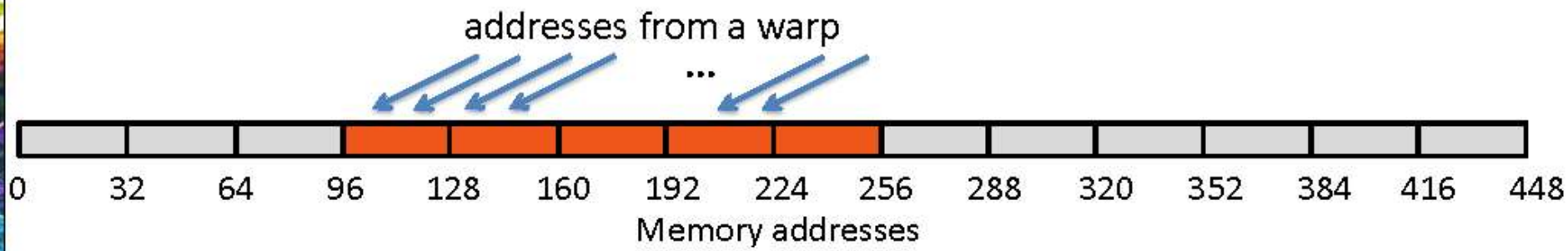
Caching Load

- **Scenario:**
  - Warp requests 32 consecutive 4-byte words, offset from perfect alignment
- **Addresses fall within 2 cache-lines**
  - 1 replay (2 transactions)
  - Bus utilization: 50%
    - Warp needs 128 bytes
    - 256 bytes move across the bus on misses



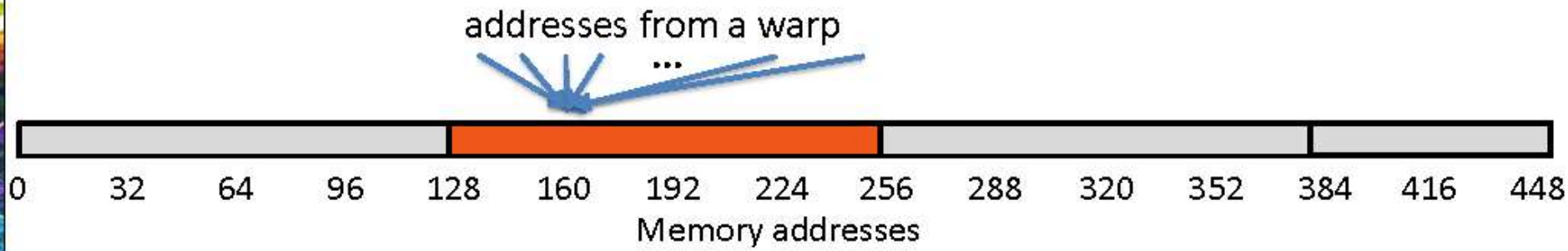
Non-caching Load

- **Scenario:**
  - Warp requests 32 consecutive 4-byte words, offset from perfect alignment
- **Addresses fall within at most 5 segments**
  - 1 replay (2 transactions)
  - Bus utilization: at least 80%
    - Warp needs 128 bytes
    - At most 160 bytes move across the bus
    - Some misaligned patterns will fall within 4 segments, so 100% utilization



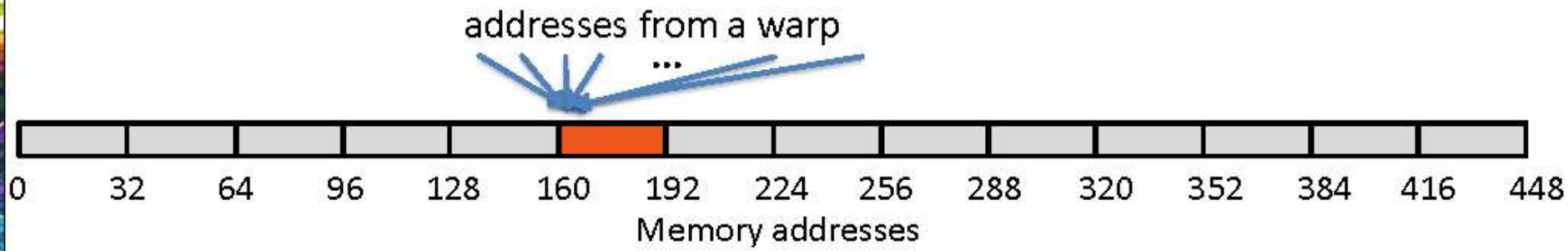
Caching Load

- **Scenario:**
  - All threads in a warp request the same 4-byte word
- **Addresses fall within a single cache-line**
  - No replays
  - Bus utilization: 3.125%
    - Warp needs 4 bytes
    - 128 bytes move across the bus on a miss



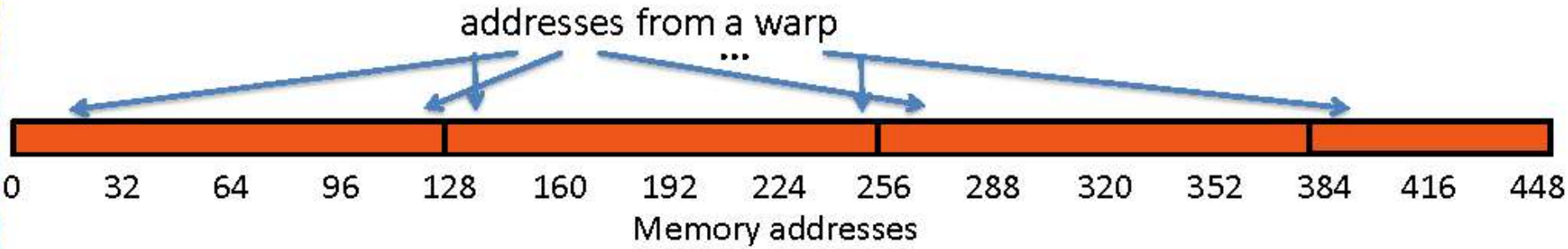
Non-caching Load

- **Scenario:**
  - All threads in a warp request the same 4-byte word
- **Addresses fall within a single segment**
  - No replays
  - Bus utilization: 12.5%
    - Warp needs 4 bytes
    - 32 bytes move across the bus on a miss



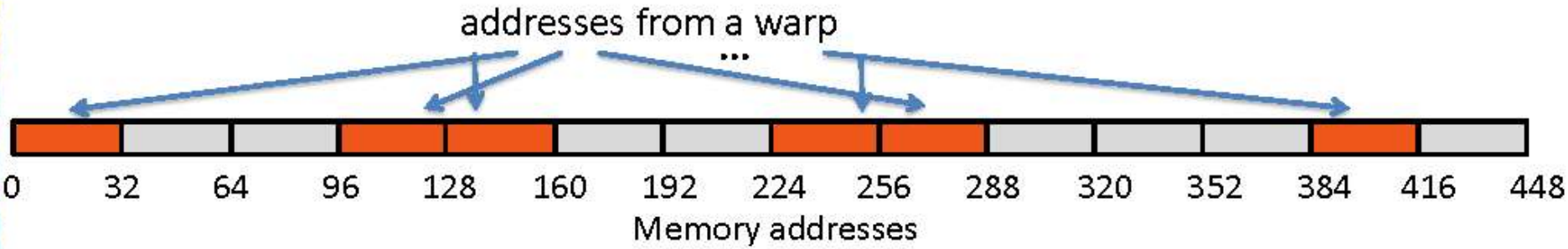
Caching Load

- **Scenario:**
  - Warp requests 32 scattered 4-byte words
- **Addresses fall within *N* cache-lines**
  - (N-1) replays (N transactions)
  - Bus utilization: 32*4B / (*N*128B)
    - Warp needs 128 bytes
    - *N*128 bytes move across the bus on a miss



Non-caching Load

- **Scenario:**
  - Warp requests 32 scattered 4-byte words
- **Addresses fall within *N* segments**
  - (N-1) replays (N transactions)
    - Could be lower some segments can be arranged into a single transaction
  - Bus utilization: 128 / (*N**32) (4x higher than caching loads)
    - Warp needs 128 bytes
    - *N**32 bytes move across the bus on a miss



Caching vs Non-caching Loads

- **Compute capabilities that can hit in L1 (CC 2.x)**
  - Caching loads are better if you count on hits
  - Non-caching loads are better if:
    - Warp address pattern is scattered
    - When kernel uses lots of LMEM (register spilling)
- **Compute capabilities that cannot hit in L1 (CC 1.x, 3.0, 3.5)**
  - Does not matter, all loads behave like non-caching
- **In general, don't rely on GPU caches like you would on CPUs:**
  - 100s of threads sharing the same L1
  - 1000s of threads sharing the same L2

L1 Sizing

- **Fermi and Kepler GPUs split 64 KB RAM between L1 and SMEM**
  - Fermi GPUs (**CC 2.x**): 16:48, 48:16
  - Kepler GPUs (**CC 3.x**):16:48, 48:16, 32:32
- **Programmer can choose the split:**
  - Default: 16 KB L1, 48 KB SMEM
  - Run-time API functions:
    - cudaDeviceSetCacheConfig(), cudaFuncSetCacheConfig()
  - Kernels that require different L1:SMEM sizing cannot run concurrently
- **Making the choice:**
  - Large L1 can help when using lots of LMEM (spilling registers)
  - Large SMEM can help if occupancy is limited by shared memory

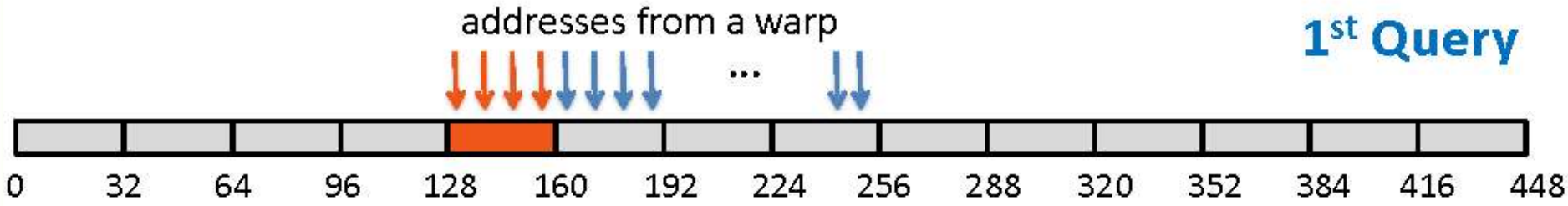
Read-Only Cache

- **An alternative to L1 when accessing DRAM**
  - Also known as *texture* cache: all texture accesses use this cache
  - CC 3.5 and higher also enable global memory accesses
    - Should not be used if a kernel reads and writes to the same addresses
- **Comparing to L1:**
  - Generally better for scattered reads than L1
    - Caching is at 32 B granularity (L1, when caching operates at 128 B granularity)
    - Does not require replay for multiple transactions (L1 does)
  - Higher latency than L1 reads, also tends to increase register use
- **Aggregate 48 KB per SM: 4 12-KB caches**
  - One 12-KB cache per scheduler
    - Warps assigned to a scheduler refer to only that cache
  - Caches are not coherent – data replication is possible

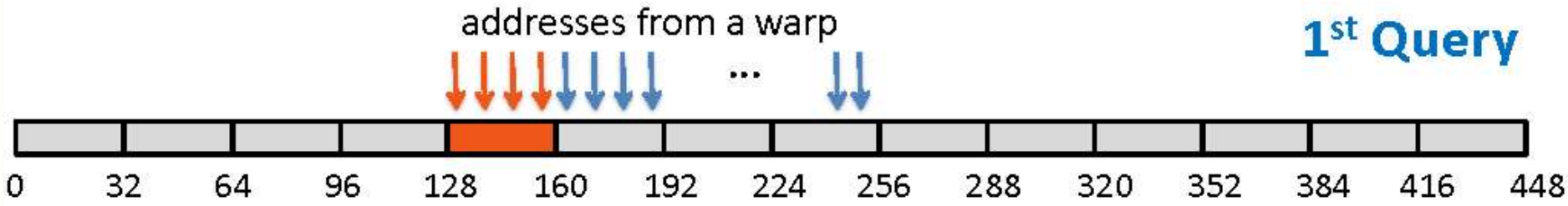
Read-Only Cache Operation

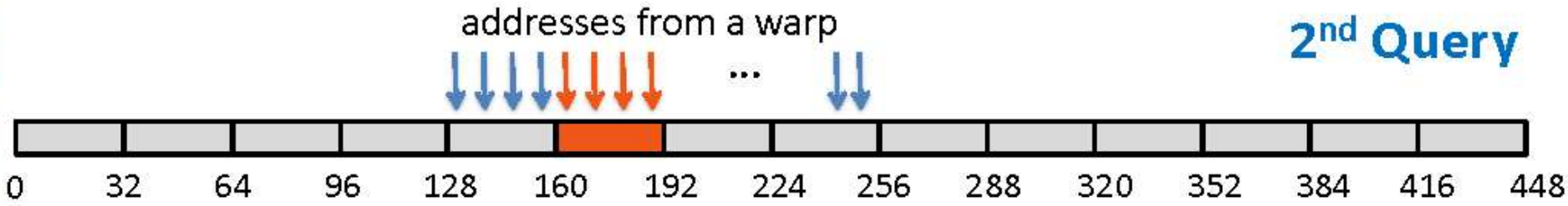
- **Always attempts to hit**
- **Transaction size: 32 B queries**
- **Warp addresses are converted to queries 4 threads at a time**
  - Thus a minimum of 8 queries per warp
  - If data within a 32-B segment is needed by multiple threads in a warp, segment misses at most once
- **Additional functionality for texture objects**
  - Interpolation, clamping, type conversion

Read-Only Cache Operation

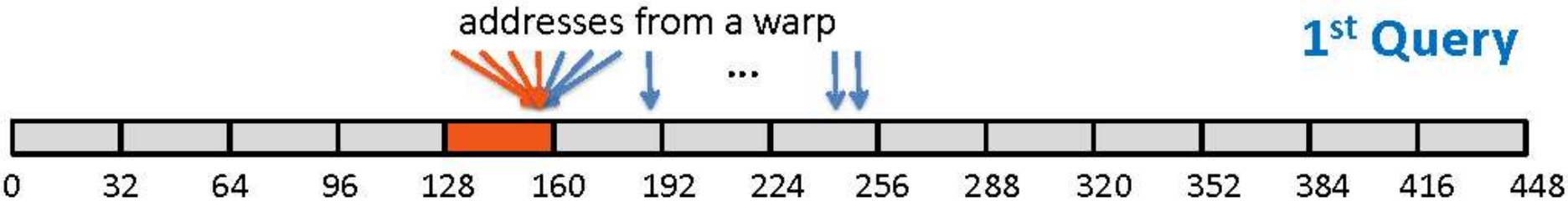


Read-Only Cache Operation

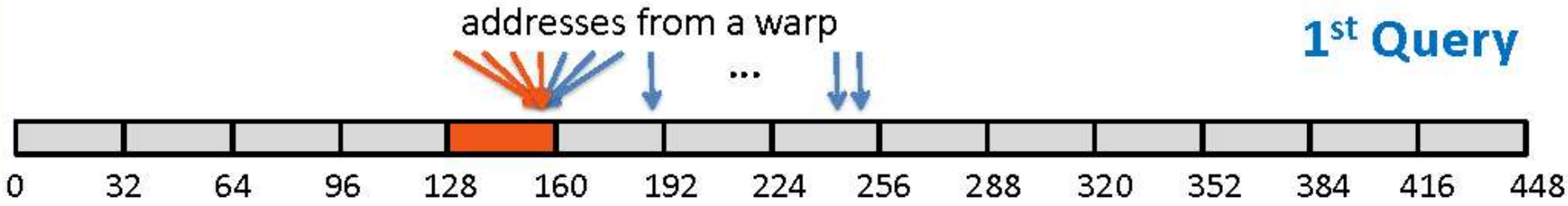


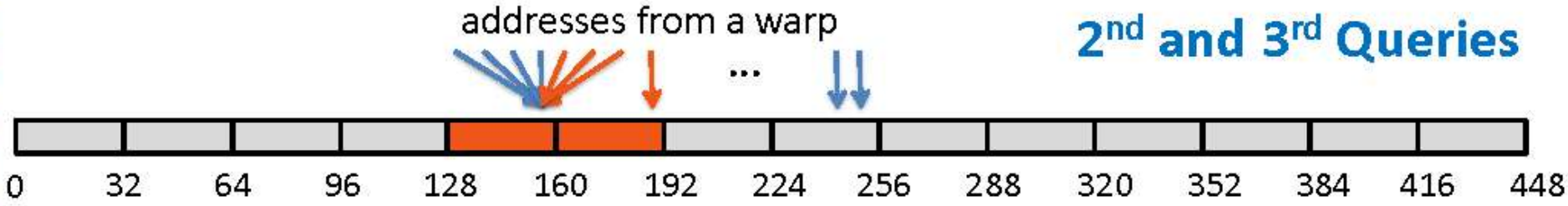


Read-Only Cache Operation

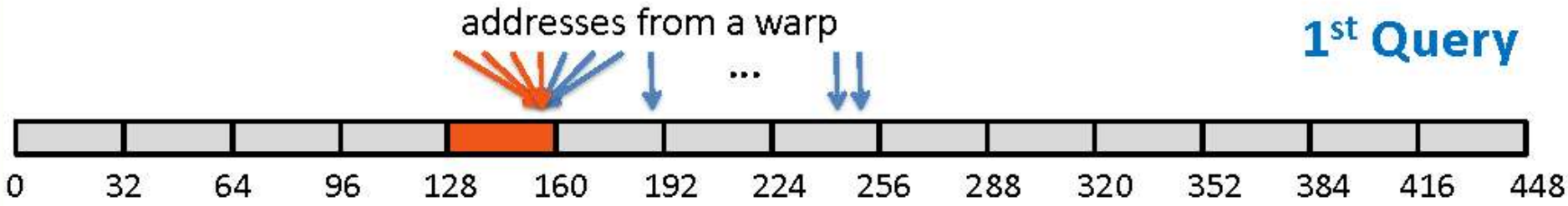


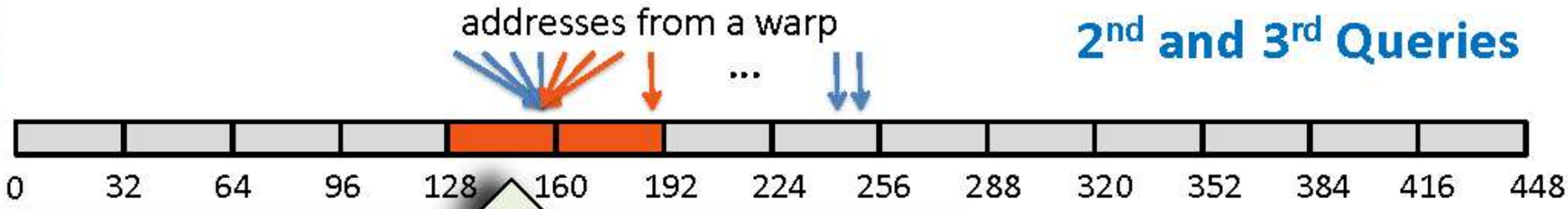
Read-Only Cache Operation





Read-Only Cache Operation





Note this segment was already requested in the 1st query: cache hit, no redundant requests to L2

GPU Reduction

- Parallel reduction is a basic parallel programming primitive; see reduction operation on a stream, e.g., in Brook for GPUs

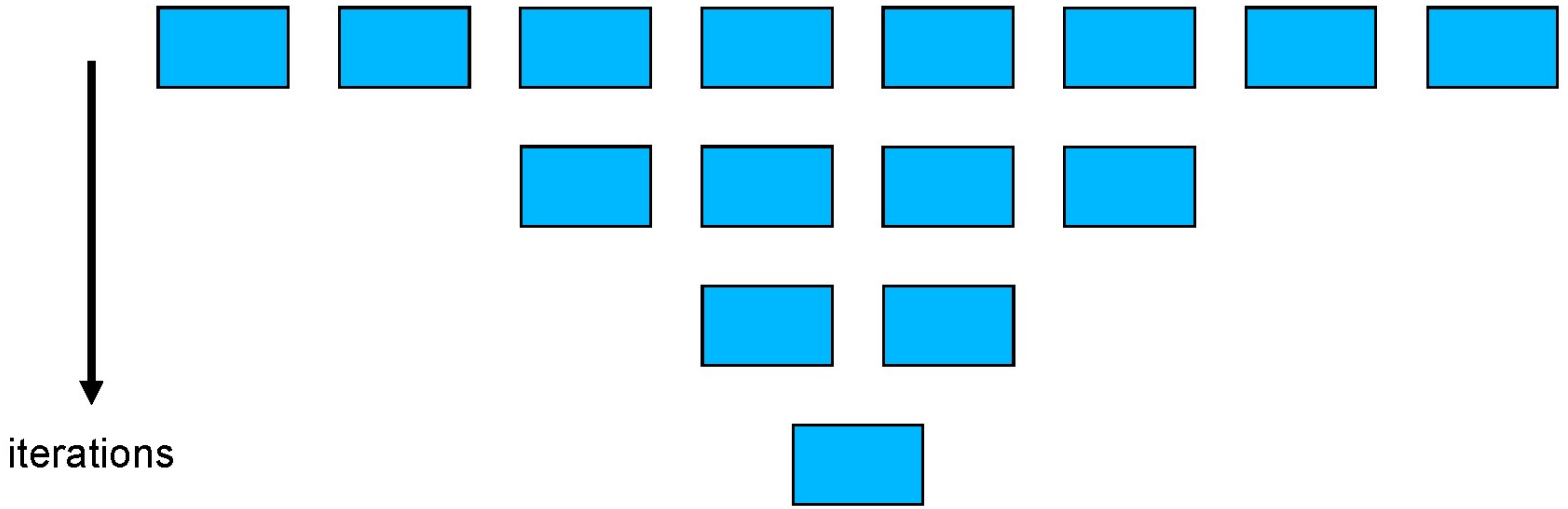
Example: Parallel Reduction

- **Given an array of values, "reduce" them to a single value in parallel**
- **Examples**
  - sum reduction: sum of all values in the array
  - Max reduction: maximum of all values in the array

- **Typical parallel implementation:**
  - Recursively halve # threads, add two values per thread
  - Takes log(n) steps for n elements, requires n/2 threads

Typical Parallel Programming Pattern

- **log(n) steps**



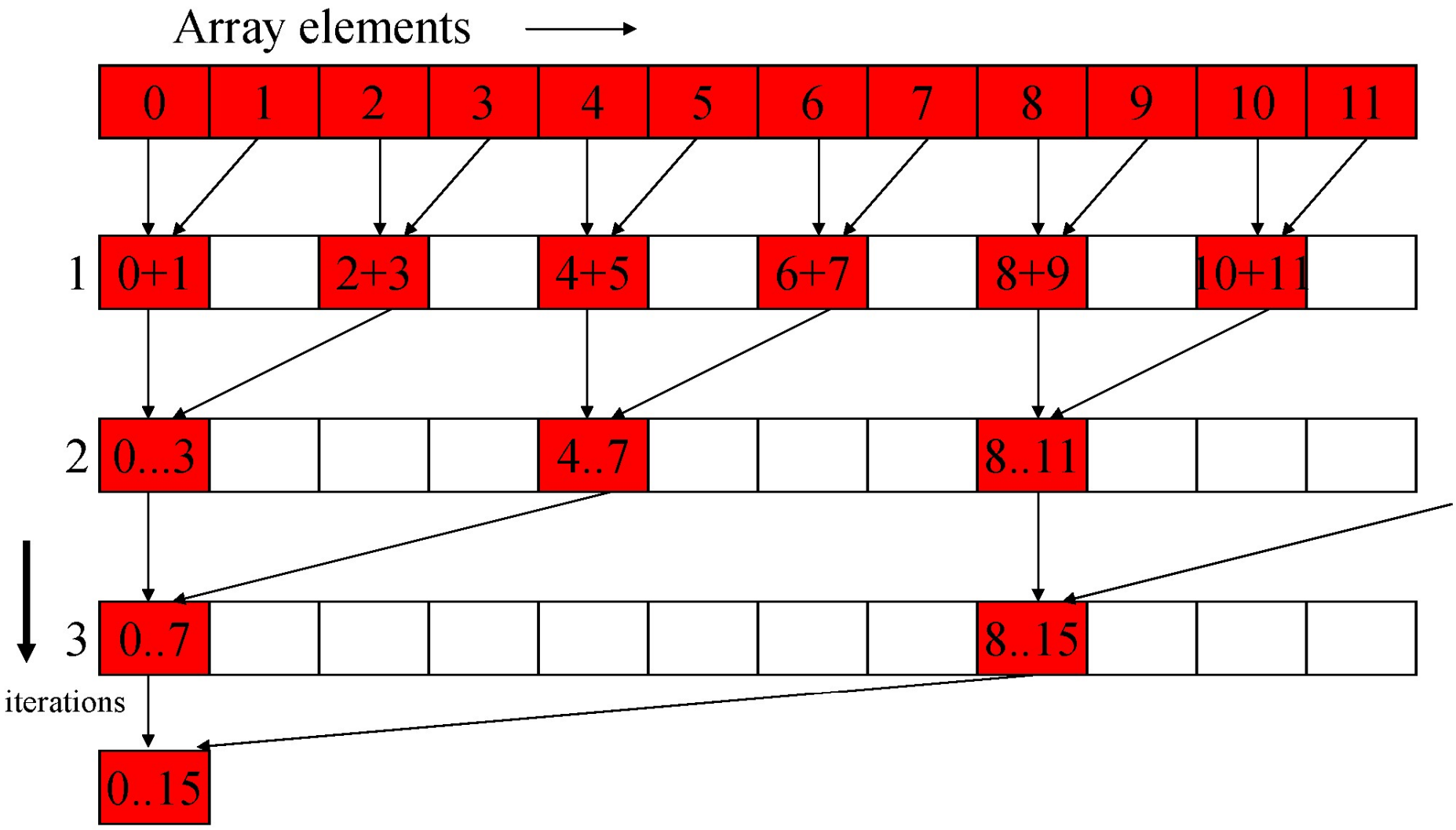
Helpful fact for counting nodes of full binary trees:
If there are N leaf nodes, there will be N-1 non-leaf nodes

Reduction – Version1

A Vector Reduction Example

- **Assume an in-place reduction using shared memory**
  - The original vector is in device global memory
  - The shared memory used to hold a partial sum vector
  - Each iteration brings the partial sum vector closer to the final sum
  - The final solution will be in element 0

Vector Reduction



A Simple Implementation

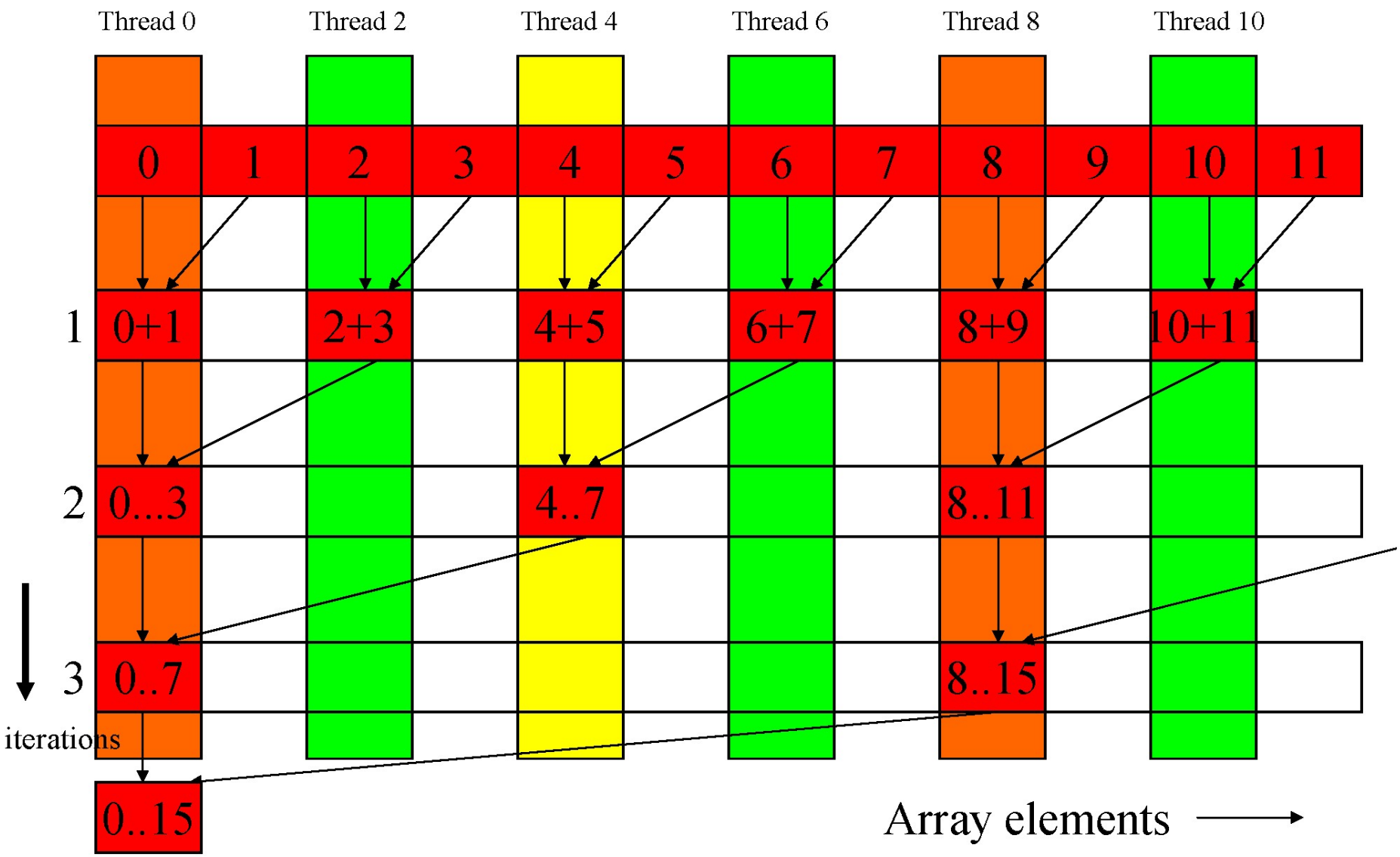
- **Assume we have already loaded array into**

```
__shared__ float partialSum[];

unsigned int t = threadIdx.x;

// loop log(n) times
for (unsigned int stride = 1;
     stride < blockDim.x;  stride *= 2)
{
  // make sure the sum of the previous iteration
  // is available
  __syncthreads();

  if (t % (2*stride) == 0)
     partialSum[t] += partialSum[t+stride];
}
```

Reduction #1: Interleaved Addressing


```
__global__ void reduce0(int *g_idata, int *g_odata) {
    extern __shared__ int sdata[];

    // each thread loads one element from global to shared mem
    unsigned int tid = threadIdx.x;
    unsigned int i = blockIdx.x*blockDim.x + threadIdx.x;
    sdata[tid] = g_idata[i];
    __syncthreads();

    // do reduction in shared mem
    for(unsigned int s=1; s < blockDim.x; s *= 2) {
        if (tid % (2*s) == 0) {
            sdata[tid] += sdata[tid + s];
        }
        __syncthreads();
    }

    // write result for this block to global mem
    if (tid == 0) g_odata[blockIdx.x] = sdata[0];
}
```

Vector Reduction with Branch Divergence



Some Observations

- **In each iterations, two control flow paths will be sequentially traversed for each warp**
  - Threads that perform addition and threads that do not
  - Threads that do not perform addition may cost extra cycles depending on the implementation of divergence

- **No more than half of threads will be executing at any time**
  - All odd index threads are disabled right from the beginning!
  - On average, less than ¼ of the threads will be activated for all warps over time.
  - After the 5^{th} iteration, entire warps in each block will be disabled, poor resource utilization but no divergence.
    - This can go on for a while, up to 4 more iterations (512/32=16= 2^4), where each iteration only has one thread activated until all warps retire

Short comings of the implementation

- **Assume we have already loaded array into**

```
__shared__ float partialSum[];

unsigned int t = threadIdx.x;
for (unsigned int stride = 1;
     stride < blockDim.x;  stride *= 2)
{
  __syncthreads();

  if (t % (2*stride) == 0)
    partialSum[t] += partialSum[t+stride];
}
```

BAD: Divergence due to interleaved branch decisions

BAD: Bank conflicts due to stride

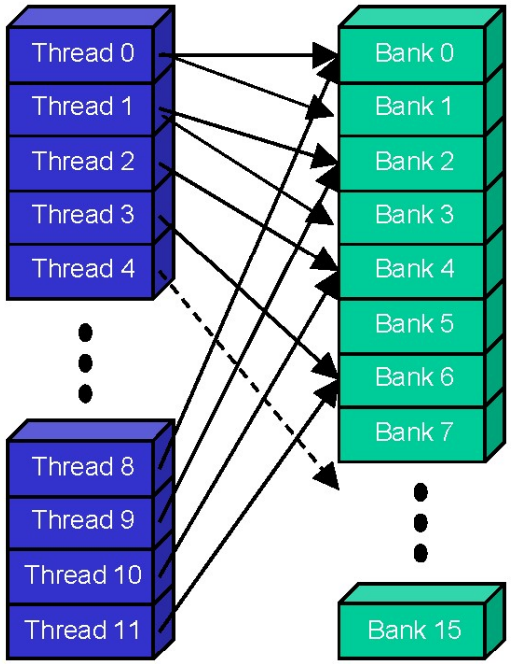
Reduction – Version2

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];
```

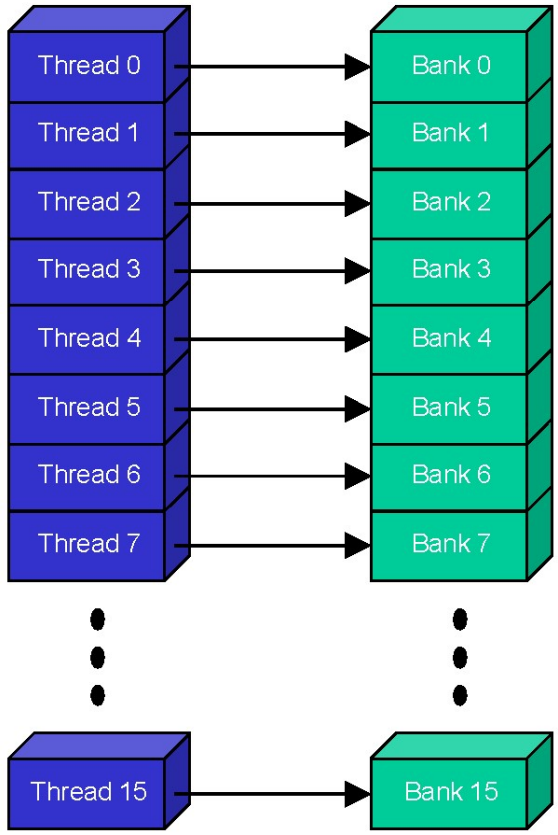
- **This makes sense for traditional CPU threads, locality in cache line usage and reduced sharing traffic.**
  - Not in shared memory usage where there is no cache line effects but banking effects



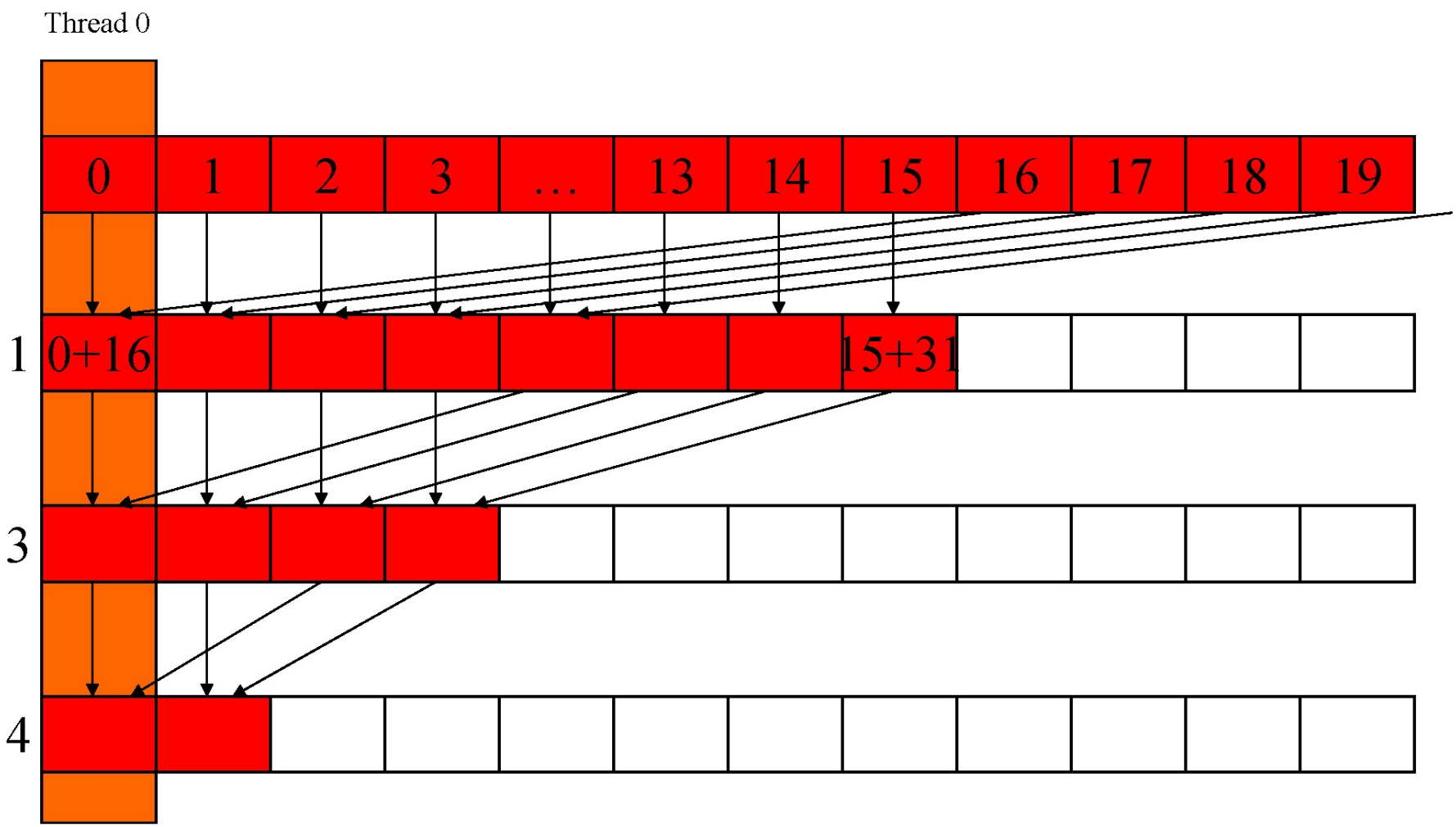
A Better Array Access Pattern

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

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



A better implementation



A better implementation

- **Assume we have already loaded array into**

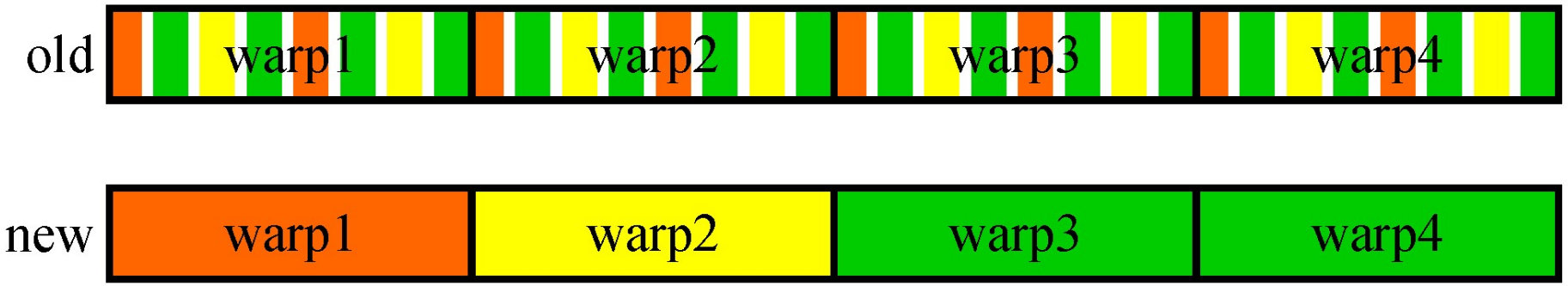
```
__shared__ float partialSum[];

unsigned int t = threadIdx.x;
for (unsigned int stride = blockDim.x;
     stride > 1;  stride >>=1)
{
  __syncthreads();
  if (t < stride)
     partialSum[t] += partialSum[t+stride];
}
```

if you want to fully retire warps, this should actually be:

```
if ( t < stride ) {
   partialSum[ t ] += partialSum[ t + stride ];
} else {
   break;
}
```

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**



Implicit Synchronization in a Warp

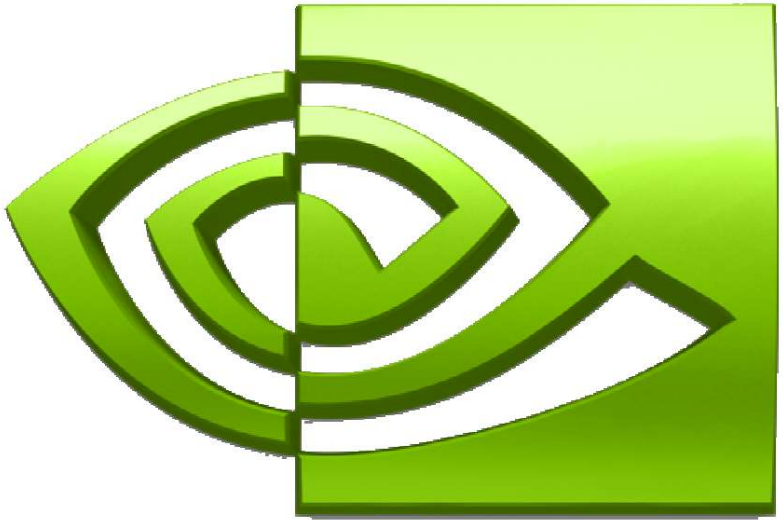
- **For last 6 loops only one warp active (i.e. tid's 0..31)**
  - Shared reads & writes SIMD synchronous within a warp
  - So skip `__syncthreads()` and unroll last 5 iterations

```
unsigned int tid = threadIdx.x
for (unsigned int d = n>>1; d
    __syncthreads();
    if (tid < d)
        shared[tid] += shared[
}
__syncthreads();
if (tid <= 32) {   // unroll last
    shared[tid] += shared[tid
    shared[tid] += shared[tid
    shared[tid] += shared[tid
    shared[tid] += shared[tid +
    shared[tid] += shared[tid + 2];
    shared[tid] += shared[tid + 1];
}
```

This would not work properly is warp size decreases; need __synchthreads() between each statement!
However, having ___synchthreads() in if statement is problematic.

now: `__syncwarp()`
or better: Cooperative Groups

Look at CUDA SDK reduction example and slides!



NVIDIA®

Optimizing Parallel Reduction in CUDA

Mark Harris
NVIDIA Developer Technology

Parallel Reduction

- Common and important data parallel primitive
- Easy to implement in CUDA
  - Harder to get it right
- Serves as a great optimization example
  - We'll walk step by step through 7 different versions
  - Demonstrates several important optimization strategies

```
template <unsigned int blockSize>
__global__ void reduce6(int *g_idata, int *g_odata, unsigned int n)
{
    extern __shared__ int sdata[];

    unsigned int tid = threadIdx.x;
    unsigned int i = blockIdx.x*(blockSize*2) + tid;
    unsigned int gridSize = blockSize*2*gridDim.x;
    sdata[tid] = 0;

    while (i < n) { sdata[tid] += g_idata[i] + g_idata[i+blockSize];  i += gridSize;  }
    __syncthreads();

    if (blockSize >= 512) { if (tid < 256) { sdata[tid] += sdata[tid + 256]; } __syncthreads(); }
    if (blockSize >= 256) { if (tid < 128) { sdata[tid] += sdata[tid + 128]; } __syncthreads(); }
    if (blockSize >= 128) { if (tid <  64) { sdata[tid] += sdata[tid +  64]; } __syncthreads(); }

    if (tid < 32) {
        if (blockSize >=  64) sdata[tid] += sdata[tid + 32];
        if (blockSize >=  32) sdata[tid] += sdata[tid + 16];
        if (blockSize >=  16) sdata[tid] += sdata[tid +  8];
        if (blockSize >=   8) sdata[tid] += sdata[tid +  4];
        if (blockSize >=   4) sdata[tid] += sdata[tid +  2];
        if (blockSize >=   2) sdata[tid] += sdata[tid +  1];
    }

    if (tid == 0) g_odata[blockIdx.x] = sdata[0];
}
```

Final Optimized Kernel

out-of-bounds check missing, see SDK code

be careful that shared variables are declared *volatile*! see SDK code

```
template <unsigned int blockSize>
__device__ void warpReduce(volatile int *sdata, unsigned int tid) {
    if (blockSize >=  64) sdata[tid] += sdata[tid + 32];
    if (blockSize >=  32) sdata[tid] += sdata[tid + 16];
    if (blockSize >=  16) sdata[tid] += sdata[tid +  8];
    if (blockSize >=   8) sdata[tid] += sdata[tid +  4];
    if (blockSize >=   4) sdata[tid] += sdata[tid +  2];
    if (blockSize >=   2) sdata[tid] += sdata[tid +  1];
}
template <unsigned int blockSize>
__global__ void reduce6(int *g_idata, int *g_odata, unsigned int n) {
    extern __shared__ int sdata[];
    unsigned int tid = threadIdx.x;
    unsigned int i = blockIdx.x*(blockSize*2) + tid;
    unsigned int gridSize = blockSize*2*gridDim.x;
    sdata[tid] = 0;

    while (i < n) { sdata[tid] += g_idata[i] + g_idata[i+blockSize];  i += gridSize;  }
    __syncthreads();

    if (blockSize >= 512) { if (tid < 256) { sdata[tid] += sdata[tid + 256]; } __syncthreads(); }
    if (blockSize >= 256) { if (tid < 128) { sdata[tid] += sdata[tid + 128]; } __syncthreads(); }
    if (blockSize >= 128) { if (tid <   64) { sdata[tid] += sdata[tid +   64]; } __syncthreads(); }

    if (tid < 32) warpReduce(sdata, tid);
    if (tid == 0) g_odata[blockIdx.x] = sdata[0];
}
```

Final Optimized Kernel

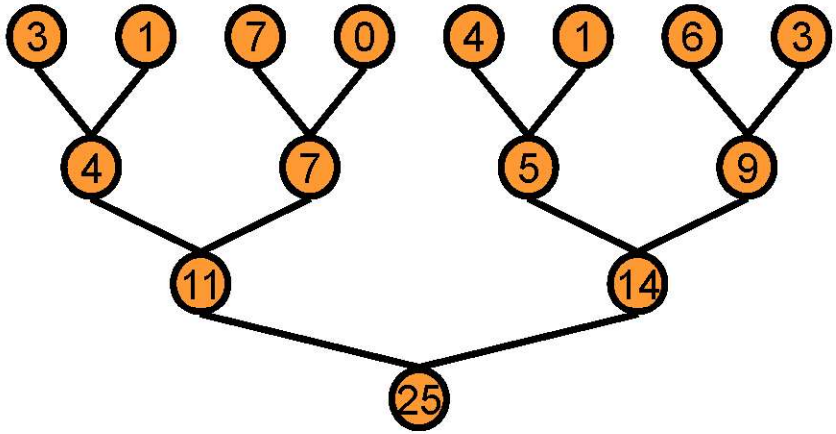
Invoking Template Kernels

- Don't we still need block size at compile time?
  - Nope, just a switch statement for 10 possible block sizes:

```
switch (threads)
    {
    case 512:
      reduce5<512><<< dimGrid, dimBlock, smemSize >>>(d_idata, d_odata); break;
    case 256:
      reduce5<256><<< dimGrid, dimBlock, smemSize >>>(d_idata, d_odata); break;
    case 128:
      reduce5<128><<< dimGrid, dimBlock, smemSize >>>(d_idata, d_odata); break;
    case 64:
      reduce5<  64><<< dimGrid, dimBlock, smemSize >>>(d_idata, d_odata); break;
    case 32:
      reduce5<  32><<< dimGrid, dimBlock, smemSize >>>(d_idata, d_odata); break;
    case 16:
      reduce5<  16><<< dimGrid, dimBlock, smemSize >>>(d_idata, d_odata); break;
    case  8:
      reduce5<   8><<< dimGrid, dimBlock, smemSize >>>(d_idata, d_odata); break;
    case  4:
      reduce5<   4><<< dimGrid, dimBlock, smemSize >>>(d_idata, d_odata); break;
    case  2:
      reduce5<   2><<< dimGrid, dimBlock, smemSize >>>(d_idata, d_odata); break;
    case  1:
      reduce5<   1><<< dimGrid, dimBlock, smemSize >>>(d_idata, d_odata); break;
    }
```

Parallel Reduction

- Tree-based approach used within each thread block



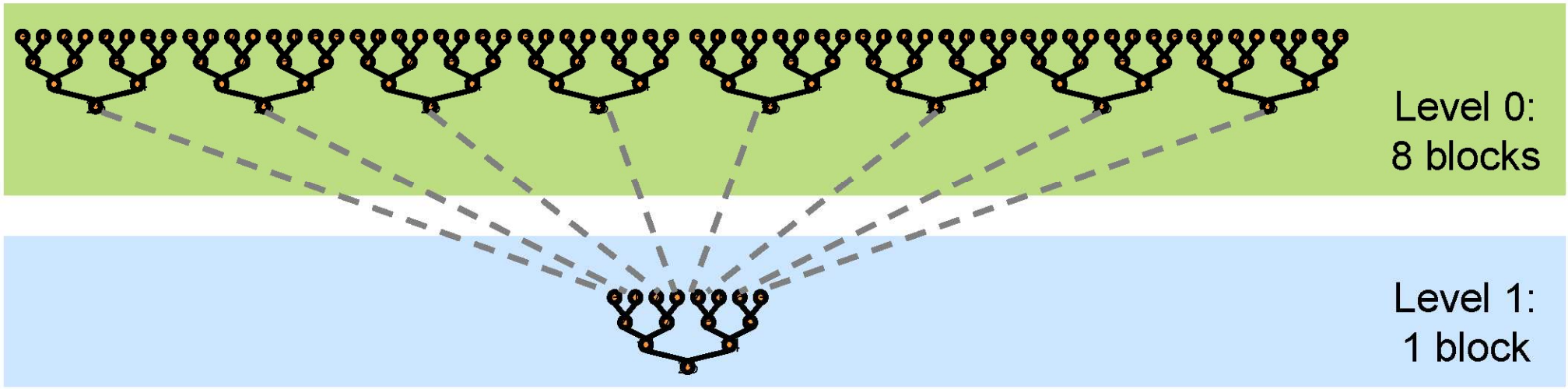
- Need to be able to use multiple thread blocks
  - To process very large arrays
  - To keep all multiprocessors on the GPU busy
  - Each thread block reduces a portion of the array
- But how do we communicate partial results between thread blocks?

Problem: Global Synchronization

- If we could synchronize across all thread blocks, could easily reduce very large arrays, right?
  - Global sync after each block produces its result
  - Once all blocks reach sync, continue recursively
- But CUDA has no global synchronization. Why?
  - Expensive to build in hardware for GPUs with high processor count
  - Would force programmer to run fewer blocks (no more than # multiprocessors * # resident blocks / multiprocessor) to avoid deadlock, which may reduce overall efficiency
- Solution: decompose into multiple kernels
  - Kernel launch serves as a global synchronization point
  - Kernel launch has negligible HW overhead, low SW overhead

Solution: Kernel Decomposition

- Avoid global sync by decomposing computation into multiple kernel invocations



- In the case of reductions, code for all levels is the same
  - Recursive kernel invocation

Performance for 4M element reduction

	Time (2^{22} ints)	Bandwidth	Step Speedup	Cumulative Speedup
Kernel 1: interleaved addressing with divergent branching	8.054 ms	2.083 GB/s		
Kernel 2: interleaved addressing with bank conflicts	3.456 ms	4.854 GB/s	2.33x	2.33x
Kernel 3: sequential addressing	1.722 ms	9.741 GB/s	2.01x	4.68x
Kernel 4: first add during global load	0.965 ms	17.377 GB/s	1.78x	8.34x
Kernel 5: unroll last warp	0.536 ms	31.289 GB/s	1.8x	15.01x
Kernel 6: completely unrolled	0.381 ms	43.996 GB/s	1.41x	21.16x
Kernel 7: multiple elements per thread	0.268 ms	62.671 GB/s	1.42x	30.04x

Kernel 7 on 32M elements: 73 GB/s!

And More...

1. On Volta and newer (Ampere, ...),
   reduction in shared memory must use
   warp synchronization! `__syncwarp()` or Cooperative Groups

2. Last optimization step for parallel reduction:

Do not use shared memory for last 5 steps, but use

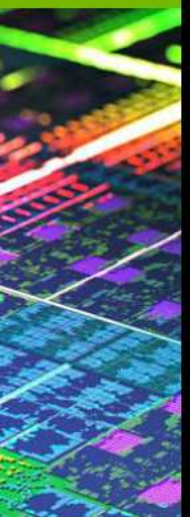
warp shuffle instructions

EXAMPLE: REDUCTION VIA SHARED MEMORY

__syncwarp

Re-converge threads and perform memory fence

```
v += shmem[tid+16]; __syncwarp();
shmem[tid] = v;     __syncwarp();
v += shmem[tid+8];  __syncwarp();
shmem[tid] = v;     __syncwarp();
v += shmem[tid+4];  __syncwarp();
shmem[tid] = v;     __syncwarp();
v += shmem[tid+2];  __syncwarp();
shmem[tid] = v;     __syncwarp();
v += shmem[tid+1];  __syncwarp();
shmem[tid] = v;
```

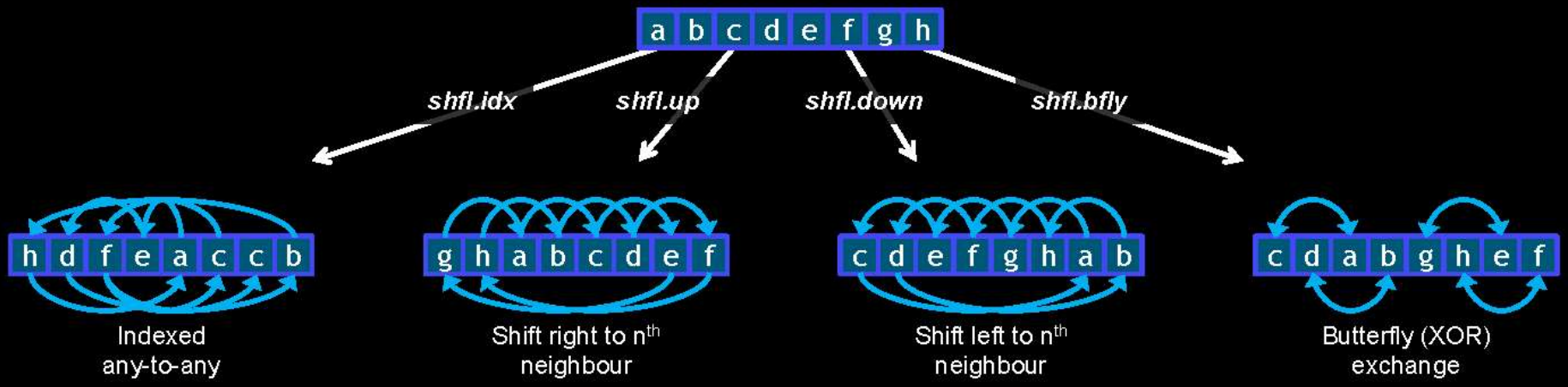


Shuffle (SHFL)

- Instruction to exchange data in a warp
- Threads can “read” other threads’ registers
- No shared memory is needed
- It is available starting from SM 3.0

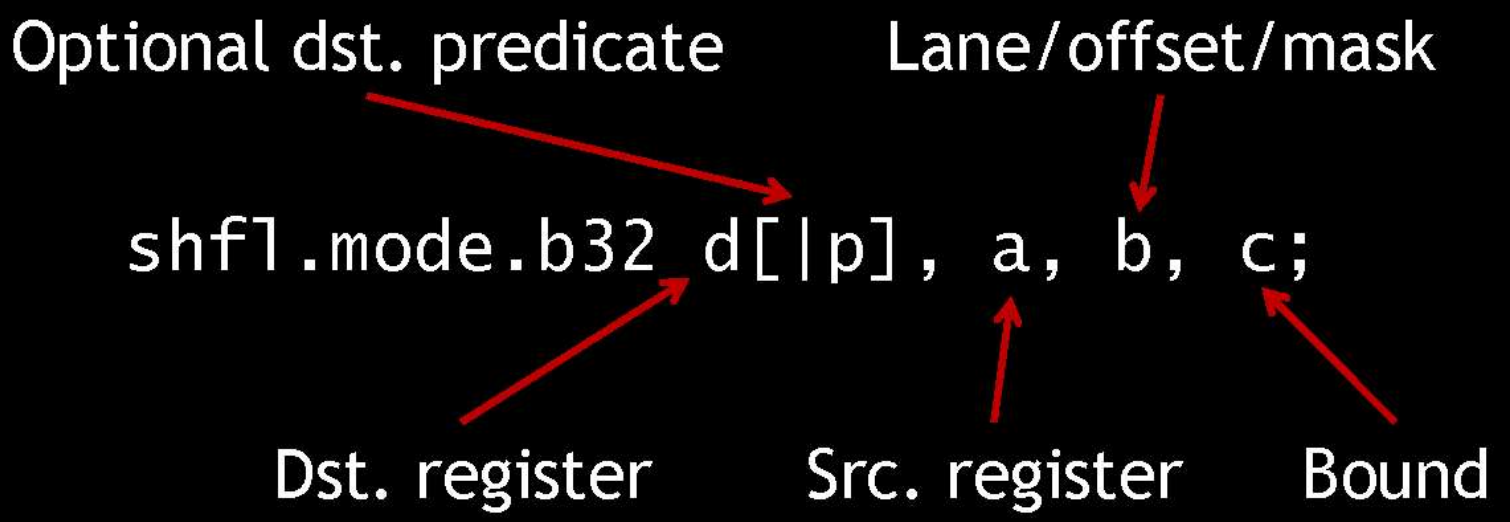
Variants

- 4 variants (idx, up, down, bfly):



Now: Use _sync variants / shuffle in cooperative thread groups!

Instruction (PTX)



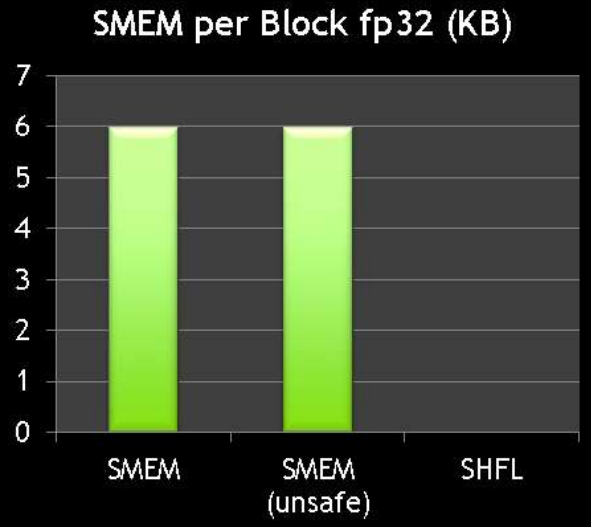
Now: Use _sync variants / shuffle in cooperative thread groups!

Reduce

- Code

```
// Threads want to reduce the value in x.

float x = …;

#pragma unroll
for(int mask = WARP_SIZE / 2 ; mask > 0 ; mask >>= 1)
    x += __shfl_xor(x, mask);

// The x variable of laneid 0 contains the reduction.
```

- Performance
  - Launch 26 blocks of 1024 threads
  - Run the reduction 4096 times



Thank you.