

# **CS 380 - GPU and GPGPU Programming**

## **Lecture 24: Scan Bank Conflicts; CUDA Memory, Pt. 4**

Markus Hadwiger, KAUST

# Reading Assignment #14 (until Dec 6)



## Read (required):

- Warp Shuffle Functions
  - CUDA Programming Guide 11.5, Appendix B.22
- CUDA Cooperative Groups
  - CUDA Programming Guide 11.5, Appendix C
  - <https://developer.nvidia.com/blog/cooperative-groups/>
- Programming Tensor Cores
  - CUDA Programming Guide 11.5, Appendix B.24 (Warp matrix functions)
  - <https://developer.nvidia.com/blog/programming-tensor-cores-cuda-9/>

## Read (optional):

- CUDA Warp-Level Primitives
  - <https://developer.nvidia.com/blog/using-cuda-warp-level-primitives/>
- Warp-aggregated atomics
  - <https://developer.nvidia.com/blog/cuda-pro-tip-optimized-filtering-warp-aggregated-atomics/>



# GPU Parallel Prefix Sum

# Parallel Prefix Sum (Scan)

---

- **Definition:**

The all-prefix-sums operation takes a binary associative operator  $\oplus$  with identity  $I$ , and an array of  $n$  elements

$$[a_0, a_1, \dots, a_{n-1}]$$

and returns the ordered set

$$[I, a_0, (a_0 \oplus a_1), \dots, (a_0 \oplus a_1 \oplus \dots \oplus a_{n-2})].$$

- **Example:**

if  $\oplus$  is addition, then scan on the set

[3 1 7 0 4 1 6 3]

returns the set

[0 3 4 11 11 15 16 22]

Exclusive scan: last input element is not included in the result

*(From Blelloch, 1990, "Prefix Sums and Their Applications")*



# Work Efficiency Considerations

---

- **The first-attempt Scan executes  $\log(n)$  parallel iterations**
  - Total adds:  $n * (\log(n) - 1) + 1 \rightarrow O(n * \log(n))$  work
- **This scan algorithm is not very work efficient**
  - Sequential scan algorithm does  $n$  adds
  - A factor of  $\log(n)$  hurts: 20x for  $10^6$  elements!
- **A parallel algorithm can be slow when execution resources are saturated due to low work efficiency**

# Balanced Trees

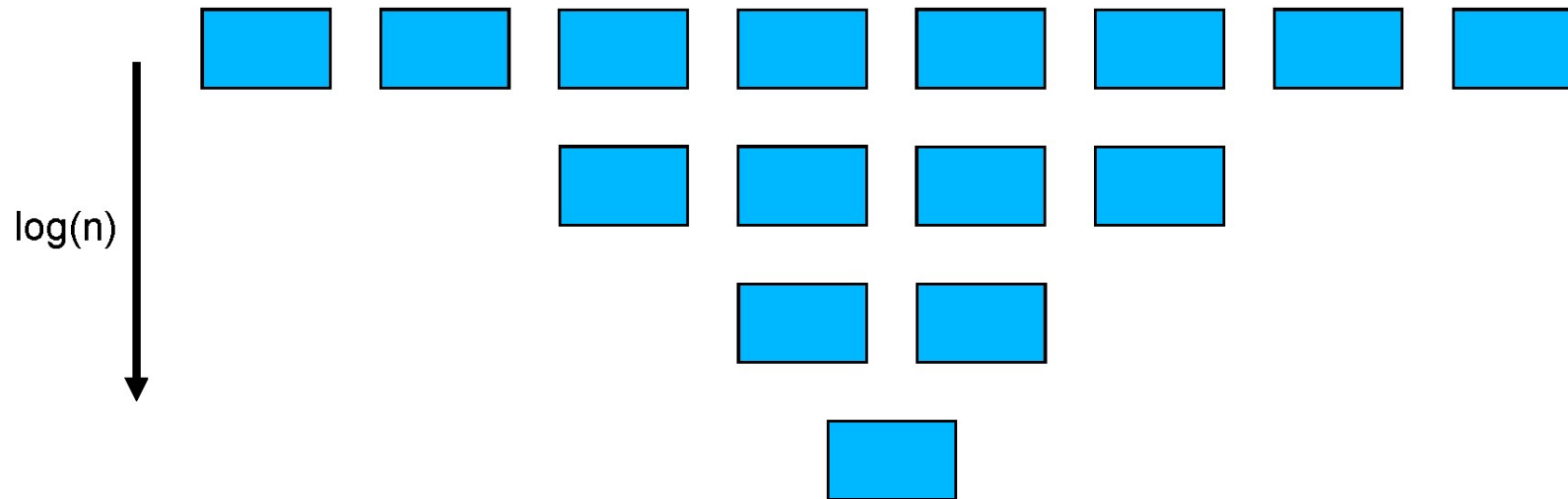
---

- **For improving efficiency**
- **A common parallel algorithm pattern:**
  - Build a balanced binary tree on the input data and sweep it to and from the root
  - Tree is not an actual data structure, but a concept to determine what each thread does at each step
- **For scan:**
  - Traverse down from leaves to root building partial sums at internal nodes in the tree
    - Root holds sum of all leaves
  - Traverse back up the tree building the scan from the partial sums

# Typical Parallel Programming Pattern

---

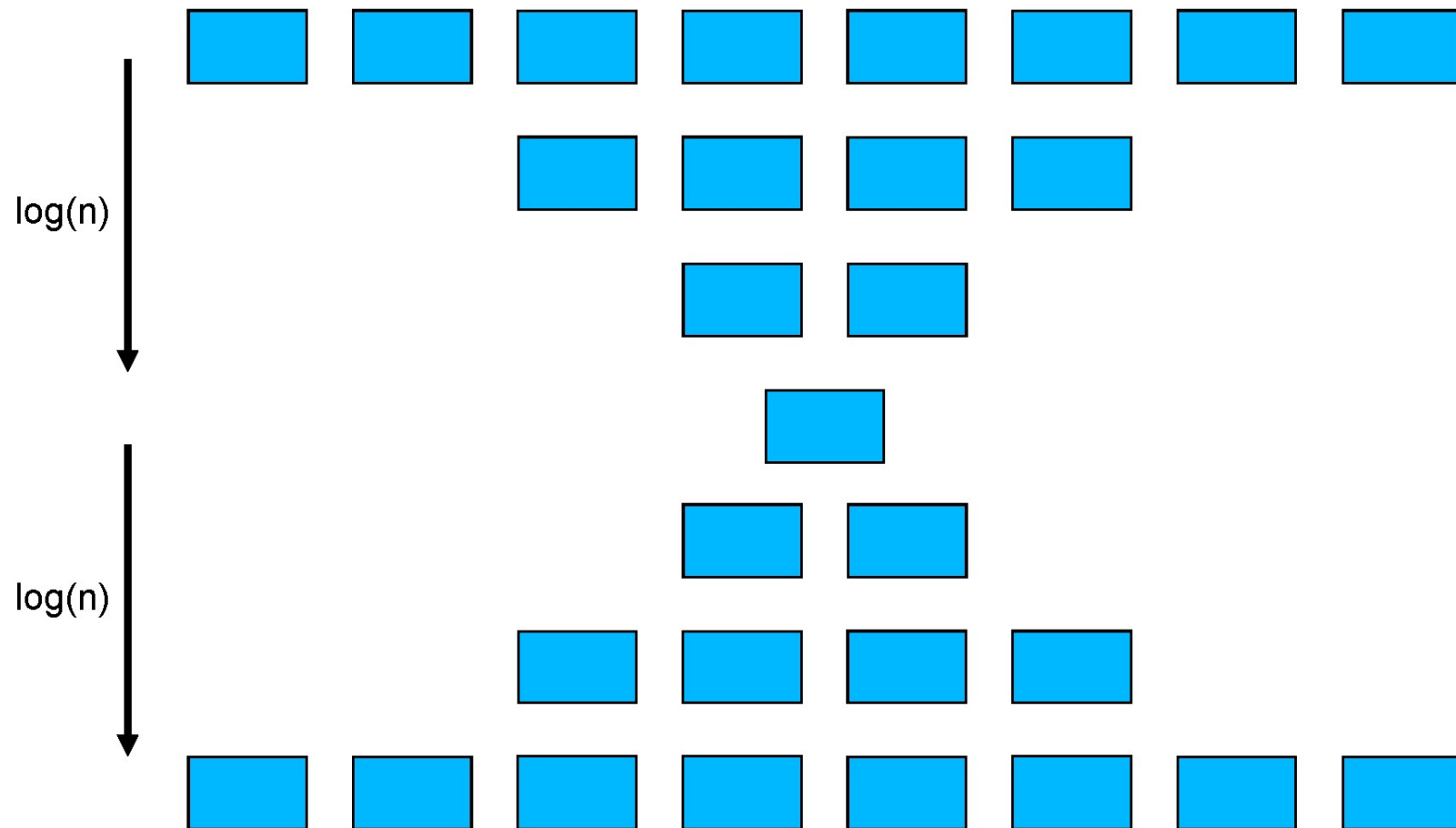
- $2 \log(n)$  steps



# Typical Parallel Programming Pattern

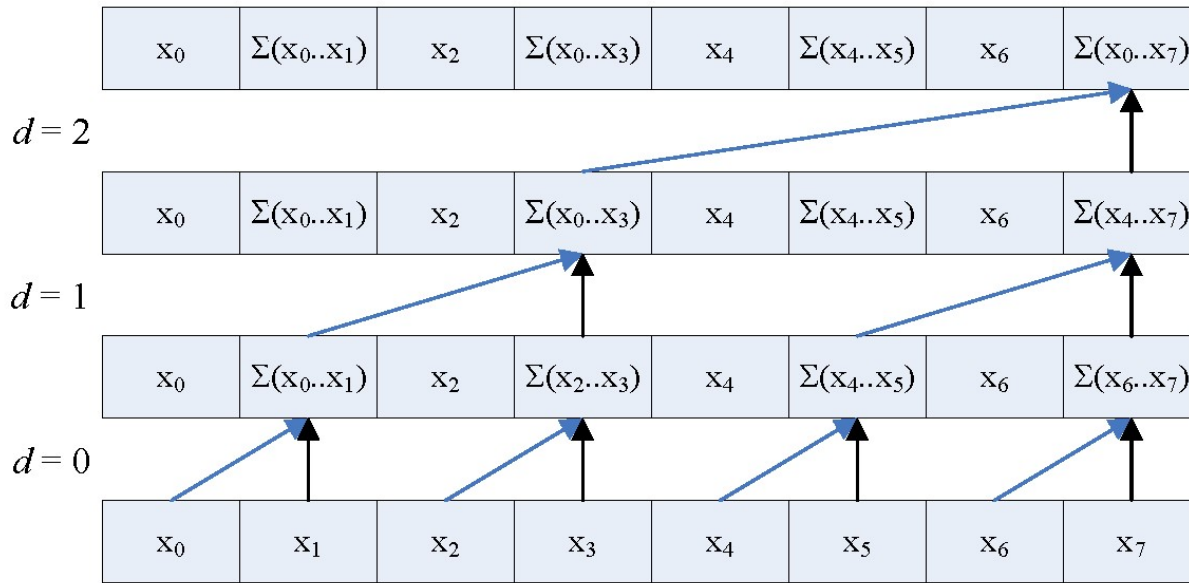
---

- $2 \log(n)$  steps

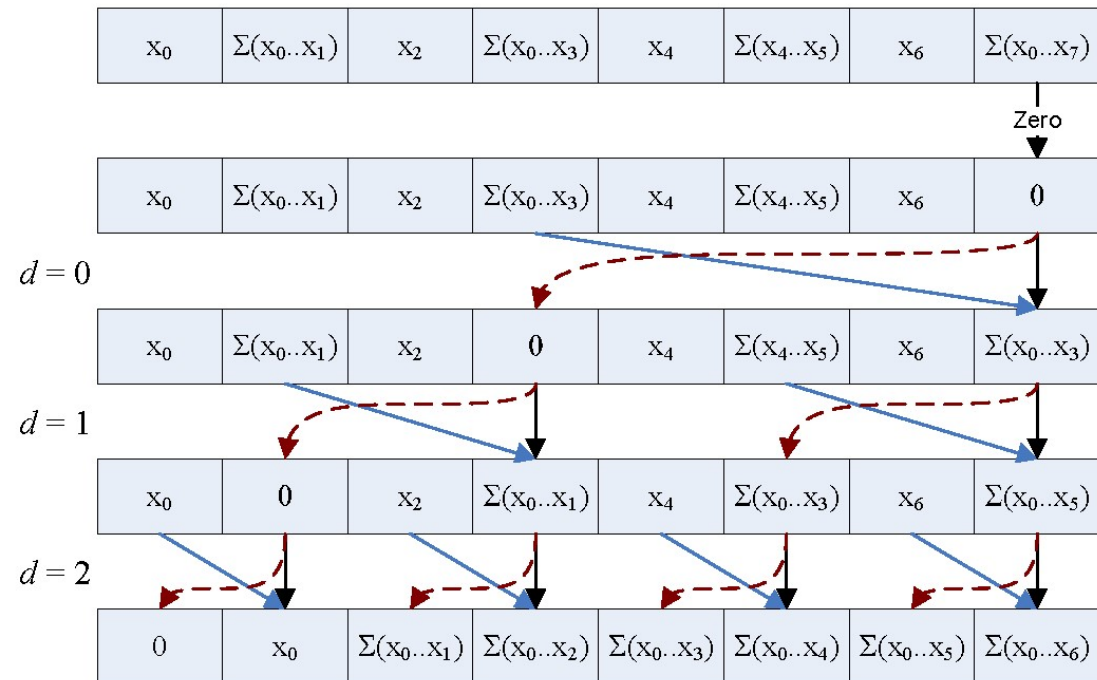


# $O(n)$ Scan [Blelloch]

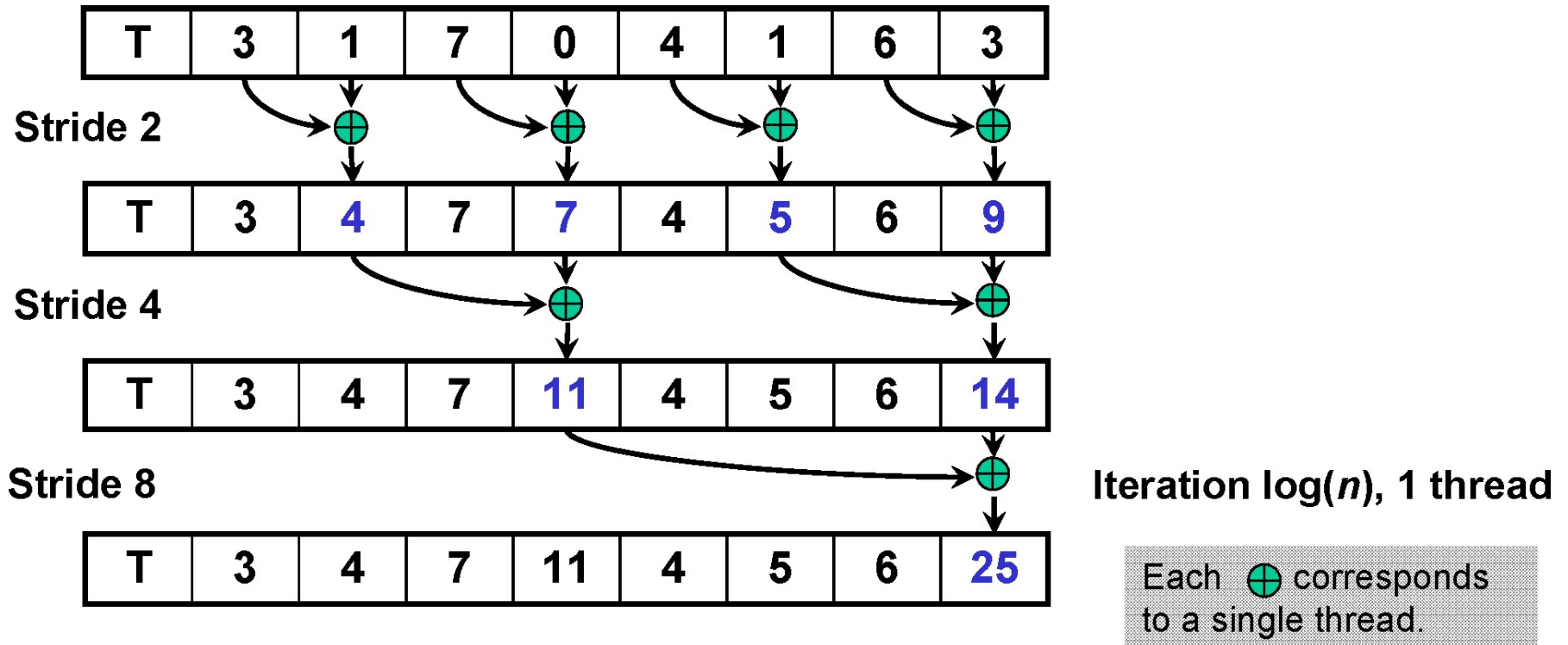
Courtesy John Owens



- Work efficient ( $O(n)$  work)
- Bank conflicts, and lots of 'em



# Build the Sum Tree



Iterate  $\log(n)$  times. Each thread adds value  $stride / 2$  elements away to its own value.

Note that this algorithm operates in-place: no need for double buffering

# Down-Sweep Variant 1: Exclusive Scan

---

T	3	4	7	11	4	5	6	0
---	---	---	---	----	---	---	---	---

We now have an array of partial sums. Since this is an exclusive scan, set the last element to zero. It will propagate back to the first element.

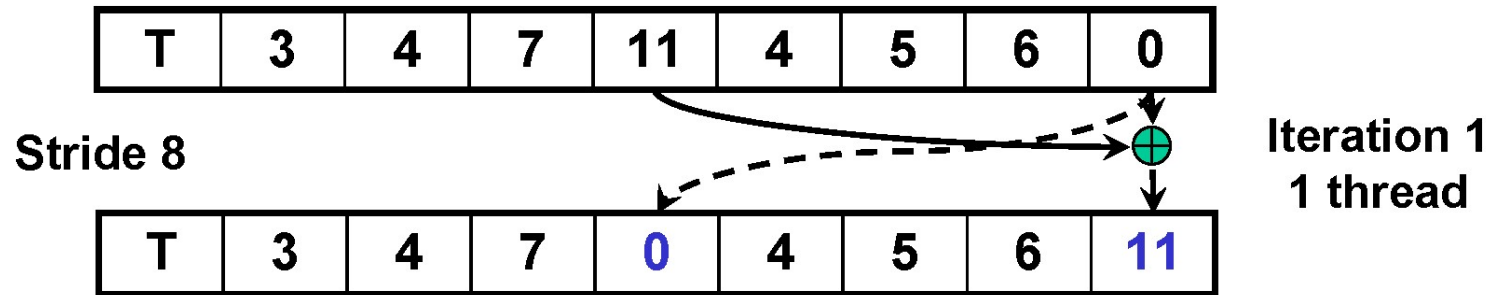
# Build Scan From Partial Sums


---

T	3	4	7	11	4	5	6	0
---	---	---	---	----	---	---	---	---



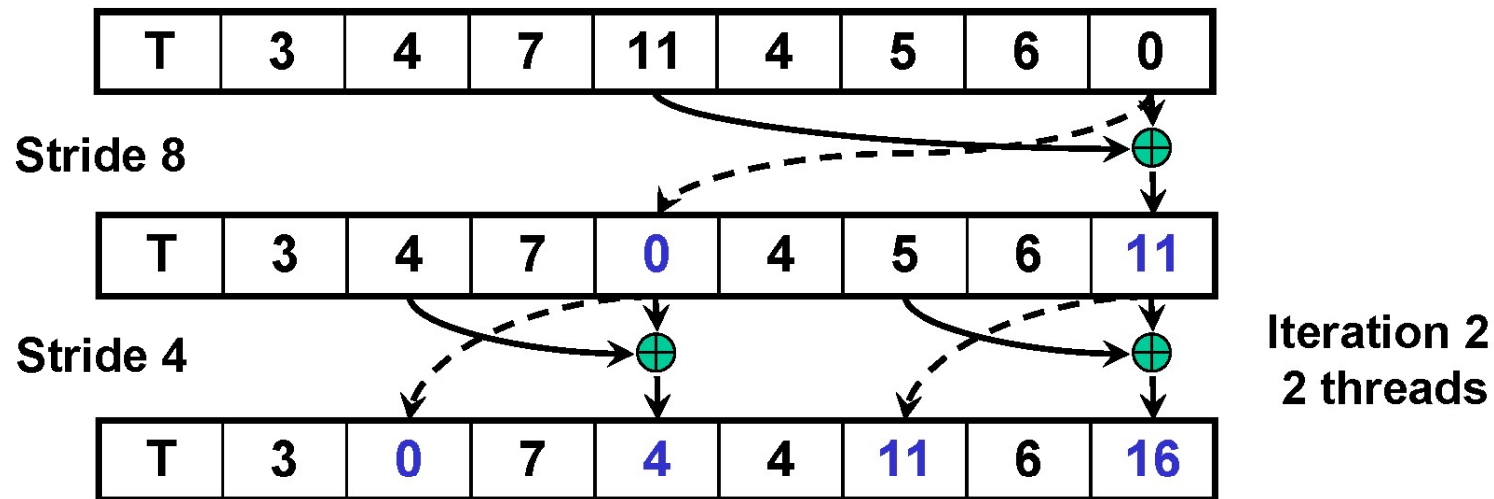
# Build Scan From Partial Sums




Each  corresponds to a single thread.

Iterate  $\log(n)$  times. Each thread adds value *stride* / 2 elements away to its own value. and sets the value *stride* elements away to its own *previous* value.

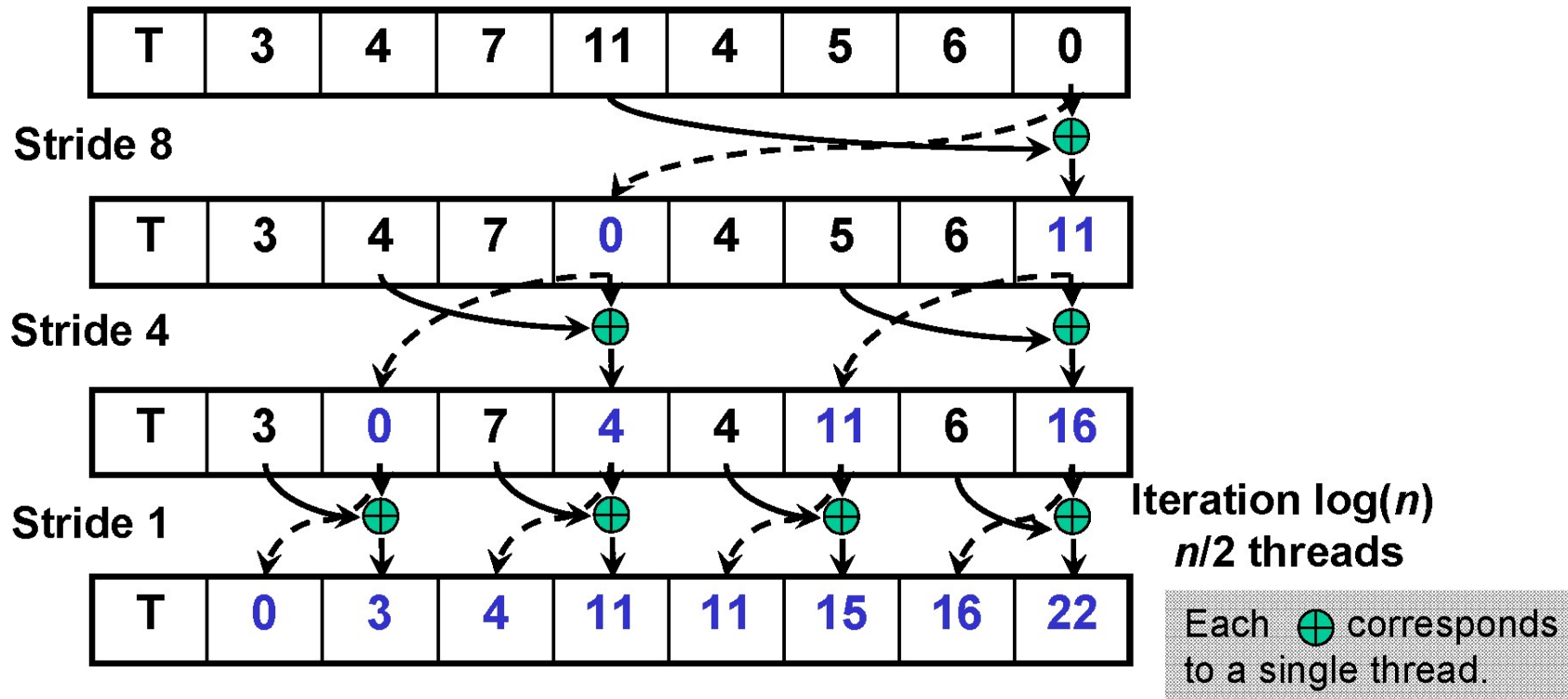
# Build Scan From Partial Sums



Each  corresponds to a single thread.

Iterate  $\log(n)$  times. Each thread adds value  $stride / 2$  elements away to its own value. and sets the value  $stride / 2$  elements away to its own *previous* value.

# Build Scan From Partial Sums



Done! We now have a completed scan that we can write out to device memory.

Total steps:  $2 * \log(n)$ .

Total work:  $2 * (n-1)$  adds =  $O(n)$  **Work Efficient!**

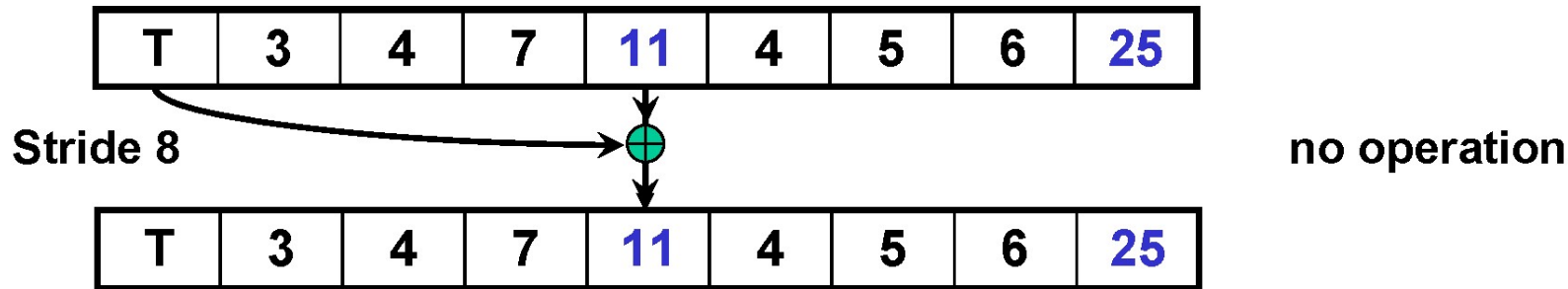
# Down-Sweep Variant 2: Inlusive Scan


---

T	3	4	7	11	4	5	6	25
---	---	---	---	----	---	---	---	----

We now have an array of partial sums. Let's propagate the sums back.

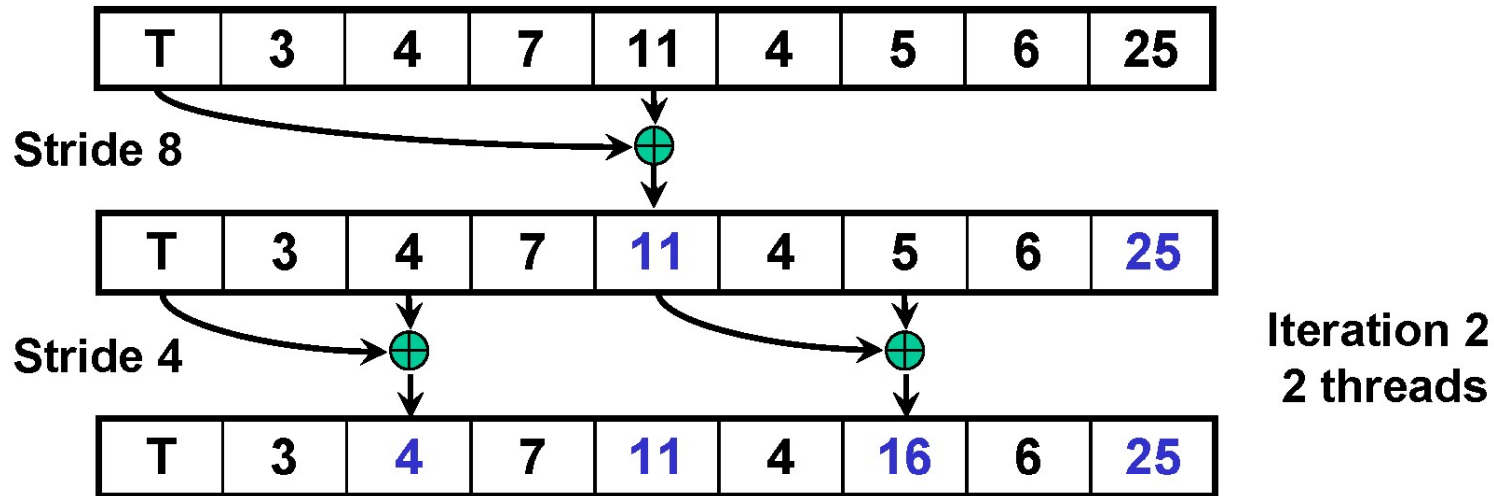
# Build Scan From Partial Sums




Each  corresponds to a single thread.

Iterate  $\log(n)$  times. Each thread adds value *stride* / 2 elements away to its own value. First element adds zero.

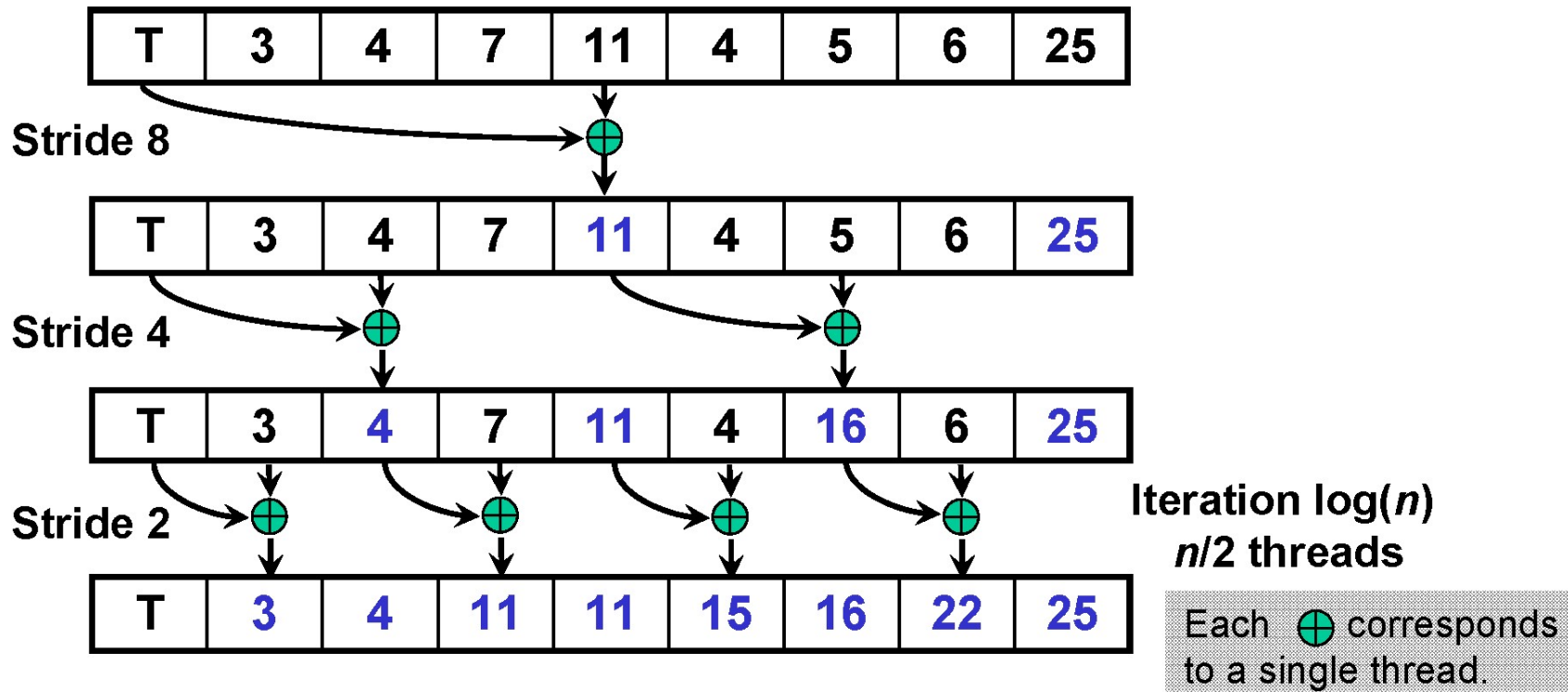
# Build Scan From Partial Sums



Each  corresponds to a single thread.

Iterate  $\log(n)$  times. Each thread adds value *stride* / 2 elements away to its own value. First element adds zero.

# Build Scan From Partial Sums



Done! We now have a completed scan that we can write out to device memory.

Total steps:  $2 * \log(n)$ .

Total work:  $< 2 * (n-1)$  adds =  $O(n)$  **Work Efficient!**

---

# Bank Conflicts in Scan - Non-power-of-two -



# Initial Bank Conflicts on Load

---

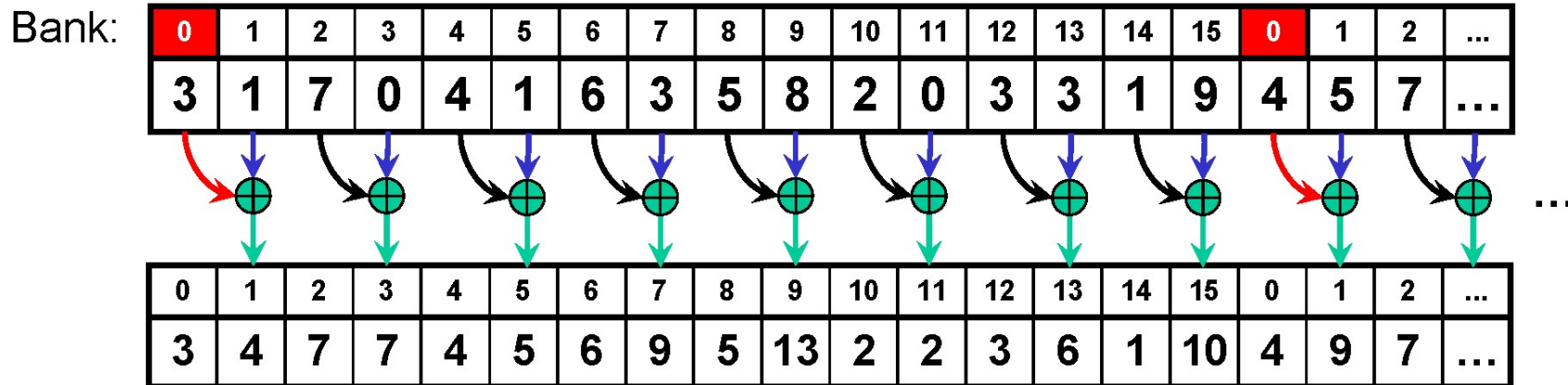
- **Each thread loads two shared mem data elements**
- **Tempting to interleave the loads**  

```
temp[2*thid]    = g_idata[2*thid];  
temp[2*thid+1] = g_idata[2*thid+1];
```
- **Threads:(0,1,2,...,8,9,10,...)→banks:(0,2,4,...,0,2,4,...)**
- **Better to load one element from each half of the array**  

```
temp[thid]      = g_idata[thid];  
temp[thid + (n/2)] = g_idata[thid + (n/2)];
```

# Bank Conflicts in the tree algorithm

- When we build the sums, each thread reads two shared memory locations and writes one:
- Th(0,8) access bank 0



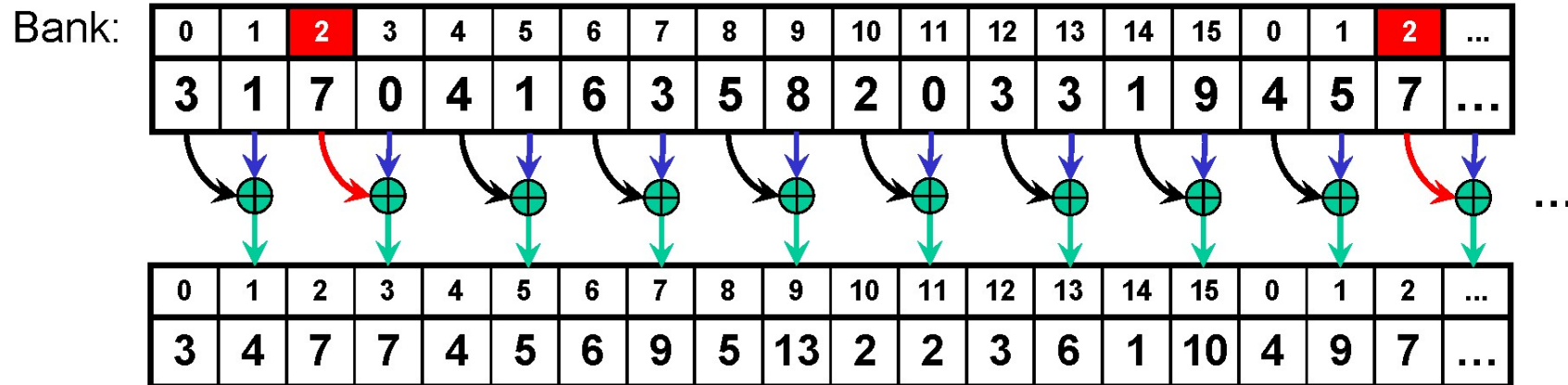
First iteration: 2 threads access each of 8 banks.

Each  corresponds to a single thread.

Like-colored arrows represent simultaneous memory accesses

# Bank Conflicts in the tree algorithm

- When we build the sums, each thread reads two shared memory locations and writes one:
- Th(1,9) access bank 2, etc.



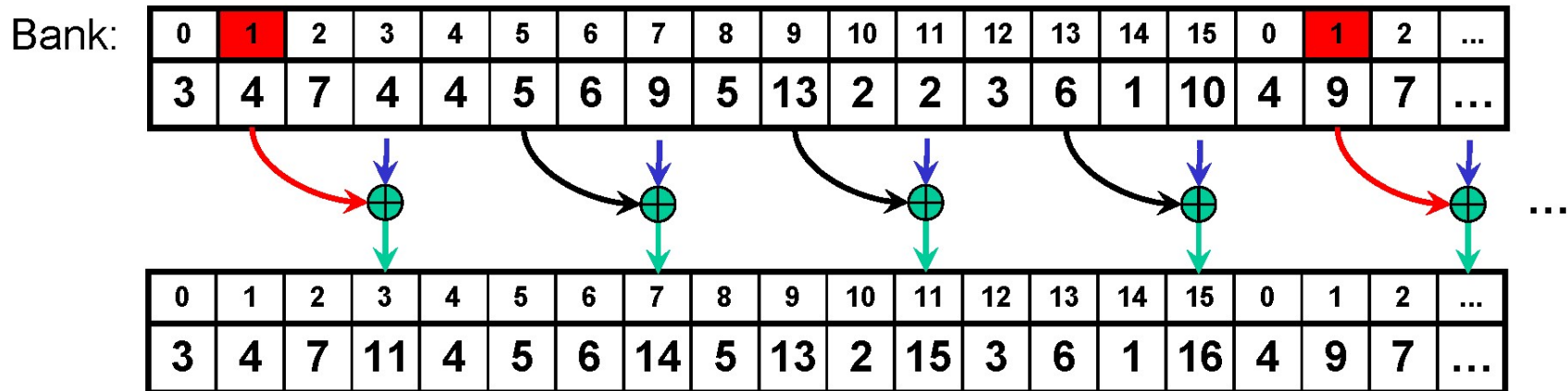
First iteration: 2 threads access each of 8 banks.

Each  corresponds to a single thread.


Like-colored arrows represent simultaneous memory accesses

# Bank Conflicts in the tree algorithm

- **2<sup>nd</sup> iteration: even worse!**
  - 4-way bank conflicts; for example:  
Th(0,4,8,12) access bank 1, Th(1,5,9,13) access Bank 5, etc.



2<sup>nd</sup> iteration: 4 threads access each of 4 banks.

Each  corresponds to a single thread.

Like-colored arrows represent simultaneous memory accesses

# Scan Bank Conflicts (1)

- **A full binary tree with 64 leaf nodes:**

Scale (s)	Thread addresses																																							
1	0	2	4	6	8	10	12	14	16	18	20	22	24	26	28	30	32	34	36	38	40	42	44	46	48	50	52	54	56	58	60	62								
2	0	4	8	12	16	20	24	28	32	36	40	44	48	52	56	60																								
4	0	8	16	24	32	40	48	56																																
8	0	16	32	48																																				
16	0	32																																						
32	0																																							
Conflicts	Banks																																							
2-way	0	2	4	6	8	10	12	14	0	2	4	6	8	10	12	14	0	2	4	6	8	10	12	14	0	2	4	6	8	10	12	14	0	2	4	6	8	10	12	14
4-way	0	4	8	12	0	4	8	12	0	4	8	12	0	4	8	12	0	4	8	12	0	4	8	12	0	4	8	12	0	4	8	12	0	4	8	12	0	4	8	12
4-way	0	8	0	8	0	8	0	8																																
4-way	0	0	0	0																																				
2-way	0	0																																						
None	0																																							

- **Multiple 2-and 4-way bank conflicts**
- **Shared memory cost for whole tree**
  - 1 32-thread warp = 6 cycles per thread w/o conflicts
    - Counting 2 shared mem reads and one write ( $s[a] += s[b]$ )
  - $6 * (2+4+4+4+2+1) = 102$  cycles
  - 36 cycles if there were no bank conflicts ( $6 * 6$ )

# Scan Bank Conflicts (2)

- It's much worse with bigger trees!
- A full binary tree with 128 leaf nodes
  - Only the last 6 iterations shown (root and 5 levels below)

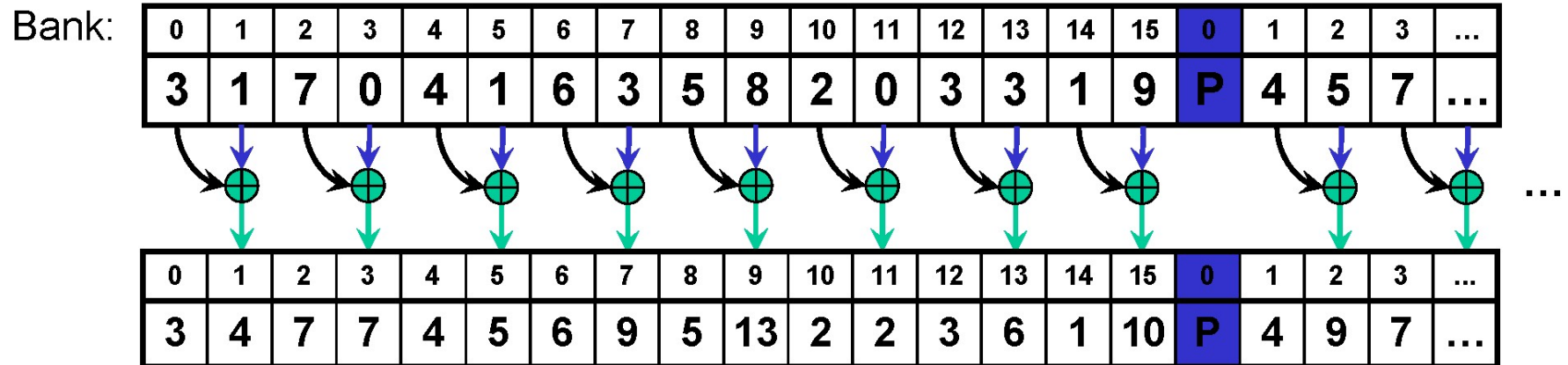
Scale (s)	Thread addresses																																			
2	0	4	8	12	16	20	24	28	32	36	40	44	48	52	56	60	64	68	72	76	80	84	88	92	96	100	104	108	112	116	120	122				
4	0	8	16	24	32	40	48	56	64	72	80	88	96	104	112	120																				
8	0	16	32	48	64	80	96	112																												
16	0	32	64	96																																
32	0	64																																		
64	0																																			
Conflicts	Banks																																			
4-way	0	4	8	12	0	4	8	12	0	4	8	12	0	4	8	12	0	4	8	12	0	4	8	12	0	4	8	12	0	4	8	12	0	4	8	10
8-way	0	8	0	8	0	8	0	8	0	8	0	8	0	8	0	8																				
8-way	0	0	0	0	0	0	0	0																												
4-way	0	0	0	0																																
2-way	0	0																																		
None	0																																			

- Cost for whole tree:
  - $12 \cdot 2 + 6 \cdot (4 + 8 + 8 + 4 + 2 + 1) = 186$  cycles
  - 48 cycles if there were no bank conflicts!  $12 \cdot 1 + (6 \cdot 6)$



# Bank Conflicts in the tree algorithm

- **We can use padding to prevent bank conflicts**
  - Just add a word of padding every 16 words:
- **No more conflicts!** 32 for full warps!



Now, within a 16-thread half-warps, all threads access different banks.

32-thread full warp!

(Note that only arrows with the same color happen simultaneously.)

# Use Padding to Reduce Conflicts

---

- This is a simple modification to the last exercise
- After you compute a shared mem address like this:

```
Address = stride * thid;
```

- Add padding like this:

```
Address += (Address >> 4); // divide by NUM_BANKS
```

- This removes most bank conflicts
  - Not all, in the case of deep trees



# Fixing Scan Bank Conflicts

---

- Insert padding every NUM\_BANKS elements

```
const int LOG_NUM_BANKS = 4; // 16 banks
int tid = threadIdx.x;
int s = 1;
// Traversal from leaves up to root
for (d = n>>1; d > 0; d >>= 1)
{
    if (thid <= d)
    {
        int a = s*(2*tid); int b = s*(2*tid+1)
        a += (a >> LOG_NUM_BANKS); // insert pad word
        b += (b >> LOG_NUM_BANKS); // insert pad word
        shared[a] += shared[b];
    }
}
```

# Fixing Scan Bank Conflicts

- **A full binary tree with 64 leaf nodes**

[illegible]

- **No more bank conflicts!**
  - However, there are ~8 cycles overhead for addressing
    - For each  $s[a] += s[b]$  (8 cycles/iter. \* 6 iter. = 48 extra cycles)
  - So just barely worth the overhead on a small tree
    - 84 cycles vs. 102 with conflicts vs. 36 optimal

# Fixing Scan Bank Conflicts

- **A full binary tree with 128 leaf nodes**
  - Only the last 6 iterations shown (root and 5 levels below)

Scale (s)	Thread addresses																															
2	0	4	8	12	17	21	25	29	34	38	42	46	51	55	59	63	68	72	76	80	85	89	93	97	102	106	110	114	119	123	127	131
4	0	8	17	25	34	42	51	59	68	76	85	93	102	110	119	127																
8	0	17	34	51	68	85	102	119																								
16	0	34	68	102																												
32	0	68																														
64	0																															

- **No more bank conflicts!**
  - Significant performance win:
    - 106 cycles vs. 186 with bank conflicts vs. 48 optimal

# Fixing Scan Bank Conflicts

- **A full binary tree with 512 leaf nodes**
  - Only the last 6 iterations shown (root and 5 levels below)

Scale (s)	Thread addresses																															
8	0	17	34	51	68	85	102	119	136	153	170	187	204	221	238	255	272	289	306	323	340	357	374	391	408	425	442	459	476	493	510	527
16	0	34	68	102	136	170	204	238	272	306	340	374	408	442	476	510																
32	0	68	136	204	272	340	408	476																								
64	0	136	272	408																												
128	0	272																														
256	0																															
</																																

- **Wait, we still have bank conflicts**
  - Method is not foolproof, but still much improved
  - 304 cycles vs. 570 with bank conflicts vs. 120 optimal
- **But it does not pay off to optimize for the rest. Address calculations are getting too expensive**

# Summary

---

- **Parallel Programming requires careful planning**
  - of the branching behavior
  - of the memory access patterns
  - of the work efficiency
- **Vector Reduction**
  - branch efficient
  - bank efficient
- **Scan Algorithm**
  - based in Balanced Tree principle:  
bottom up, top down traversal



# CUDA Memory Continued

# 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

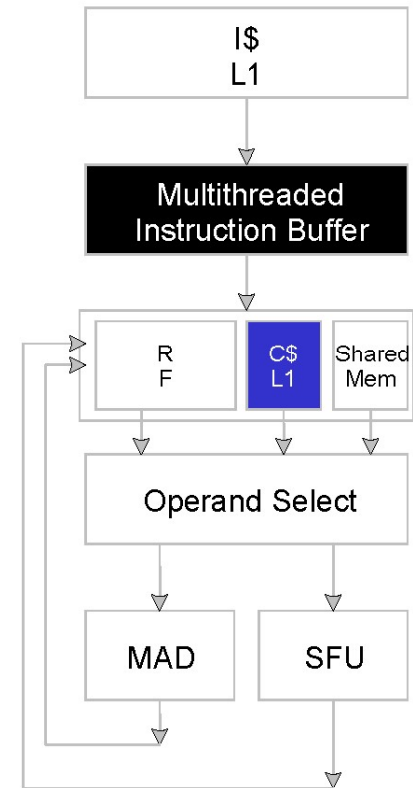
Constant memory: Constant (uniform) cache

Unified memory programming: Device/host memory sharing

# Constants

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

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





# Texture Memory

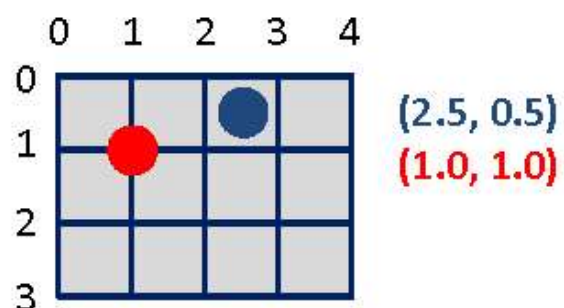
---

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

# Additional Texture Functionality

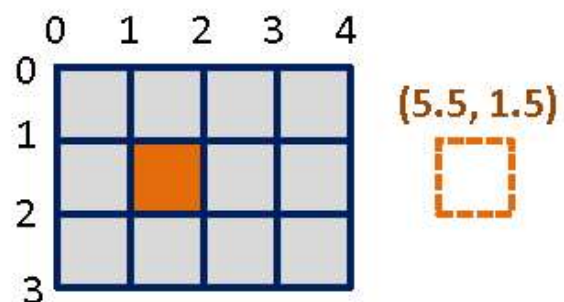
- **All of these are “free”**
  - Dedicated hardware
  - Must use CUDA texture objects
    - See CUDA Programming Guide for more details
    - Texture objects can interoperate graphics (OpenGL, DirectX)
- **Out-of-bounds index handling: clamp or wrap-around**
- **Optional interpolation**
  - Think: using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

# Examples of Texture Object Indexing

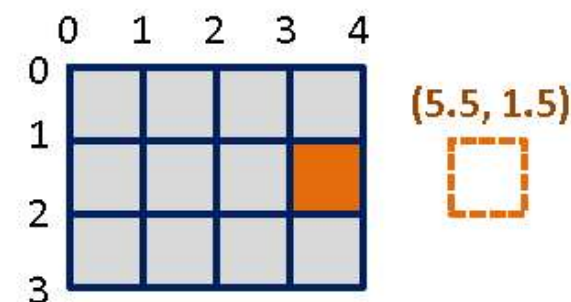


**Integer indices fall between elements**  
**Optional interpolation:**  
 Weights are determined by coordinate distance

**Index Wrap:**



**Index Clamp:**




# Native Memory Layout – Data Locality

---


## CPU

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


Input



Color coded locality  
red (near), blue (far)

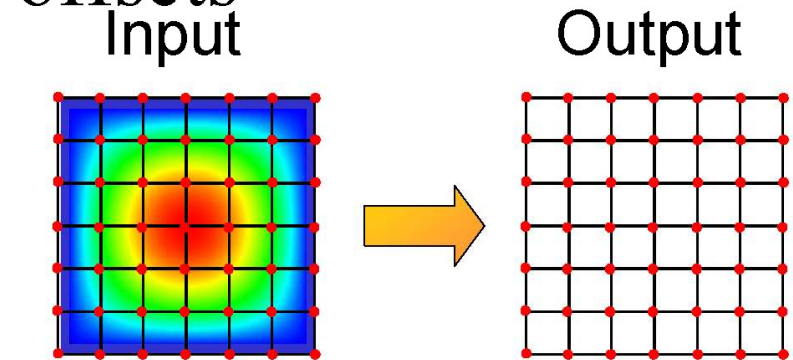


Output



## GPU

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

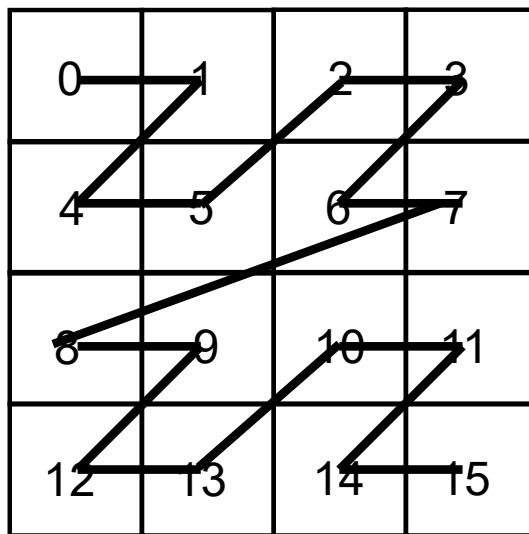


# Space-Filling Curves: Morton Order (Z Order)



Map higher-dimensional space to 1D

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



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

0	1	4	5	2	3	6	7	8	9	12	13	10	11	14	15
0000	0001	0100	0101	0010	0011	0110	0111	1000	1001	1100	1101	1010	1011	1110	1111
0000	0001	0010	0011	0100	0101	0110	0111	1000	1001	1010	1011	1100	1101	1110	1111
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

# 1D Access

---

- **Access to linear Cuda memory**

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

- **Texture reference**

- type
- access/filtering mode

```
// global texture reference
```

```
texture< float4, 1, cudaReadModeElementType> texPos;
```

- **Bind to linear array**

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

- **Within kernel**

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

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

# 2D Access

---

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

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

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

```
cudaChannelFormatDesc floatTex =  
cudaCreateChannelDesc<float4>();  
float4* src;  
cudaArray* img;  
cudaMallocArray( &img, &floatTex, w, h);  
cudaMemcpyToArray(img, 0, 0, src, w*h*sizeof(float4),  
    cudaMemcpyHostToDevice);  
cudaBindTextureToArray( texImg, img, floatTex) );  
cudaUnbindTexture(texImg);
```

# 2D Access

---

- **Within kernel**

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

- **Pros**

- optimized for 2D locality (optimized memory layout / spacefilling curve)

- **Cons**

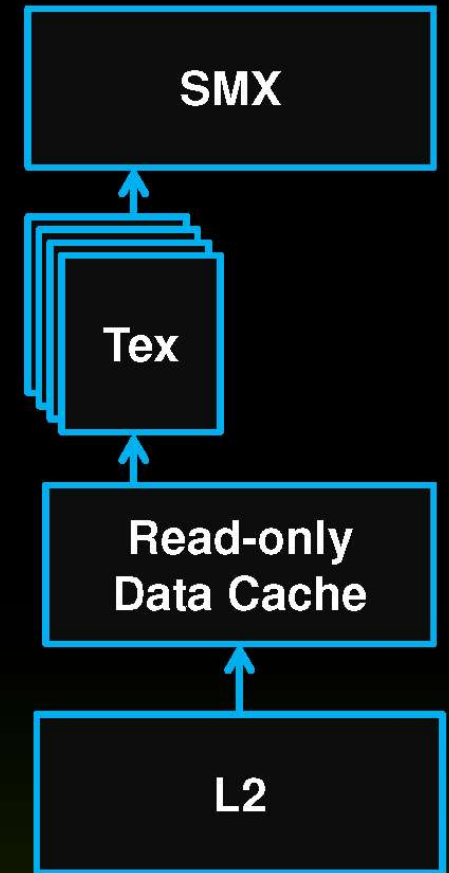
- If the result of some kernel should be used as 2D texture  
    `cudaMemcpyToArray` is required
- You cannot write to a texture which is currently read from

- **CUDA “surfaces” are writeable textures!**



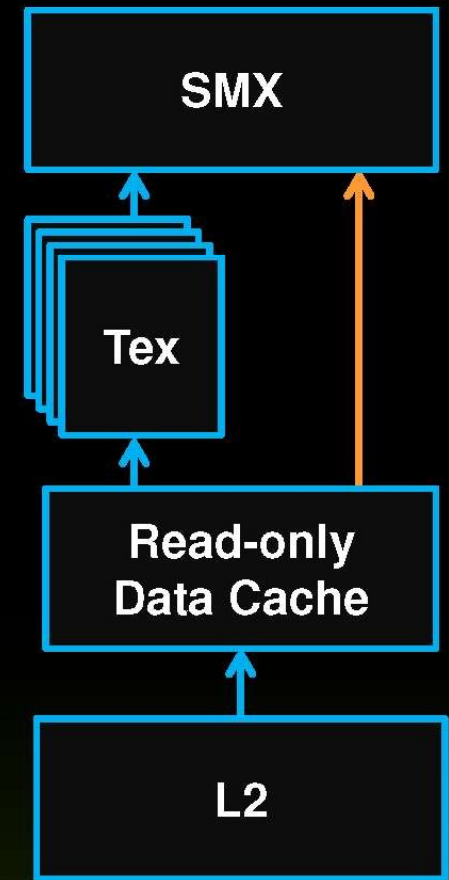
# Texture performance

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



# Texture Cache Unlocked

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





## **Global Memory Accesses**

- Memory coalescing
- Cached memory access

# Memory Layout of a Matrix in C

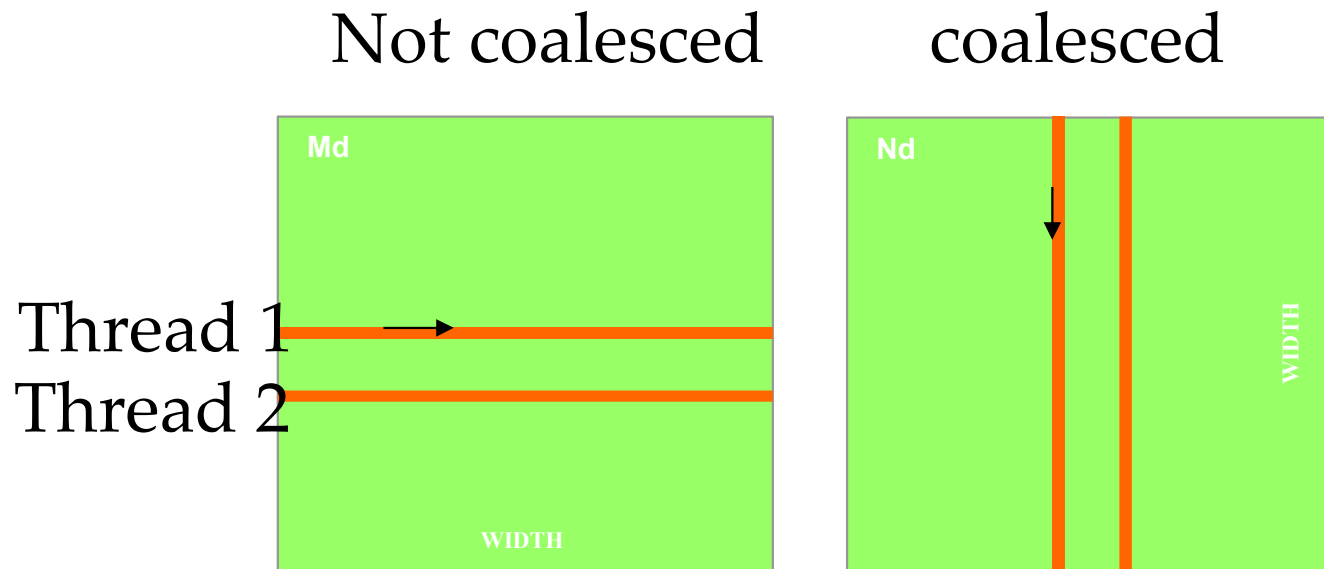
$M_{0,0}$	$M_{1,0}$	$M_{2,0}$	$M_{3,0}$
$M_{0,1}$	$M_{1,1}$	$M_{2,1}$	$M_{3,1}$
$M_{0,2}$	$M_{1,2}$	$M_{2,2}$	$M_{3,2}$
$M_{0,3}$	$M_{1,3}$	$M_{2,3}$	$M_{3,3}$

M  
↓

$M_{0,0}$	$M_{1,0}$	$M_{2,0}$	$M_{3,0}$	$M_{0,1}$	$M_{1,1}$	$M_{2,1}$	$M_{3,1}$	$M_{0,2}$	$M_{1,2}$	$M_{2,2}$	$M_{3,2}$	$M_{0,3}$	$M_{1,3}$	$M_{2,3}$	$M_{3,3}$
-----------	-----------	-----------	-----------	-----------	-----------	-----------	-----------	-----------	-----------	-----------	-----------	-----------	-----------	-----------	-----------

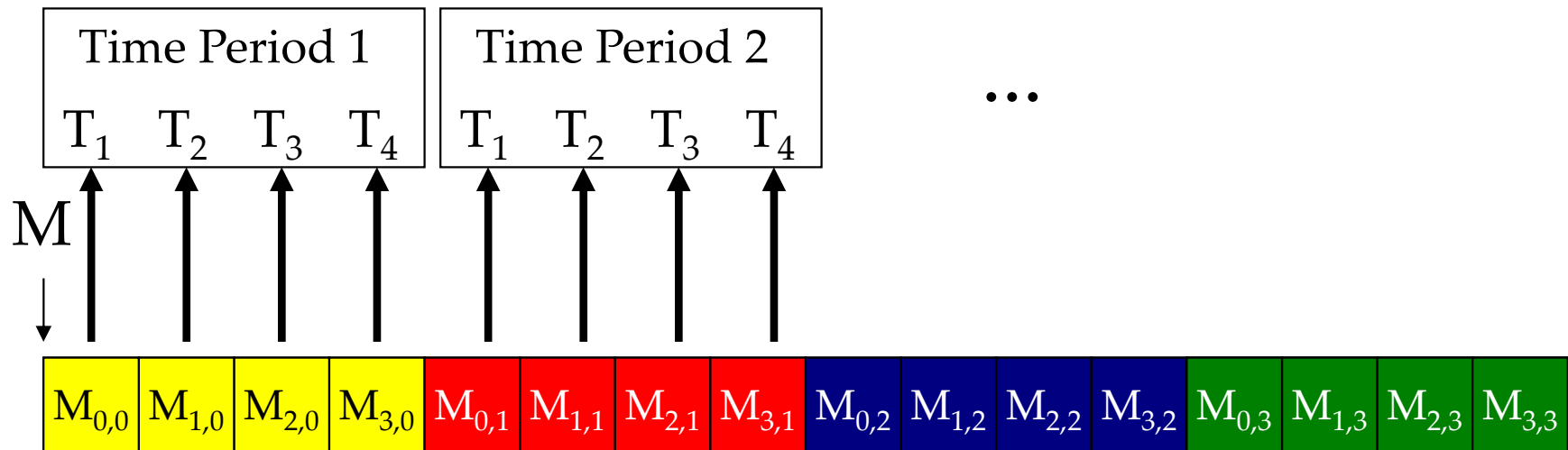
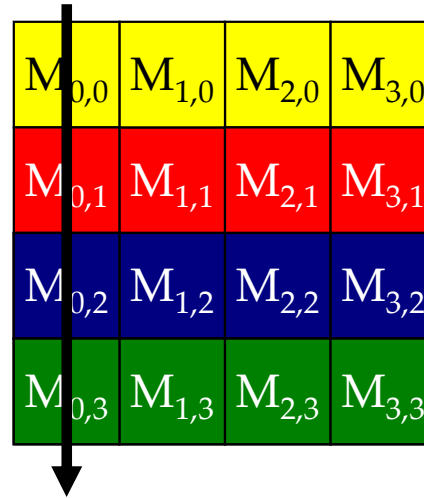
# Memory Coalescing

- When accessing global memory, peak performance utilization occurs when all threads in a half warp (**full warp on Fermi**) access continuous memory locations.
- Requirements relaxed on  $\geq 1.2$  devices; L1 cache on Fermi!

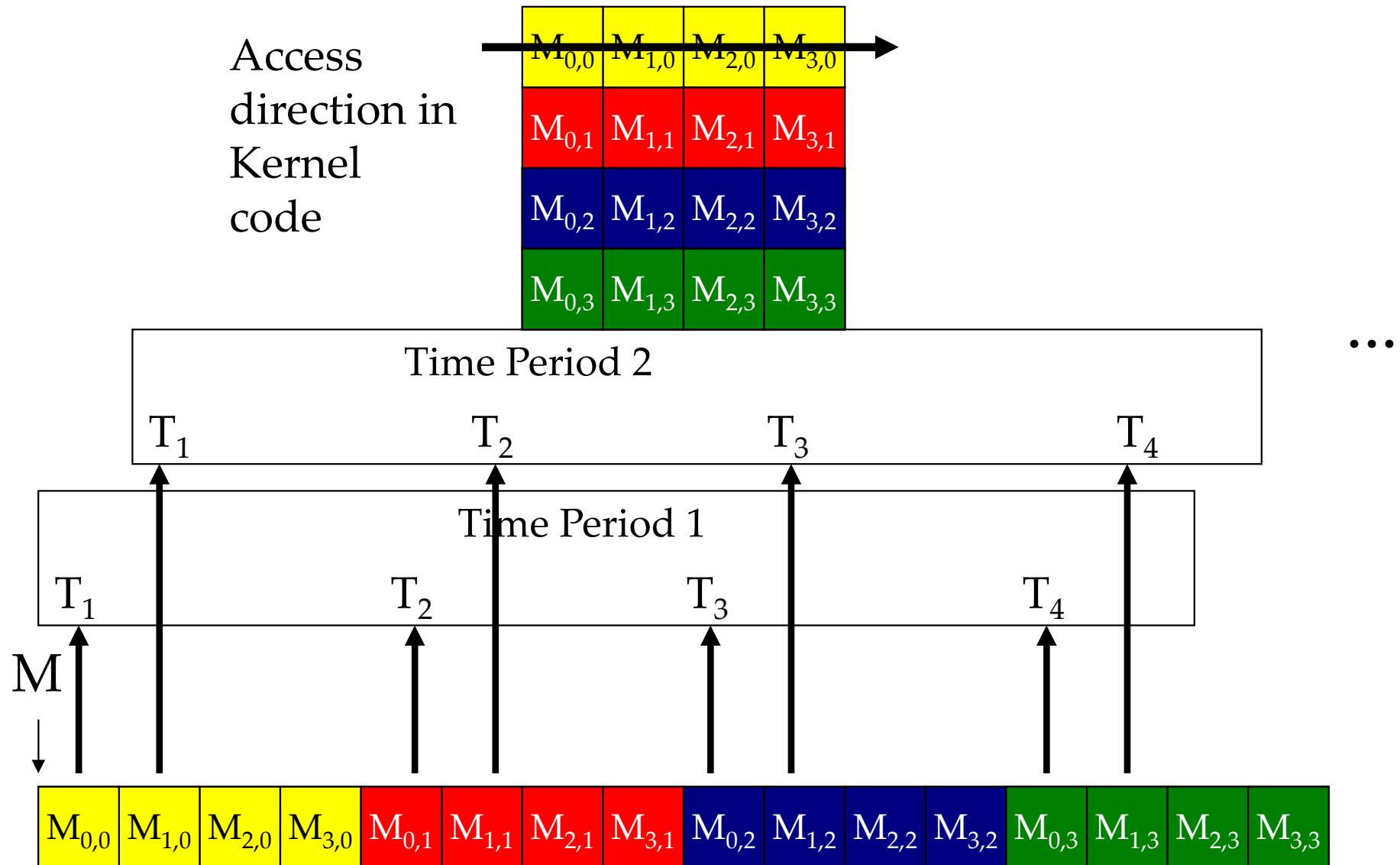


# Memory Layout of a Matrix in C

Access  
direction in  
Kernel  
code



# Memory Layout of a Matrix in C



# Global Memory Acc.

CUDA 6.5 (cc. 1.0 – 3.x)

Cached on Fermi (cc. 2.x)  
and higher

L1 cache per SM

Global L2 cache

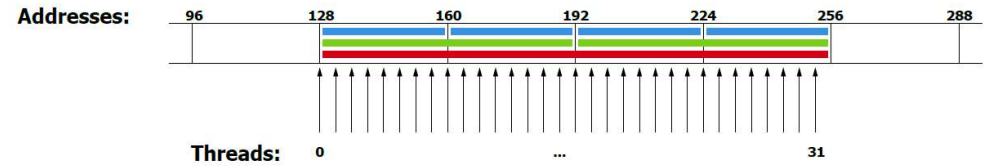
Compile time flag can choose:

- Caching in both L1 and L2
- Caching only in L2

Cache line size (L1, L2):

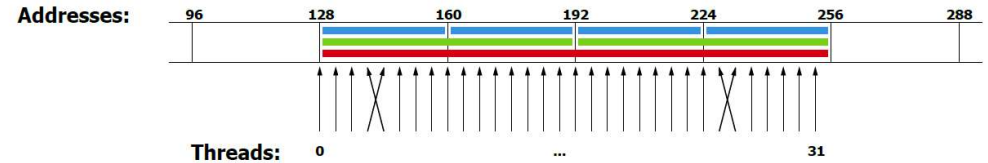
- 128 bytes

## Aligned and sequential



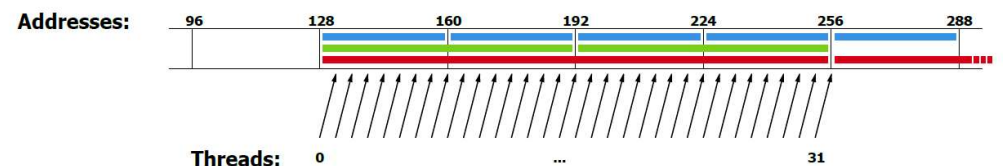
Compute capability:	1.0 and 1.1	1.2 and 1.3	2.x and 3.x	
Memory transactions:	Uncached		Uncached	Cached
	1x 64B at 128 1x 64B at 192	1x 64B at 128 1x 64B at 192	1x 32B at 128 1x 32B at 160 1x 32B at 192 1x 32B at 224	1x 128B at 128

## Aligned and non-sequential



Compute capability:	1.0 and 1.1	1.2 and 1.3	2.x and 3.x	
Memory transactions:	Uncached		Uncached	Cached
	8x 32B at 128 8x 32B at 160 8x 32B at 192 8x 32B at 224	1x 64B at 128 1x 64B at 192	1x 32B at 128 1x 32B at 160 1x 32B at 192 1x 32B at 224	1x 128B at 128

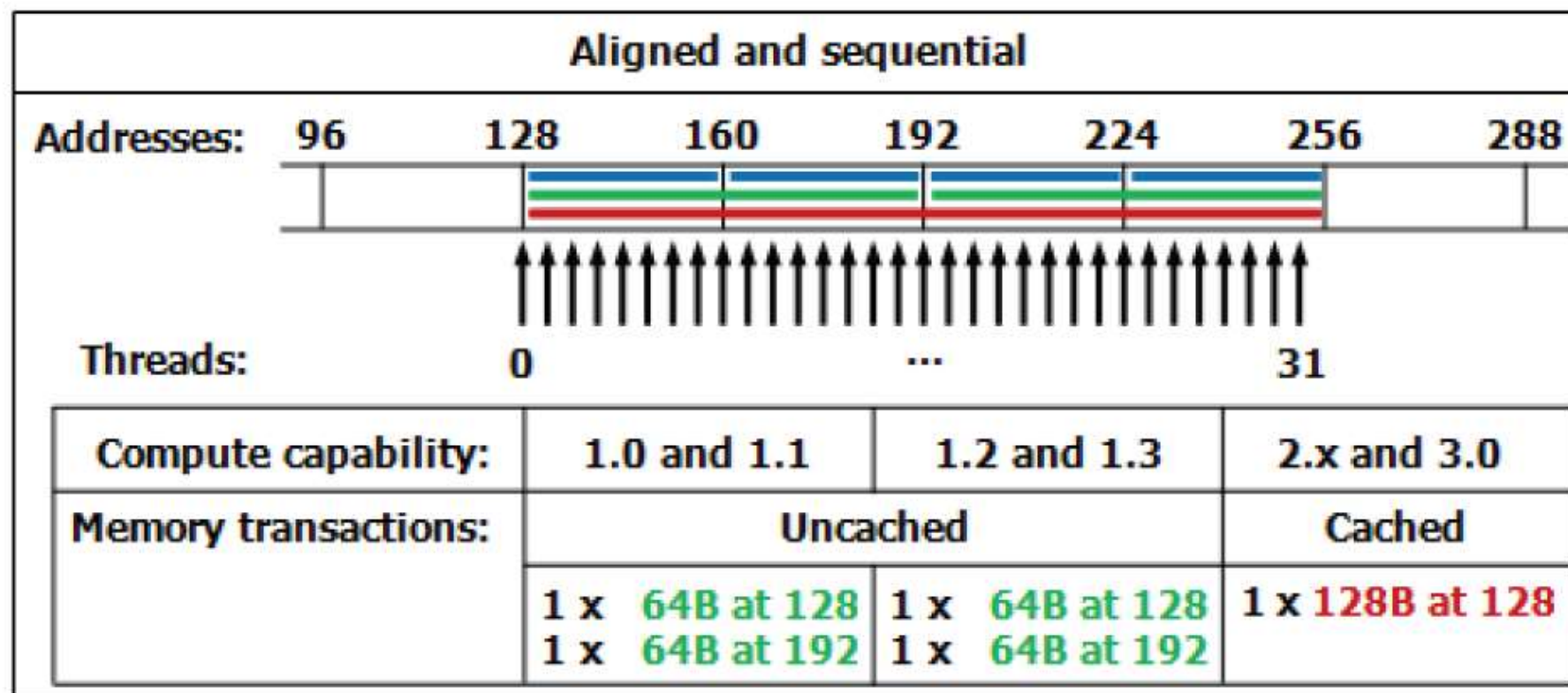
## Mis-aligned and sequential



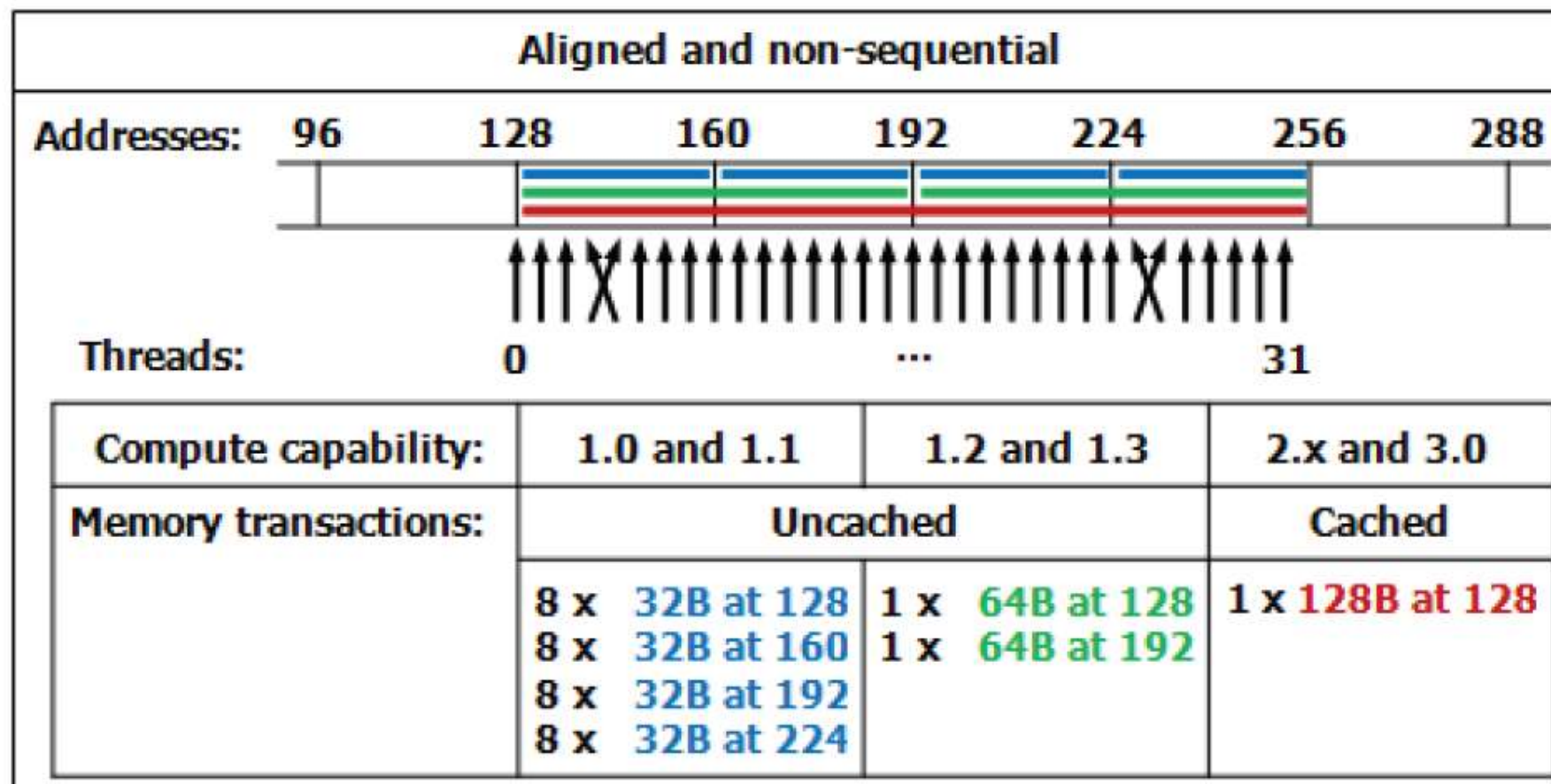
Compute capability:	1.0 and 1.1	1.2 and 1.3	2.x and 3.x	
Memory transactions:	Uncached		Uncached	Cached
	7x 32B at 128 8x 32B at 160 8x 32B at 192 8x 32B at 224 1x 32B at 256	1x 128B at 128 1x 64B at 192 1x 32B at 256	1x 32B at 128 1x 32B at 160 1x 32B at 192 1x 32B at 224 1x 32B at 256	1x 128B at 128 1x 128B at 256



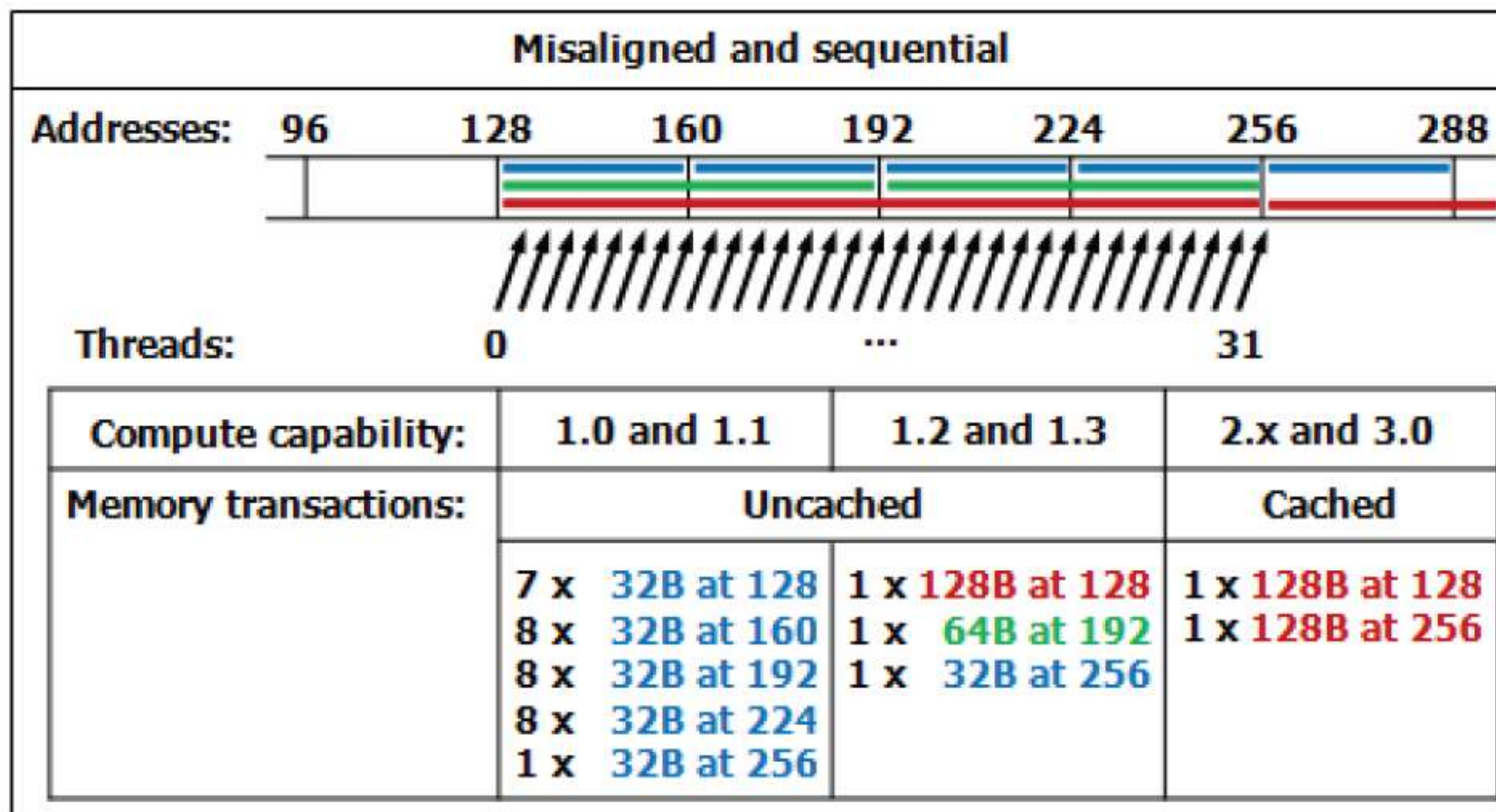
# Global Memory Access Arch. Differences (1)



# Global Memory Access Arch. Differences (2)



# Global Memory Access Arch. Differences (3)



# 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

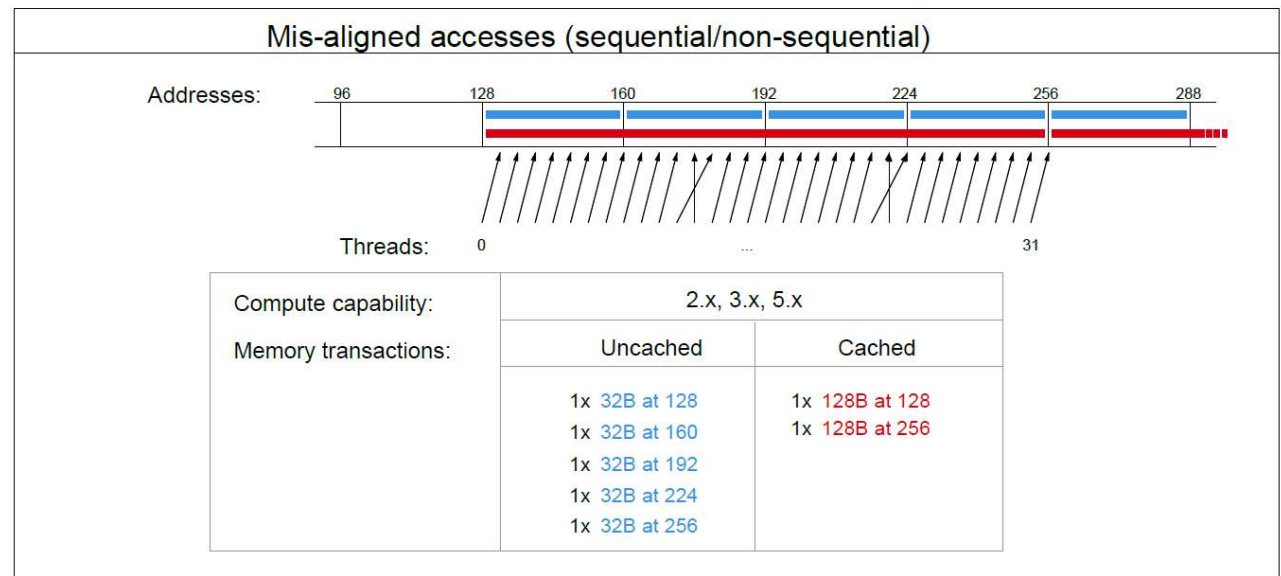
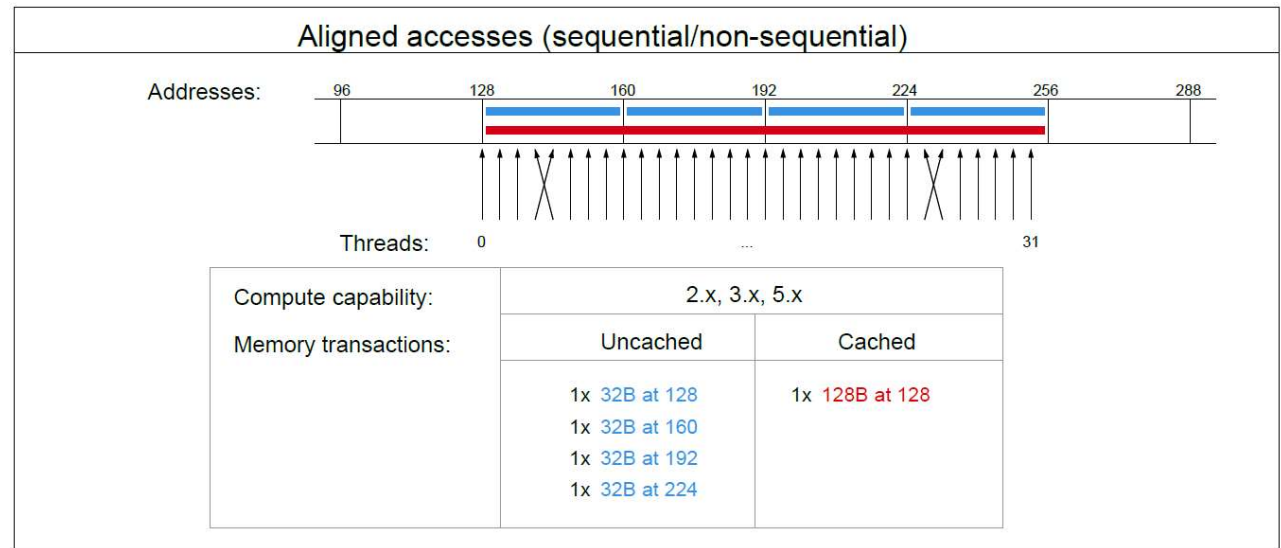


all recent  
compute capabilities  
(- 8.x)

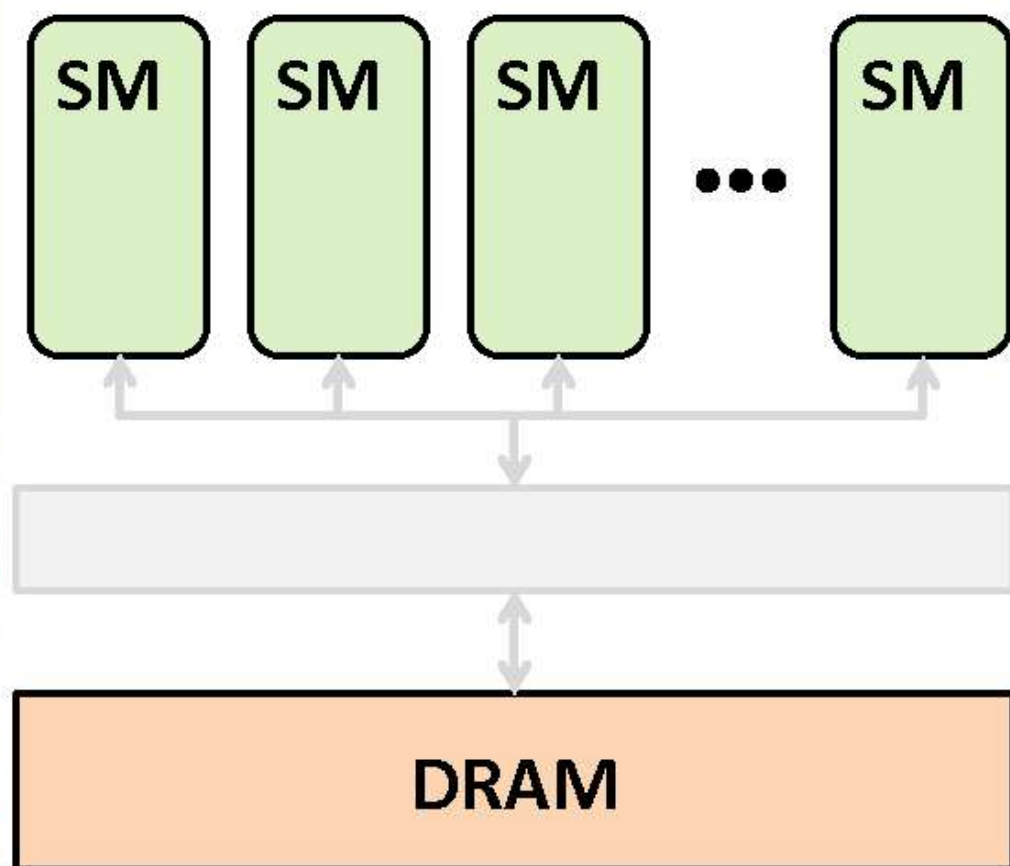
**Beware:**

*Uncached here means  
not cached in L1*

*the L2 cache is  
always used!*

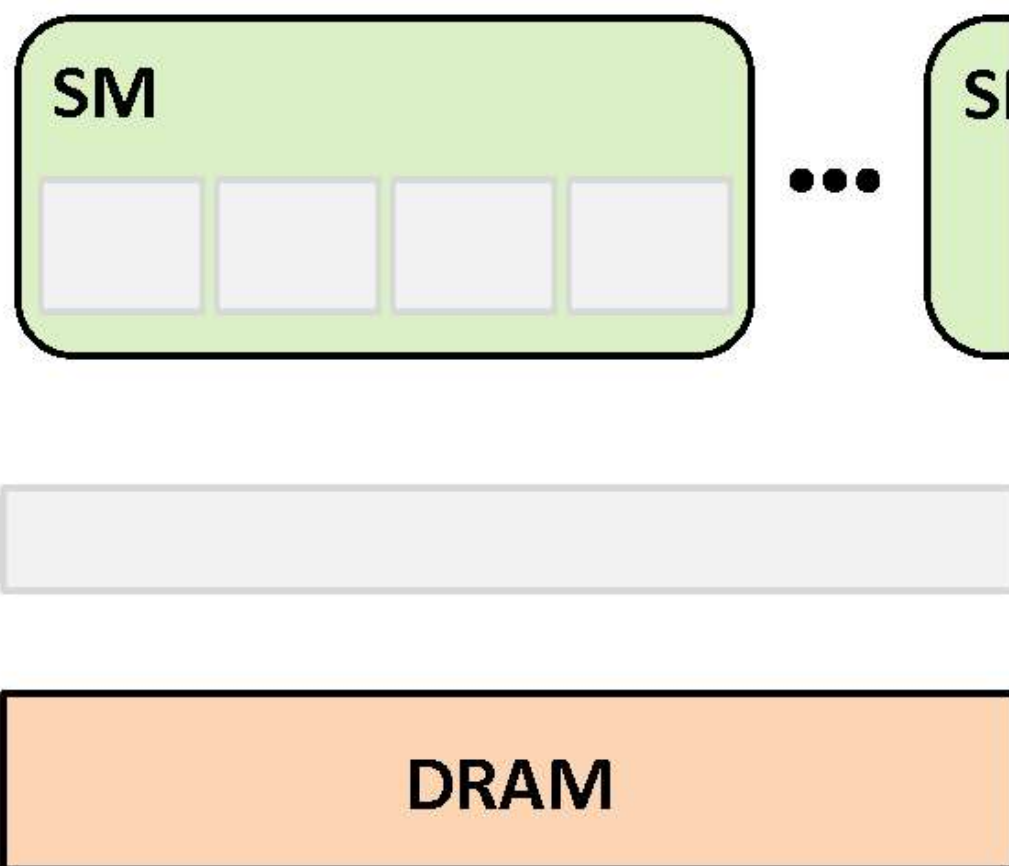


# Maximize Byte Use



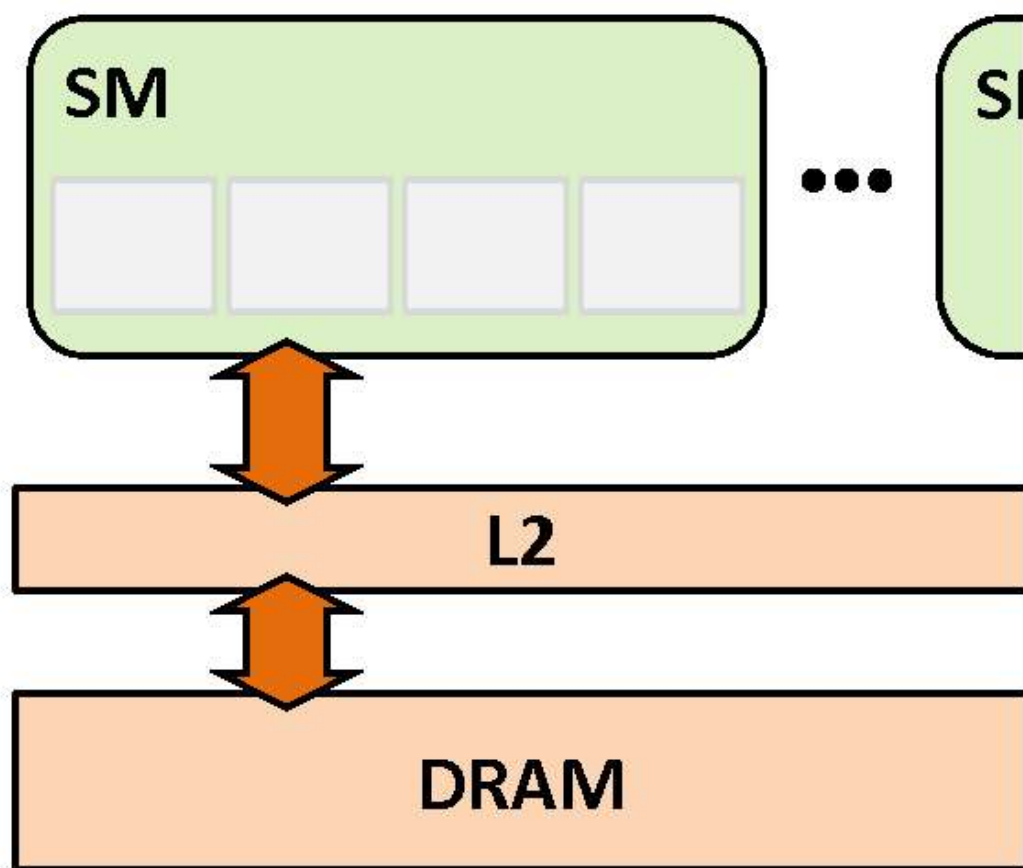
- **Two things to keep in mind:**
  - Memory accesses are per warp
  - Memory is accessed in discrete chunks
    - lines/segments
    - want to make sure that bytes that travel from DRAM to SMs get used
      - For that we should understand how memory system works
- **Note: not that different from CPUs**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

# GPU Memory System



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

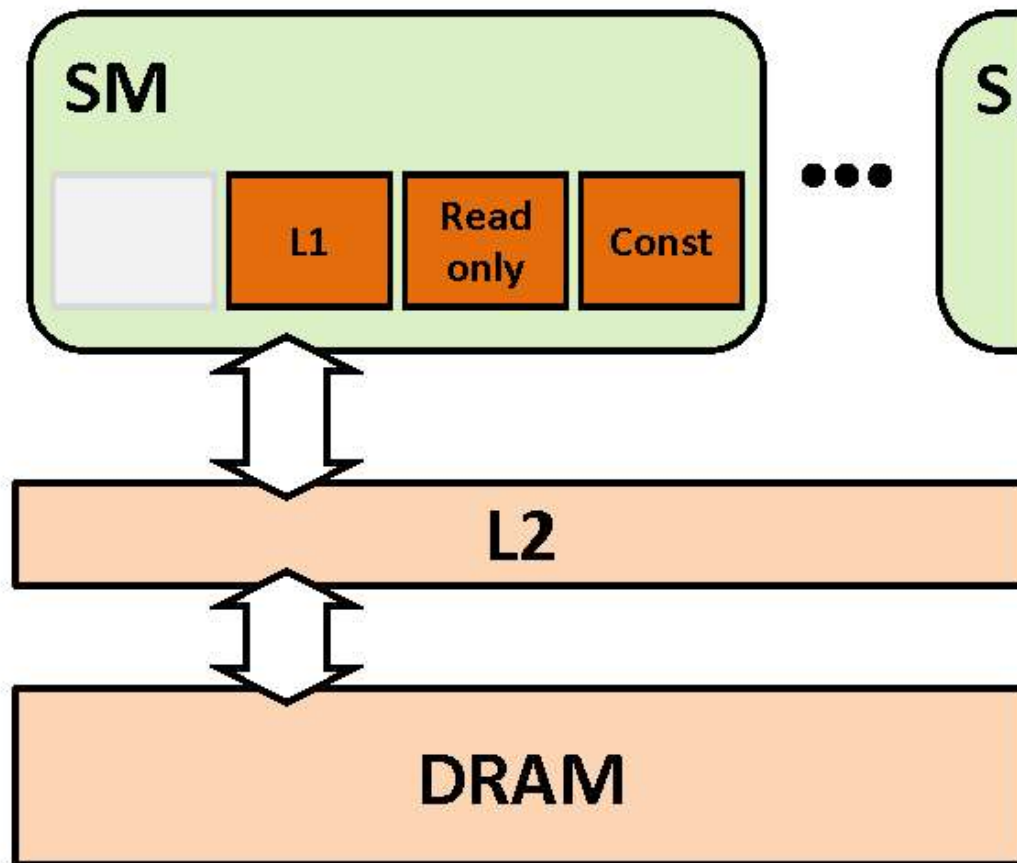
# GPU Memory System



- 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

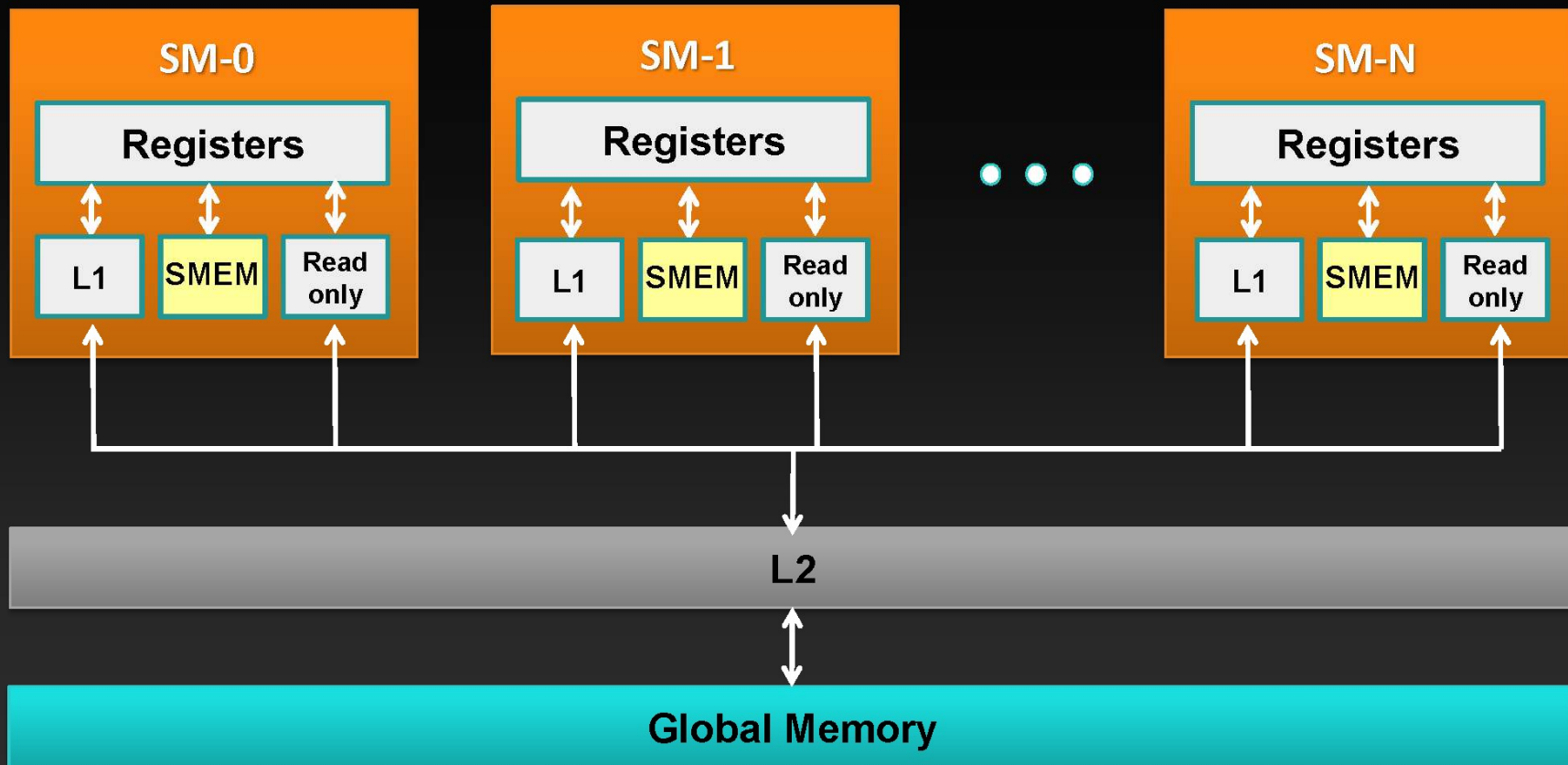
- **Not cached in the SM**
  - Invalidate the line in L1, go to L2
- **Access is at 32 B segment granularity**
- **Transaction to memory: 1, 2, or 4 segments**
  - Only the required segments will be sent
- **If multiple threads in a warp write to the same address**
  - One of the threads will “win”
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

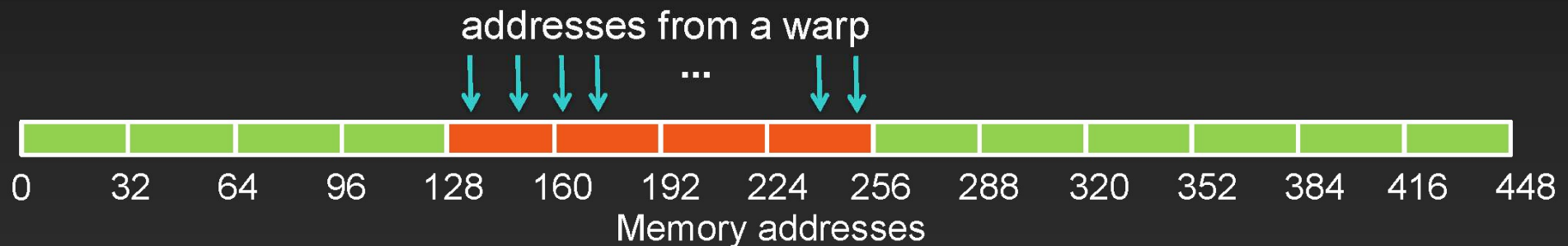
- **Two perspectives on the throughput:**
  - **Application's point of view:**
    - count only bytes requested by application
  - **HW point of view:**
    - count all bytes moved by hardware
- **The two views can be different:**
  - **Memory is accessed at 32 byte granularity**
    - **Scattered/offset 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%



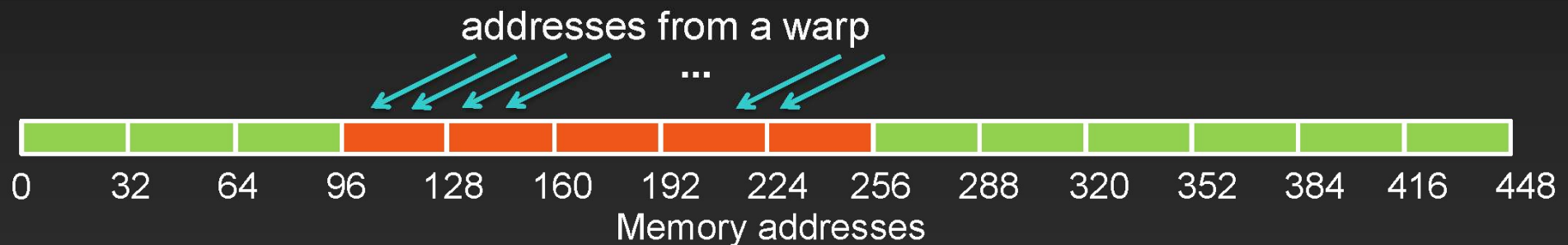
# Access Patterns vs. Memory Throughput

- **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

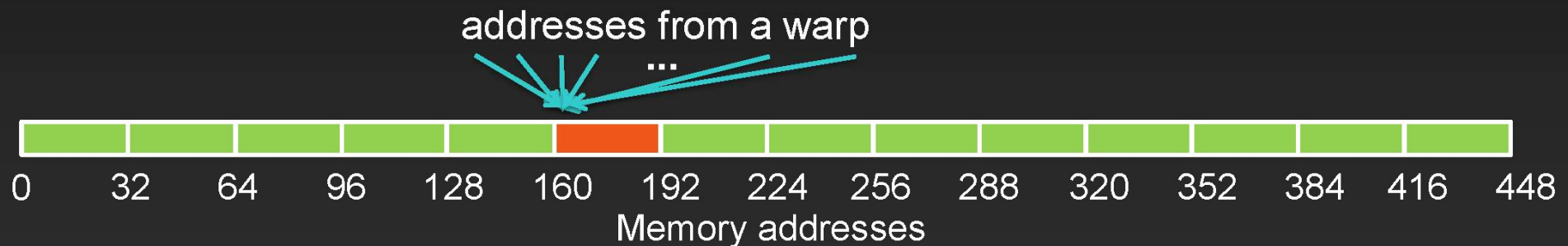
- **Scenario:**
  - Warp requests 32 misaligned, consecutive 4-byte words
- **Addresses fall within at most 5 segments**
  - Warp needs 128 bytes
  - At most 160 bytes move across the bus
  - Bus utilization: at least 80%
    - Some misaligned patterns will fall within 4 segments, so 100% utilization





# Access Patterns vs. Memory Throughput

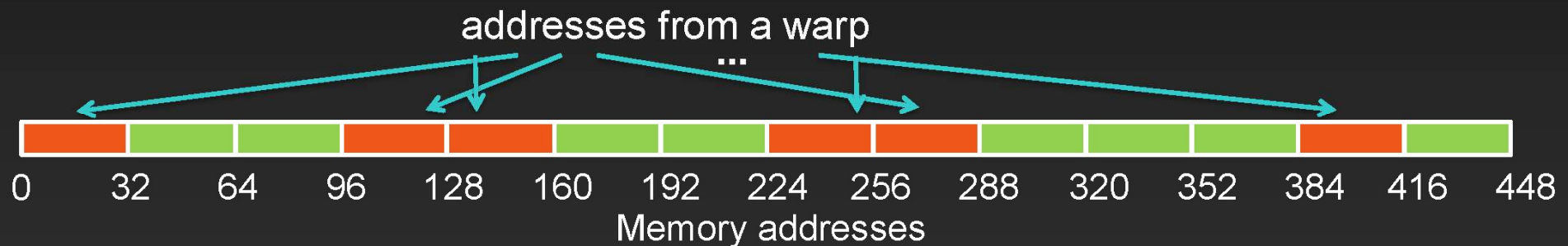
- **Scenario:**
  - All threads in a warp request the same 4-byte word
- **Addresses fall within a single segment**
  - Warp needs 4 bytes
  - 32 bytes move across the bus
  - Bus utilization: 12.5%





# 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

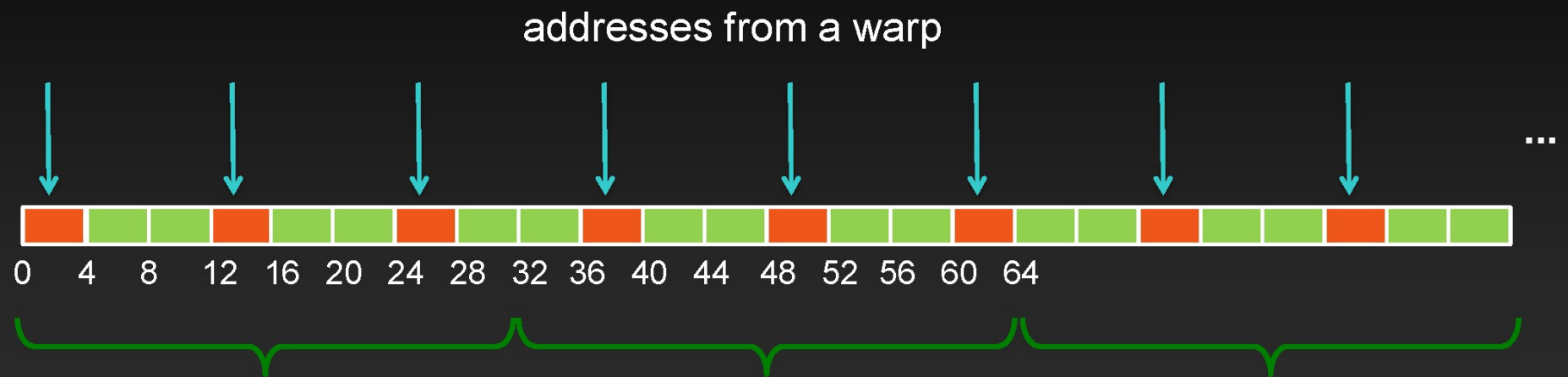
- Say we are reading a 12-byte structure per thread

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

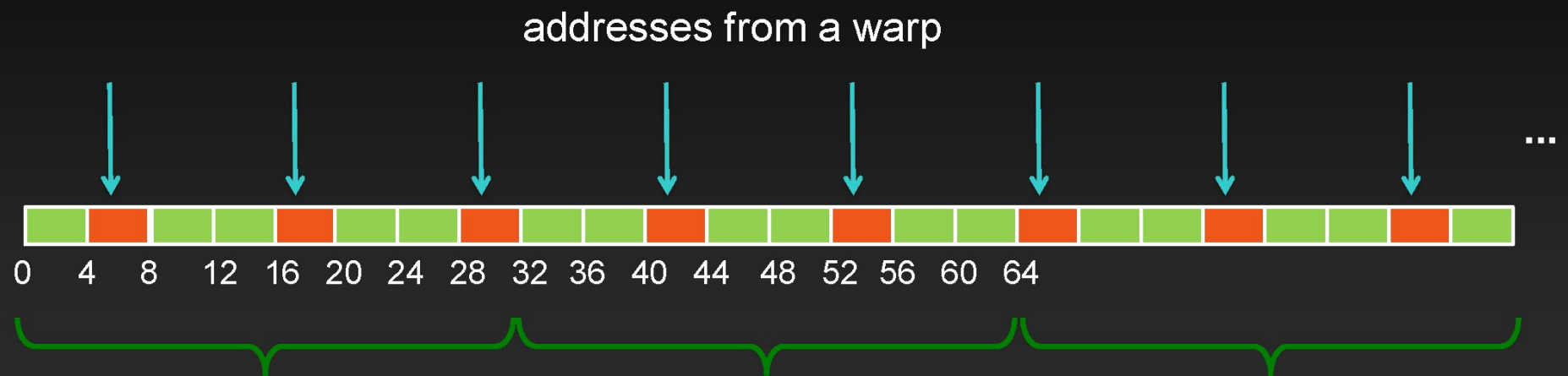
# Structure of Non-Native Size

- 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

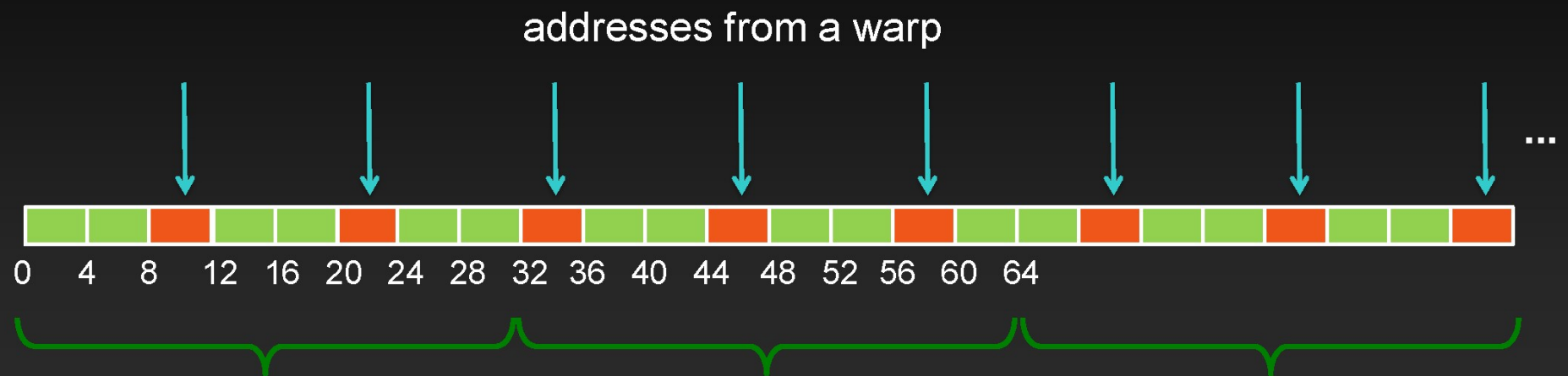
# First Load Instruction



## Second Load Instruction



# Third Load Instruction



# Performance and Solutions

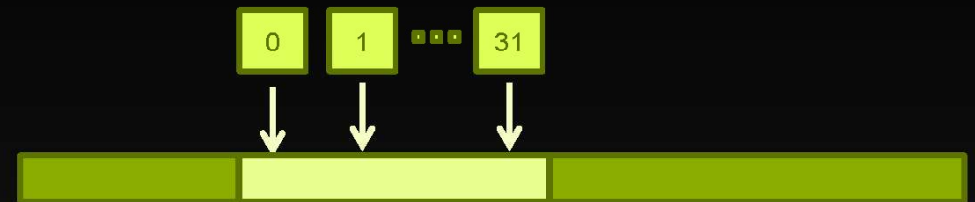
- **Because of the address pattern, we end up moving 3x more bytes than application requests**
  - We waste a lot of bandwidth, leaving performance on the table
- **Potential solutions:**
  - **Change data layout from array of structures to structure of arrays**
    - In this case: 3 separate arrays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

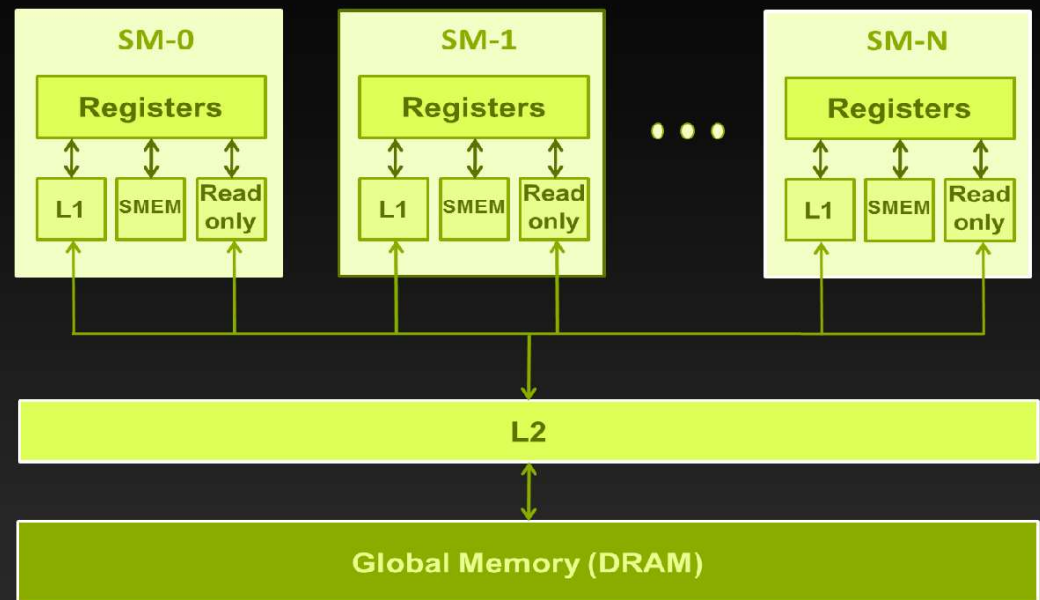


# Summary: GMEM Optimization

- **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
  - 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
  - Using **\_\_ldg()** intrinsic
    - Requires no pointer decoration

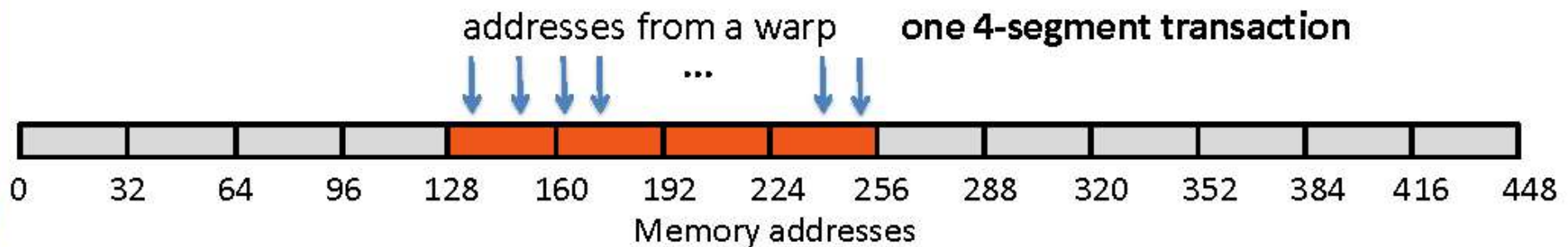
```
__global__ void kernel( int *output,  
                        int *input )  
{  
    ...  
    output[idx] = __ldg( &input[idx] );  
}
```

# 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**

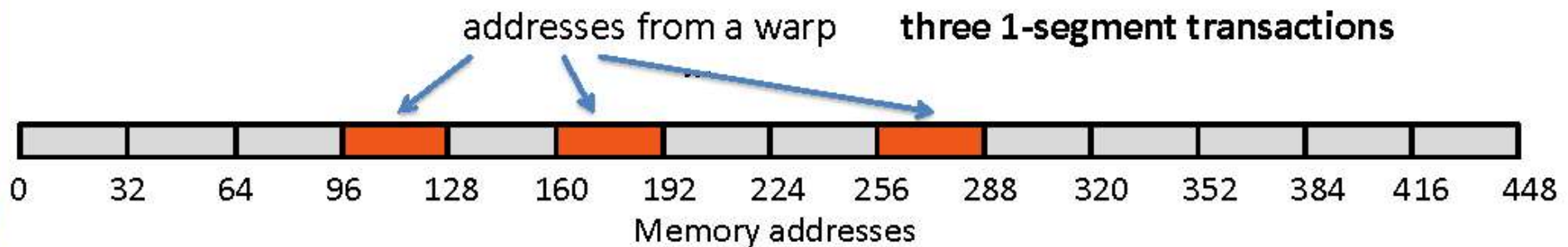


# Some Store Pattern Examples

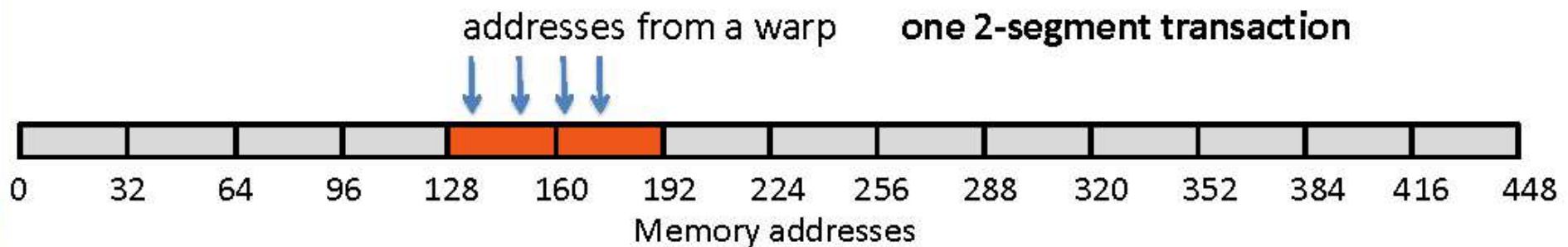




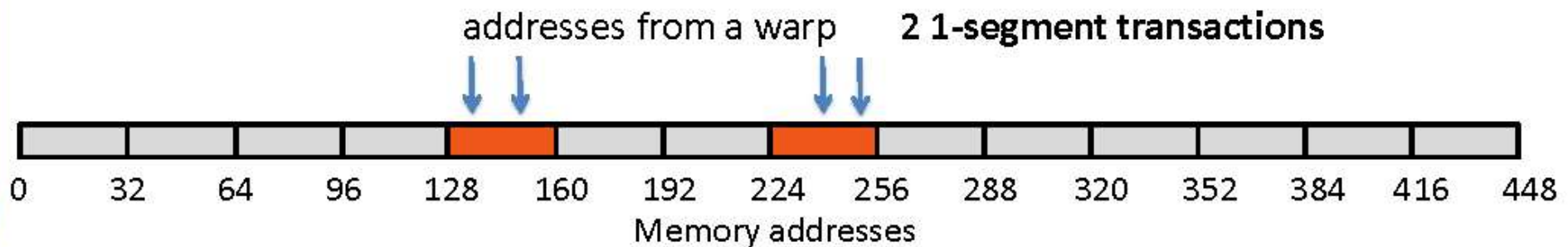
# Some Store Pattern Examples



# Some Store Pattern Examples



# Some Store Pattern Examples

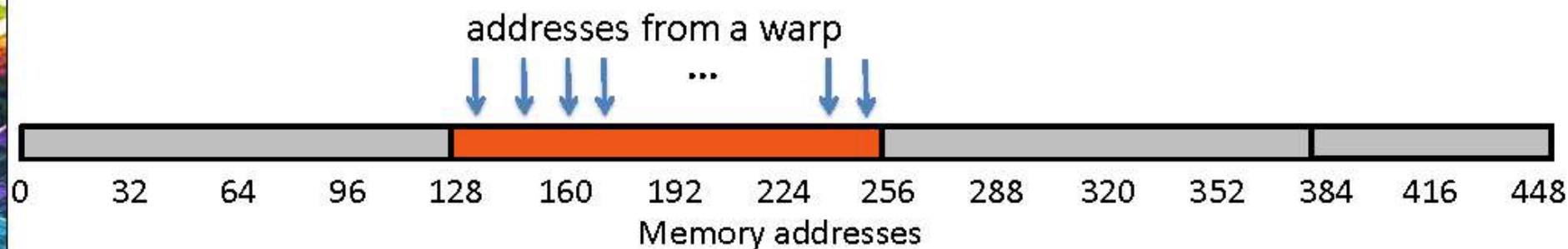


# GMEM Reads

- **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

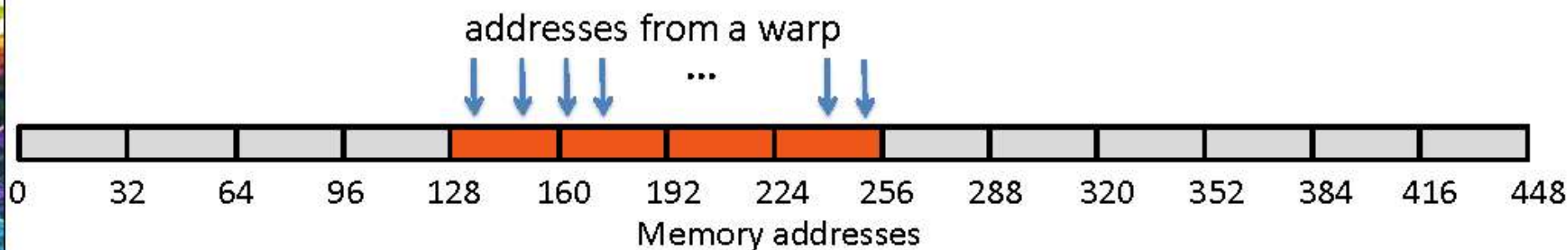
- **Scenario:**
  - Warp requests 32 aligned, consecutive 4-byte words
- **Addresses fall within 1 cache-line**
  - No replays
  - Bus utilization: 100%
    - Warp needs 128 bytes
    - 128 bytes move across the bus on a miss





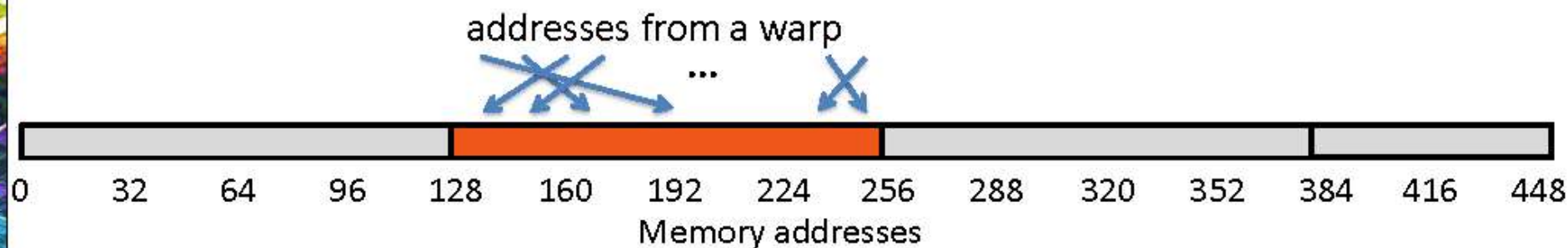
# Non-caching Load

- **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

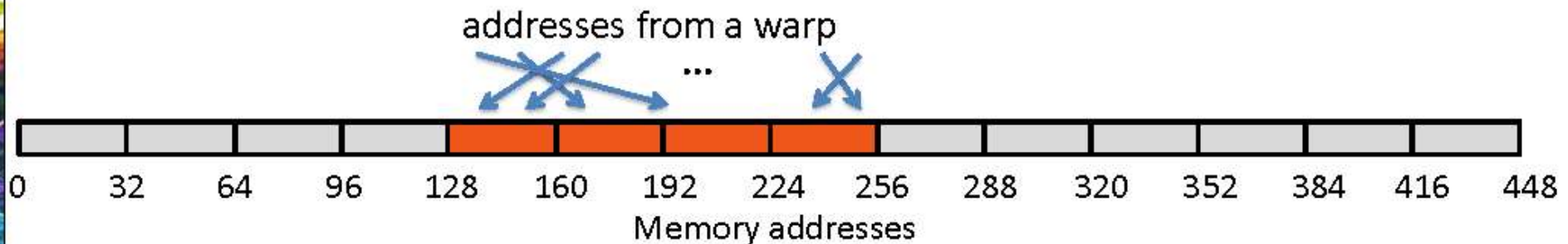
- **Scenario:**
  - Warp requests 32 aligned, permuted 4-byte words
- **Addresses fall within 1 cache-line**
  - No replays
  - Bus utilization: 100%
    - Warp needs 128 bytes
    - 128 bytes move across the bus on a miss





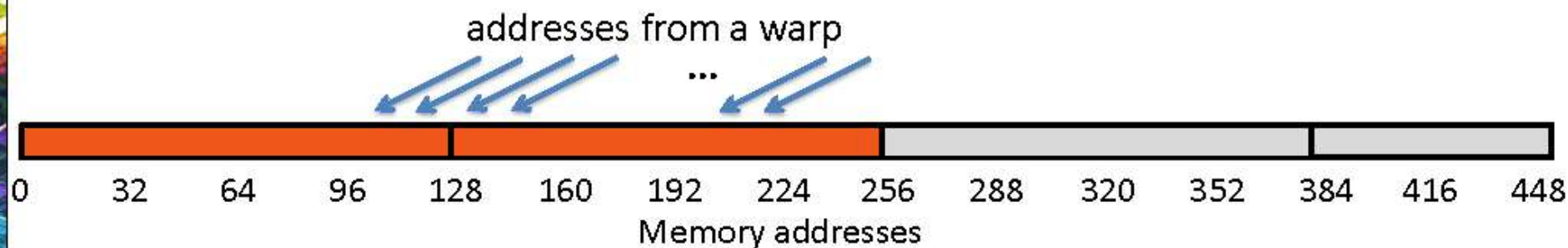
# Non-caching Load

- **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



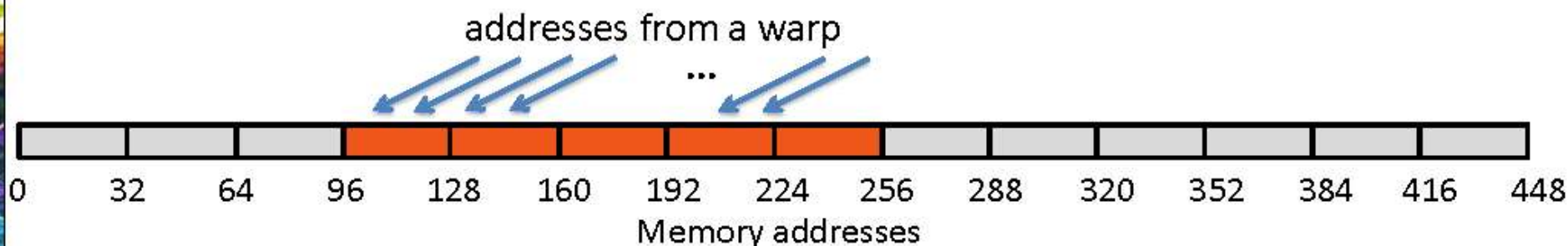
# Caching Load

- **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



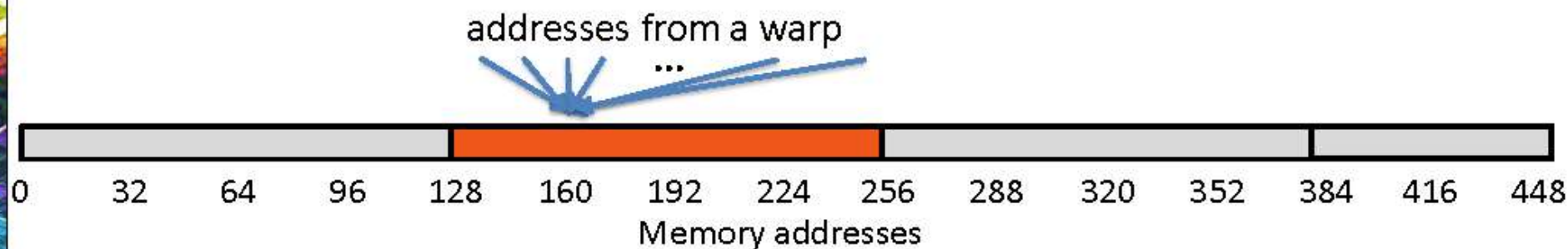
# Non-caching Load

- **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



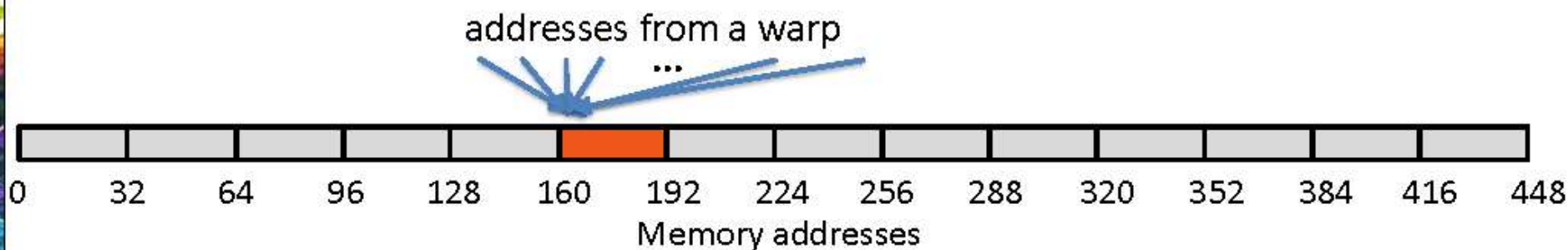
# Caching Load

- **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

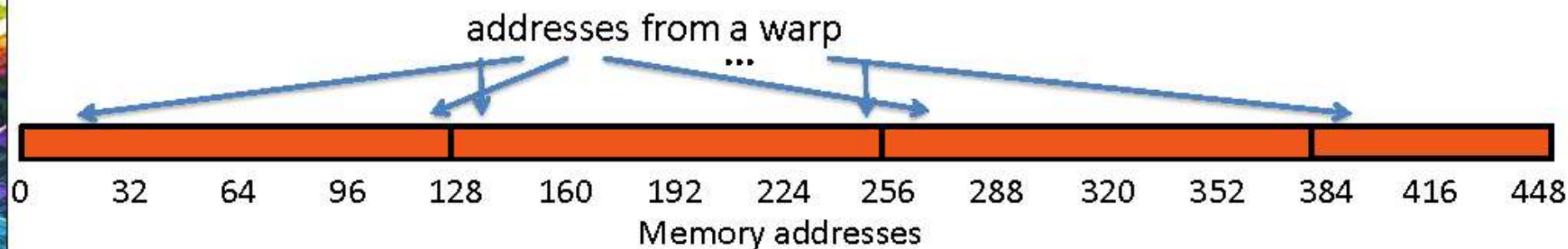
- **Scenario:**
  - All threads in a warp request the same 4-byte word
- **Addresses fall within a single segment**
  - No replays
  - Bus utilization: 12.5%
    - Warp needs 4 bytes
    - 32 bytes move across the bus on a miss





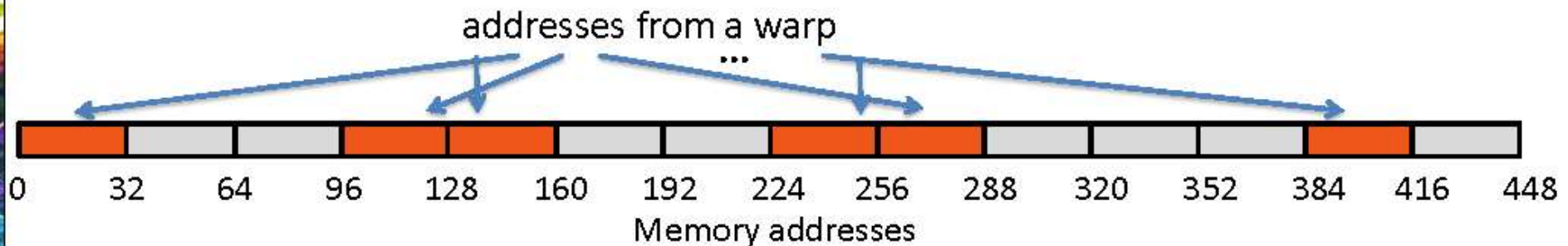
# Caching Load

- **Scenario:**
  - Warp requests 32 scattered 4-byte words
- **Addresses fall within  $N$  cache-lines**
  - $(N-1)$  replays  $(N)$  transactions
  - Bus utilization:  $32 \cdot 4B / (N \cdot 128B)$ 
    - Warp needs 128 bytes
    - $N \cdot 128$  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

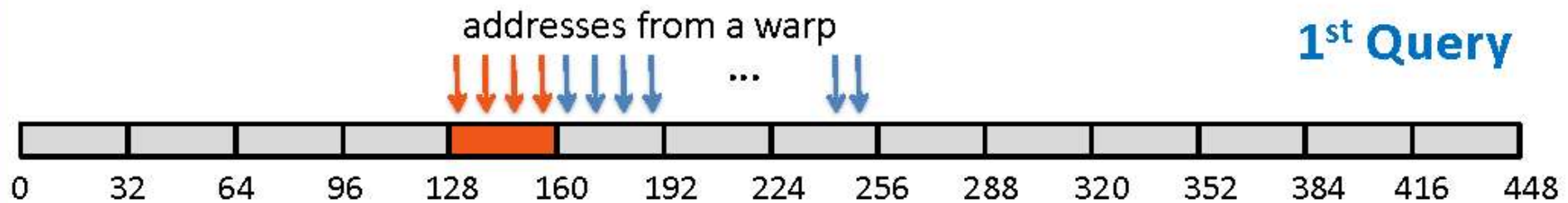
- **An alternative to L1 when accessing DRAM**
  - Also known as *texture* cache: all texture accesses use this cache
  - CC 3.5 and higher also enable global memory accesses
    - Should not be used if a kernel reads and writes to the same addresses
- **Comparing to L1:**
  - Generally better for scattered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



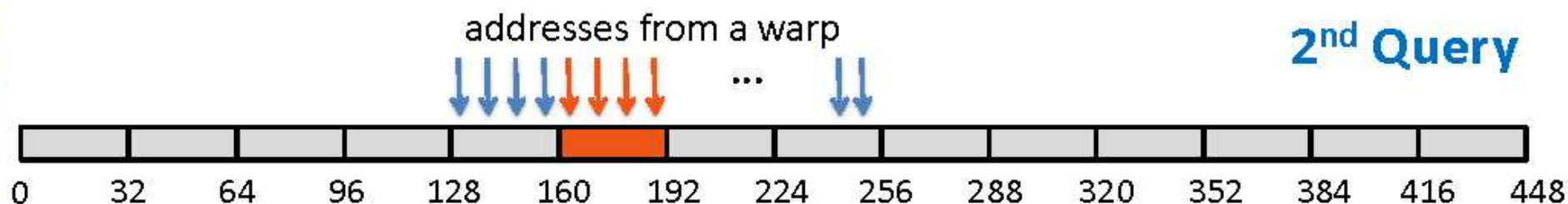
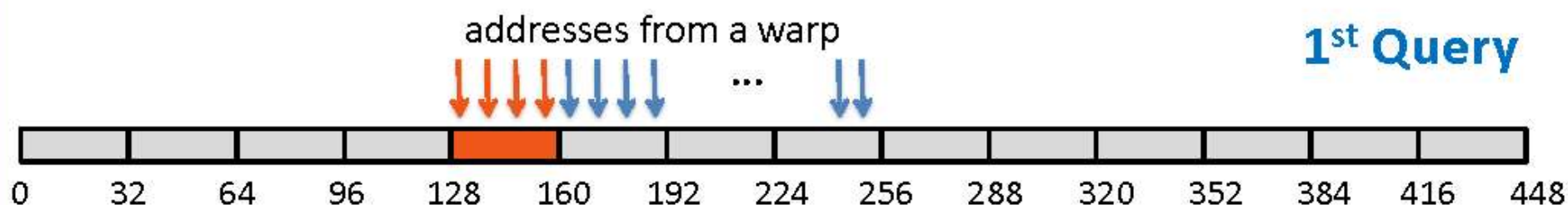
# Read-Only Cache Operation

- **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

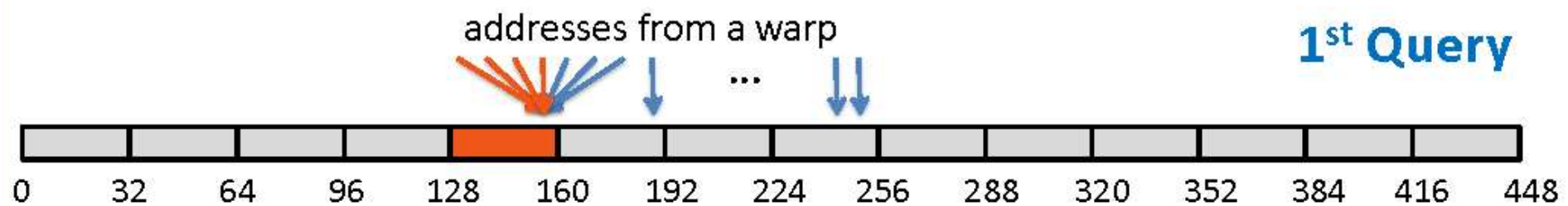
# Read-Only Cache Operation



# Read-Only Cache Operation

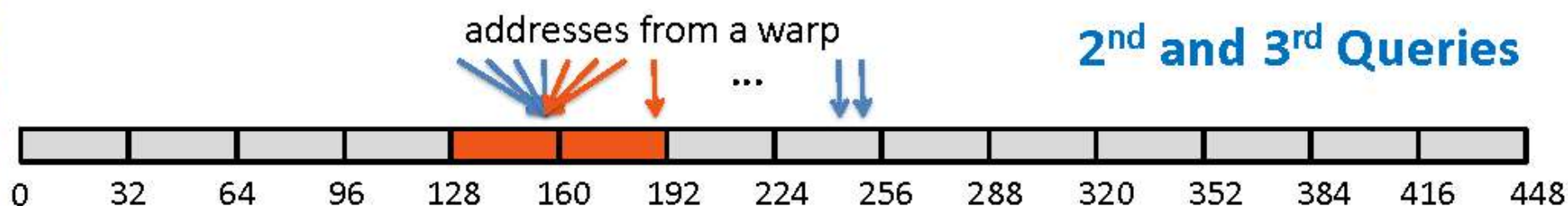
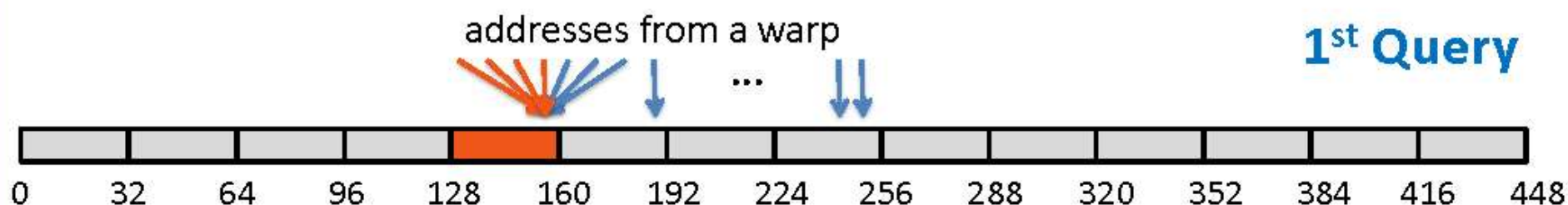


# Read-Only Cache Operation

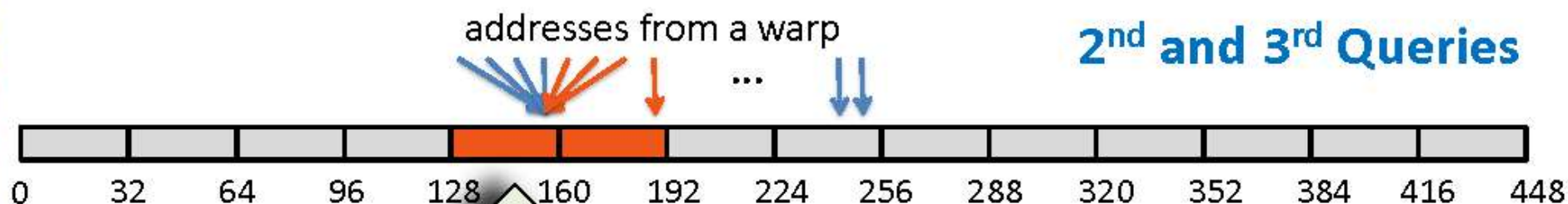
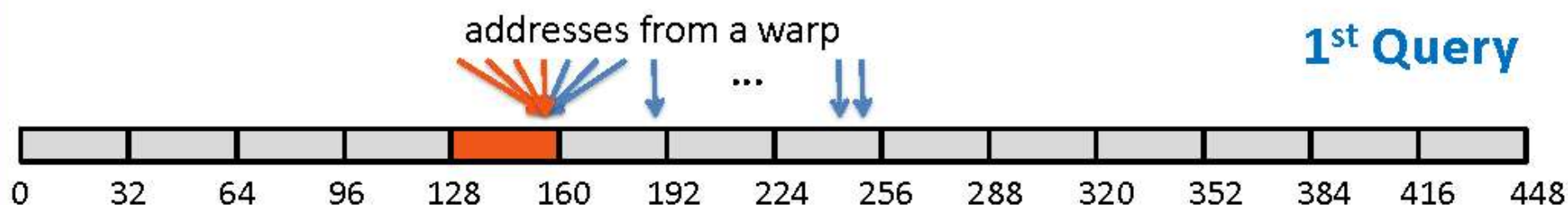




# Read-Only Cache Operation



# Read-Only Cache Operation



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

# Thank you.

- Hendrik Lensch, Robert Strzodka
- NVIDIA