



CUDA Performance Optimization

Markus Hrywniak, Senior DevTech Compute, April 2023

Outline

What you will (not) learn this session

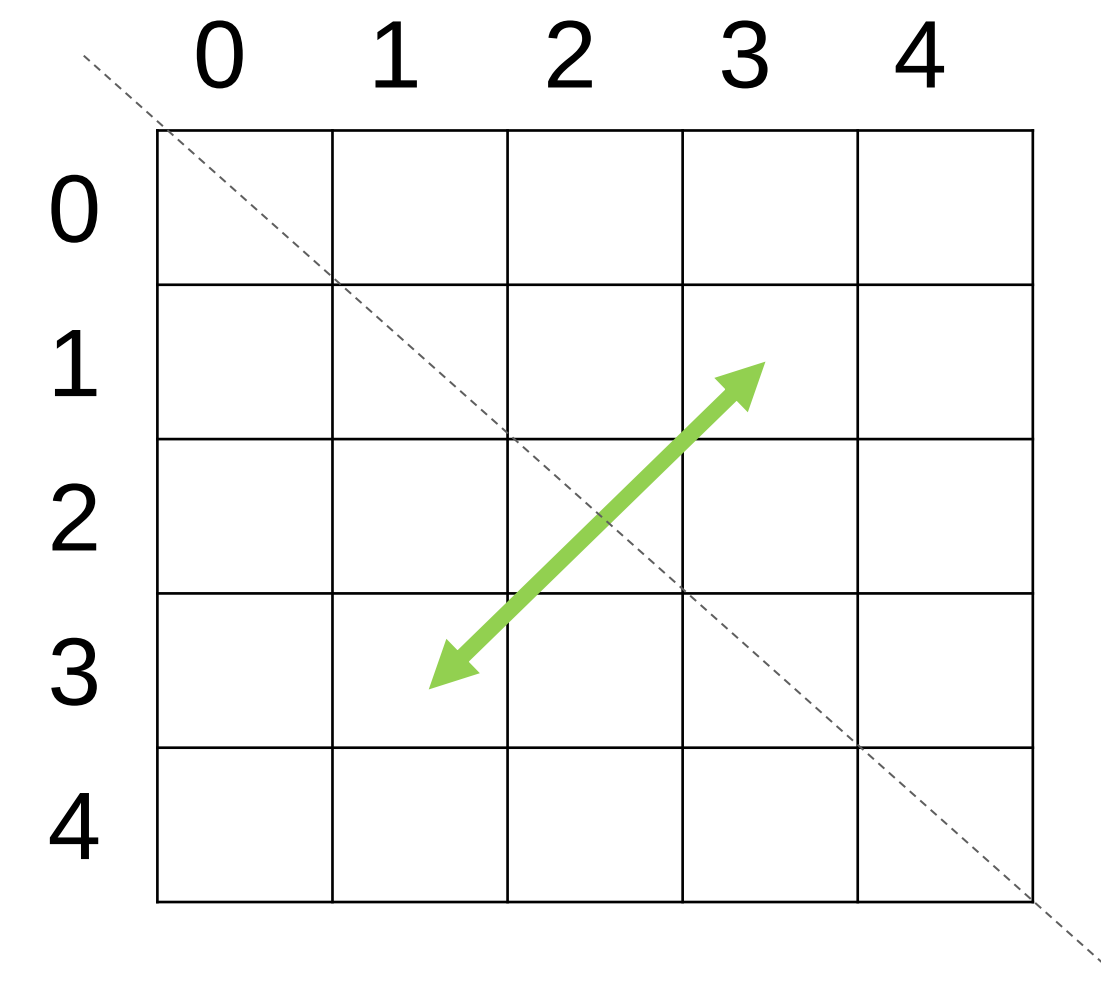
- Memory access patterns
 - Coalesced access
- Branch divergence
- Using profiling tools (some more) for Kernel level
- Also important, but not covered here:
 - Exposing enough parallelism
 - Expressing data locality
 - Application-wide profiling

Motivating Example: Matrix Transpose

No FLOPs

CPU VERSION:

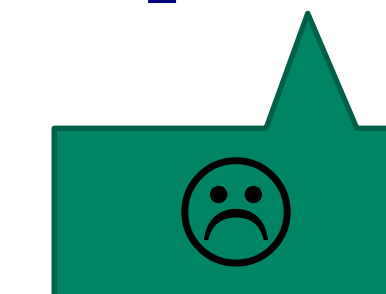
```
void transpose(
Integer* a_trans,
Integer* a,
Integer n)
{
    for (Integer row = 0; row < n; ++row)
        for (Integer col = 0; col < n; ++col)
            a_trans[col][row] = a[row][col];
}
```



GPU VERSION:

```
__global__ void transpose(
Integer* a_trans,
Integer* a,
Integer n)
{
    Integer row =
        blockIdx.y*blockDim.y+threadIdx.y;
    Integer col =
        blockIdx.x*blockDim.x+threadIdx.x;

    if (row < n && col < n)
        a_trans[col][row] = a[row][col];
}
```



Row-major
ordering

Using Nsight compute (demo)

Source Counters

All

Source metrics, including branch efficiency and sampled warp stall reasons. Sampling Data metrics are periodically sampled over the kernel runtime. They indicate when warps were stalled and couldn't be scheduled. See the documentation for a description of all stall reasons. Only focus on stalls if the schedulers fail to issue every cycle.

Branch Instructions [inst]

4.194.304

Branch Efficiency [%]

Branch Instructions Ratio [%]

0,07

Avg. Divergent Branches

Sampling Data (All)

Location	Value	Value (%)
transpose.cu:33 (0x14beeadb03e0 in transpose)	163.458	50,00
transpose.cu:15 (0x14beeadb03c0 in transpose)	111.189	33,33
transpose.cu:17 (0x14beeadb0210 in transpose)	7,684	2,30
transpose.cu:18 (0x14beeadb0240 in transpose)	5,406	1,62
transpose.cu:15 (0x14beeadb0350 in transpose)	3,437	1,03

Sampling Data (Not Issued)

Location	Value	Value (%)
transpose.cu:33 (0x14beeadb03e0 in transpose)	159.279	50,00
transpose.cu:15 (0x14beeadb03c0 in transpose)	105.609	33,33
transpose.cu:17 (0x14beeadb0210 in transpose)	7,388	2,30
transpose.cu:18 (0x14beeadb0240 in transpose)	4,984	1,62
transpose.cu:15 (0x14beeadb0350 in transpose)	2,977	1,03

Most Instructions Executed

Location	Value	Value (%)
transpose.cu:15 (0x14beeadb0200 in transpose)	2.097.152	63,33
transpose.cu:17 (0x14beeadb0210 in transpose)	2.097.152	63,33
transpose.cu:17 (0x14beeadb0220 in transpose)	2.097.152	63,33
transpose.cu:15 (0x14beeadb0230 in transpose)	2.097.152	63,33
transpose.cu:18 (0x14beeadb0240 in transpose)	2.097.152	63,33

Recommendations

Warning

Uncoalesced Global Accesses

[Warning] Uncoalesced global access, expected 16777216 sectors, got 67108864 (4.00x) at PC [0x14beeadb03c0](#) at /p/home/jusers/hrywniak1/juwels/CUDA-Course/4-Performance_Optimization/exercises/tasks/transpose.cu:15

View: Source

Source: transpose.cu Find... Navigation: L2 Sectors Global

# Source	Live Registers	Sampling Data (All)	Sampling Data (Not Issued)	Instructions Executed	Instructions On Thread
13 <code>__global__ void transpose(Integer* const a_trans, const Integer* const a, Integer* const b) {</code>		0	0		
14 <code>{</code>		0	0		
15 <code>const Integer col_block = blockIdx.x;</code>	4	438	208	4,194,304	134,217,728
16 <code>const Integer row_block = blockIdx.y;</code>	6	187	42	2,097,152	67,108,864
17 <code>const Integer block_col = threadIdx.x;</code>	3	7,719	7,388	4,194,304	134,217,728
18 <code>const Integer block_row = threadIdx.y;</code>	6	5,720	5,034	4,194,304	134,217,728
19 <code>const Integer col = col_block*BLOCK_SIZE+block_col;</code>	6	2,805	1,691	2,097,152	67,108,864
20 <code>const Integer row = row_block*BLOCK_SIZE+block_row;</code>	6	2,980	2,330	2,097,152	67,108,864
21 <code></code>		0	0		
22 <code>//TODO: declare shared memory for tile</code>		0	0		
23 <code>//_shared_ ... a_tile ...</code>		0	0		
24 <code></code>		0	0		
25 <code>if (row < n && col < n)</code>	6	2,342	333	10,485,760	268,435,456
26 <code>{</code>		0	0		
27 <code>//TODO: load tile of a into shared memory</code>		0	0		
28 <code>//TODO: call __syncthreads() to ensure all shared</code>		0	0		
29 <code>//TODO: read from a_tile with correct index:</code>		0	0		
30 <code>//a_trans[col_block*BLOCK_SIZE+block_row] * n + (</code>		0	0		
31 <code>a_trans[col*n+row] = a[row*n+col];</code>	9	119,250	109,013	31,457,280	1,006,632,960
32 <code>}</code>		0	0		
33 <code>}</code>	1	163,492	1		7,108,864
34 <code></code>		0	0		
35 <code>int main()</code>		0	0		
36 <code>{</code>		0	0		
37 <code>const Integer n = 8192;</code>		0	0		
38 <code>Integer* a = new Integer[n*n];</code>		0	0		
39 <code></code>		0	0		
40 <code>for (Integer row = 0; row < n; ++row)</code>		0	0		

Total Sample Count: 109013

0.02% Dispatch Stall (18)

0.20% Math Pipe Throttle (223)

0.24% Wait (264)

0.36% Mio Throttle (397)

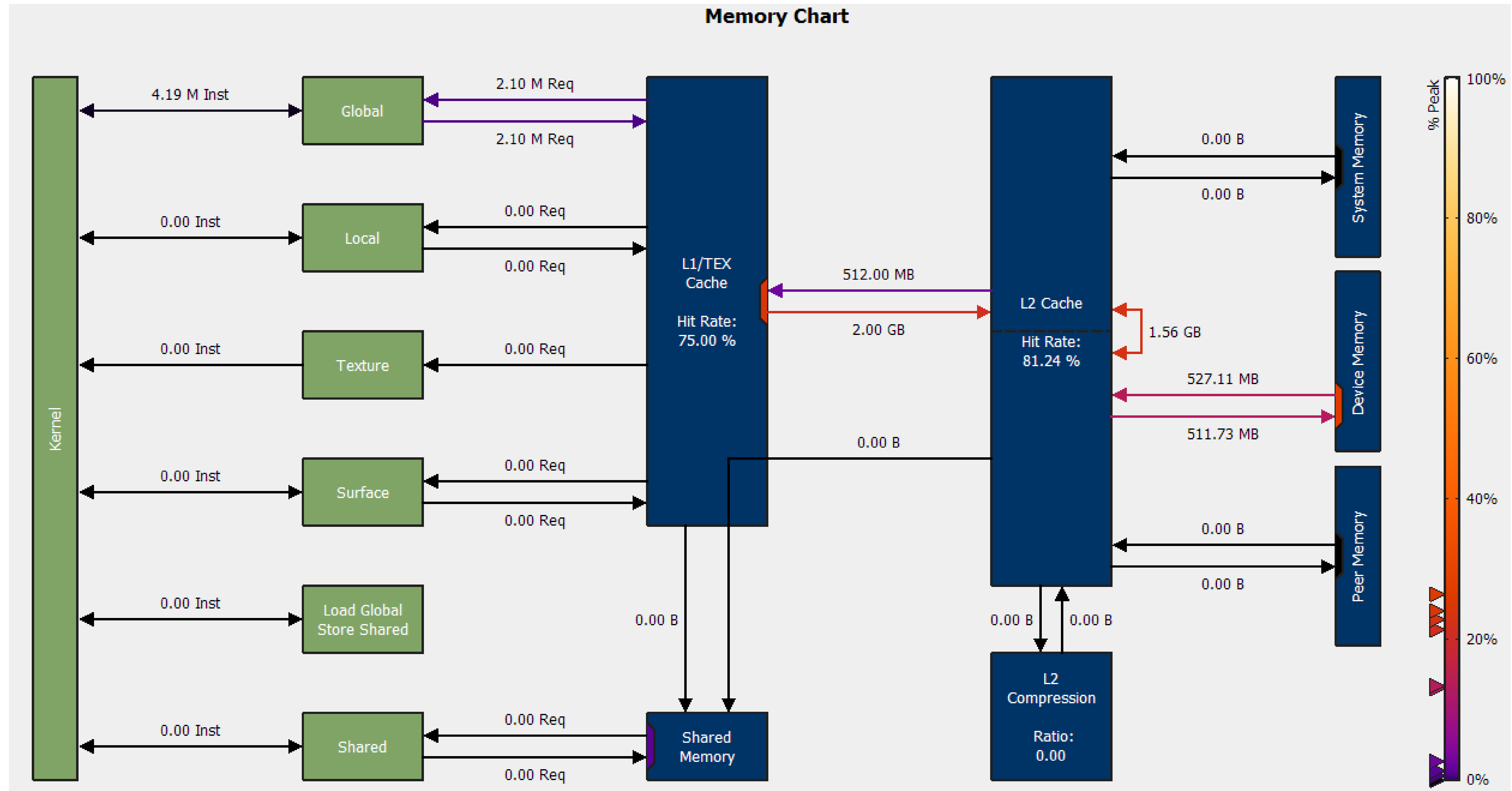
17.37% Lg Throttle (18937)

81.80% Long Scoreboard (89174)

„Uncoalesced **global** access [...] 4.00x“

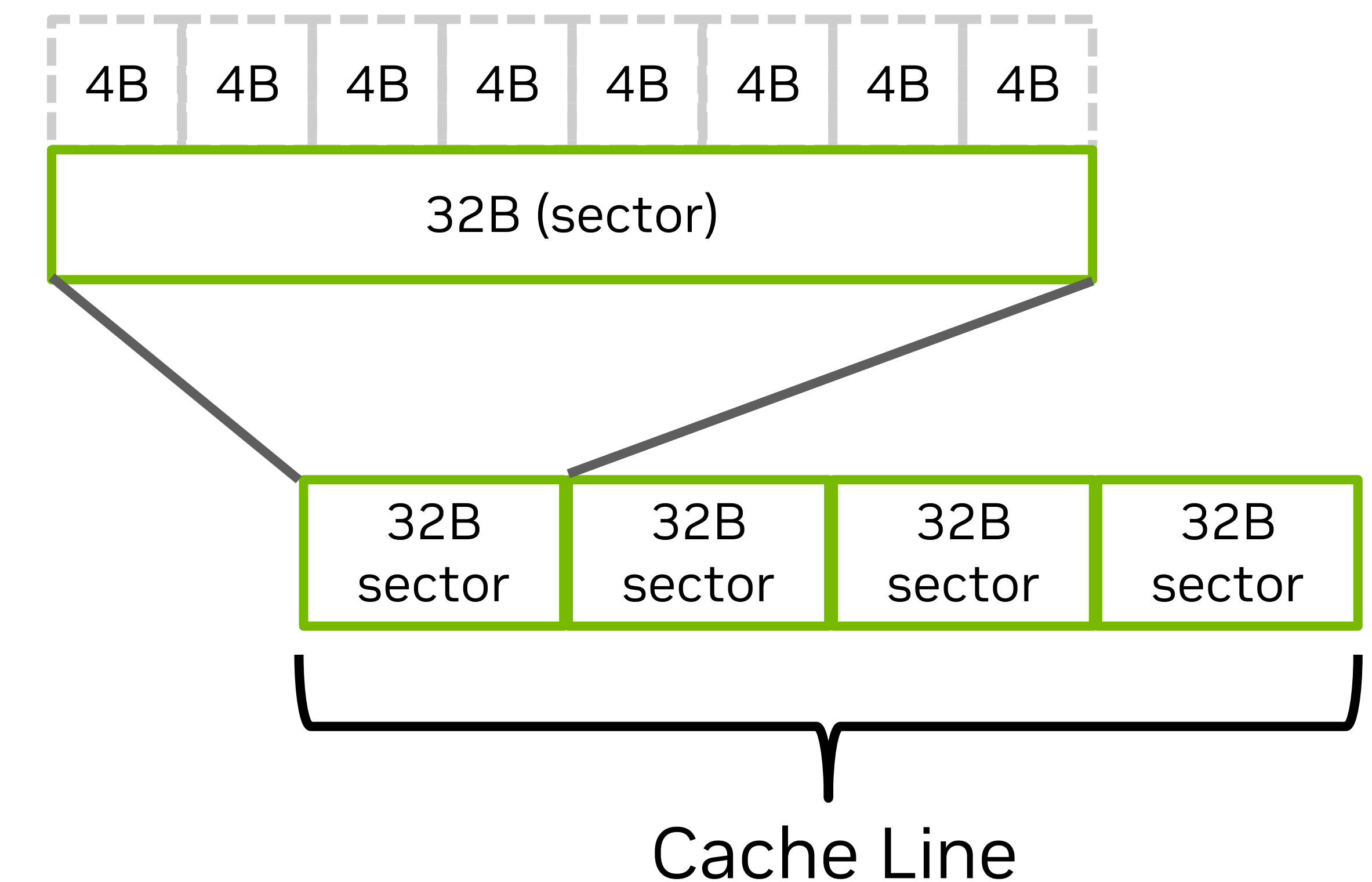
Global, local, l1, l2?

Understanding the memory hierarchy



Memory Transactions and Coalescing

- Access to global memory triggers transactions ([Device Mem Access](#))
- Memory access granularity = 32 bytes = 1 sector
- Cache line = 128 bytes = 4 consecutive sectors
 - Example: 4 byte per thread $\rightarrow 4B * 32 \text{ threads (1 warp)} = 128B$
- Data goes from **global** device memory through L2 cache
- Granularity*: Sector for L2, Cache line for L1



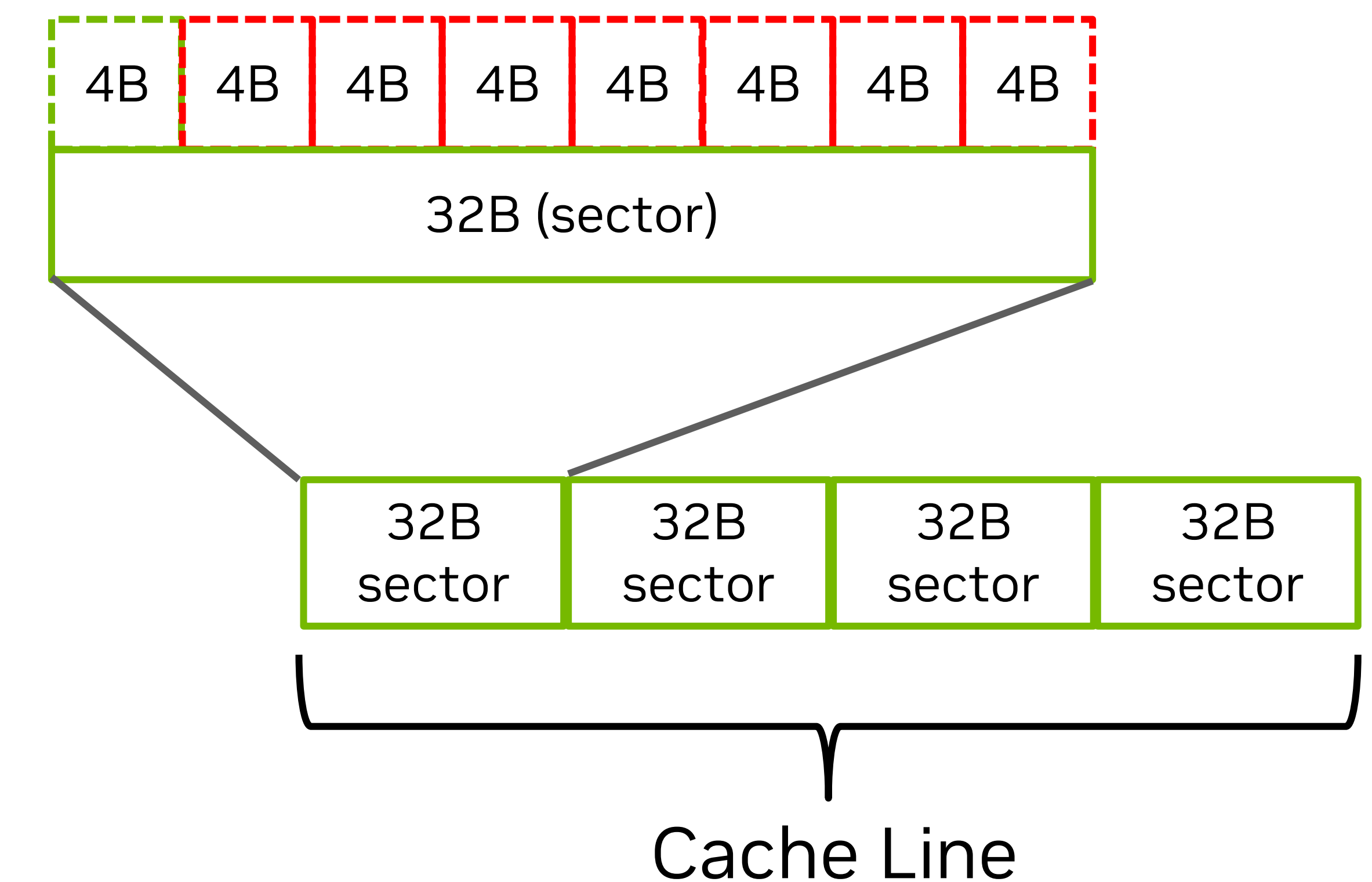
*The full picture:
[S32089: Understanding and Optimizing Memory-Bound Kernels with Nsight Compute](#)

Memory Transactions and Coalescing

Coalescing details

- Coalescing: Adjacent accesses can share transactions
- Transactions must be “Naturally Aligned”: First address % size == 0
- All bytes in a transaction are transferred. Use them!

For example, if a 32-byte memory transaction is generated for each thread's 4-byte access, throughput is divided by 8.



$$\text{degree of coalescing} = \frac{\text{\#bytes requested}}{\text{\#bytes transferred}}$$

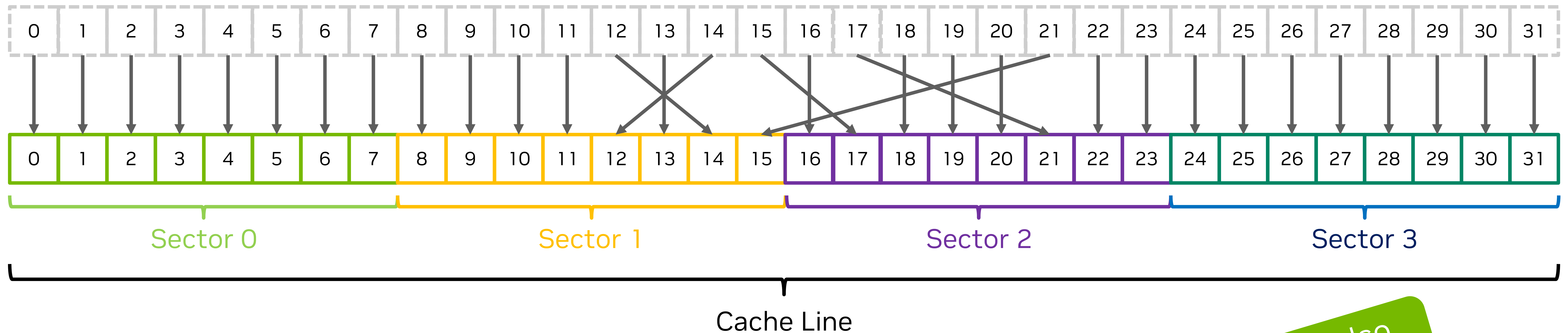
Accessing Global Memory

optimal access pattern (4byte words) – fully coalesced

```
int x_val = x[threadIdx.x];
```

All addresses fall within 4 sectors

Bus utilization: 100%



Permutations are also fine

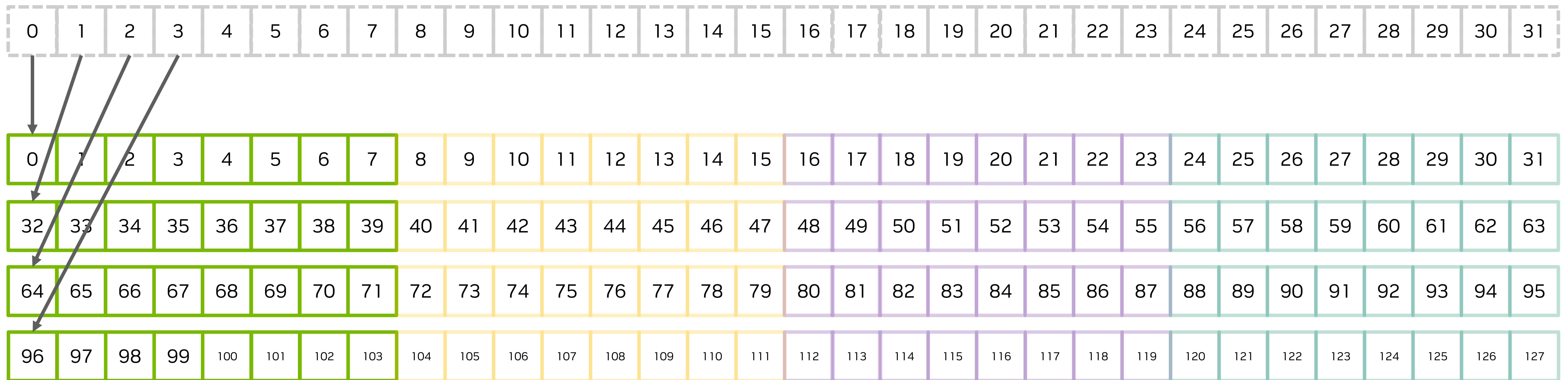
Accessing Global Memory

worst case access pattern (4byte words) – fully uncoalesced

```
// stride 32
int x_val = x[32*threadIdx.x];
// „random“ (pointer chasing, lists, tree, ...)
int x_val = x[lookup[threadIdx.x]];
```

All addresses fall in 32 different sectors

Bus utilization: 12.5%



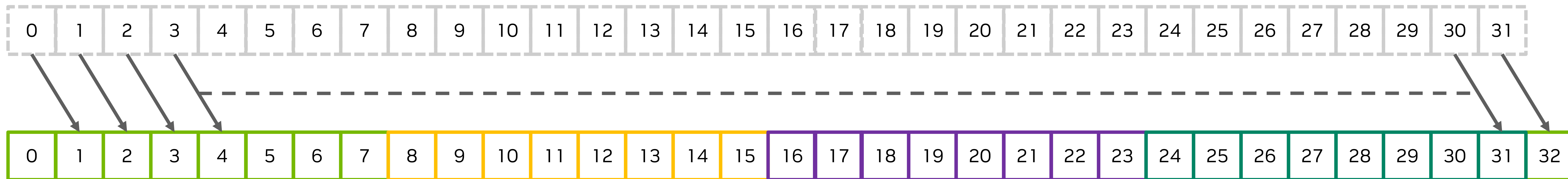
Accessing Global Memory

shifted access

```
int x_val = x[threadIdx.x+1];
```

All addresses fall within 5 sectors

Bus utilization: 80% = 128B/160B



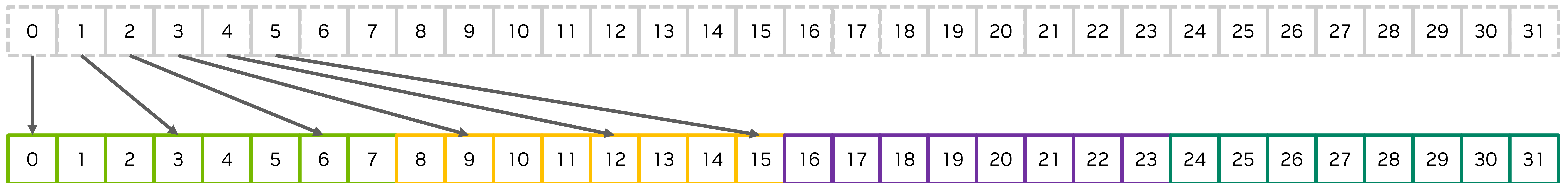
Accessing Global Memory

common access pattern: stride 3

```
int x_val = x[3*threadIdx.x];
```

All addresses fall within 12 sectors (4 byte words)

Bus utilization: 33%



```
struct {float x,y,z;} a; ... a[tid].x
```

→ use structure-of-arrays (SoA): `a.x[tid]`

```
float a[M][N]; ... a[tid][42]
```

→ multi-dimensional arrays: pay attention to coalescing (row-major, column-major?)

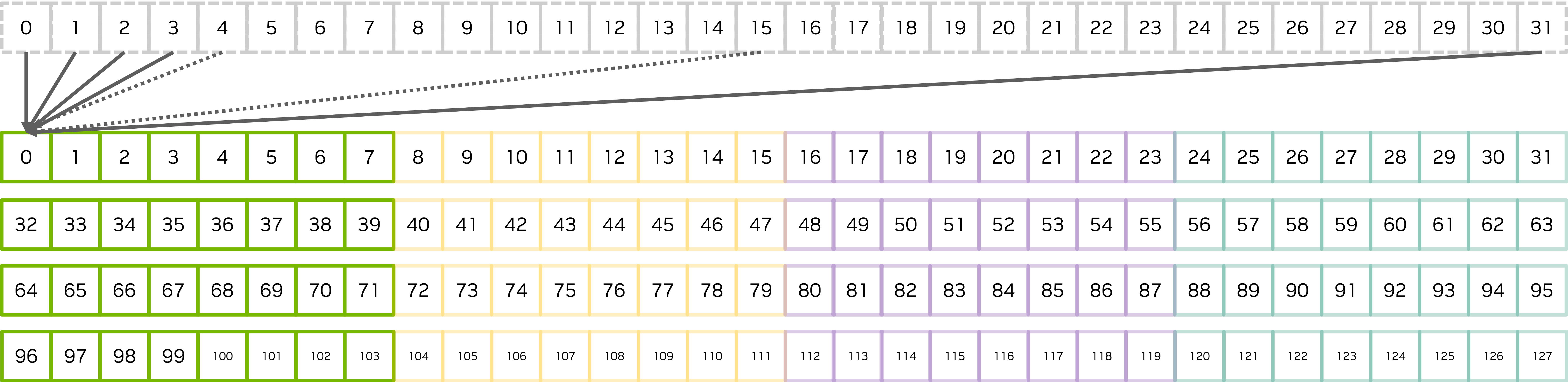
Accessing Global Memory

another “worst case” access pattern?

```
// same for all threads – e.g. loop index
int x_val = x[i];
```

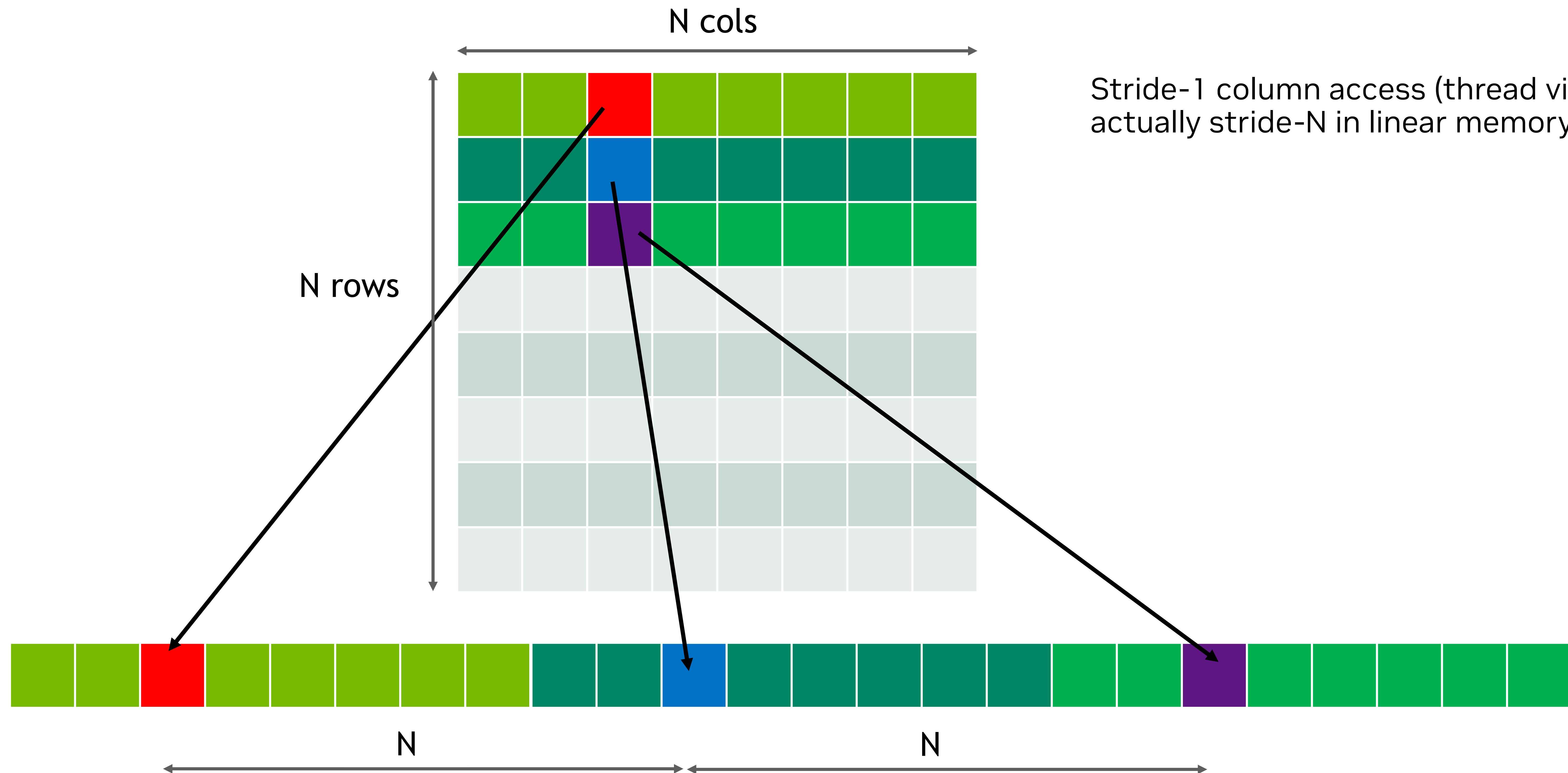
Single address, single sector

Bus utilization: 12.5%



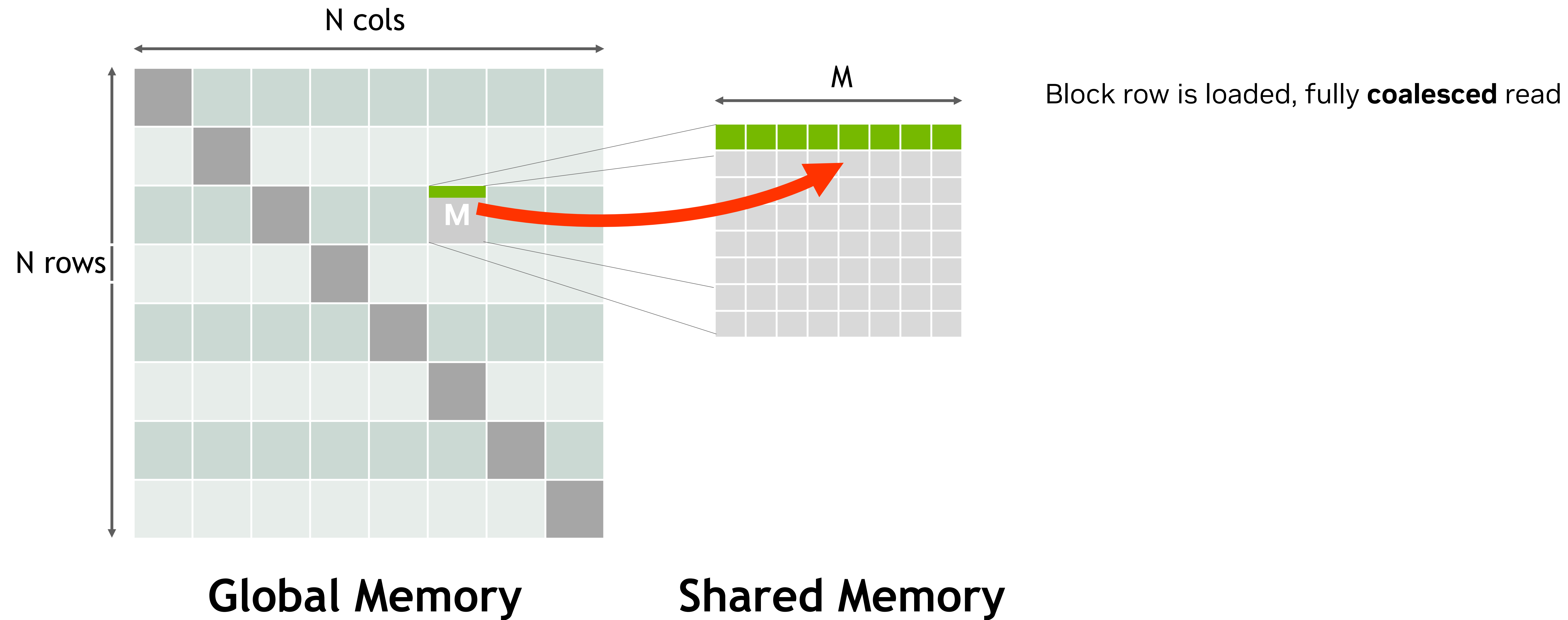
Matrix Transpose

Access pattern



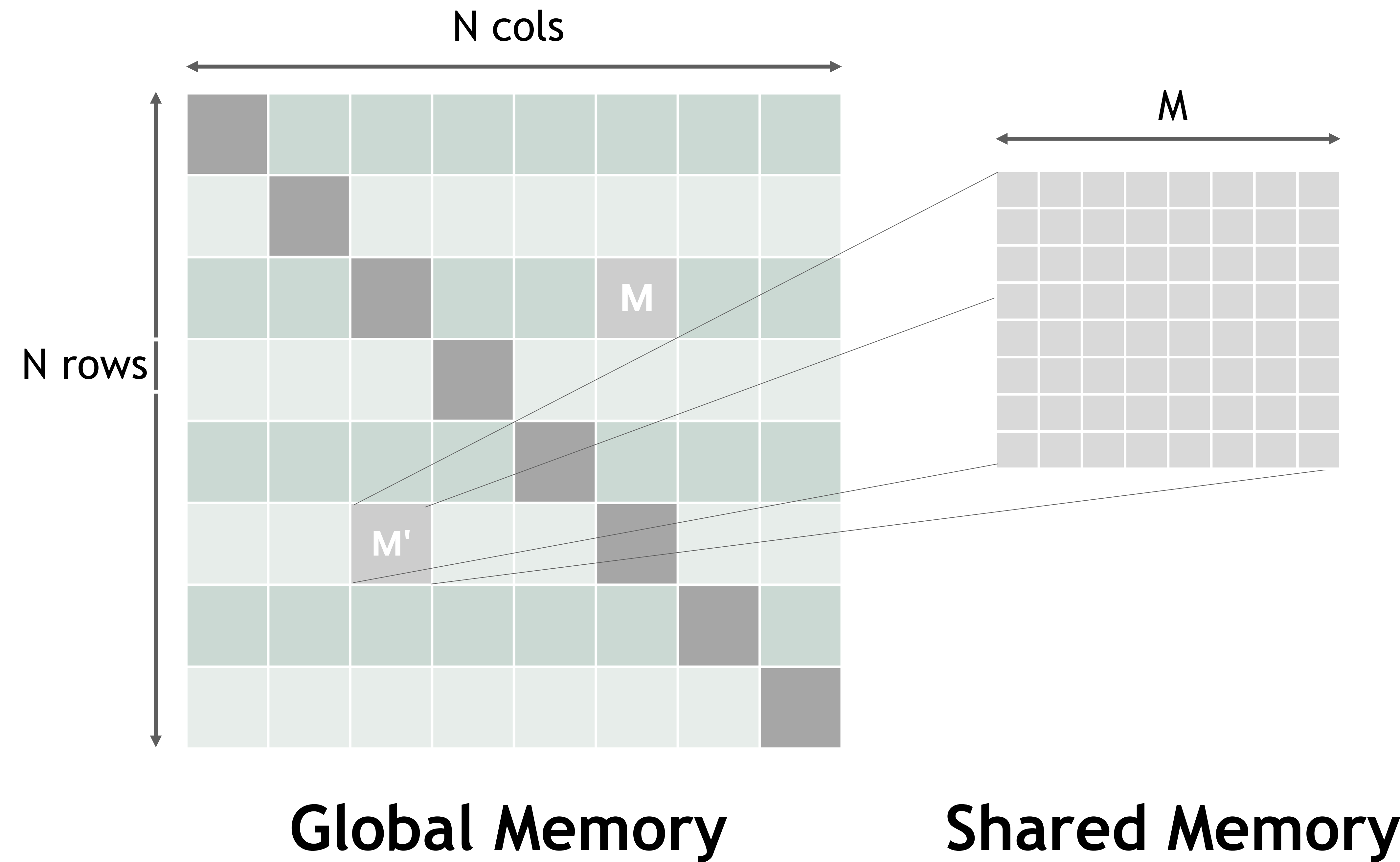
Matrix Transpose

Using shared memory



Matrix Transpose

Using shared memory

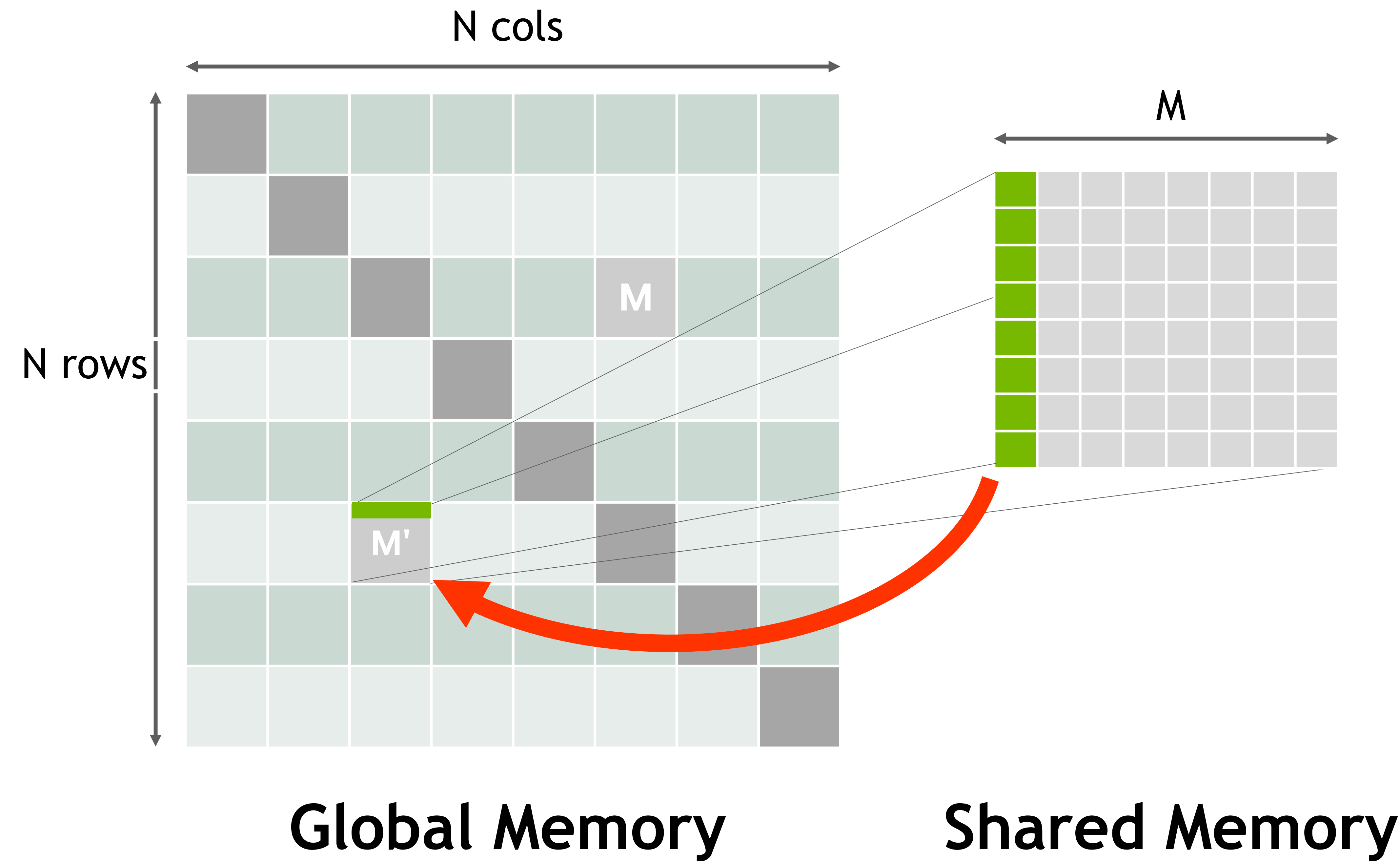


Block row is loaded, fully **coalesced** read

Indexing: Location of block is reflected on the diagonal

Matrix Transpose

Using shared memory



Block row is loaded, fully **coalesced** read

Indexing: Location of block is reflected on the diagonal

Block *column* of shared is written to *row* of matrix

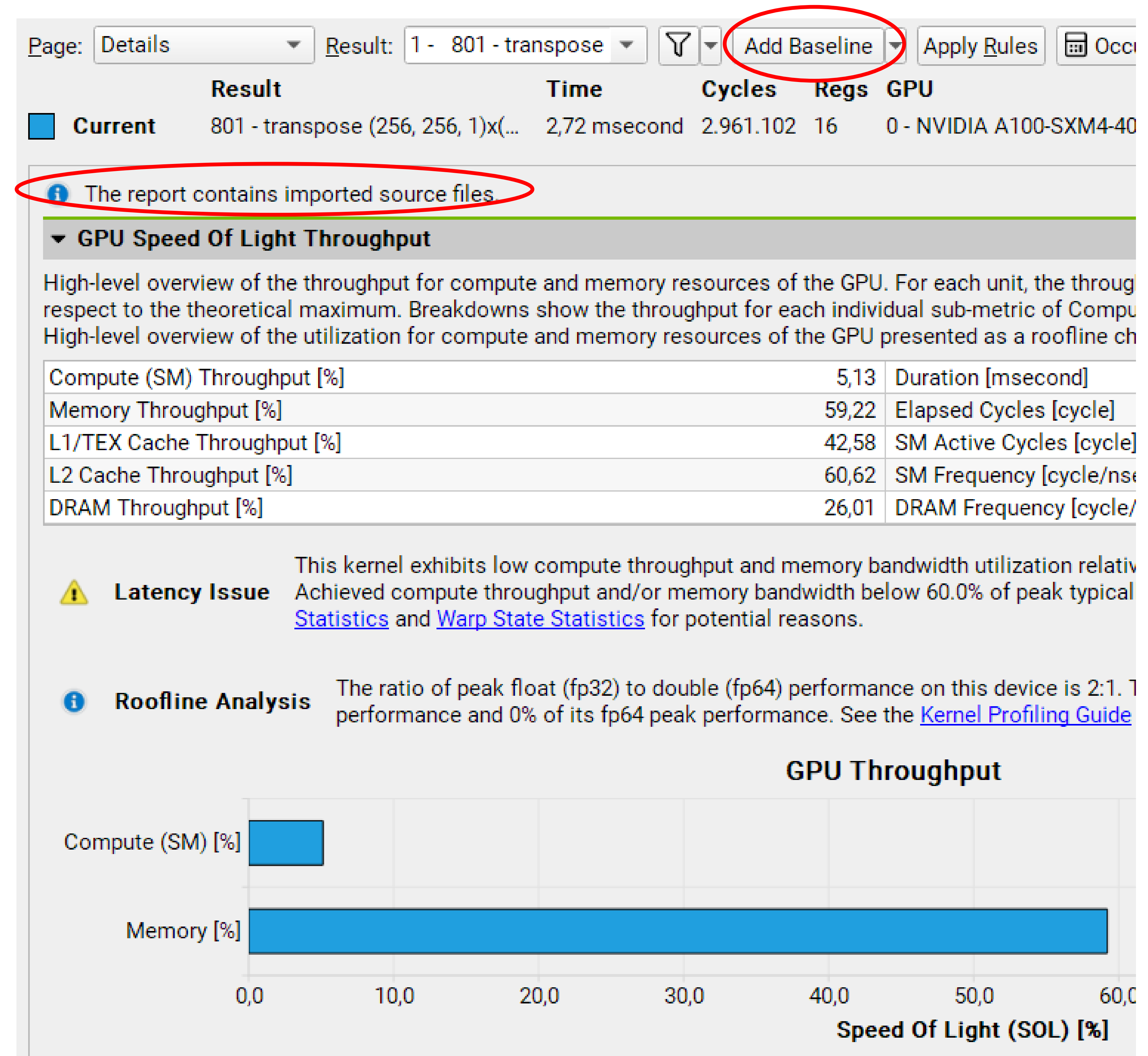
Transposes the block $M \rightarrow M'$

Coalesced write

Analysis with Nsight Compute

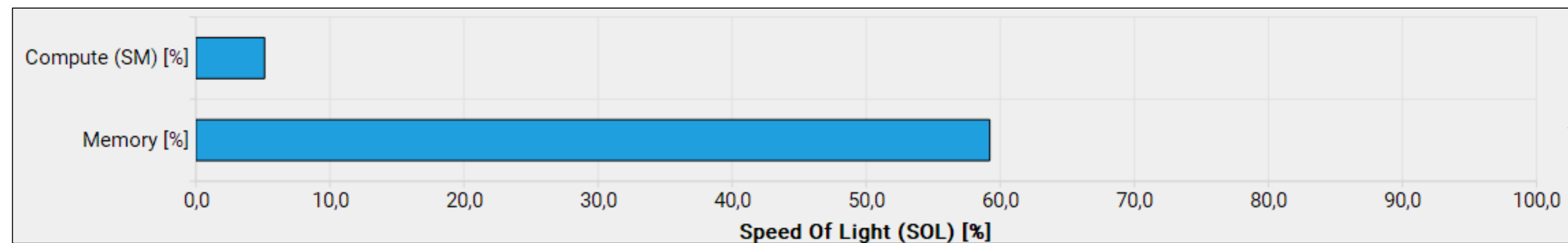
First steps

- Add baseline for comparison
 - Recommended: Always add `--import-source true` if possible
- Check the „Breakdown“ tables and how they change
- Look at the other sections, warp state statistics
- Tooltips on mouseover over metrics/names/...

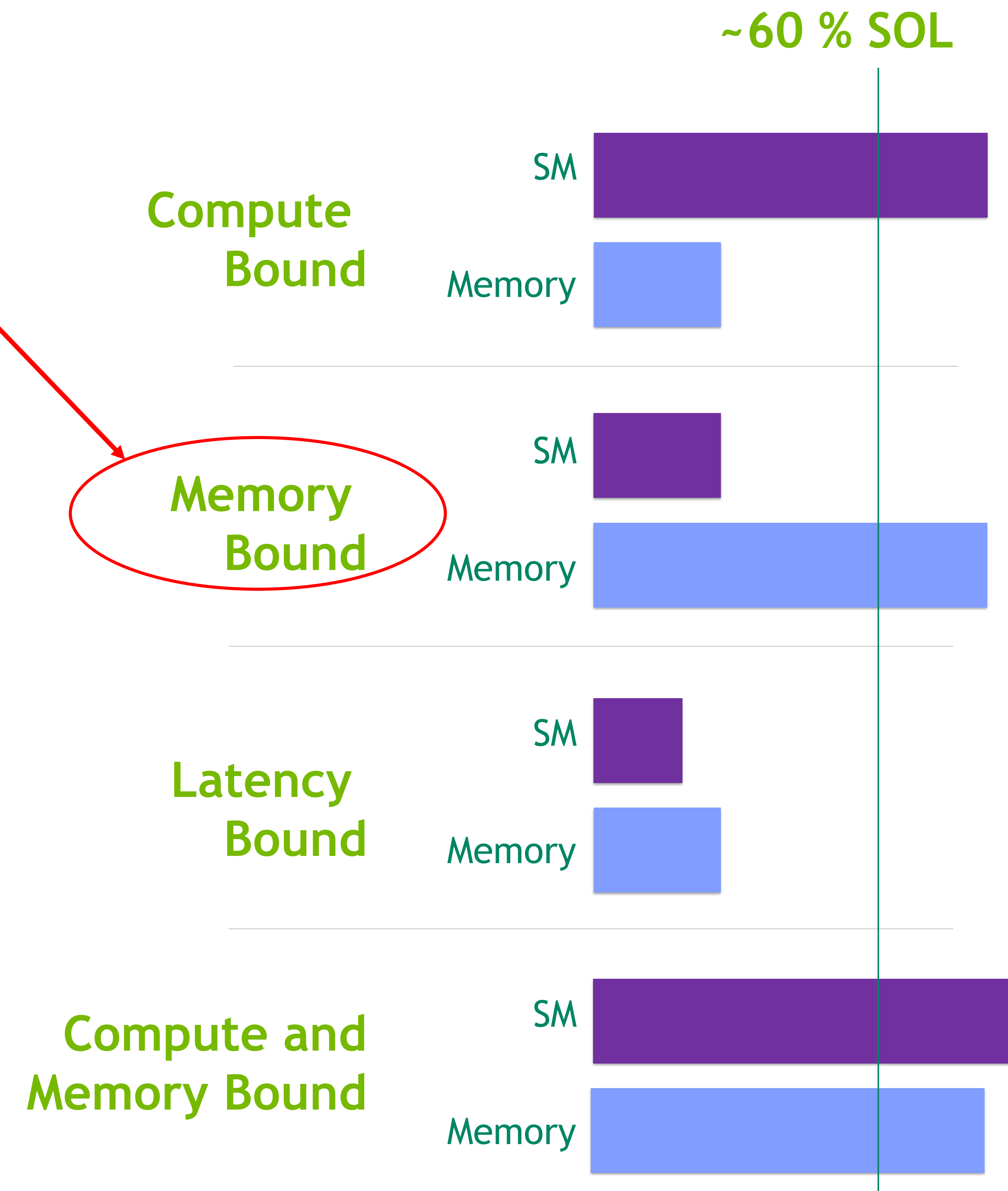


Kernel-level Profiling

Performance limiter categories



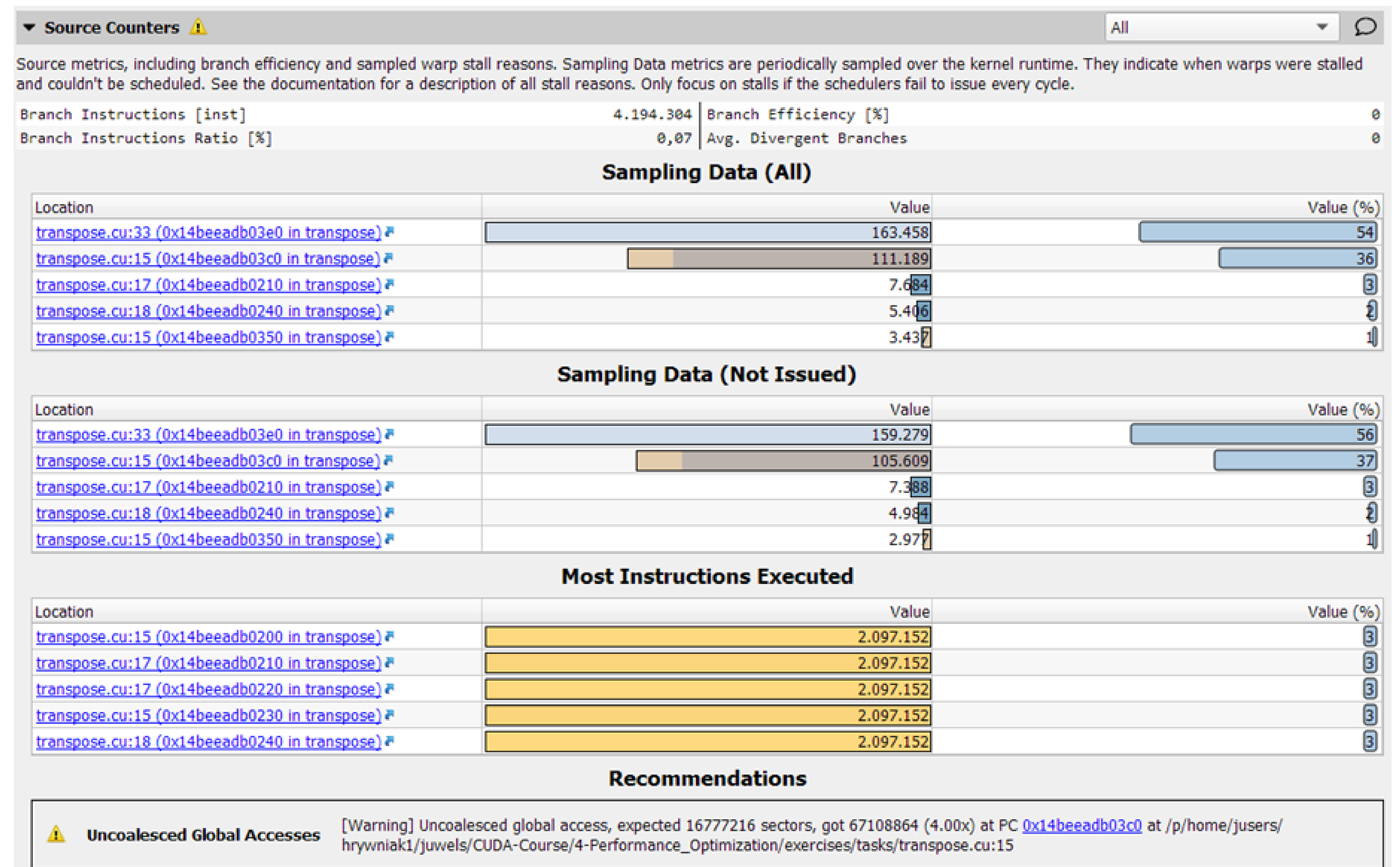
- Four possible combinations of high/low...
 - ...memory utilization
 - ...compute utilization
- Good? Bad?
- Depends on problem and its implementation



Analysis with Nsight Compute

Iterating and comparing

- Check the Source Counters section (also on CLI)
- Links will take you to Source/SASS view



Analysis with Nsight Compute

Source and SASS view

Page: Source

Result: 1 - 801 - transpose

Add Baseline

Apply Rules

Occupancy Calculator

Copy as Image

Current

801 - transpose (256, 256, 1)x(32, 32, 1)

2,72 msecond

2.961.102

16

0 - NVIDIA A100-SXM4-40GB

1,09 cycle/nsecond

8.0

[7060] transpose

View: Source and SASS

Source: transpose.cu

Find...

Navigation: L2 Theoretical Sectors Global Excessive

Source

12

13 __global__ void transpose(Integer* const a_trans, const Integer* const a, const Integer n)

14 {

15 const Integer col_block = blockIdx.x;

16 const Integer row_block = blockIdx.y;

17 const Integer block_col = threadIdx.x;

18 const Integer block_row = threadIdx.y;

19 const Integer col = col_block*BLOCK_SIZE+block_col;

20 const Integer row = row_block*BLOCK_SIZE+block_row;

21

22 //TODO: declare shared memory for tile

23 //__shared__ ... a_tile ...

24

25 if (row < n && col < n)

26 {

27 //TODO: load tile of a into shared memory

28 //TODO: call __syncthreads() to ensure all shared memory writes are completed

29 //TODO: read from a_tile with correct index:

30 //a_trans[(col_block*BLOCK_SIZE+block_row) * n + (row_block*BLOCK_SIZE+block_col)] = a_

31 a_trans[col*n+row] = a[row*n+col];

32 }

33 }

34

35 int main()

36 {

37 const Integer n = 8192;

38 Integer* a = new Integer[n*n];

oretical Sectors

lobal Excessive

L2 Theoretical

Sectors Global

L2

50.331.648

83.886.080

Source: transpose

Find...

Navigation: L2 Theoretical Sectors Global Excessive

Address

Source

4 00001524 c6fad730

S2R R5, SR_CTAID.X

5 00001524 c6fad740

S2R R4, SR_TID.Y

6 00001524 c6fad750

S2R R7, SR_CTAID.Y

7 00001524 c6fad760

IMAD.WIDE.U32 R2, R5, 0x20, R2

8 00001524 c6fad770

IMAD.MOV.U32 R5, RZ, RZ, RZ

9 00001524 c6fad780

ISETP.GE.U32.AND P1, PT, R2, c[0x0][0x170], PT

10 00001524 c6fad790

IMAD.WIDE.U32 R4, R7, 0x20, R4

11 00001524 c6fad7a0

ISETP.GE.AND.EX P1, PT, R3, c[0x0][0x174], PT, P

12 00001524 c6fad7b0

ISETP.GE.U32.AND P0, PT, R4, c[0x0][0x170], PT

13 00001524 c6fad7c0

ISETP.GE.OR.EX P0, PT, R5, c[0x0][0x174], P1, P0

14 00001524 c6fad7d0 @P0

EXIT

15 00001524 c6fad7e0

IMAD R9, R5, c[0x0][0x170], RZ

16 00001524 c6fad7f0

ULDC.64 UR4, c[0x0][0x118]

17 00001524 c6fad800

IMAD.WIDE.U32 R6, R4, c[0x0][0x170], R2

18 00001524 c6fad810

IMAD R9, R4, c[0x0][0x174], R9

19 00001524 c6fad820

LEA R8, P0, R6, c[0x0][0x168], 0x3

20 00001524 c6fad830

IADD3 R7, R7, R9, RZ

21 00001524 c6fad840

LEA.HI.X R9, R6, c[0x0][0x16c], R7, 0x3, P0

22 00001524 c6fad850

LDG.E.64 R6, [R8.64]

23 00001524 c6fad860

IMAD R3, R3, c[0x0][0x170], RZ

24 00001524 c6fad870

IMAD.WIDE.U32 R4, R2, c[0x0][0x170], R4

25 00001524 c6fad880

IMAD R3, R2, c[0x0][0x174], R3

26 00001524 c6fad890

LEA R2, P0, R4, c[0x0][0x160], 0x3

27 00001524 c6fad8a0

IMAD.IADD R3, R5, 0x1, R3

28 00001524 c6fad8b0

LEA.HI.X R3, R4, c[0x0][0x164], R3, 0x3, P0

29 00001524 c6fad8c0

STG.E.64 [R2.64], R6

30 00001524 c6fad8d0

EXIT

oretical Sectors

lobal Excessive

L2 Theoretical

Sectors Global

L2 Theoretical

Sectors Global Ideal

L2 Explicit

Evict Policies

0

16.777.216

16.777.216

50.331.648

67.108.864

16.777.216

Task: Coalesced Matrix Transpose

- Location of code: *4-Performance_Optimization/exercises/tasks*
- Steps (see also Instructions.ipynb):
 - Follow TODOs in *transpose.cu*, use shared memory to coalesce writes to *a_trans*
 - Profile with Nsight Compute and observe your changes

Roofline Analysis

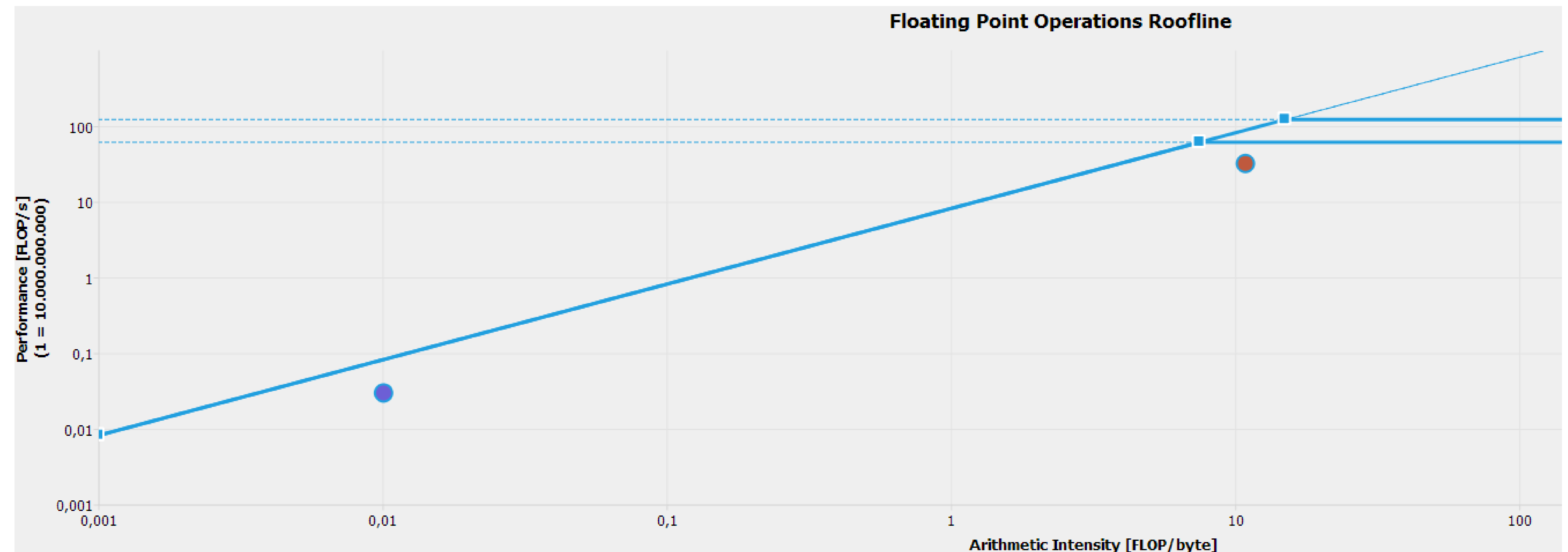
How well is the hardware utilized?

Transpose does zero floating point computations... more interesting example (here: on V100)

Counting flops and transferred bytes $\rightarrow$ AI, x-axis

Measuring achieved performance $\rightarrow$ FLOP/s, y-axis

Rooflines from device peak bandwidth / compute



GTC session:

[S32062: Performance Tuning CUDA Applications with the Roofline Model](#)

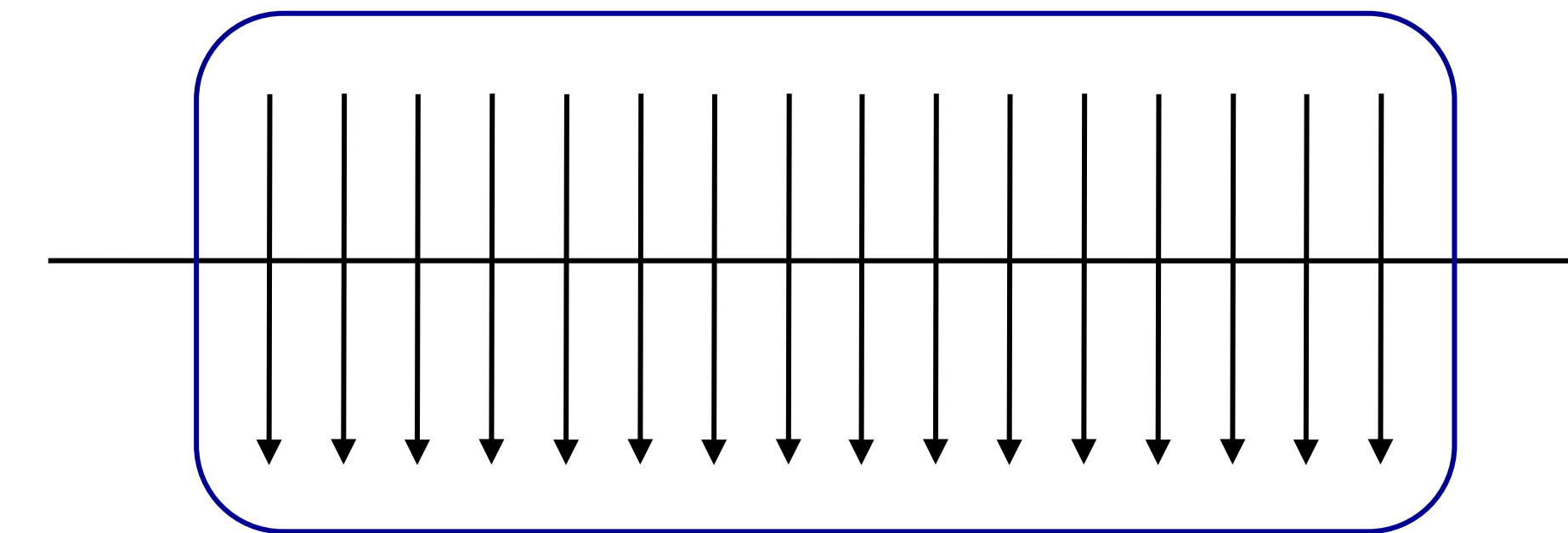
Roofline Hackathon:

<https://www.youtube.com/watch?v=ZXZ2SrM3pmE&t=2382s>

Branch Divergence

Recap: Warp execution

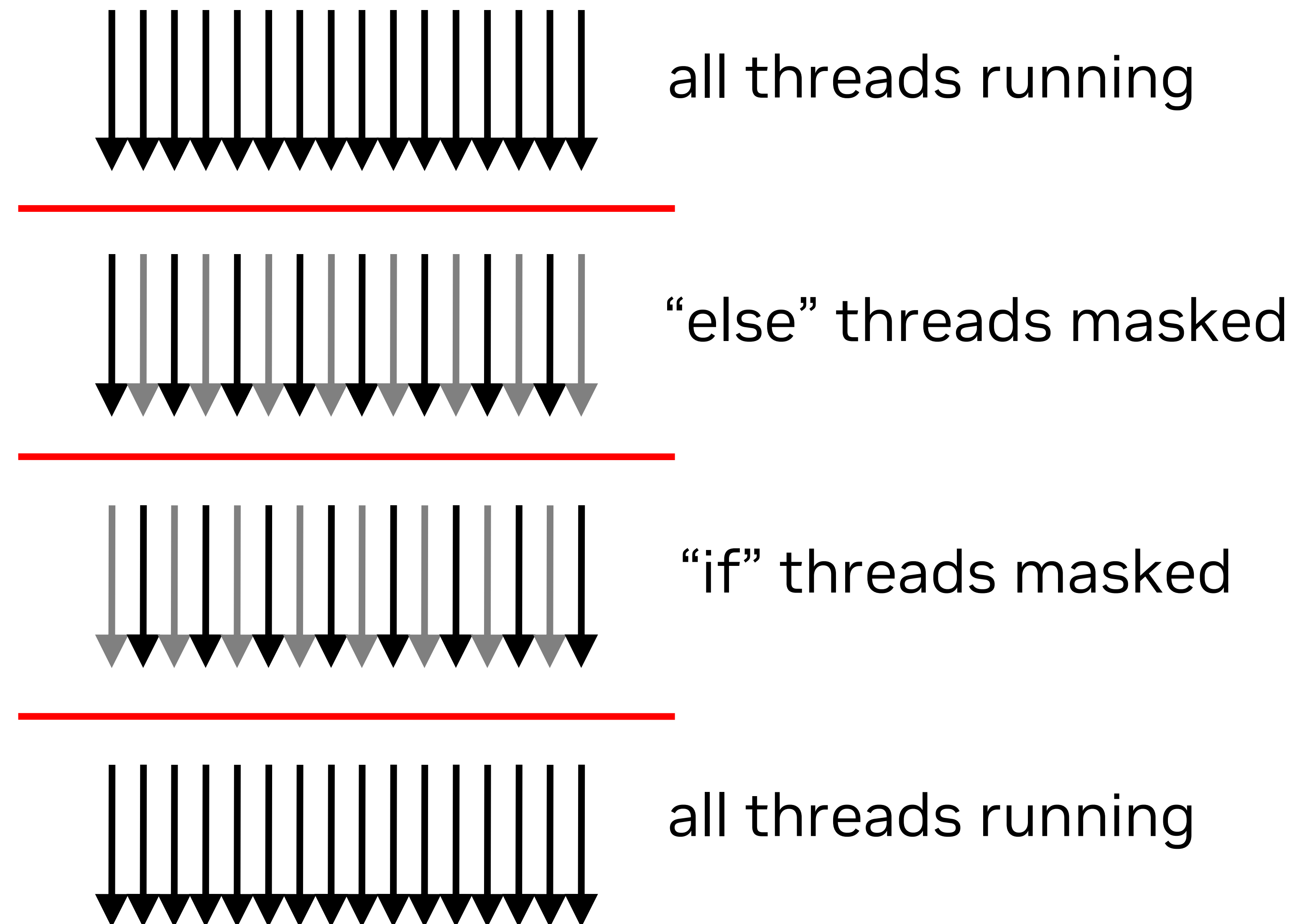
- GPUs use the Single Instruction Multiple Threads (SIMT) execution
 - functionally transparent to the programmer
 - but has performance implications
- warp
 - group of synchronously* executing threads (*since Volta: Independent Thread Scheduling)
 - neighbor threads (mostly x dimension)
 - basic unit of scheduling



Branch Divergence

Within Warp

```
// ...  
if (condition) {  
    // do stuff  
    // ...  
} else {  
    // do other stuff  
    // ...  
}  
// ...
```



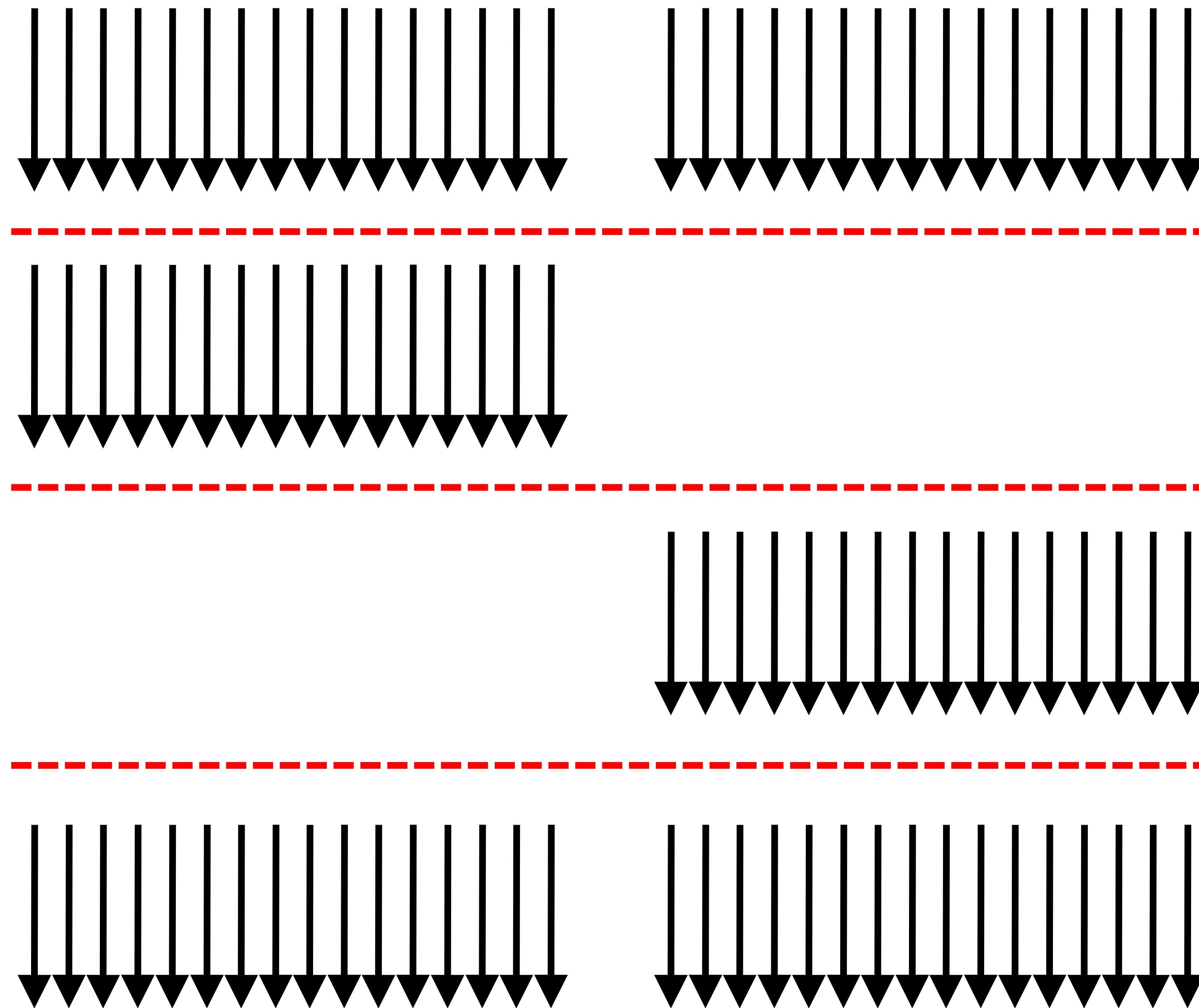
divergence *within* warp → performance penalty

```
if(threadIdx.x % 2 == 0) ...
```


Branch Divergence

Between Warps

```
// ...  
if (condition) {  
    // do stuff  
    // ...  
} else {  
    // do other stuff  
    // ...  
}  
// ...
```



divergence *between* warps → no penalty

```
if(blockIdx.x % 2 == 0) ...
```


Conclusion

To achieve coalesced global memory access:

- Usually: Fix your access pattern

- Try to use shared memory (but first, check cache behavior)

- Look for different way of storage or better algorithm

Avoid divergent branches

Use the tools!

