

CS 380 - GPU and GPGPU Programming

Lecture 25: CUDA Memory, Pt. 5; Shuffle Instructions; Cooperative Groups

Markus Hadwiger, KAUST

Reading Assignment #15++



Further suggested reading:

- Raihan et al., arXiv, Feb 2019, Modeling Deep Learning Accelerator Enabled GPUs
 - <https://arxiv.org/abs/1811.08309>
 - See also GPGPU-SIM: <http://www.gpgpu-sim.org/>
- CUTLASS 2.8 template library (last update Nov 2021)
 - <https://developer.nvidia.com/blog/cutlass-linear-algebra-cuda/>
 - <https://github.com/NVIDIA/cutlass>
- Register Cache: Caching for Warp-Centric CUDA Programs
 - <https://developer.nvidia.com/blog/register-cache-warp-cuda/>
- cuSPARSE library description in the CUDA SDK
- CUSP library: <http://cusplibrary.github.io/>
- Incomplete-LU and Cholesky Preconditioned Iterative Methods Using CUSPARSE and CUBLAS, Maxim Naumov
 - https://developer.download.nvidia.com/assets/cuda/files/psts_white_paper_final.pdf



Global Memory Accesses

- Memory coalescing
- Cached memory access



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

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.



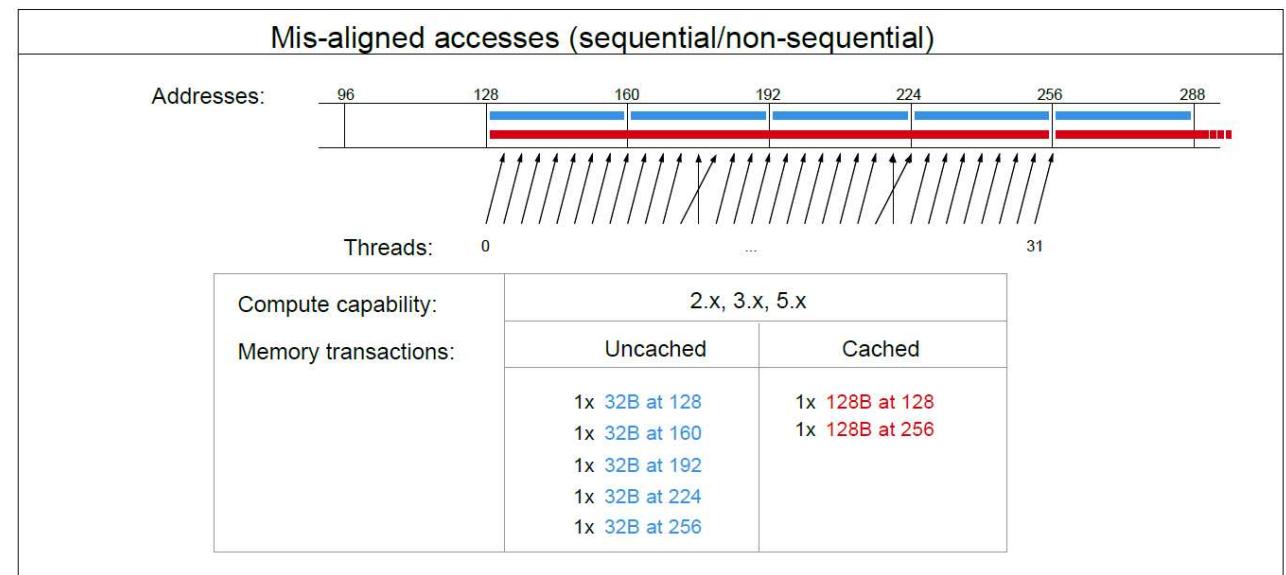
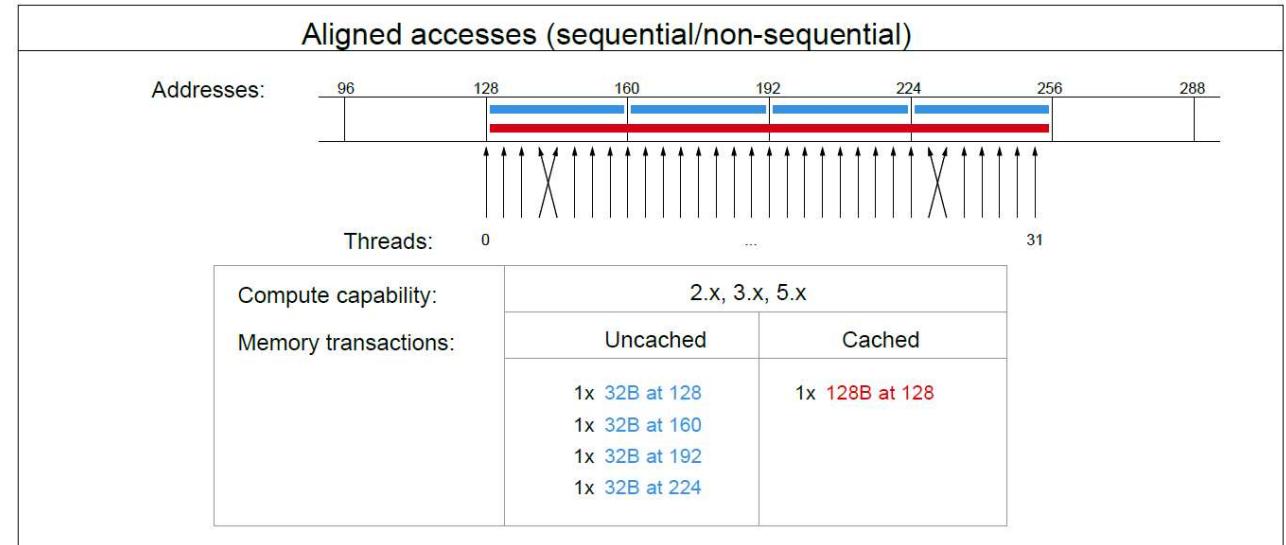
Global Memory Access

all recent
compute capabilities
(- 8.x)

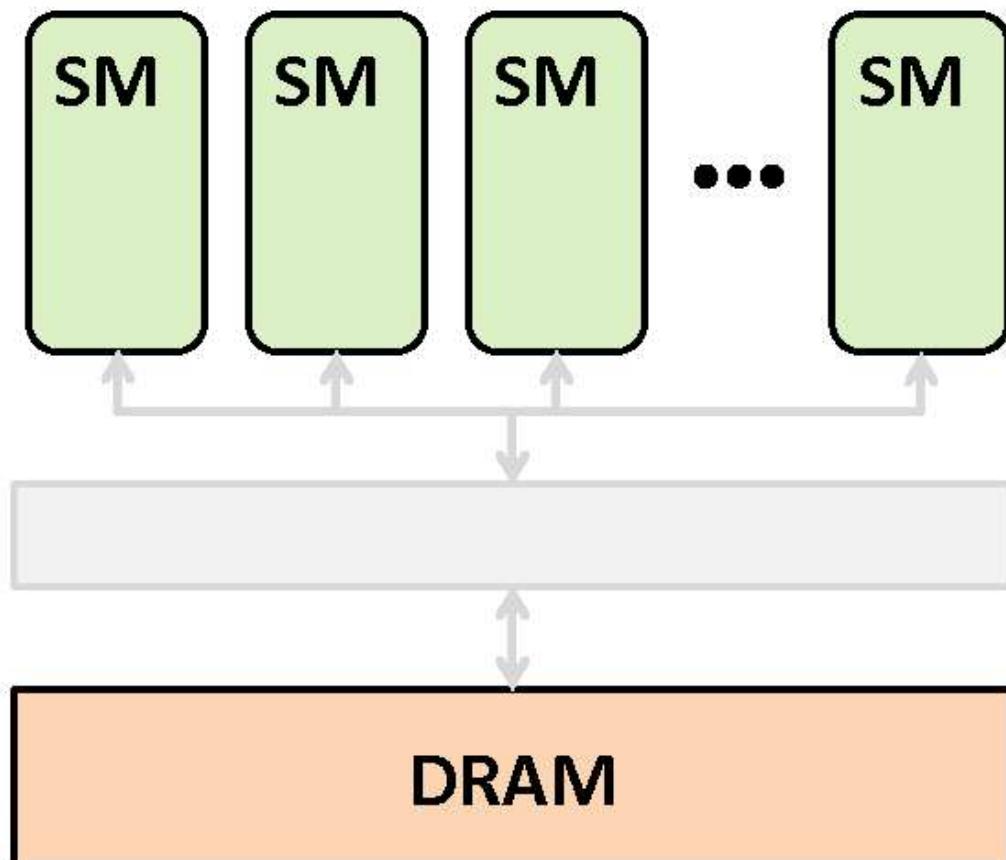
Beware:

*Uncached here means
not cached in L1*

*the L2 cache is
always used!*

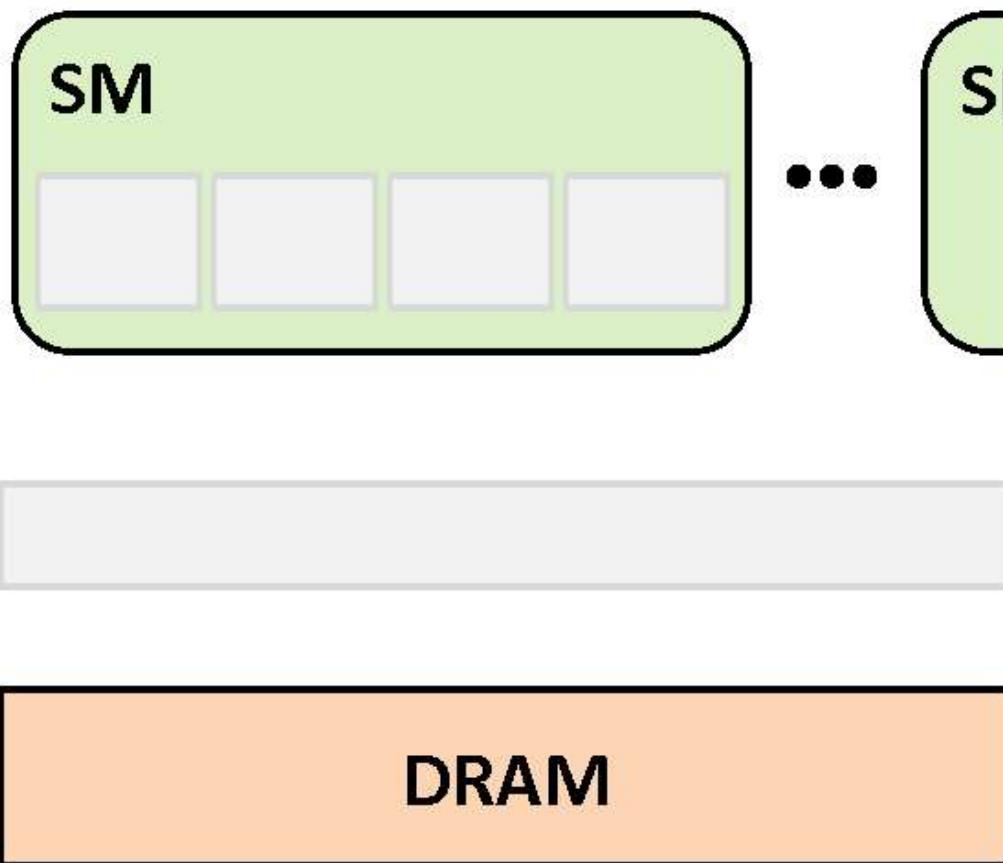


Maximize Byte Use



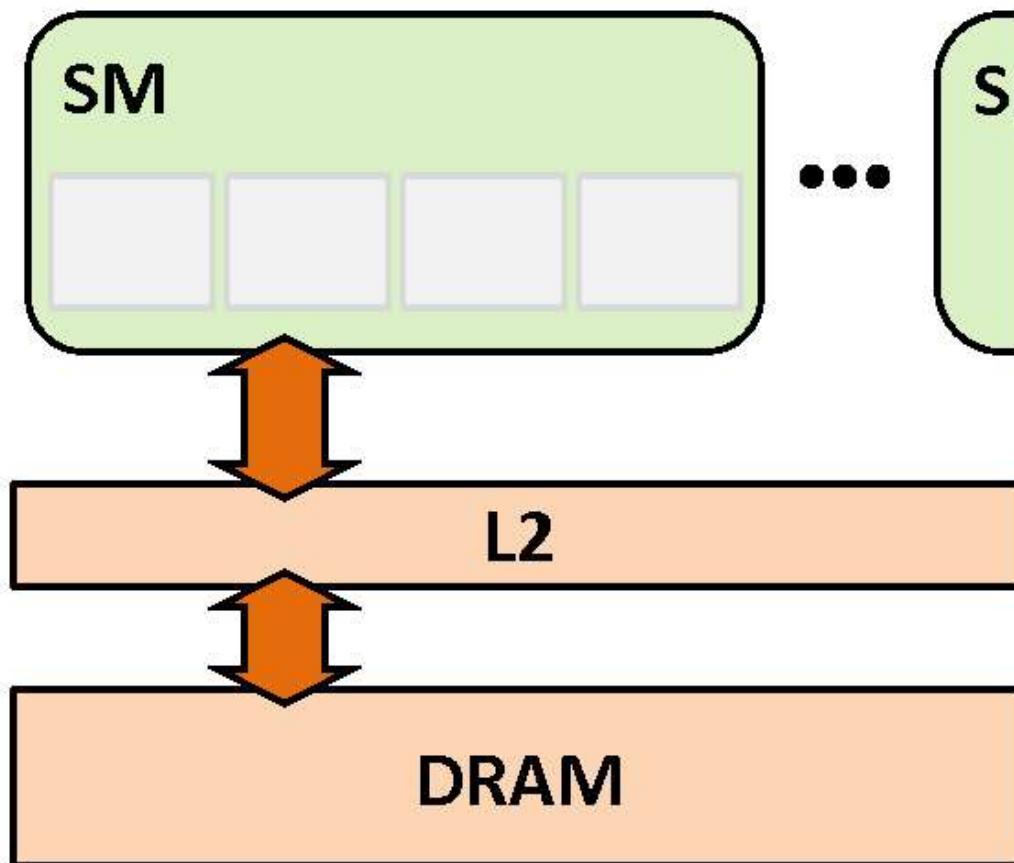
- Two things to keep in mind:
 - Memory accesses are per warp
 - Memory is accessed in discrete chunks
 - lines/segments
 - want to make sure that bytes that travel from DRAM to SMs get used
 - For that we should understand how memory system works
- Note: not that different from CPUs
 - x86 needs SSE/AVX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

GPU Memory System



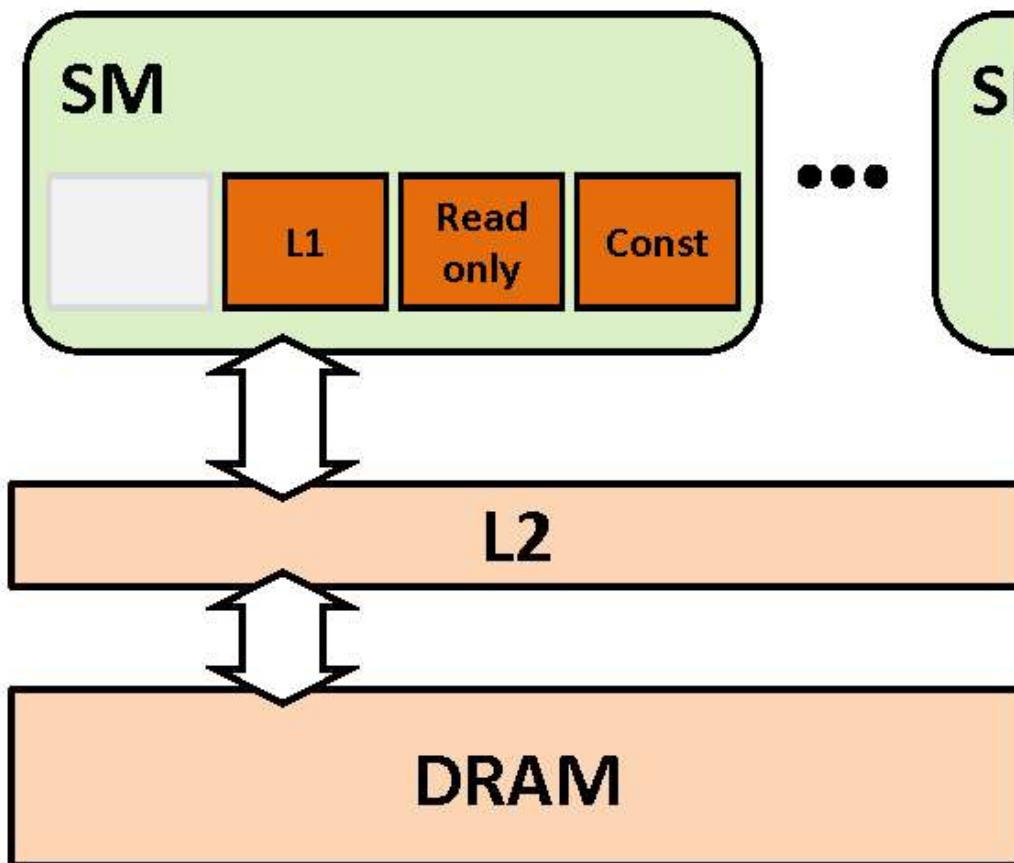
- All data lives in DRAM
 - Global memory
 - Local memory
 - Textures
 - Constants

GPU Memory System



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

GPU Memory System

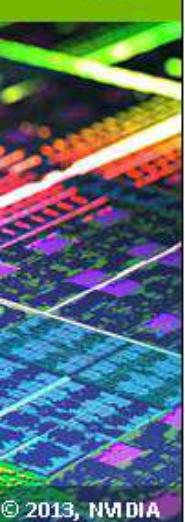


- Once in an SM, data goes into one of 3 caches/buffers
- Programmer's choice
 - L1 is the “default”
 - Read-only, Const require explicit code



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



Compute Capab. 3.x (Kepler, Part 4)

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.

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.

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.



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



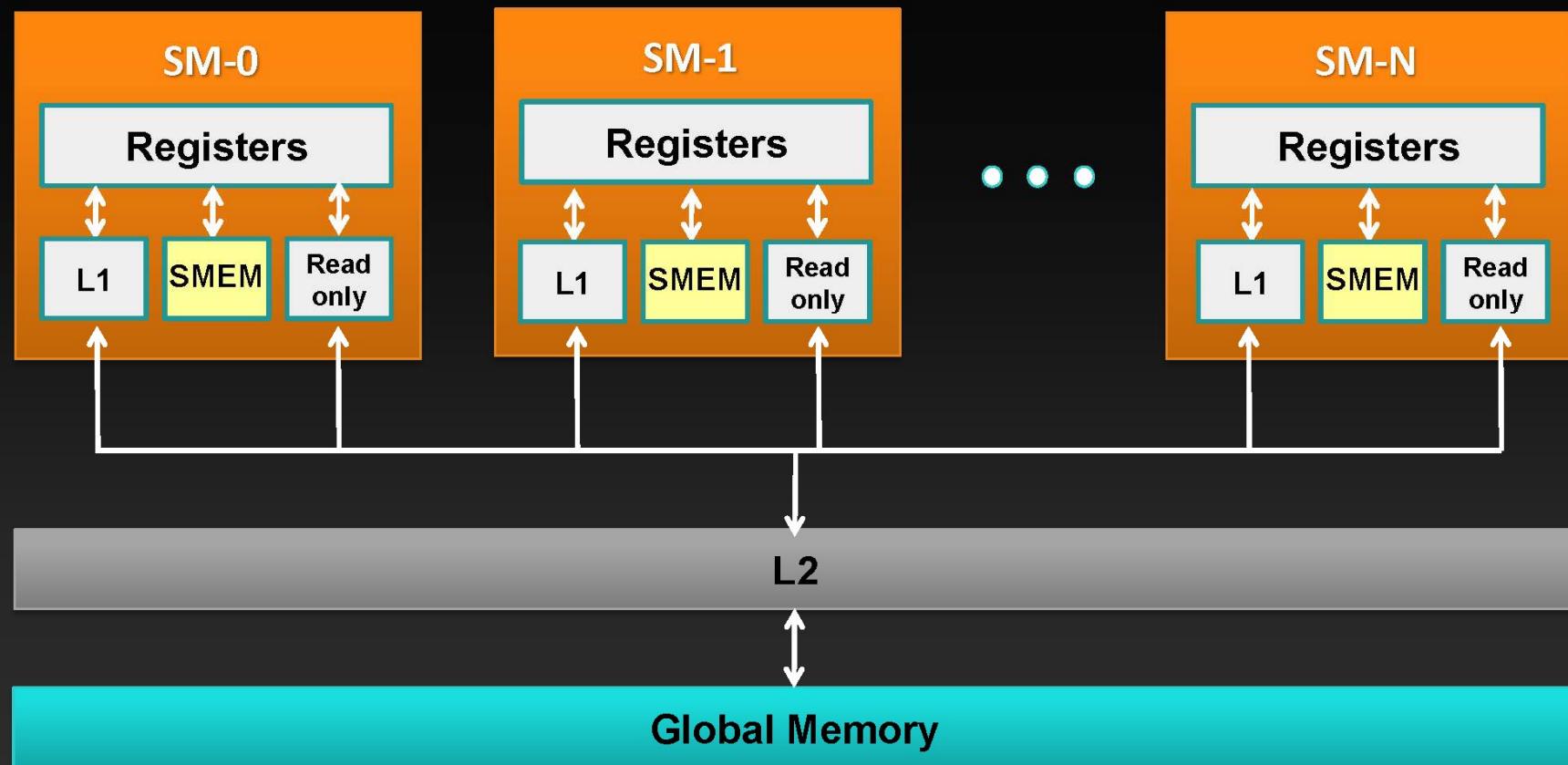
GMEM Writes

- **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

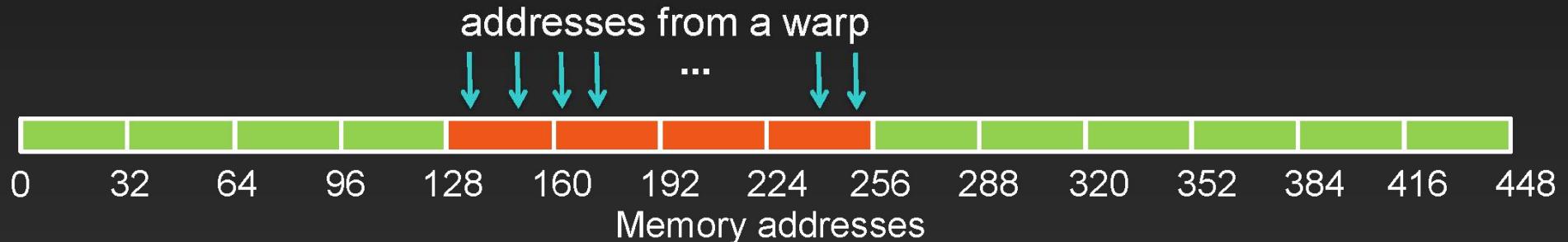
- Two perspectives on the throughput:
 - Application's point of view:
 - count only bytes requested by application
 - HW point of view:
 - count all bytes moved by hardware
- The two views can be different:
 - Memory is accessed at 32 byte granularity
 - Scattered/offset pattern: application doesn't 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%



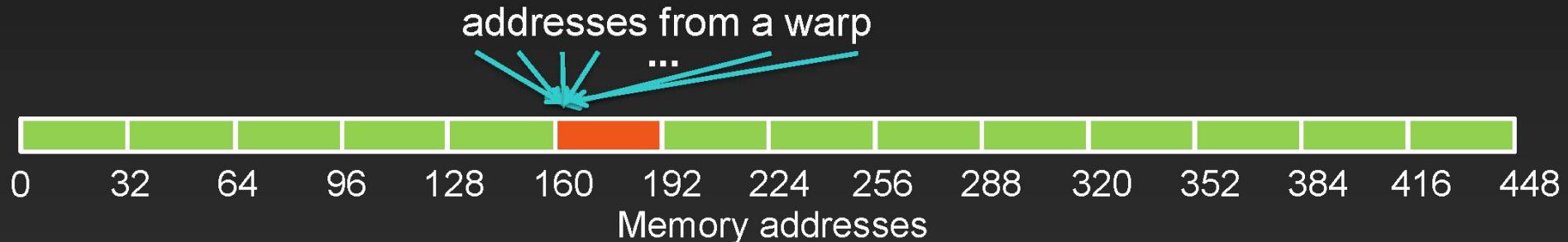
Access Patterns vs. Memory Throughput

- 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



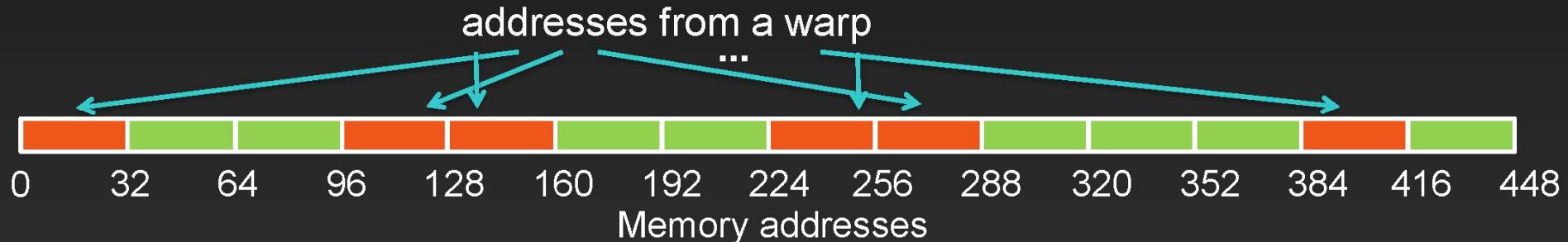
Access Patterns vs. Memory Throughput

- **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
 - $N \times 32$ bytes move across the bus
 - Bus utilization: $128 / (N \times 32)$



Structures of Non-Native Size

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

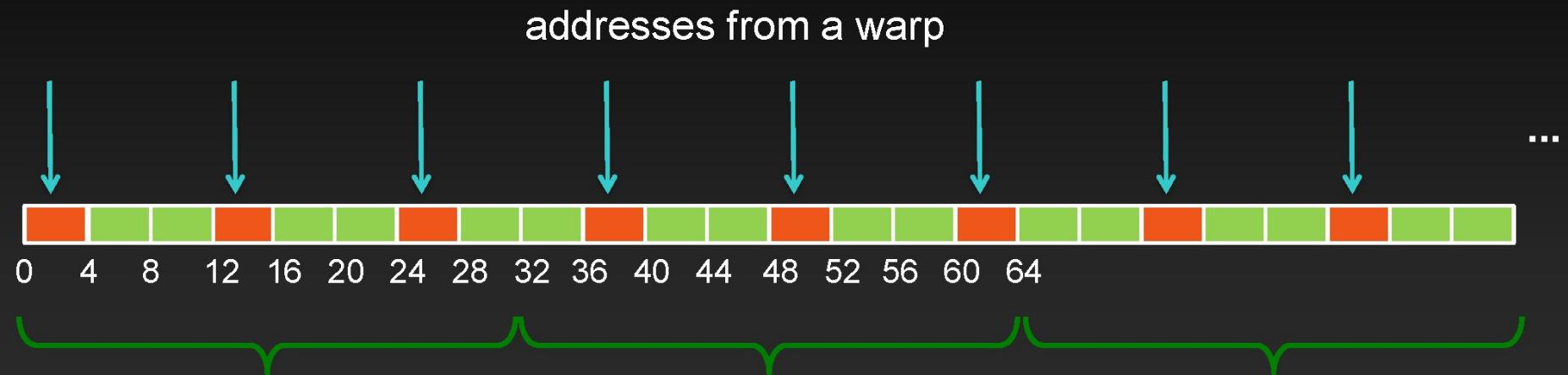
```
struct Position
{
    float x, y, z;
};

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

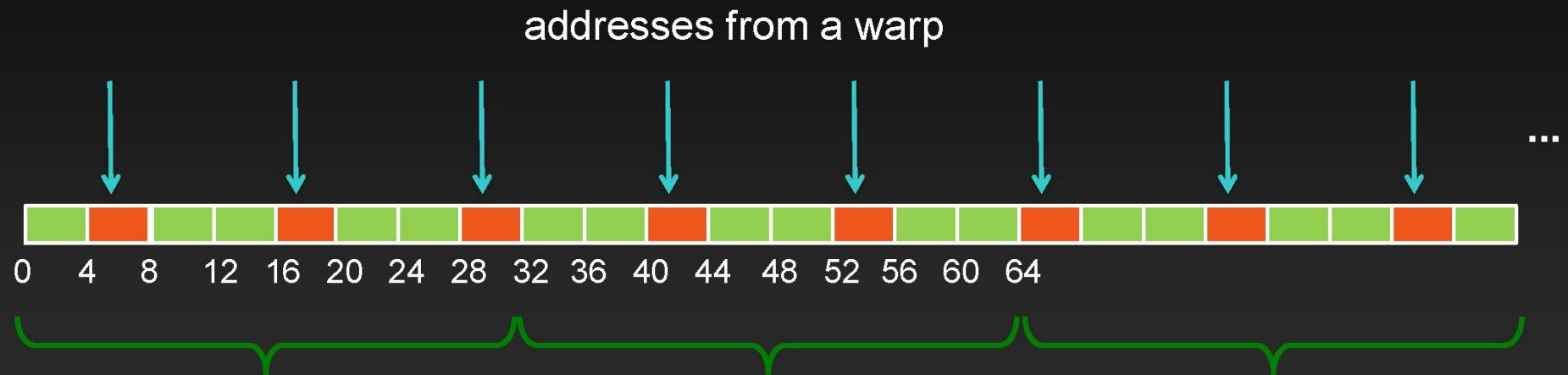
Structure of Non-Native Size

- 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

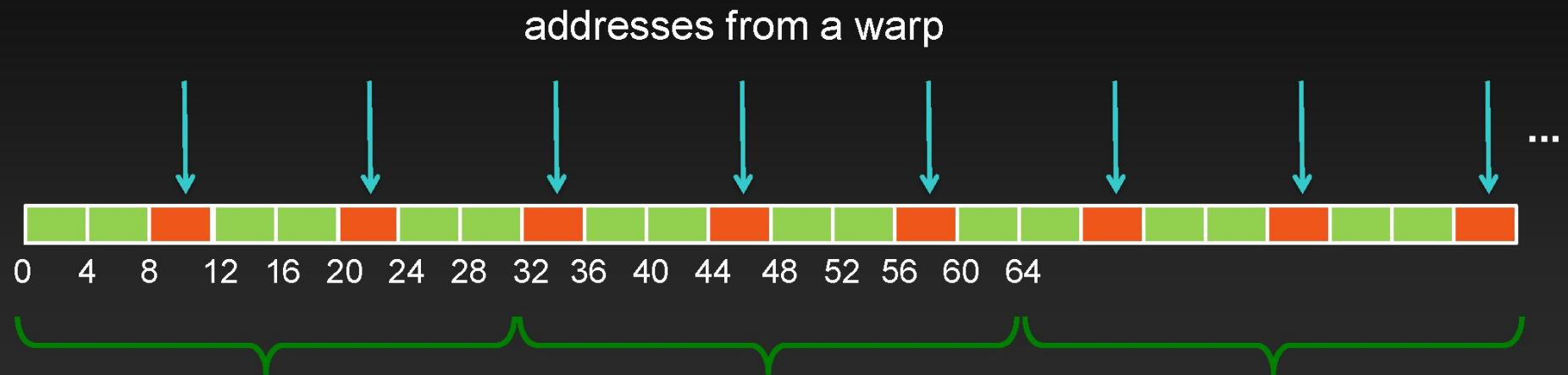
First Load Instruction



Second Load Instruction



Third Load Instruction



Performance and Solutions

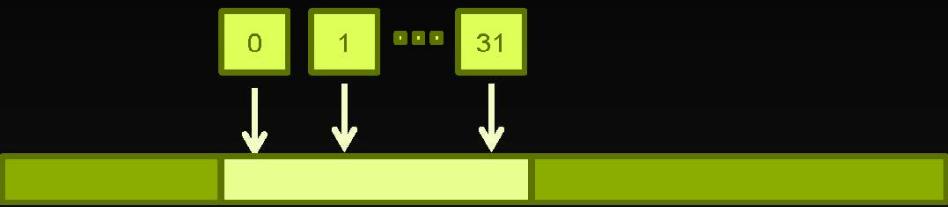
- Because of the address pattern, we end up moving 3x more bytes than application requests
 - We waste a lot of bandwidth, leaving performance on the table
- Potential solutions:
 - Change data layout from array of structures to structure of arrays
 - In this case: 3 separate arrays of floats
 - The 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

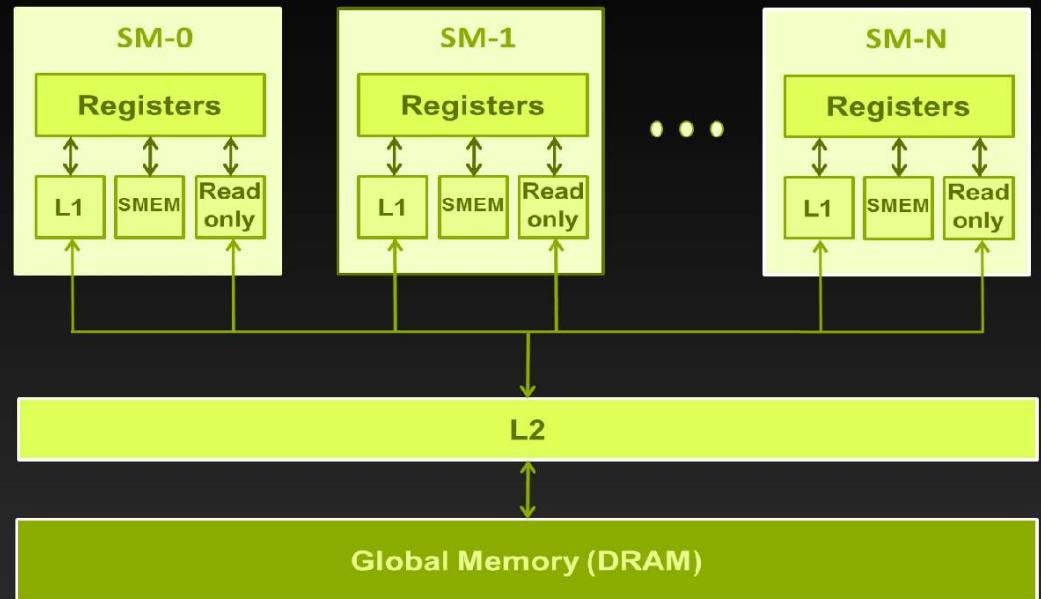
A note about caches

- **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 arguments
 - Pointer of interest: `const`
 - All other pointer arguments
 - Conveys to compiler that they are not modified
 - Using `__ldg()` intrinsic
 - Requires no pointer decoration

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

Read-only Data Cache

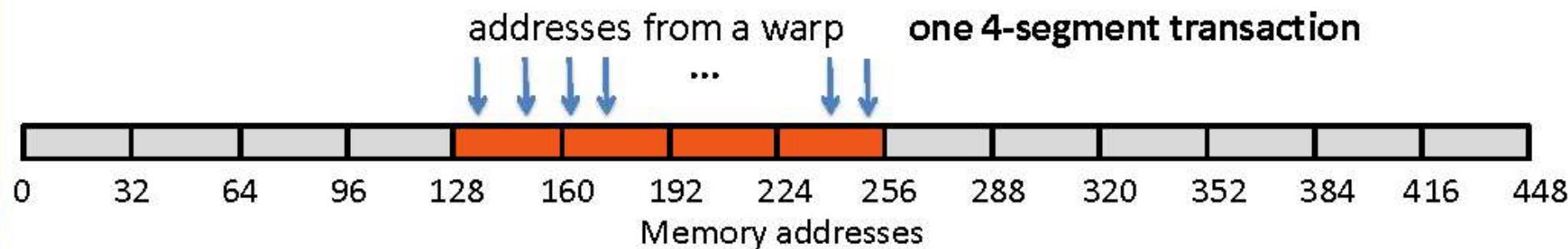
- Go through the read-only cache
 - Not coherent with writes
 - Thus, addresses must not be written by the same kernel
- Two ways to enable:
 - Decorating pointer arguments
 - Pointer of interest: `const`
 - All other pointer arguments
 - Conveys to compiler that they are read-only
 - Using `__ldg()`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

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

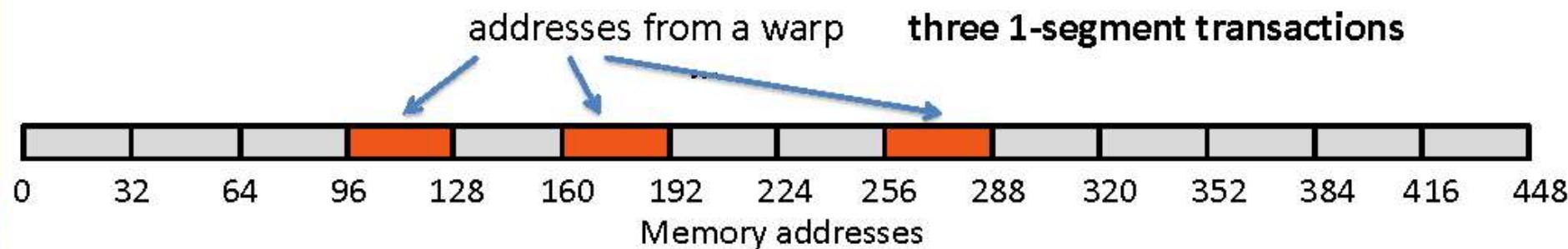
Blocking for L1, Read-only, L2 Caches

- 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

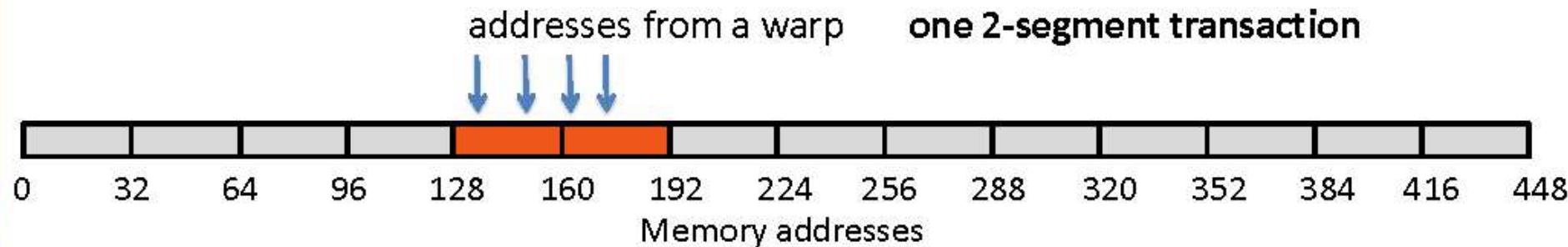
Some Store Pattern Examples



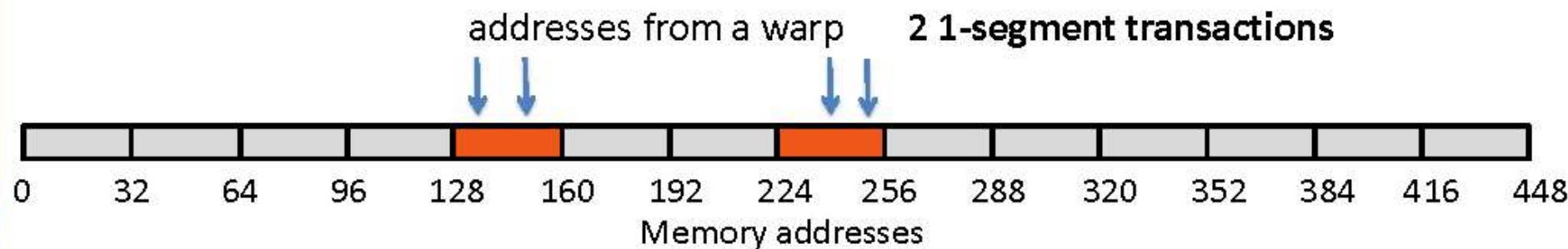
Some Store Pattern Examples



Some Store Pattern Examples



Some Store Pattern Examples

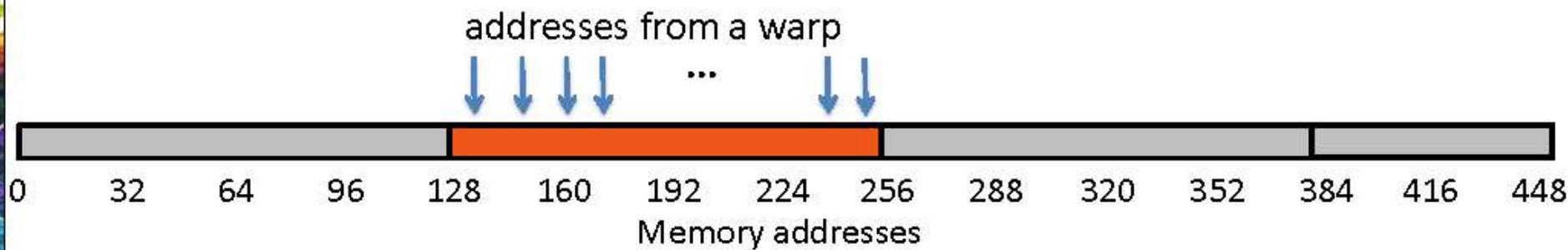


GMEM Reads

- 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)

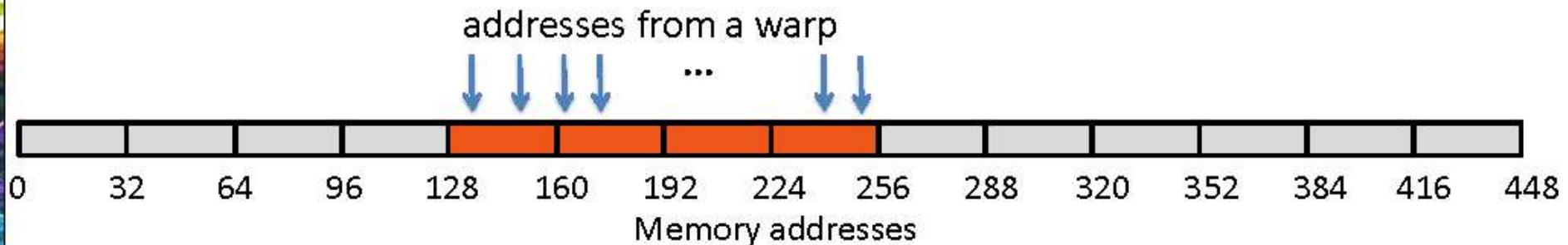
Caching Load

- **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



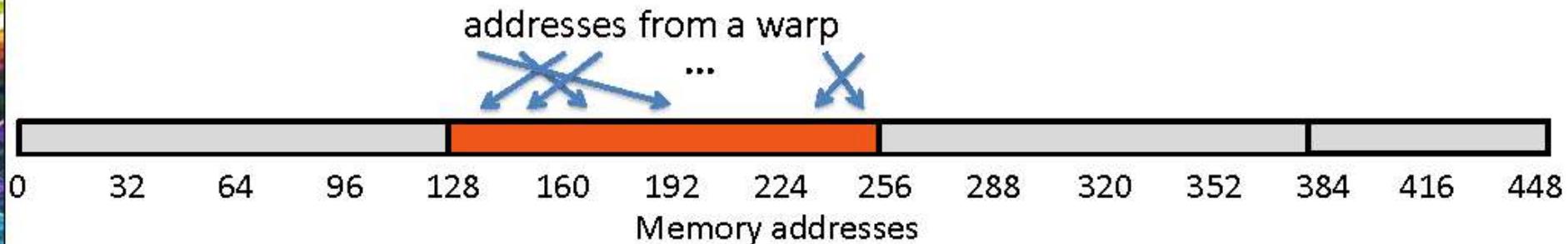
Non-caching Load

- **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



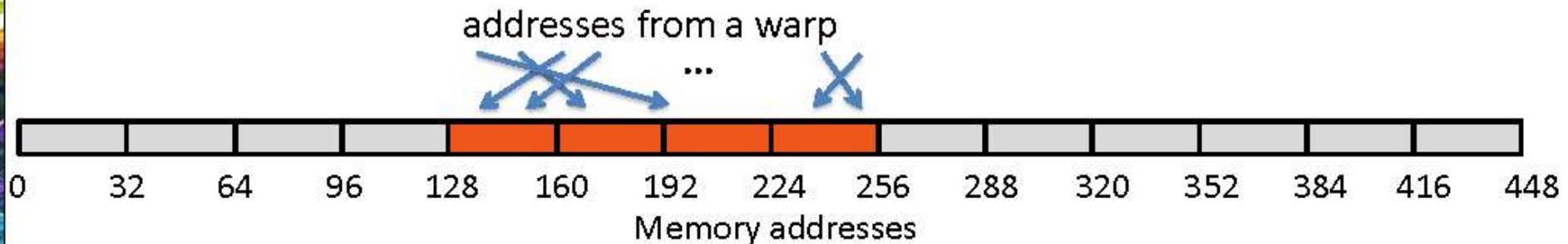
Caching Load

- **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



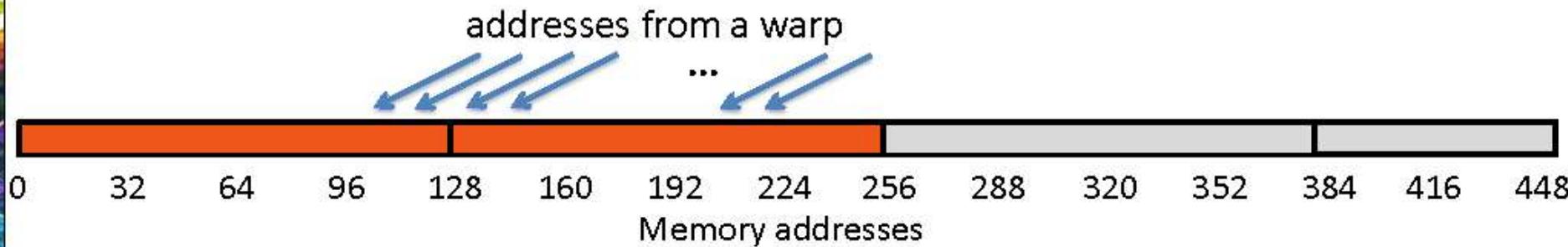
Non-caching Load

- **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



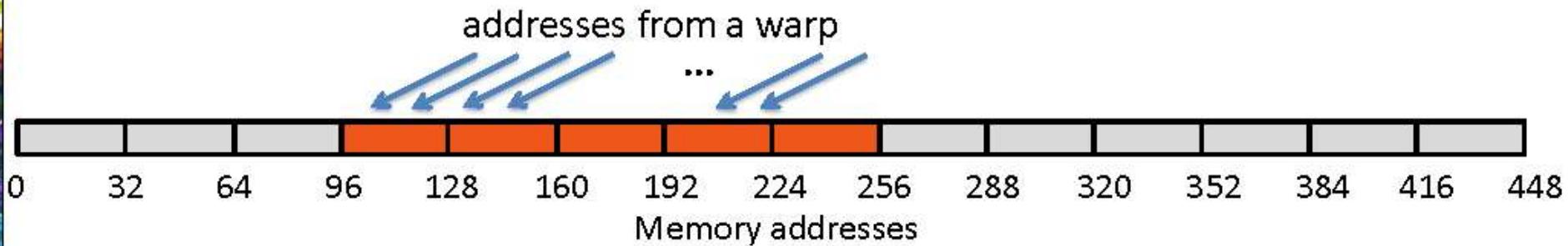
Caching Load

- **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



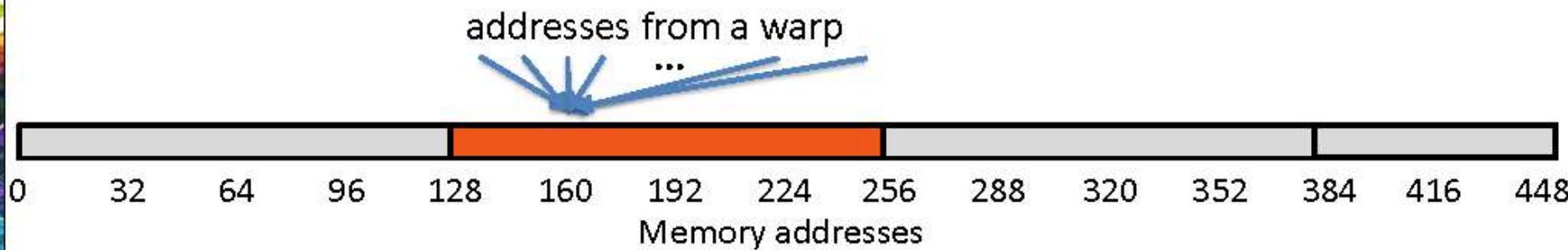
Non-caching Load

- **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



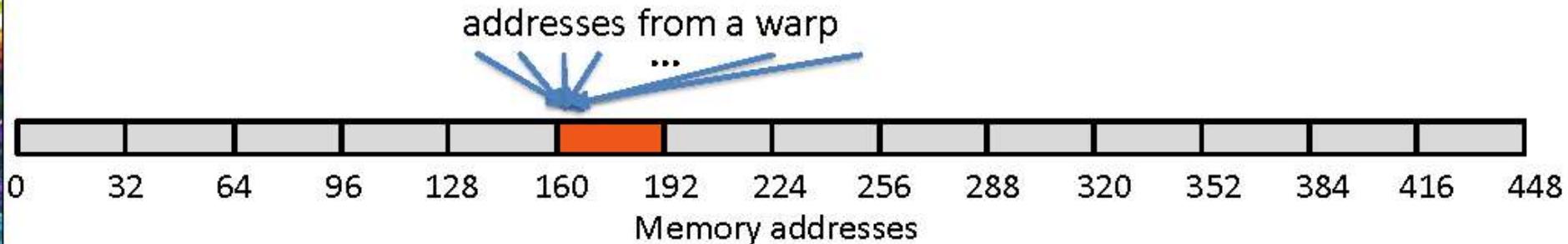
Caching Load

- **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



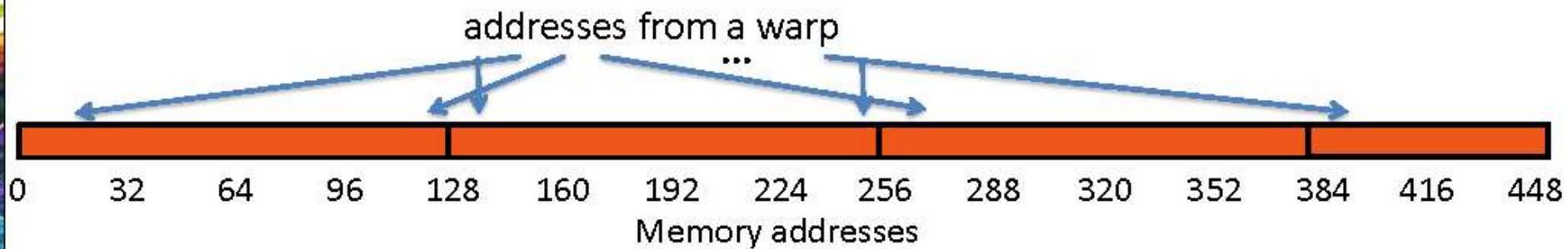
Non-caching Load

- **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



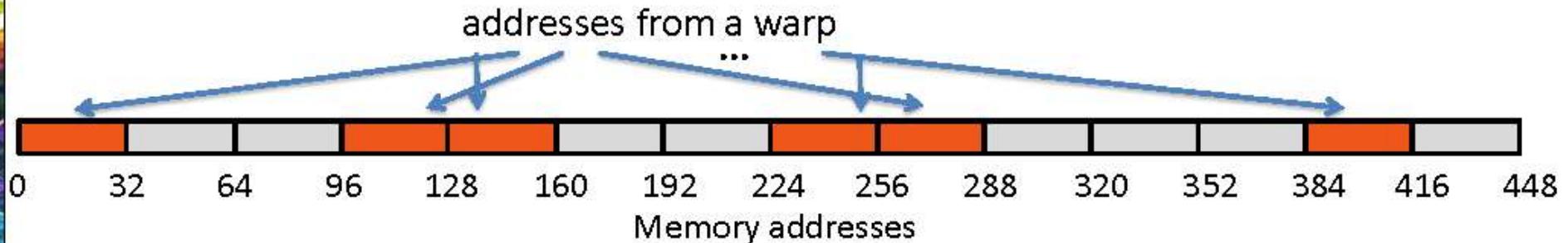
Caching Load

- **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

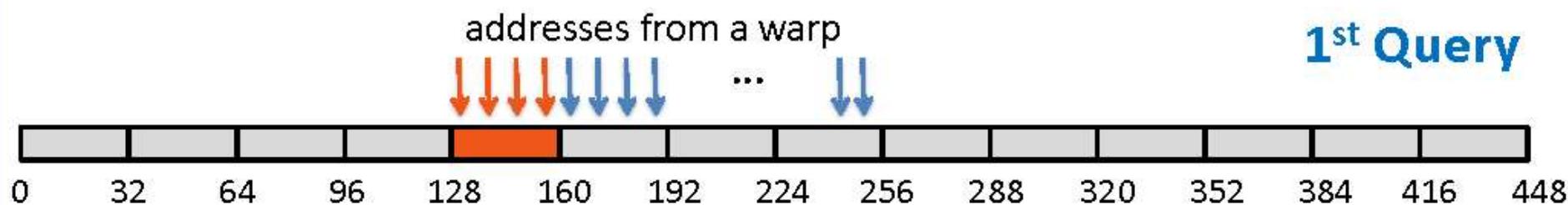
- **An alternative to L1 when accessing DRAM**
 - Also known as *texture* cache: all texture accesses use this cache
 - CC 3.5 and higher also enable global memory accesses
 - Should not be used if a kernel reads and writes to the same addresses
- **Comparing to L1:**
 - Generally better for scattered 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



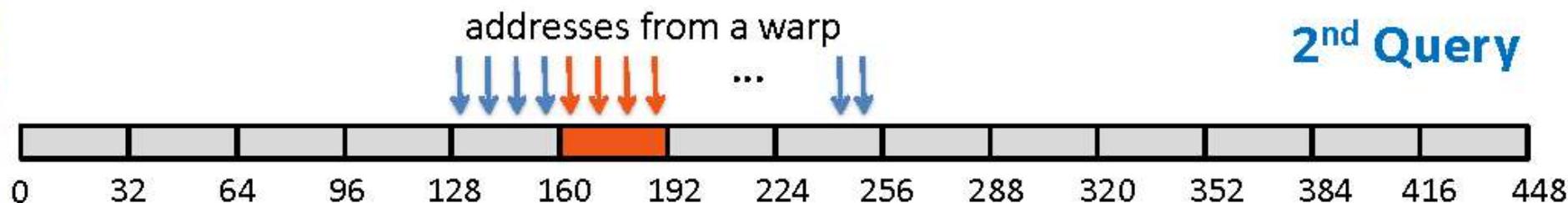
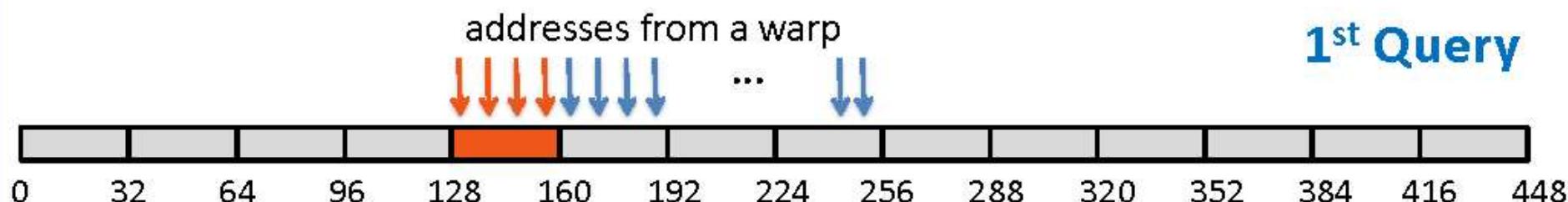
Read-Only Cache Operation

- **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

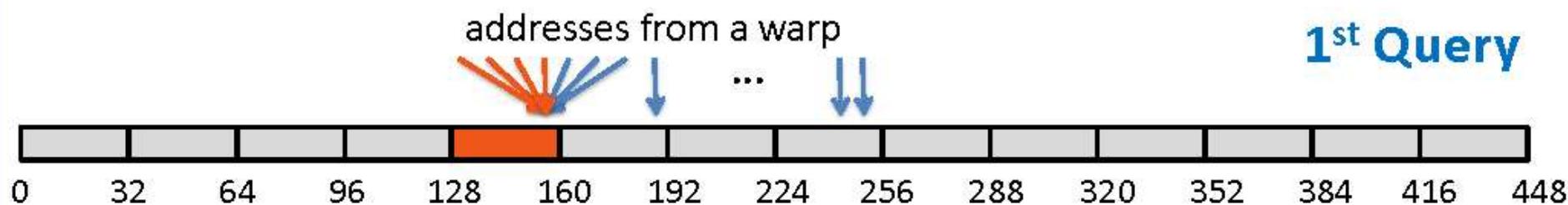
Read-Only Cache Operation



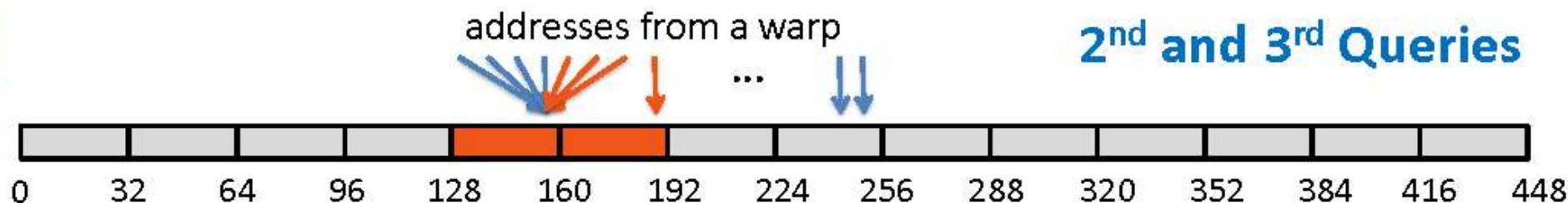
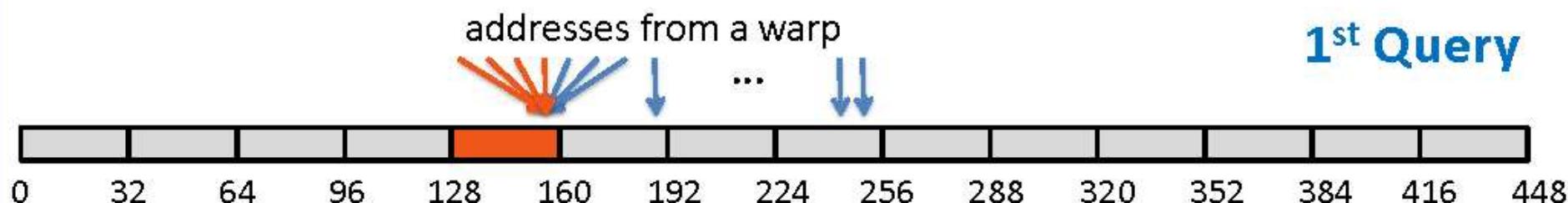
Read-Only Cache Operation



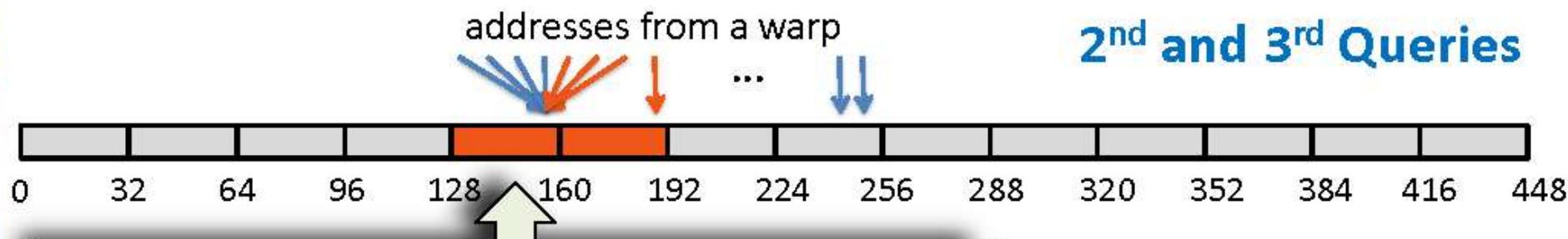
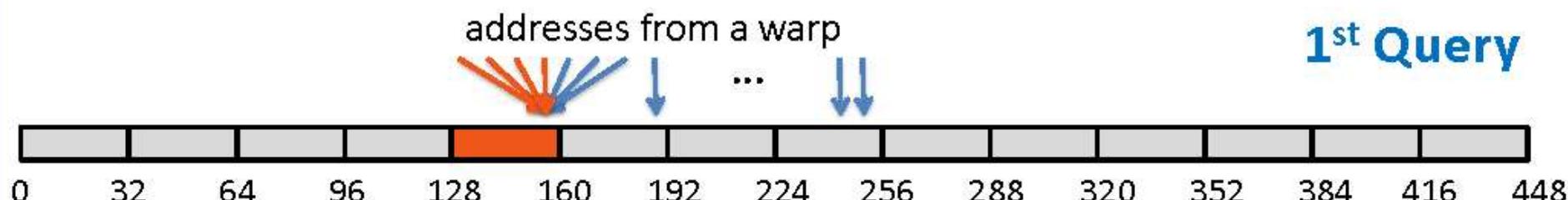
Read-Only Cache Operation



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



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 <code>ld.ca</code> , 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 <code>ld.ca</code> loads cached in L1.
.cg	Cache at global level (cache in L2 and below, not L1). Use <code>ld.cg</code> 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 <code>ld.cs</code> 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 <code>ld.cs</code> is applied to a Local window address, it performs the <code>ld.lu</code> operation.
.lu	Last use. The compiler/programmer may use <code>ld.lu</code> when restoring spilled registers and popping function stack frames to avoid needless write-backs of lines that will not be used again. The <code>ld.lu</code> instruction performs a load cached streaming operation (<code>ld.cs</code>) on global addresses.
.cv	Don't cache and fetch again (consider cached system memory lines stale, fetch again). The <code>ld.cv</code> 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.)



GPU TECHNOLOGY
CONFERENCE

Shuffle: Tips and Tricks

Julien Demouth, NVIDIA

Glossary

Safer with cooperative thread groups!

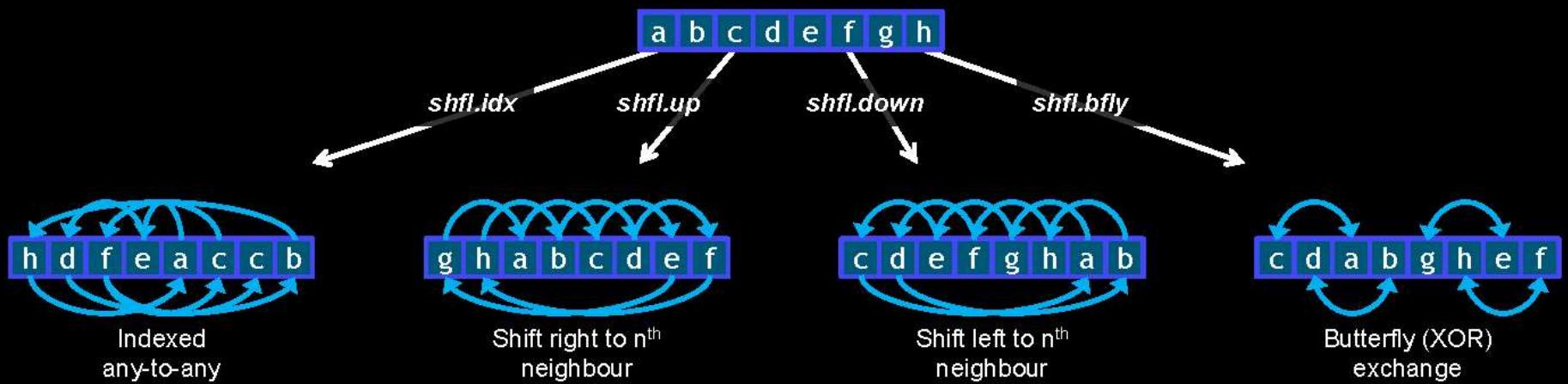
- Warp
 - ~~Implicitly synchronized~~ group of threads (32 on current HW)
- Warp ID (warpid)
 - Identifier of the warp in a block: `threadIdx.x / 32`
- Lane ID (laneid)
 - Coordinate of the thread in a warp: `threadIdx.x % 32`
 - Special register (available from PTX): `%laneid`

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)

Optional dst. predicate Lane/offset/mask
shfl.mode.b32 d[|p], a, b, c;
 ↑
 Dst. register Src. register Bound

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

Implement SHFL for 64b Numbers

```
__device__ __inline__ double shfl(double x, int lane)
{
    // split the double number into 2 32b registers.
    int lo, hi;
    asm volatile( "mov.b32 {%0,%1}, %2;" : "=r"(lo), "=r"(hi) : "d"(x));

    // Shuffle the two 32b registers.
    lo = __shfl(lo, lane);
    hi = __shfl(hi, lane);

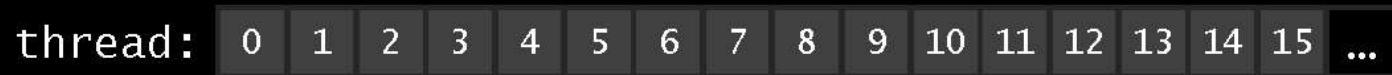
    // Recreate the 64b number.
    asm volatile( "mov.b64 %0, {%1,%2};" : "=d(x)" : "r"(lo), "r"(hi));

    return x;
}
```

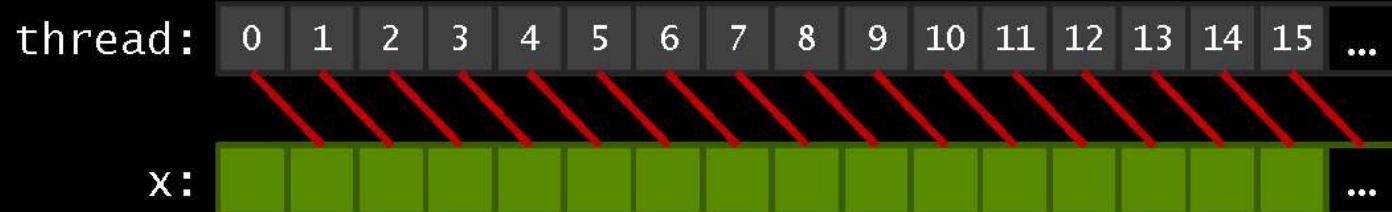
- Generic SHFL: <https://github.com/BryanCatanzaro/generics>

Performance Experiment

- One element per thread



- Each thread takes its right neighbor



Performance Experiment

- We run the following test on a K20

```
T x = input[tidx];
for(int i = 0 ; i < 4096 ; ++i)
    x = get_right_neighbor(x);
output[tidx] = x;
```

- We launch 26 blocks of 1024 threads
 - On K20, we have 13 SMs
 - We need 2048 threads per SM to have 100% of occupancy
- We time different variants of that kernel

Performance Experiment

- Shared memory (SMEM)

```
smem[threadIdx.x] = smem[32*warpid + ((laneid+1) % 32)];  
__syncthreads();
```

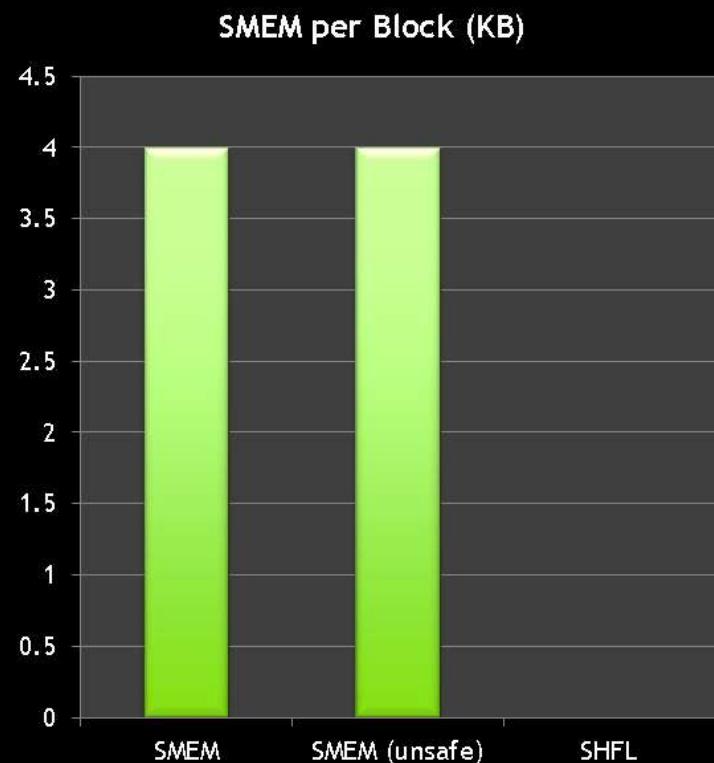
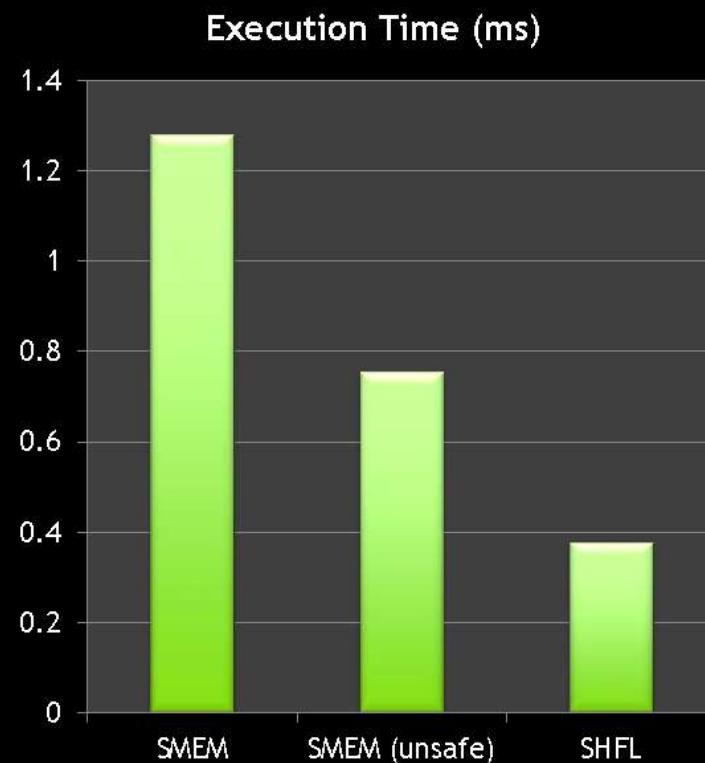
- Shuffle (SHFL)

```
x = __shfl(x, (laneid+1) % 32);
```

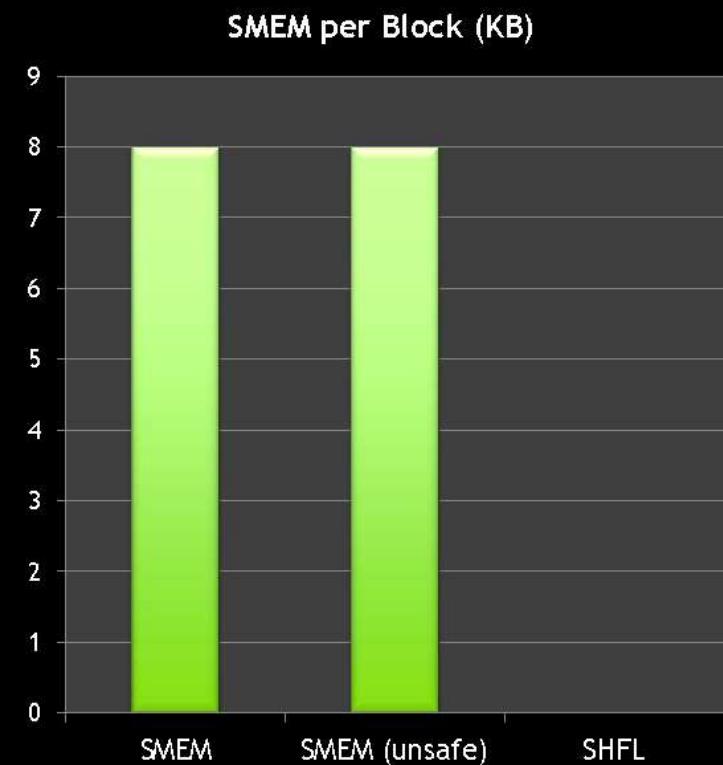
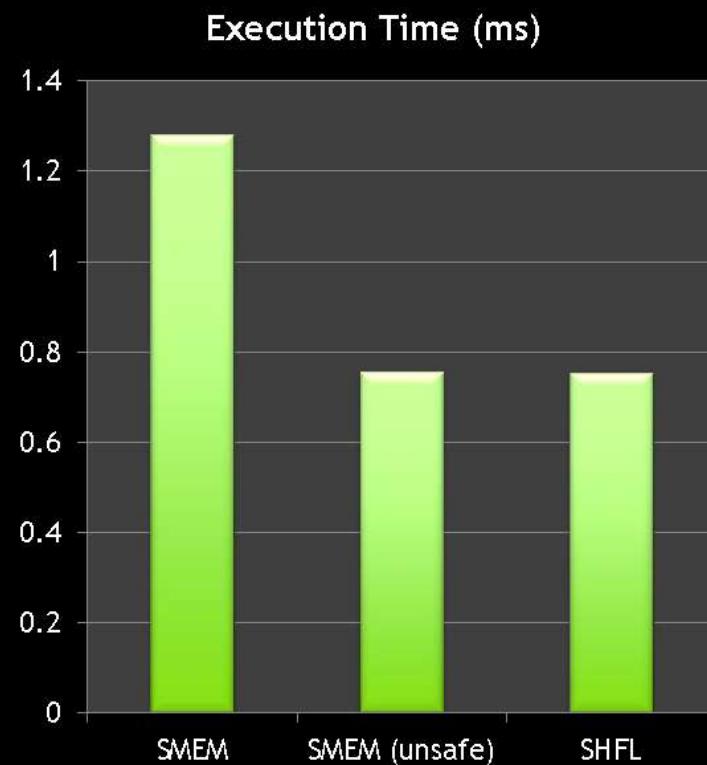
- Shared memory without `__syncthreads` + volatile (*unsafe*)

```
__shared__ volatile T *smem = ...;  
smem[threadIdx.x] = smem[32*warpid + ((laneid+1) % 32)];
```

Performance Experiment (fp32)



Performance Experiment (fp64)



Performance Experiment

- Always faster than shared memory
- Much safer than using no `__syncthreads` (and volatile)
 - And never slower
- Does not require shared memory
 - Useful when occupancy is limited by SMEM usage

Broadcast

Now: Use cooperative thread groups!

- All threads read from a single lane

```
x = __shfl(x, 0); // All the threads read x from laneid 0.
```

- More complex example

```
// All threads evaluate a predicate.  
int predicate = ...;  
  
// All threads vote.  
unsigned vote = __ballot(predicate);  
  
// All threads get x from the "last" lane which evaluated the predicate to true.  
if(vote)  
    x = __shfl(x, __bfind(vote));  
  
// __bind(unsigned i): Find the most significant bit in a 32/64 number (PTX).  
__bfind(&b, i) { asm volatile("bfind.u32 %0, %1;" : "=r"(b) : "r"(i)); }
```

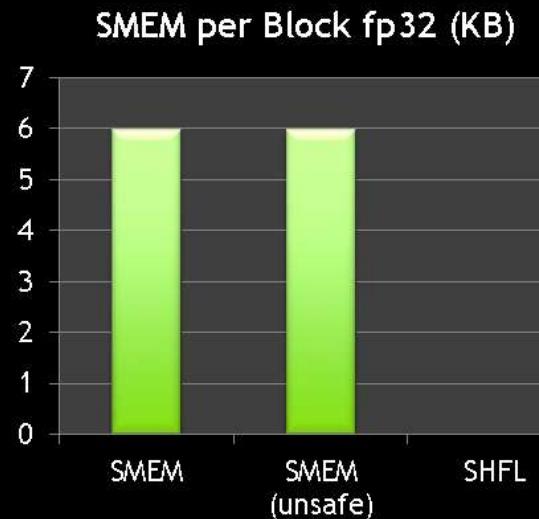
Reduce

■ 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



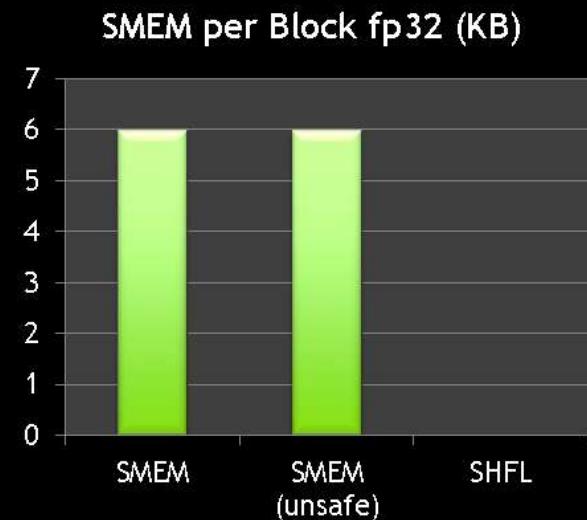
Scan

- Code

```
#pragma unroll
for( int offset = 1 ; offset < 32 ; offset <= 1 )
{
    float y = __shfl_up(x, offset);
    if(laneid() >= offset)
        x += y;
}
```

- Performance

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



Scan

- Use the predicate from SHFL

```
#pragma unroll
for( int offset = 1 ; offset < 32 ; offset <= 1 )
{
    asm volatile( "{"
        "    .reg .f32 r0;" 
        "    .reg .pred p;" 
        "    shfl.up.b32 r0 |p, %0, %1, 0x0;" 
        "    @p add.f32 r0, r0, %0;" 
        "    mov.f32 %0, r0;" 
        "}" : "+f"(x) : "r"(offset));
}
```

- Use CUB:
<https://nvlabs.github.com/cub>

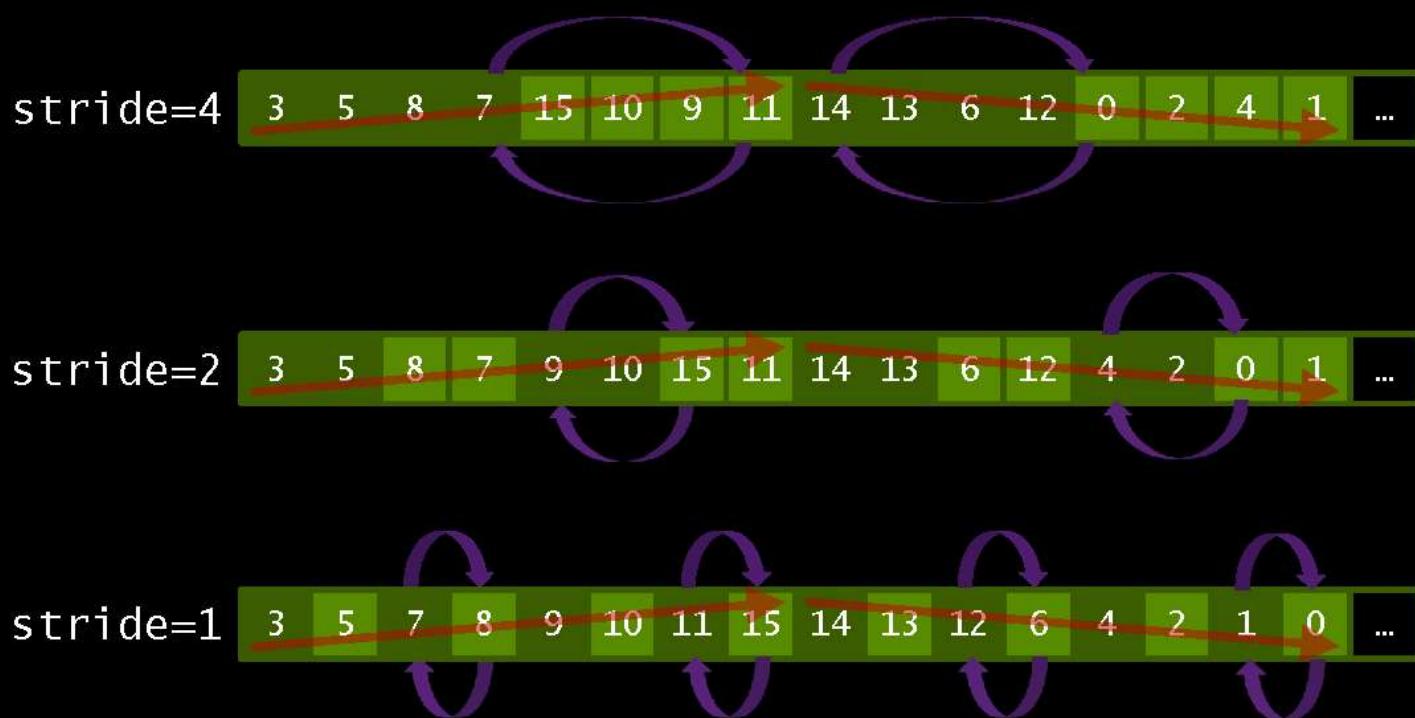


Bitonic Sort

x: 11 3 8 5 10 15 9 7 12 4 2 0 14 13 6 1 ...



Bitonic Sort



Bitonic Sort

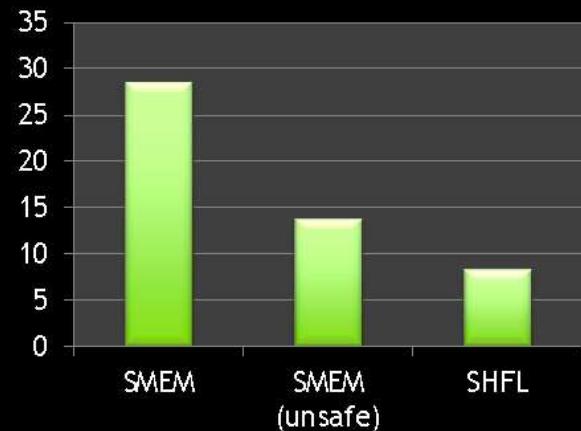
```
int swap(int x, int mask, int dir)
{
    int y = __shfl_xor(x, mask);
    return x < y == dir ? y : x;
}

x = swap(x, 0x01, bfe(laneid, 1) ^ bfe(laneid, 0)); // 2
x = swap(x, 0x02, bfe(laneid, 2) ^ bfe(laneid, 1)); // 4
x = swap(x, 0x01, bfe(laneid, 2) ^ bfe(laneid, 0));
x = swap(x, 0x04, bfe(laneid, 3) ^ bfe(laneid, 2)); // 8
x = swap(x, 0x02, bfe(laneid, 3) ^ bfe(laneid, 1));
x = swap(x, 0x01, bfe(laneid, 3) ^ bfe(laneid, 0));
x = swap(x, 0x08, bfe(laneid, 4) ^ bfe(laneid, 3)); // 16
x = swap(x, 0x04, bfe(laneid, 4) ^ bfe(laneid, 2));
x = swap(x, 0x02, bfe(laneid, 4) ^ bfe(laneid, 1));
x = swap(x, 0x01, bfe(laneid, 4) ^ bfe(laneid, 0));
x = swap(x, 0x10,
           bfe(laneid, 4)); // 32
x = swap(x, 0x08,
           bfe(laneid, 3));
x = swap(x, 0x04,
           bfe(laneid, 2));
x = swap(x, 0x02,
           bfe(laneid, 1));
x = swap(x, 0x01,
           bfe(laneid, 0));

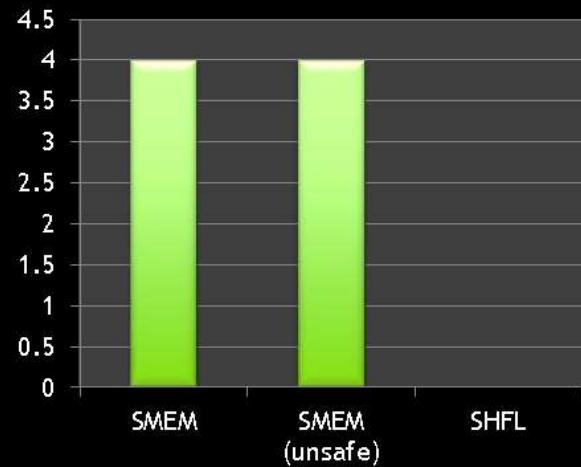
// int bfe(int i, int k): Extract k-th bit from i

// PTX: bfe dst, src, start, len (see p.81, ptx_isa_3.1)
```

Execution Time int32 (ms)

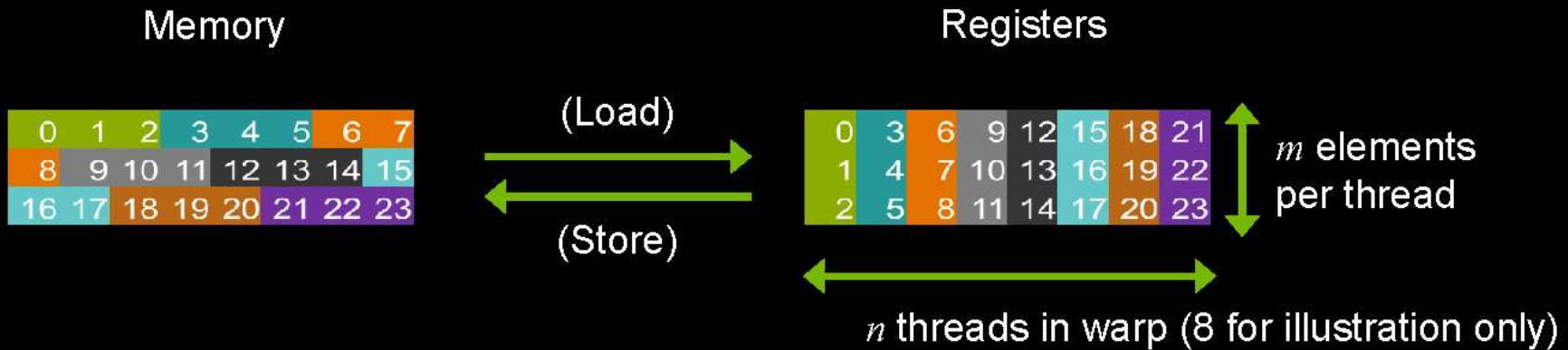


SMEM per Block (KB)



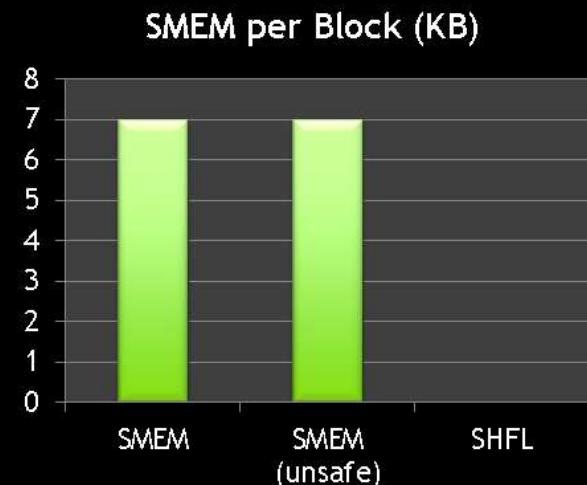
Transpose

- When threads load or store arrays of structures, transposes enable fully coalesced memory operations
- e.g. when loading, have the warp perform coalesced loads, then transpose to send the data to the appropriate thread



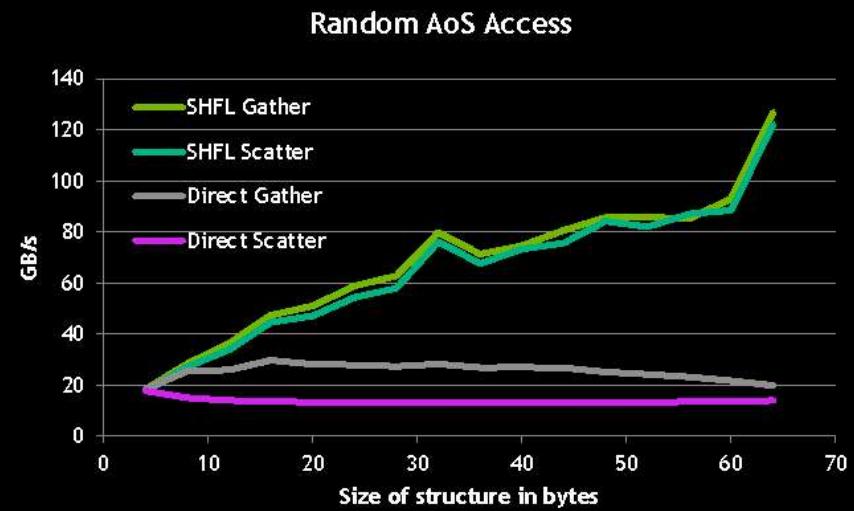
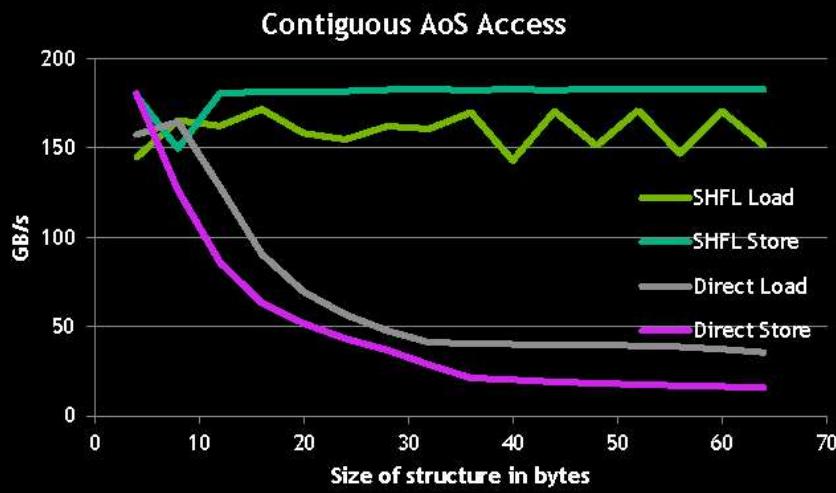
Transpose

- You can use SMEM to implement this transpose, or you can use SHFL
- Code:
<http://github.com/bryancatanzaro/trove>
- Performance
 - Launch 104 blocks of 256 threads
 - Run the transpose 4096 times



Array of Structures Access via Transpose

- Transpose speeds access to arrays of structures
- High-level interface: `coalesced_ptr<T>`
 - Just dereference like any pointer
 - Up to 6x faster than direct compiler generated access



Conclusion

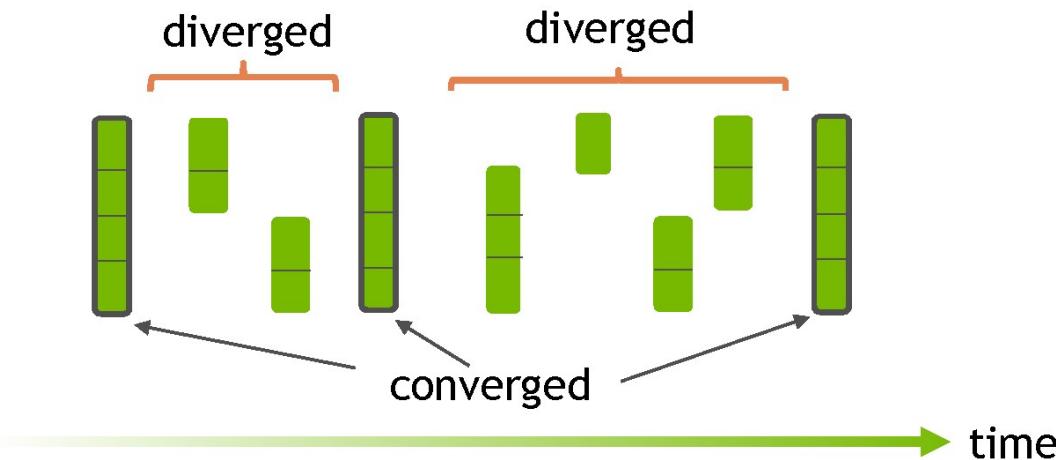
- SHFL is available for SM \geq SM 3.0
- It is always faster than “safe” shared memory
- It is never slower than “unsafe” shared memory
- It can be used in many different algorithms

WARP SYNCHRONOUS PROGRAMMING IN CUDA 9.0

CUDA WARP THREADING MODEL

NVIDIA GPU multiprocessors create, manage, schedule and execute threads in **warps** (32 parallel threads).

Threads in a warp may diverge and re-converge during execution.



Full efficiency may be realized when all 32 threads of a warp are converged.

WARP SYNCHRONOUS PROGRAMMING

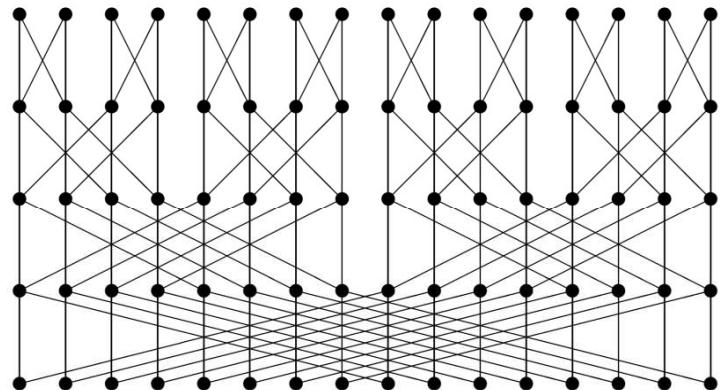
Warp synchronous programming is a CUDA programming technique that leverages warp execution for efficient inter-thread communication.

- e.g. reduction, scan, aggregated atomic operation, etc.

CUDA C++ supports warp synchronous programming by providing warp synchronous built-in functions and cooperative group collectives.

EXAMPLE: SUM ACROSS A WARP

```
val = input[lane_id];
val += __shfl_xor_sync(0xffffffff, val, 1);
val += __shfl_xor_sync(0xffffffff, val, 2);
val += __shfl_xor_sync(0xffffffff, val, 4);           val =  $\sum_{i=0}^{32} input[i]$ 
val += __shfl_xor_sync(0xffffffff, val, 8);
val += __shfl_xor_sync(0xffffffff, val, 16);
```

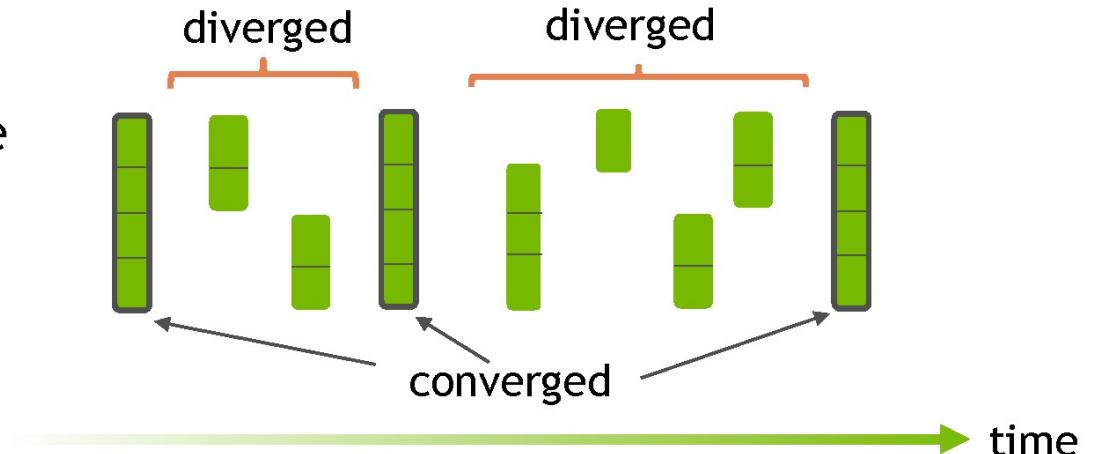


HOW TO WRITE WARP SYNCHRONOUS PROGRAMMING

Make Sync Explicit

Thread re-convergence

- Use built-in functions to converge threads explicitly
- Do not rely on implicit thread re-convergence.

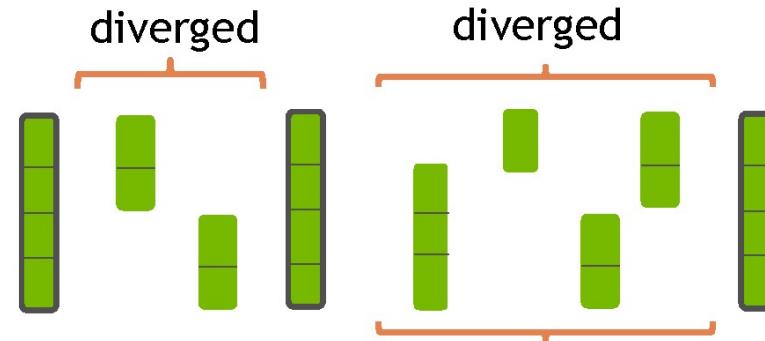


HOW TO WRITE WARP SYNCHRONOUS PROGRAMMING

Make Sync Explicit

Thread re-convergence

- Use built-in functions to converge threads explicitly
- Do not rely on implicit thread re-convergence.



Reading and writing the same memory location by different threads may cause data races.

Data exchange between threads

- Use built-in functions to sync threads and exchange data in one step.
- When using shared memory, avoid data races between convergence points.

WARP SYNCHRONOUS BUILT-IN FUNCTIONS

Three Categories (New in CUDA 9.0)

Active-mask query: which threads in a warp are active

- `__activemask`

Synchronized data exchange: exchange data between threads in warp

- `__all_sync`, `__any_sync`, `__uni_sync`, `__ballot_sync`
- `__shfl_sync`, `__shfl_up_sync`, `__shfl_down_sync`, `__shfl_xor_sync`
- `__match_any_sync`, `__match_all_sync`

Threads synchronization: synchronize threads in a warp and provide a memory fence

- `__syncwarp`

EXAMPLE: ALIGNED MEMORY COPY

`__activemask __all_sync`

```
// pick the optimal memory copy based on the alignment

__device__ void memorycopy(char *tptr, char *sptr, size_t size) {
    unsigned mask = __activemask();

    if (__all_sync(mask, is_all_aligned(tptr, sptr, 16)))
        return memcpy_aligned_16(tptr, sptr, size);

    if (__all_sync(mask, is_all_aligned(tptr, sptr, 8)))
        return memcpy_aligned_8(tptr, sptr, size);
    ...
}
```

EXAMPLE: ALIGNED MEMORY COPY

`__activemask __all_sync`

```
// pick the optimal memory copy based on the alignment
__device__ void memorycopy(char *tptr, char *sptr, size_t size) {
    unsigned mask = __activemask();
    if (__all_sync(mask, is_all_aligned(tptr, sptr, 16))
        return memcpy_aligned_16(tptr, sptr, size);
    if (__all_sync(mask, is_all_aligned(tptr, sptr, 8))
        return memcpy_aligned_8(tptr, sptr, size);
    ...
}
```

Find the active threads

EXAMPLE: ALIGNED MEMORY COPY

`__activemask __all_sync`

```
// pick the optimal memory copy based on the alignment
```

Find the active threads

```
__device__ void memorycopy(char *tptr, char *sptr, size_t size) {
```

```
    unsigned mask = __activemask();
```

```
    if (__all_sync(mask, is_all_aligned(tptr, sptr, 16))  
        return memcpy_aligned_16(tptr, sptr, size);
```

```
    if (__all_sync(mask, is_all_aligned(tptr, sptr, 8))  
        return memcpy_aligned_8(tptr, sptr, size);
```

```
    ...
```

```
}
```

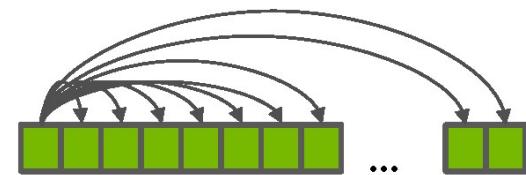
Returns true when all threads in ‘mask’ have the same predicate value

EXAMPLE: SHUFFLE

`__shfl_sync, __shfl_down_sync`

Broadcast: all threads get the value of ‘x’ from lane id 0

```
y = __shfl_sync(0xffffffff, x, 0);
```

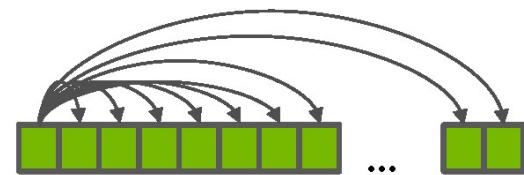


EXAMPLE: SHUFFLE

`__shfl_sync, __shfl_down_sync`

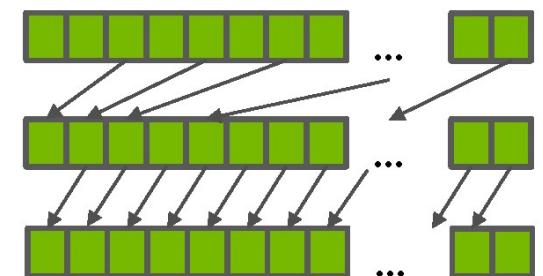
Broadcast: all threads get the value of ‘x’ from lane id 0

```
y = __shfl_sync(0xffffffff, x, 0);
```



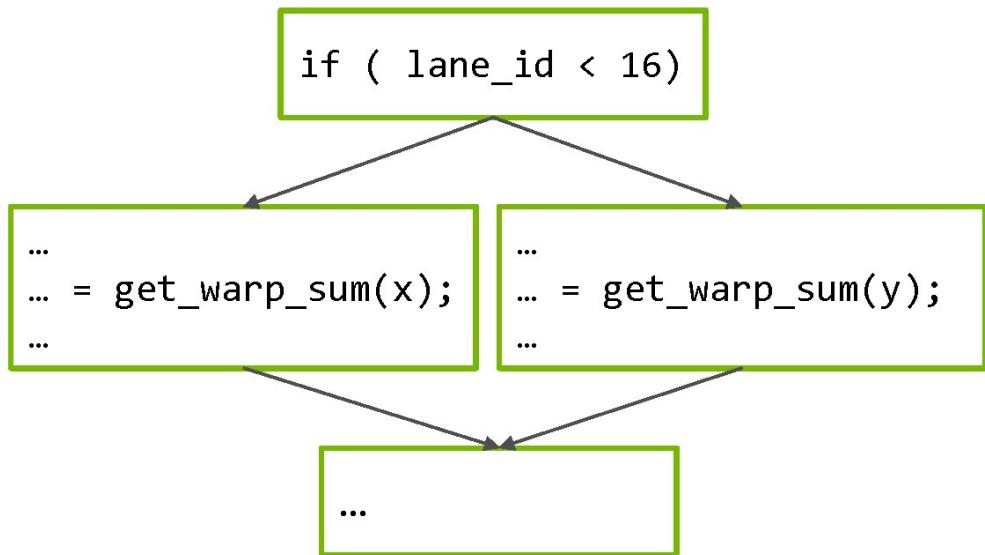
Reduction:

```
for (int offset = 16; offset > 0; offset /= 2)  
    val += __shfl_down_sync(0xffffffff, val, offset);
```



EXAMPLE: DIVERGENT BRANCHES

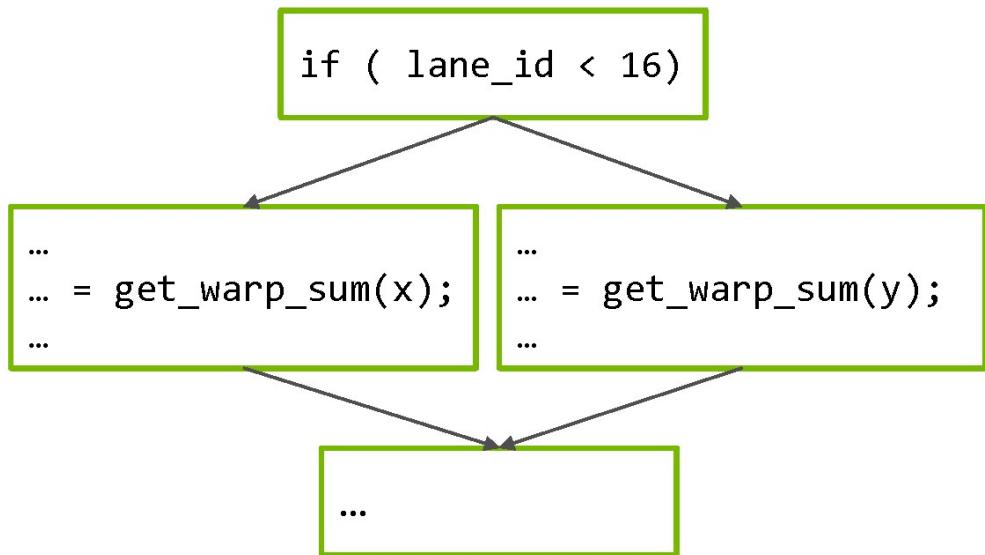
All *_sync built-in functions can be used in divergent branches on Volta



```
#define FULLMASK 0xffffffff  
  
__device__ int get_warp_sum(int v) {  
    for (int i = 1; i < 32; i = i*2)  
        v += __shfl_xor_sync(FULLMASK, v, i);  
    return v;  
}
```

EXAMPLE: DIVERGENT BRANCHES

All *_sync built-in functions can be used in divergent branches on Volta



```
#define FULLMASK 0xffffffff  
  
__device__ int get_warp_sum(int v) {  
    for (int i = 1; i < 32; i = i*2)  
        v += __shfl_xor_sync(FULLMASK, v, i);  
    return v;  
}
```

Possible to write a library function that performs warp synchronous programming w/o requiring it to be called convergently.

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;
```

BUT WHAT'S WRONG WITH THIS CODE?

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

IMPLICIT WARP SYNCHRONOUS PROGRAMMING

Unsafe and Unsupported

Implicit warp synchronous programming builds upon two unreliable assumptions,

- implicit thread re-convergence points, and
- Implicit lock-step execution of threads in a warp.

Implicit warp synchronous programming is unsafe and unsupported.

Make warp synchronous programming safe by making synchronizations explicit.

IMPLICIT THREAD RE-CONVERGENCE

Unreliable Assumption 1

Example 1:

```
if (lane_id < 16)
    A;
else
    B;
assert(__activemask() == 0xffffffff);
```

IMPLICIT THREAD RE-CONVERGENCE

Unreliable Assumption 1

Example 1:

```
if (lane_id < 16)
    A;
else
    B;
assert(__activemask() == 0xffffffff); not guaranteed to be true
```

Solution

- Do not rely on implicit thread re-convergence
- Use warp synchronous built-in functions to ensure convergence

IMPLICIT LOCK-STEP EXECUTION

Unreliable Assumption 2

Example 2

```
if (__activemask() == 0xffffffff) {  
    assert(__activemask() == 0xffffffff);  
}
```

IMPLICIT LOCK-STEP EXECUTION

Unreliable Assumption 2

Example 2

```
if (__activemask() == 0xffffffff) {  
    assert(__activemask() == 0xffffffff); not guaranteed to be true  
}
```

Solution

- Do not rely on implicit lock-step execution
- Use warp synchronous built-in functions to ensure convergence

IMPLICIT LOCK-STEP EXECUTION

Unreliable Assumption 2

Example 3

```
shmem[tid] += shmem[tid+16];
shmem[tid] += shmem[tid+8];
shmem[tid] += shmem[tid+4];
shmem[tid] += shmem[tid+2];
shmem[tid] += shmem[tid+1];
```

IMPLICIT LOCK-STEP EXECUTION

Unreliable Assumption 2

Example 3

```
shmem[tid] += shmem[tid+16];
shmem[tid] += shmem[tid+8];
shmem[tid] += shmem[tid+4];
shmem[tid] += shmem[tid+2];
shmem[tid] += shmem[tid+1];
```

} data race

Solution

- Make sync explicit

```
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;
```

LEGACY WARP-LEVEL BUILT-IN FUNCTIONS

Deprecated in CUDA 9.0

Legacy built-in functions

- `__all()`, `__any()`, `__ballot()`, `__shfl()`, `__shfl_up()`, `__shfl_down()`, `__shfl_xor()`

These legacy warp-level built-in functions can perform data exchange between the active threads in a warp.

They do not ensure which threads are active.

They are deprecated in CUDA 9.0 on all architectures.

COOPERATIVE GROUPS VS BUILT-IN FUNCTIONS

Example: warp aggregated atomic

```
// increment the value at ptr by 1 and return the old value
__device__ int atomicAggInc(int *p);
```

```
coalesced_group g = coalesced_threads();
int res;
if (g.thread_rank() == 0)
    res = atomicAdd(p, g.size());
res = g.shfl(res, 0);
return g.thread_rank() + res;
```

```
int mask = __activemask();
int rank = __popc(mask & __lanemask_lt());
int leader_lane = __ffs(mask) - 1;
int res;
if (rank == 0)
    res = atomicAdd(p, __popc(mask));
res = __shfl_sync(mask, res, leader_lane);
return rank + res;
```

WARP SYNCHRONOUS PROGRAMMING IN CUDA 9.0

New warp synchronous built-in functions ensure reliable synchronizations.

New warp synchronous built-in functions can be used divergently on Volta.

Legacy warp built-in functions are deprecated.

Cooperative groups offers

- Higher-level abstraction of thread groups
- Four levels of thread grouping
- More scalable code and better software decomposition



BETTER COMPOSITION

Barrier synchronization hidden within functions

```
__device__ int sum(int *x, int n)
{
    ...
    __syncthreads();
    ...
    return total;
}

__global__ void parallel_kernel(float *x)
{
    ...
    // Entire thread block must call sum
    sum(x, n);
}
```

All threads in thread block
must arrive at this barrier.

Hidden constraint on
caller due to
implementation of *sum*.

BETTER COMPOSITION

Explicit cooperative interfaces

```
__device__ int sum(thread_group g, int *x, int n)
{
    ...
    g.sync()                                Participating thread group
    ...
    return total;
}

__global__ void parallel_kernel(...)
{
    ...
    // Entire thread block must call sum      The need to synchronize
    sum(this_thread_block(), x, n);           in sum is visible in code.
    ...
}
```

FUTURE ROADMAP

Partition by label or predicate, more complex scopes

0 1 0 1 0 1 0 1 (Volta specific)

```
thread_group cta = this_thread_block();
thread_group g = partition(cta, cta.thread_rank() & 1);
```

Warp 32



Warp 32

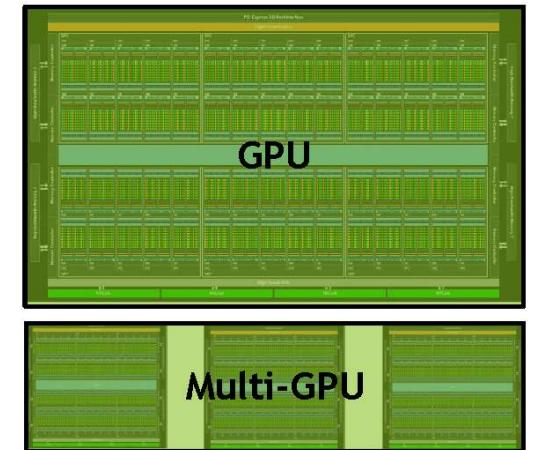


Warp 32



```
thread_group g = tiled_partition(cta, 64);
```

At all scopes!



FUTURE ROADMAP

Library of collectives (sort, reduce, etc.)

```
template <int BlockThreads>
__global__ int BlockReduce(float *d_in, ...)
{
    static_thread_block<BlockThreads> cta = this_thread_block();
    // Statically allocate shared reduction storage
    __shared__ reduce_storage<decltype(cta), float> group_reduce;

    // Compute the block-wide sum for thread-0
    float total = cooperative_groups::reduce(
                    cta, d_in[cta.rank()], group_reduce);
}
```

On a simpler note:

```
// Collective key-value sort, default allocator
cooperative_groups::sort(this_thread_block(), myValues, myKeys);
```

HONORABLE MENTION

The ones that didn't make it into their own slide

`_CG_DEBUG` : Define to enable various runtime safety checks. This helps debug incorrect API usage, incorrect synchronization, or similar issues (Automatically turned on with `-G`).

Tools help detect incorrect warp-synchronization with the racecheck tool.

Match is a new Volta instruction that is able to return who in your warp has the same 32 or 64 bit value

Developers **now have** a flexible model for synchronization and communication between groups of threads.

Shipping in CUDA 9.0

Provides safety, composability, and high performance

Flexibility to synchronize at various architecture and program defined scopes.

Deploy everywhere from Kepler to Volta

COOPERATIVE GROUPS

Kyrylo Perelygin, Yuan Lin

GTC 2017



Cooperative Groups: a flexible model for synchronization and communication within groups of threads.

At a glance

Scalable Cooperation among groups of threads

Flexible parallel decompositions

Composition across software boundaries

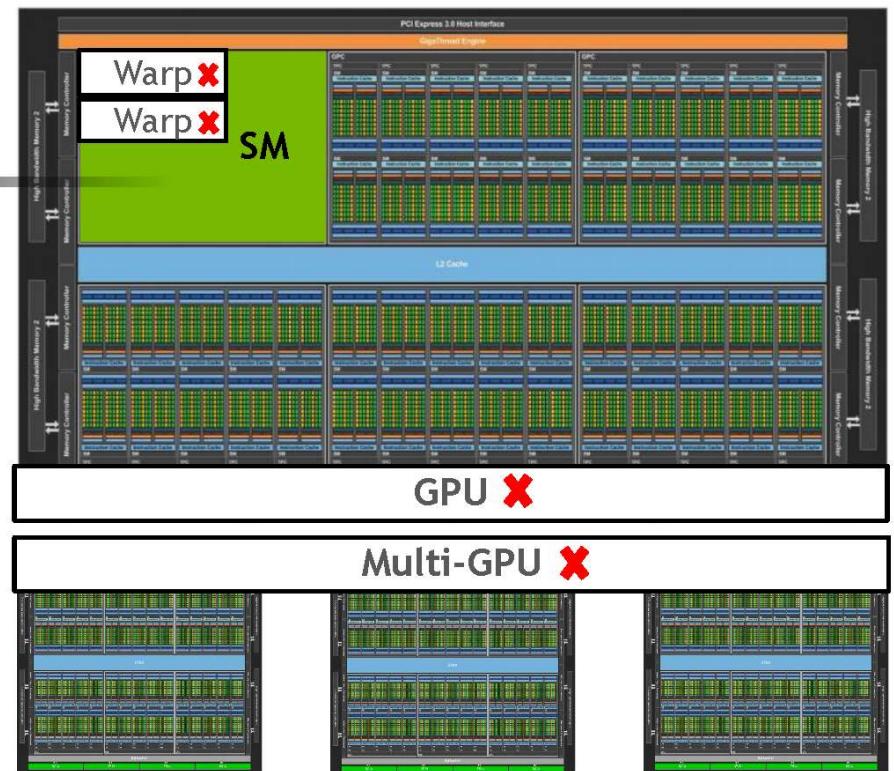
Deploy Everywhere

Benefits all applications

Examples include:
Persistent RNNs
Physics
Search Algorithms
Sorting

LEVELS OF COOPERATION: TODAY

`__syncthreads(): block level synchronization barrier in CUDA`



LEVELS OF COOPERATION: CUDA 9.0

For current coalesced set of threads:

```
auto g = coalesced_threads();
```

For warp-sized group of threads:

```
auto block = this_thread_block();  
auto g = tiled_partition<32>(block)
```

For CUDA thread blocks:

```
auto g = this_thread_block();
```

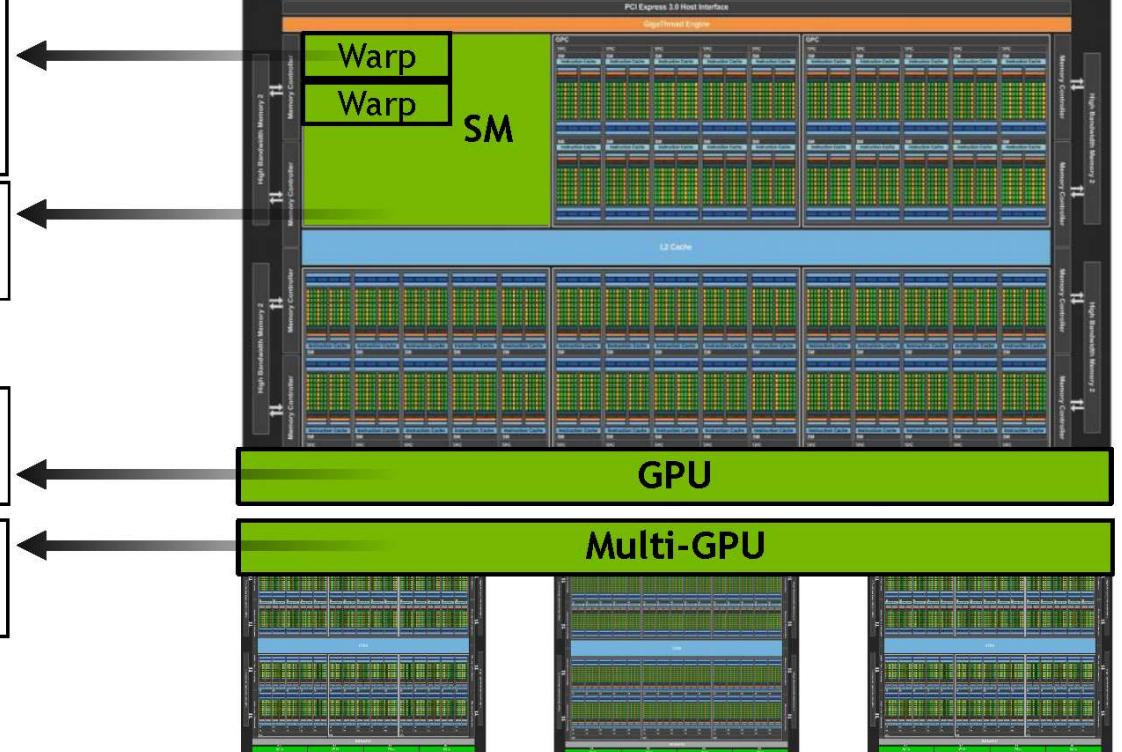
For device-spanning grid:

```
auto g = this_grid();
```

For multiple grids spanning GPUs:

```
auto g = this_multi_grid();
```

All Cooperative Groups functionality is
within a **cooperative_groups::** namespace

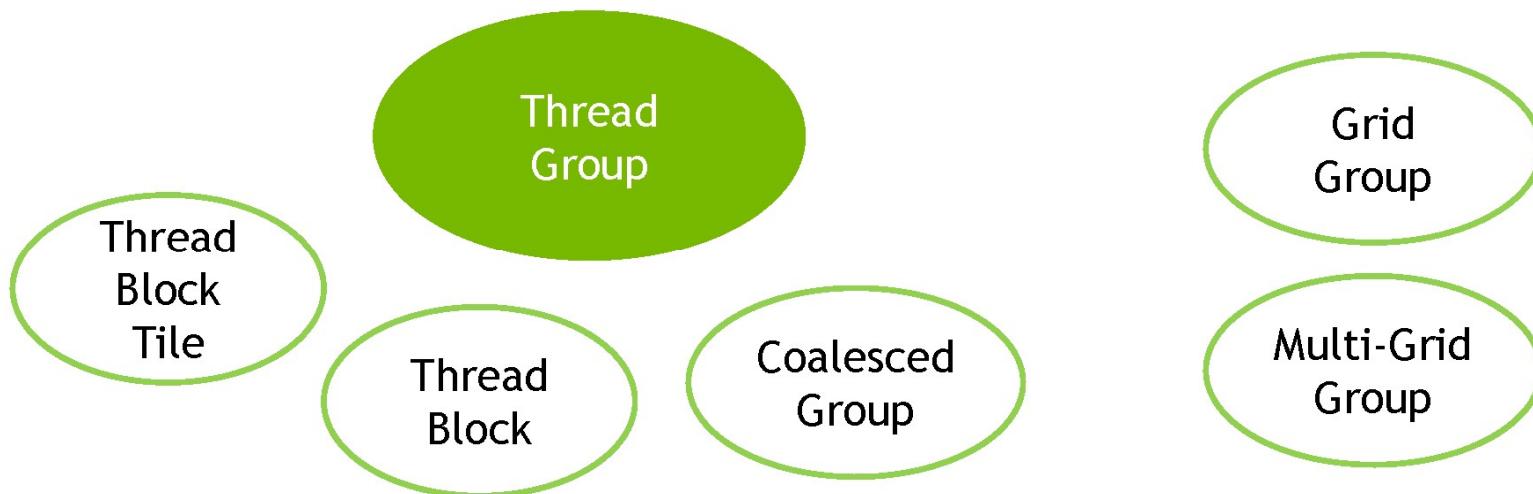


THREAD GROUP

Base type, the implementation depends on its construction.

Unifies the various group types into one general, collective, thread group.

We need to extend the CUDA programming model with handles that can represent the groups of threads that can communicate/synchronize



THREAD BLOCK

Implicit group of all the threads in the launched thread block

Implements the same interface as `thread_group`:

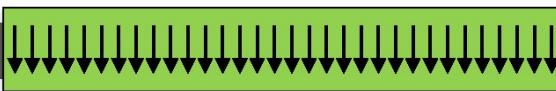
```
void sync();           // Synchronize the threads in the group  
unsigned size();      // Total number of threads in the group  
unsigned thread_rank(); // Rank of the calling thread within [0, size]  
bool is_valid();       // Whether the group violated any API constraints
```

And additional `thread_block` specific functions:

```
dim3 group_index();    // 3-dimensional block index within the grid  
dim3 thread_index();   // 3-dimensional thread index within the block
```

PROGRAM DEFINED DECOMPOSITION

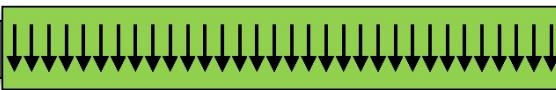
CUDA KERNEL



All threads launched

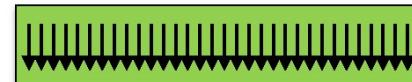
```
thread_block g = this_thread_block();
```

foobar(thread_block g)



All threads in thread block

```
thread_group tile32 = tiled_partition(g, 32);
```



```
thread_group tile4 = tiled_partition(tile32, 4);
```



Restricted to powers of two,
and <= 32 in initial release

GENERIC PARALLEL ALGORITHMS

Per-Block

```
g = this_thread_block();
reduce(g, ptr, myVal);
```

Per-Warp

```
g = tiled_partition(this_thread_block(), 32);
reduce(g, ptr, myVal);
```



```
__device__ int reduce(thread_group g, int *x, int val) {
    int lane = g.thread_rank();
    for (int i = g.size()/2; i > 0; i /= 2) {
        x[lane] = val;           g.sync();
        val += x[lane + i];    g.sync();
    }
    return val;
}
```

THREAD BLOCK TILE

A subset of threads of a thread block, divided into tiles in row-major order

```
thread_block_tile<32> tile32 = tiled_partition<32>(this_thread_block());
```



```
thread_block_tile<4> tile4 = tiled_partition<4>(this_thread_block());
```



Expose additional functionality:

Size known at compile time = fast!

.shfl()
.shfl_down()
.shfl_up()
.shfl_xor()

.any()
.all()
.ballot()
.match_any()
.match_all()

STATIC TILE REDUCE

Per-Tile of 16 threads

```
g = tiled_partition<16>(this_thread_block());
tile_reduce(g, myVal);
```



```
template <unsigned size>
__device__ int tile_reduce(thread_block_tile<size> g, int val) {
    for (int i = g.size()/2; i > 0; i /= 2) {
        val += g.shfl_down(val, i);
    }
    return val;
}
```

GRID GROUP

A set of threads within the same grid, guaranteed to be resident on the device

New CUDA Launch API to opt-in:
`cudaLaunchCooperativeKernel(...)`

```
__global__ kernel() {
    grid_group grid = this_grid();
    // load data
    // loop - compute, share data
    grid.sync();
    // devices are now synced
}
```



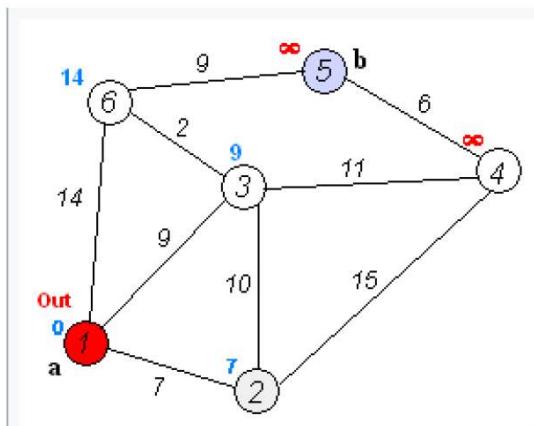
Device needs to support the `cooperativeLaunch` property.

```
cudaOccupancyMaxActiveBlocksPerMultiprocessor(&numBlocksPerSm, kernel, numThreads, 0));
```

GRID GROUP

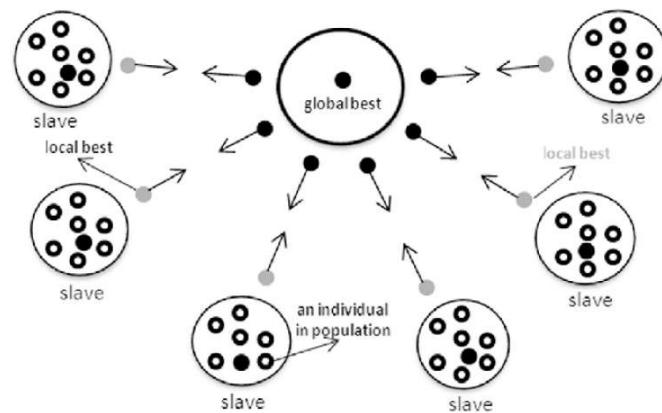
The goal: keep as much state as possible resident

Shortest Path / Search



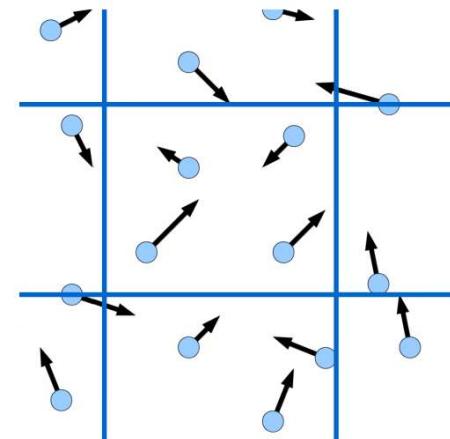
Weight array perfect for persistence
Iteration over vertices?
Fuse!

Genetic Algorithms / Master driven algorithms



Synchronization
between a master block
and slaves

Particle Simulations

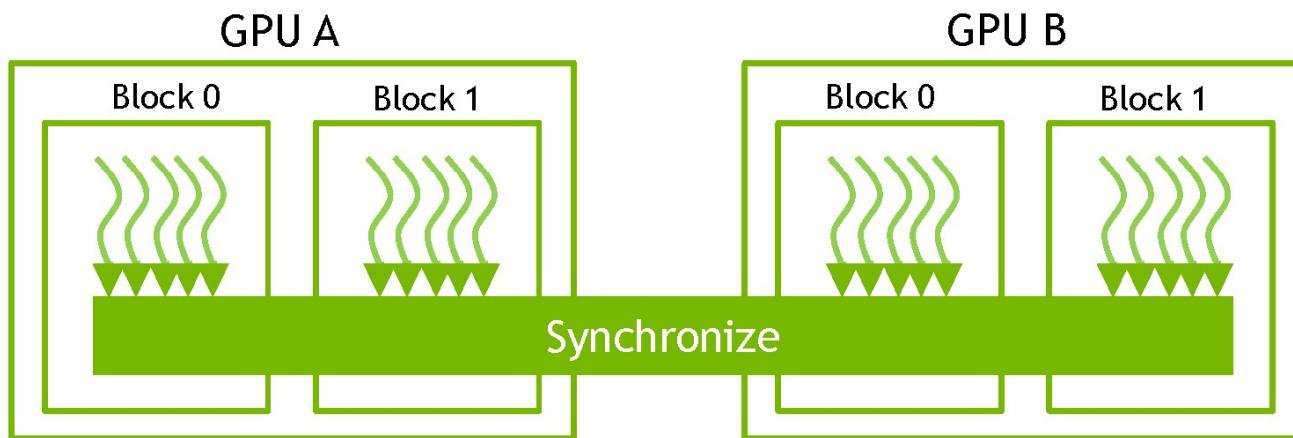


Synchronization
between update and
collision simulation

MULTI GRID GROUP

A set of threads guaranteed to be resident on the same system, on multiple devices

```
__global__ void kernel() {
    multi_grid_group multi_grid = this_multi_grid();
    // load data
    // loop - compute, share data
    multi_grid.sync();
    // devices are now synced, keep on computing
}
```



MULTI GRID GROUP

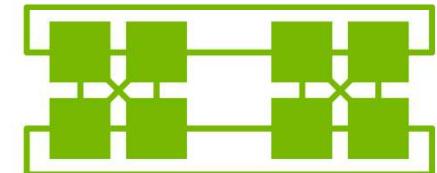
Launch on multiple devices at once

New CUDA Launch API to opt-in:

```
cudaLaunchCooperativeKernelMultiDevice(...)
```

Devices need to support the `cooperativeMultiDeviceLaunch` property.

```
struct cudaLaunchParams params[numDevices];
for (int i = 0; i < numDevices; i++) {
    params[i].func = (void *)kernel;
    params[i].gridDim = dim3(...); // Use occupancy calculator
    params[i].blockDim = dim3(...);
    params[i].sharedMem = ...;
    params[i].stream = ...; // Cannot use the NULL stream
    params[i].args = ...;
}
cudaLaunchCooperativeKernelMultiDevice(params, numDevices);
```



COALESCED GROUP

Discover the set of coalesced threads, i.e. a group of converged threads executing in SIMD

```
coalesced_group active = coalesced_threads();
```



Size: 8

COALESCED GROUP

Discover the set of coalesced threads, i.e. a group of converged threads executing in SIMD



COALESCED GROUP

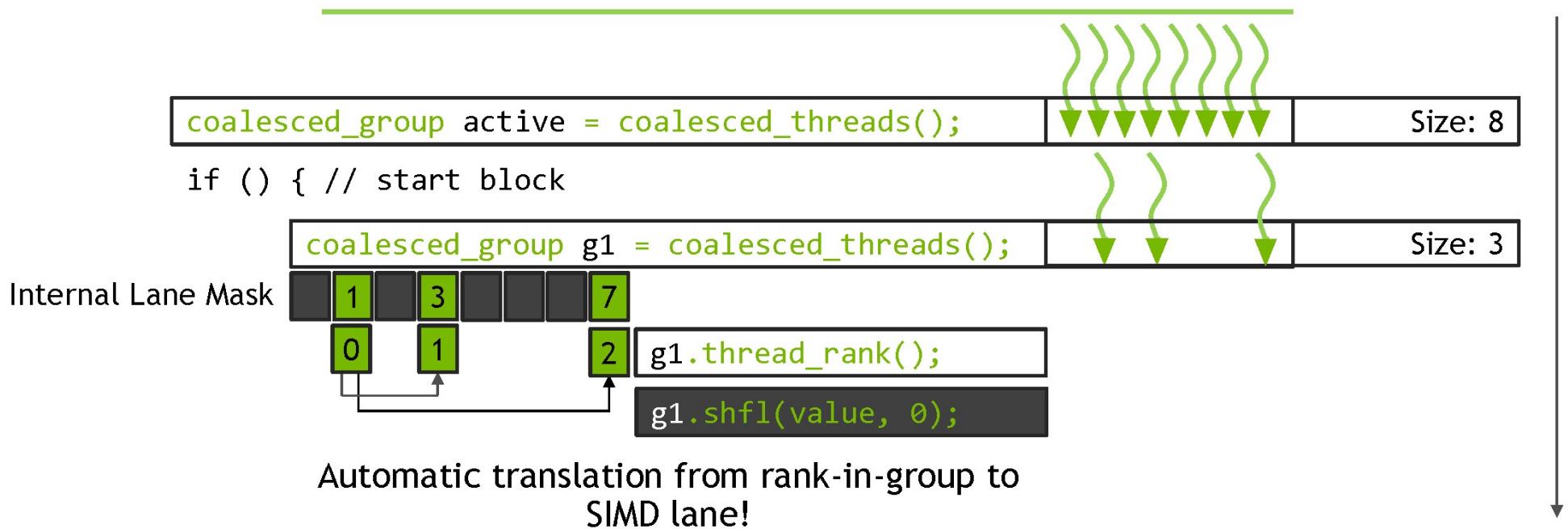
Discover the set of coalesced threads, i.e. a group of converged threads executing in SIMD



Automatic translation to rank-in-group!

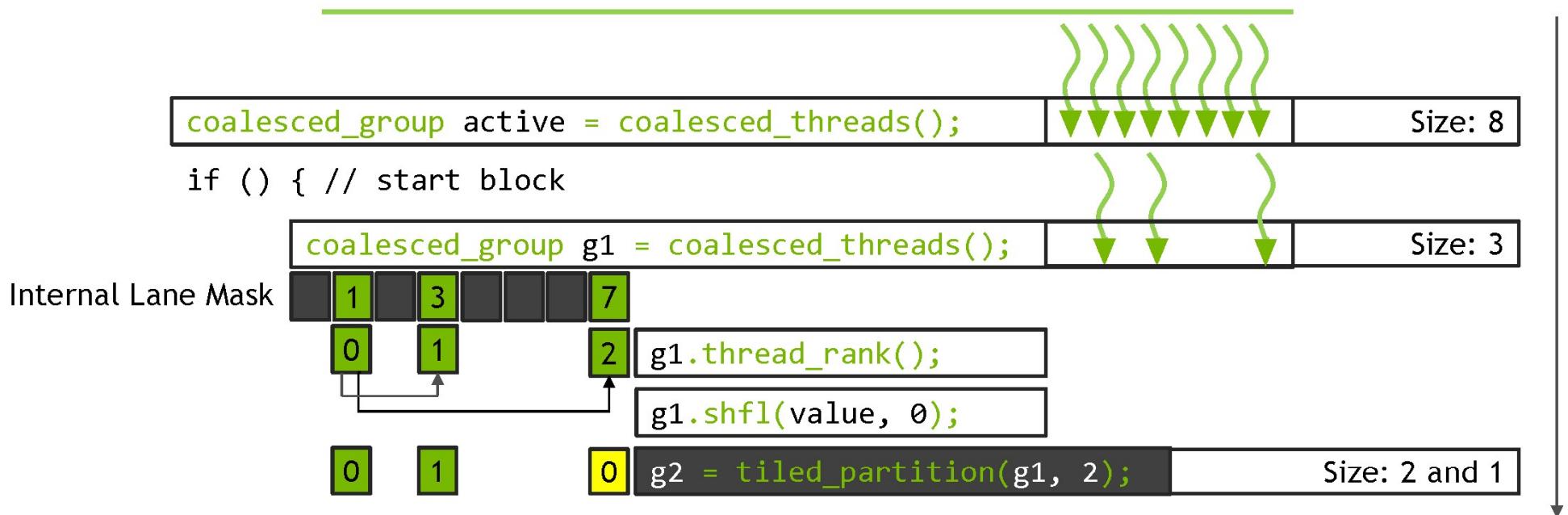
COALESCED GROUP

Discover the set of coalesced threads, i.e. a group of converged threads executing in SIMD



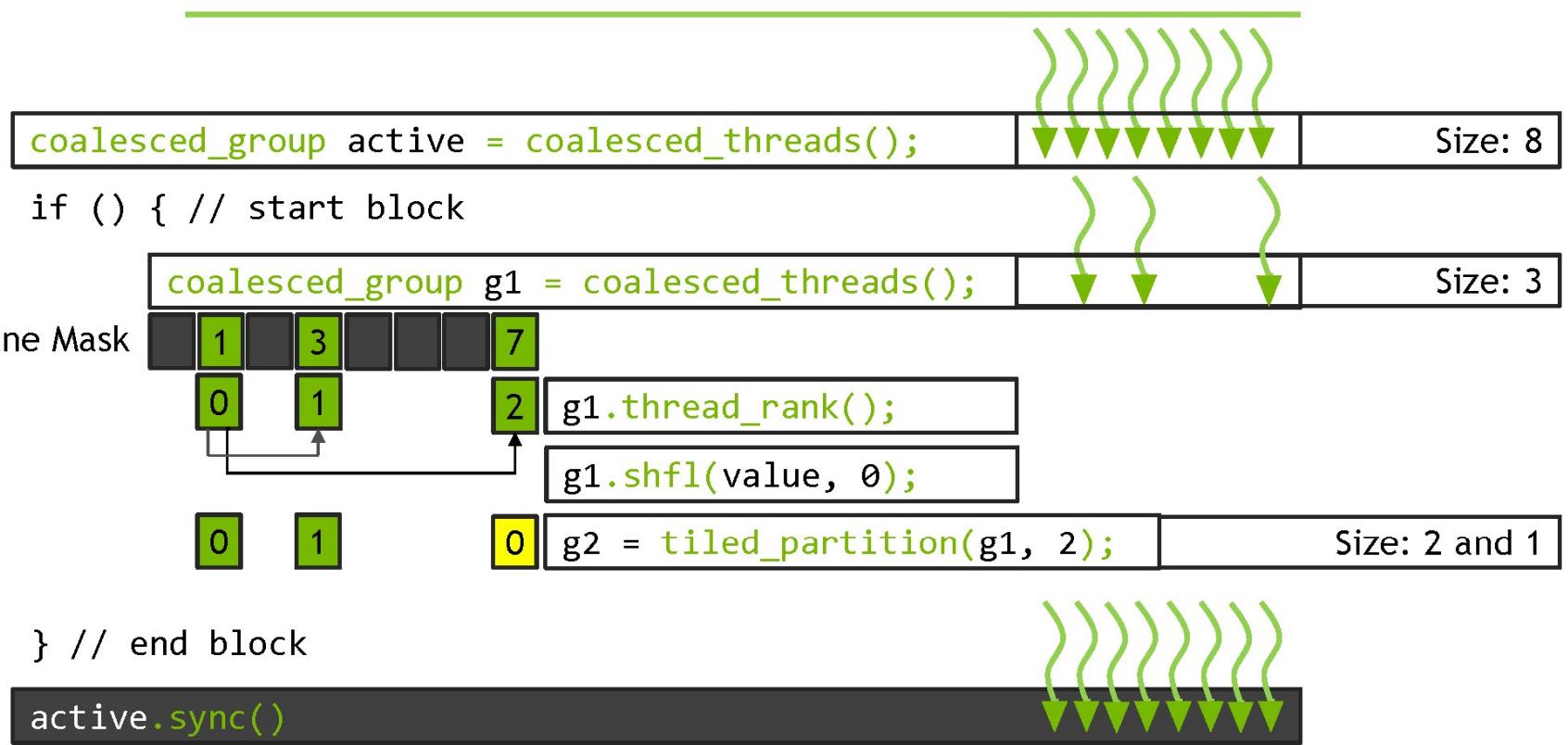
COALESCED GROUP

Discover the set of coalesced threads, i.e. a group of converged threads executing in SIMD



COALESCED GROUP

Discover the set of coalesced threads, i.e. a group of converged threads executing in SIMD

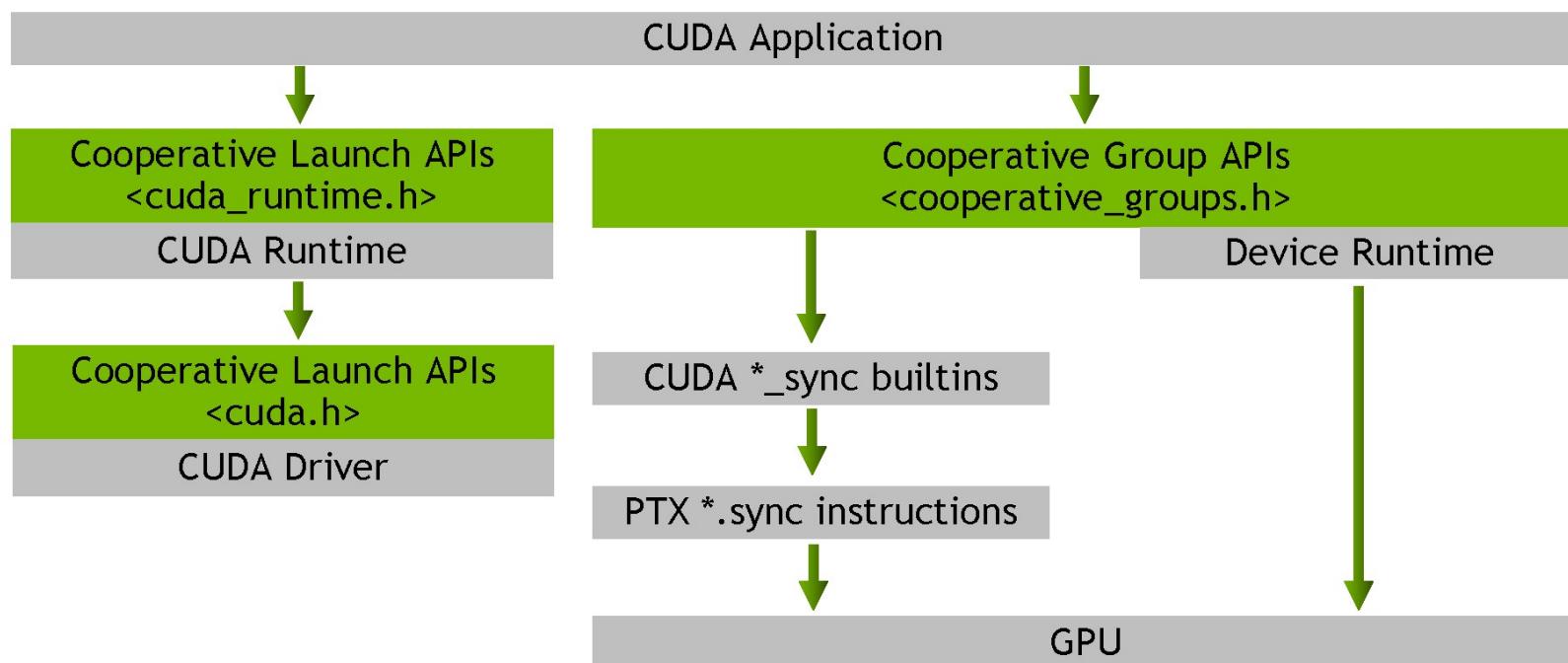


ATOMIC AGGREGATION

Opportunistic cooperation within a warp

```
inline __device__ int atomicAggInc(int *p)
{
    coalesced_group g = coalesced_threads();
    int prev;
    if (g.thread_rank() == 0) {
        prev = atomicAdd(p, g.size());
    }
    prev = g.thread_rank() + g.shfl(prev, 0);
    return prev;
}
```

ARCHITECTURE



Thank you.

- NVIDIA