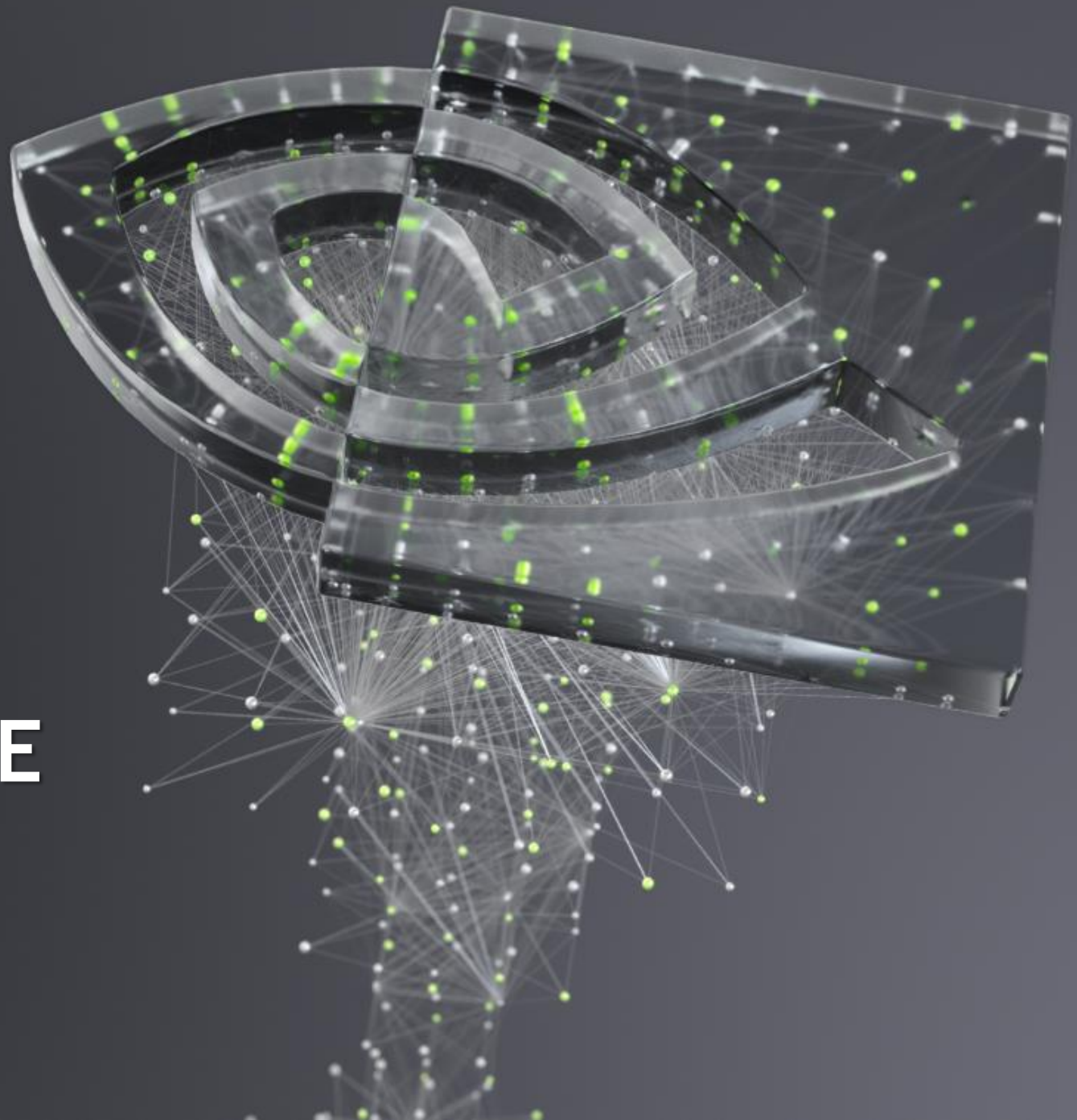


CUDA PERFORMANCE OPTIMIZATION

Markus Hrywniak, DevTech Compute, April 29, 2021

OUTLINE

What you will (not) learn this session

Memory access patterns

- Coalesced access

Branch divergence

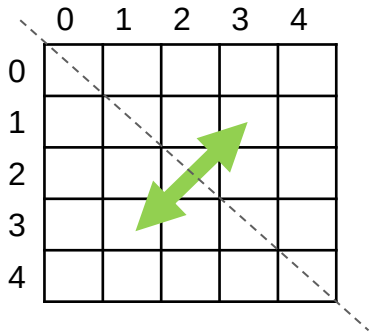
Using profiling tools (some more) for Kernel level

Also important, but not covered here:

- Exposing enough parallelism
- Expressing data locality
- Application-wide profiling

MOTIVATING EXAMPLE: MATRIX TRANSPOSE

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];

}
```

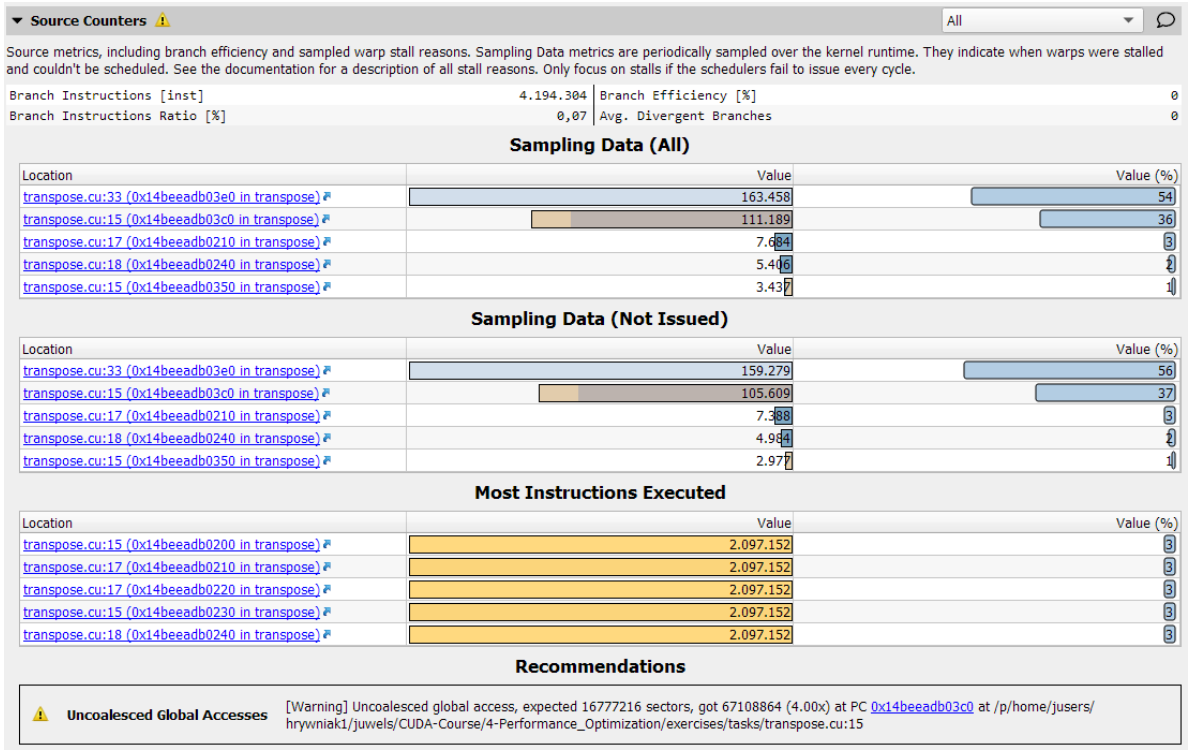
GPU VERSION:

```
__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 [%]	0
Branch Instructions Ratio [%]	0,07	Avg. Divergent Branches	0

Sampling Data (All)

Location	Value	Value (%)
transpose.cu:33 (0x14beeadb03e0 in transpose)	163.458	54
transpose.cu:15 (0x14beeadb03c0 in transpose)	111.189	36
transpose.cu:17 (0x14beeadb0210 in transpose)	7.684	3
transpose.cu:18 (0x14beeadb0240 in transpose)	5.406	2
transpose.cu:15 (0x14beeadb0350 in transpose)	3.437	1

Sampling Data (Not Issued)

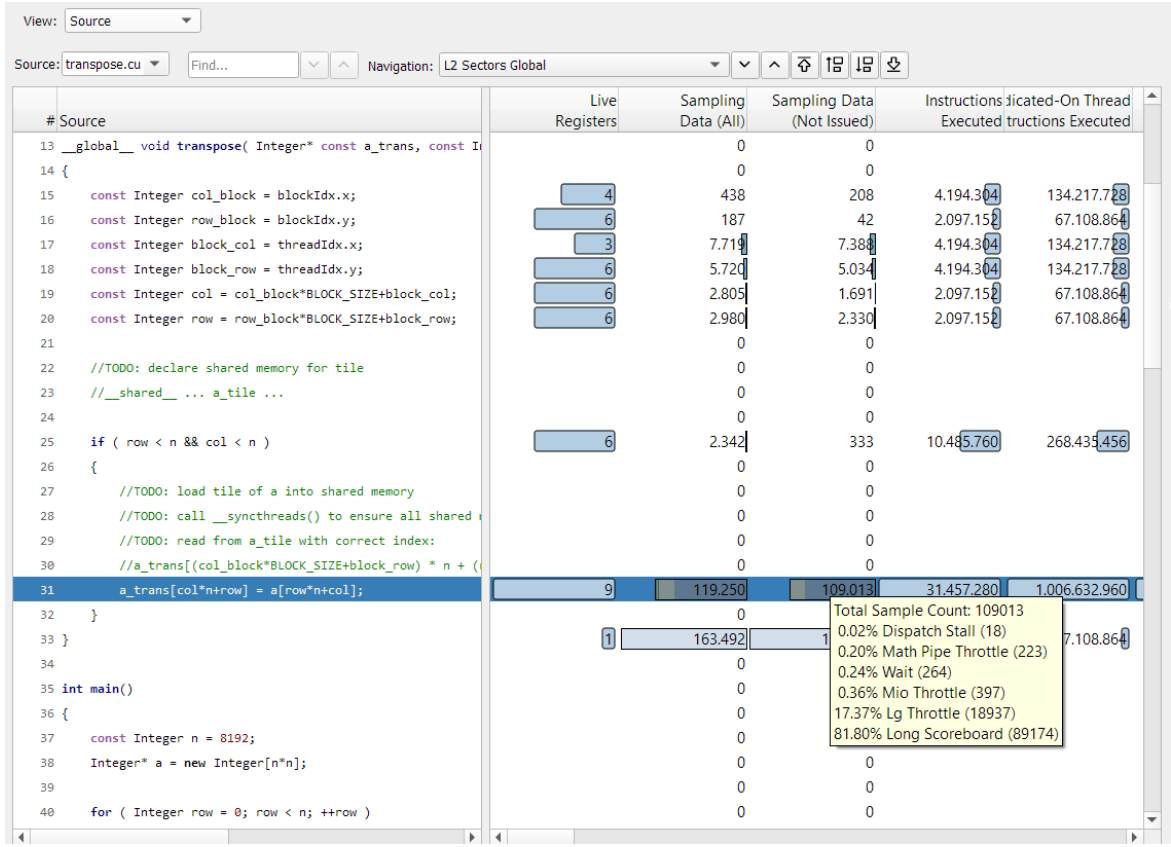
Location	Value	Value (%)
transpose.cu:33 (0x14beeadb03e0 in transpose)	159.279	56
transpose.cu:15 (0x14beeadb03c0 in transpose)	105.609	37
transpose.cu:17 (0x14beeadb0210 in transpose)	7.388	3
transpose.cu:18 (0x14beeadb0240 in transpose)	4.984	2
transpose.cu:15 (0x14beeadb0350 in transpose)	2.977	1

Most Instructions Executed

Location	Value	Value (%)
transpose.cu:15 (0x14beeadb0200 in transpose)	2.097.152	3
transpose.cu:17 (0x14beeadb0210 in transpose)	2.097.152	3
transpose.cu:17 (0x14beeadb0220 in transpose)	2.097.152	3
transpose.cu:15 (0x14beeadb0230 in transpose)	2.097.152	3
transpose.cu:18 (0x14beeadb0240 in transpose)	2.097.152	3

Recommendations

⚠ **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

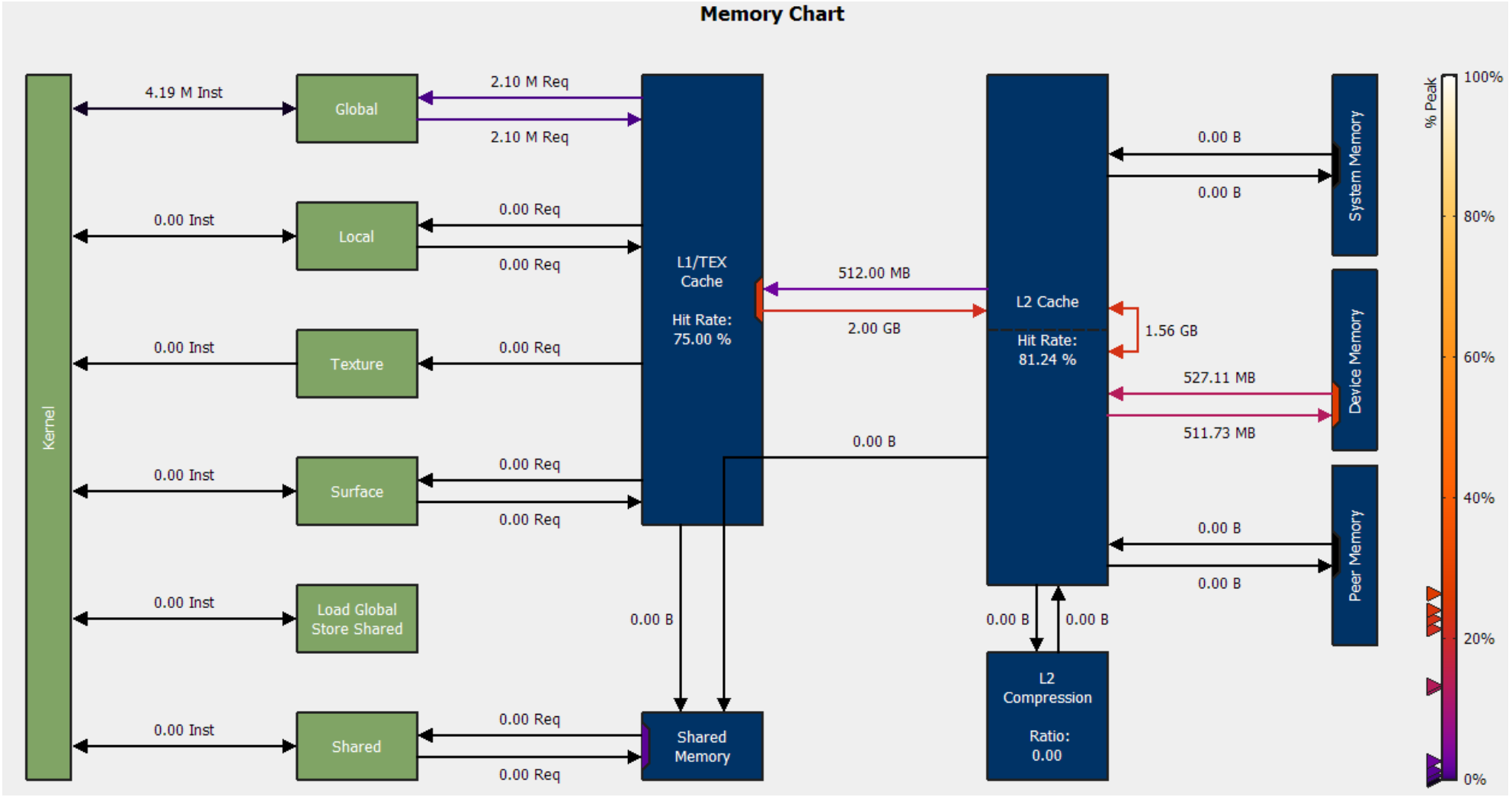
#	Source	Live Registers	Sampling Data (All)	Sampling Data (Not Issued)	Instructions Executed	Dedicated-On Thread Instructions Executed
13	`__global__ void transpose( Integer* const a_trans, const I`		0	0		
14	`{`		0	0		
15	`const Integer col_block = blockIdx.x;`	4	438	208	4.194.304	134.217.728
16	`const Integer row_block = blockIdx.y;`	6	187	42	2.097.152	67.108.864
17	`const Integer block_col = threadIdx.x;`	3	7.719	7.388	4.194.304	134.217.728
18	`const Integer block_row = threadIdx.y;`	6	5.720	5.034	4.194.304	134.217.728
19	`const Integer col = col_block*BLOCK_SIZE+block_col;`	6	2.805	1.691	2.097.152	67.108.864
20	`const Integer row = row_block*BLOCK_SIZE+block_row;`	6	2.980	2.330	2.097.152	67.108.864
21			0	0		
22	`//TODO: declare shared memory for tile`		0	0		
23	`//__shared__ ... a_tile ...`		0	0		
24			0	0		
25	`if ( row < n && col < n )`	6	2.342	333	10.485.760	268.435.456
26	`{`		0	0		
27	`//TODO: load tile of a into shared memory`		0	0		
28	`//TODO: call __syncthreads() to ensure all shared`		0	0		
29	`//TODO: read from a_tile with correct index:`		0	0		
30	`//a_trans[(col_block*BLOCK_SIZE+block_row) * n + (`		0	0		
31	`a_trans[col*n+row] = a[row*n+col];`	9	119.250	109.013	31.457.280	1.006.632.960
32	`}`		0			
33	`}`	1	163.492	1		7.108.864
34			0			
35	`int main()`		0			
36	`{`		0			
37	`const Integer n = 8192;`		0			
38	`Integer* a = new Integer[n*n];`		0	0		
39			0	0		
40	`for ( Integer row = 0; row < n; ++row )`		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 → 4B * 32 threads (1 warp) = 128B

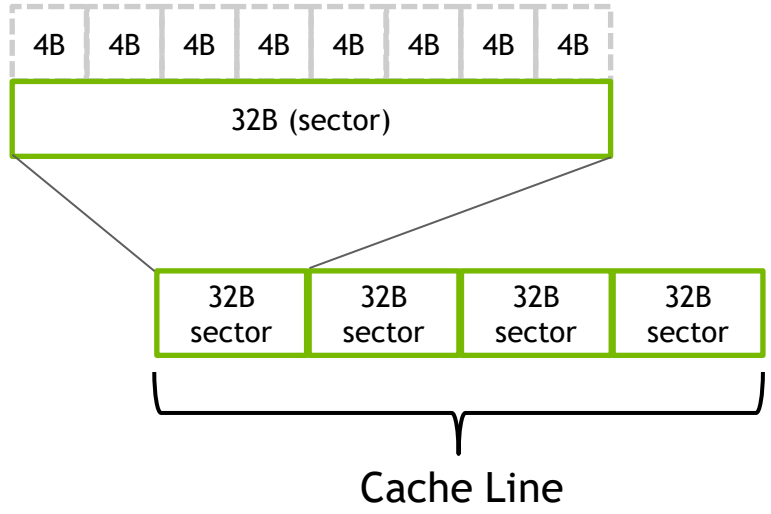
L1-cached access:

128-byte transactions

L2-cached access:

32-byte transactions

Used for global memory



The full picture:
S32089: Understanding and Optimizing Memory-Bound Kernels with Nsight Compute

MEMORY TRANSACTIONS AND COALESCING

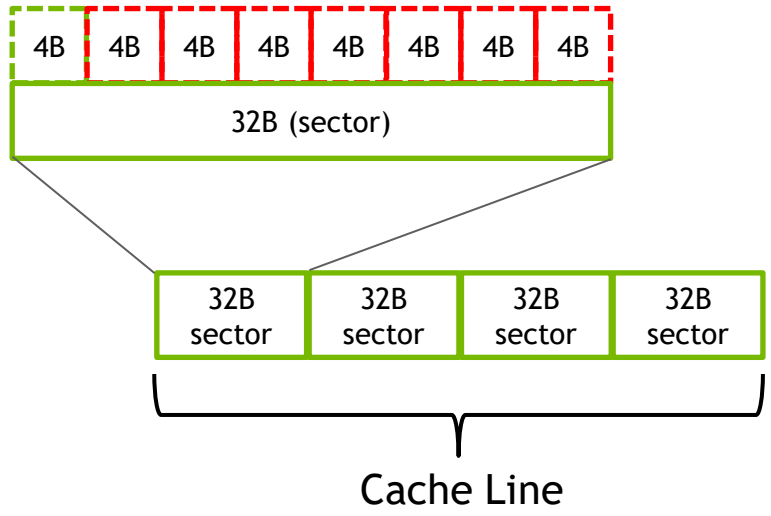
Coalescing details

Coalescing: Adjacent threads 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.



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

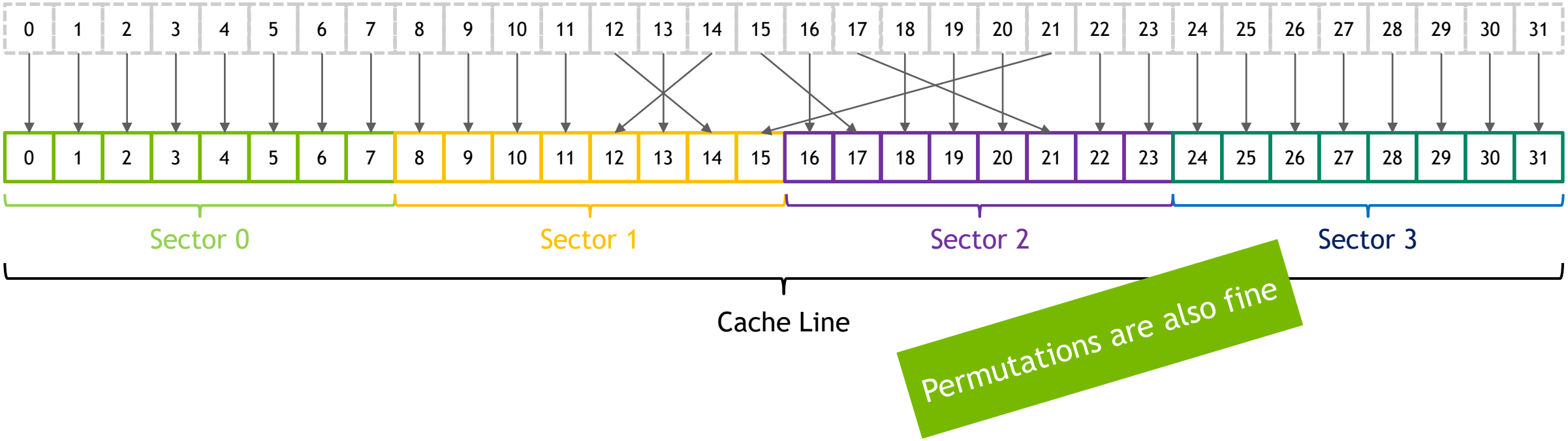
ACCESSING GLOBAL MEMORY

optimal access pattern (4byte words) – fully coalesced

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

All addresses fall within 4 sectors

Bus utilization: 100%



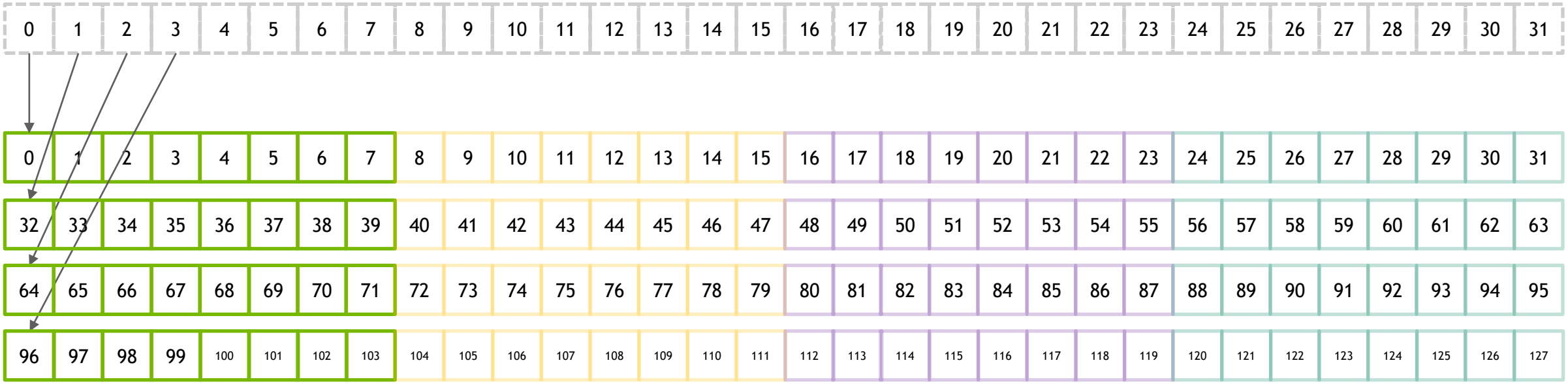
ACCESSING GLOBAL MEMORY

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%



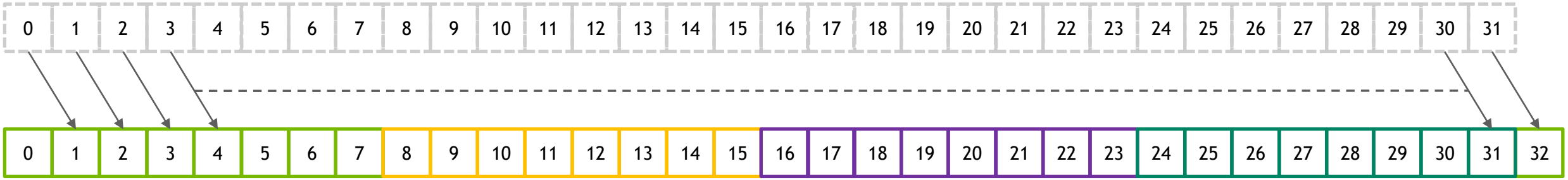
ACCESSING GLOBAL MEMORY

shifted access

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

All addresses fall within 5 sectors

Bus utilization: 80% = 128B/160B



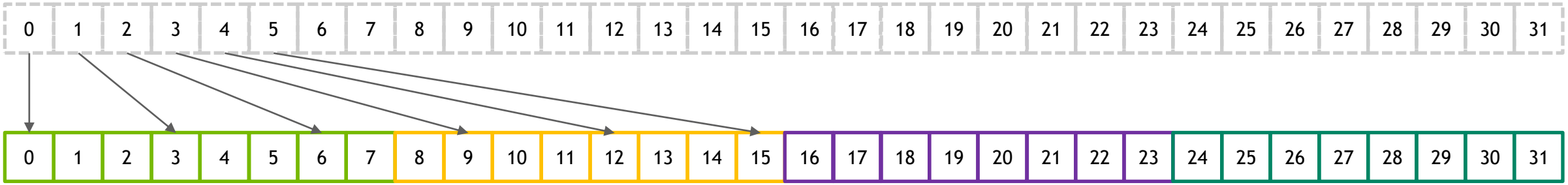
ACCESSING GLOBAL MEMORY

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?)

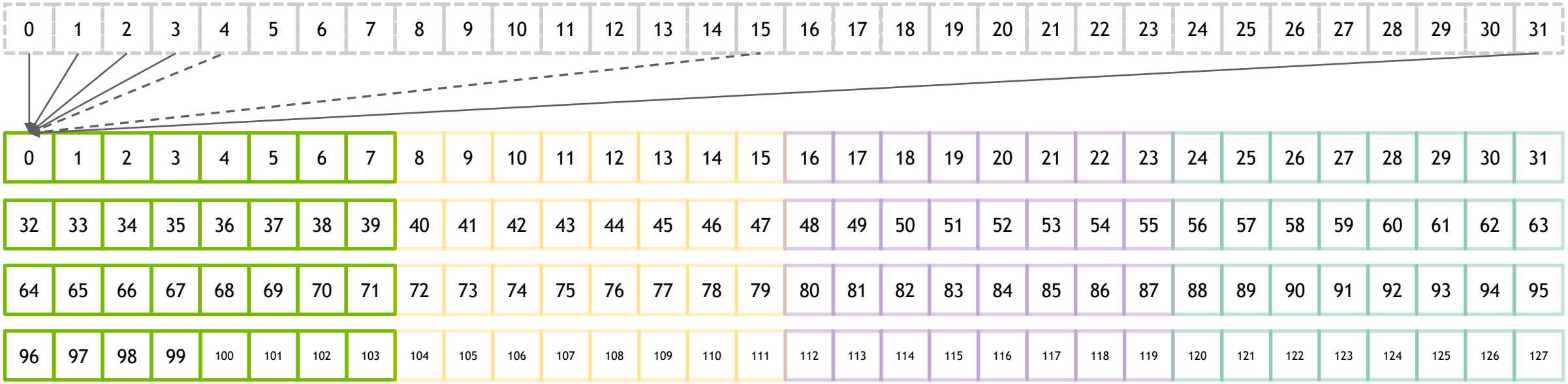
ACCESSING GLOBAL MEMORY

another “worst case” access pattern?

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

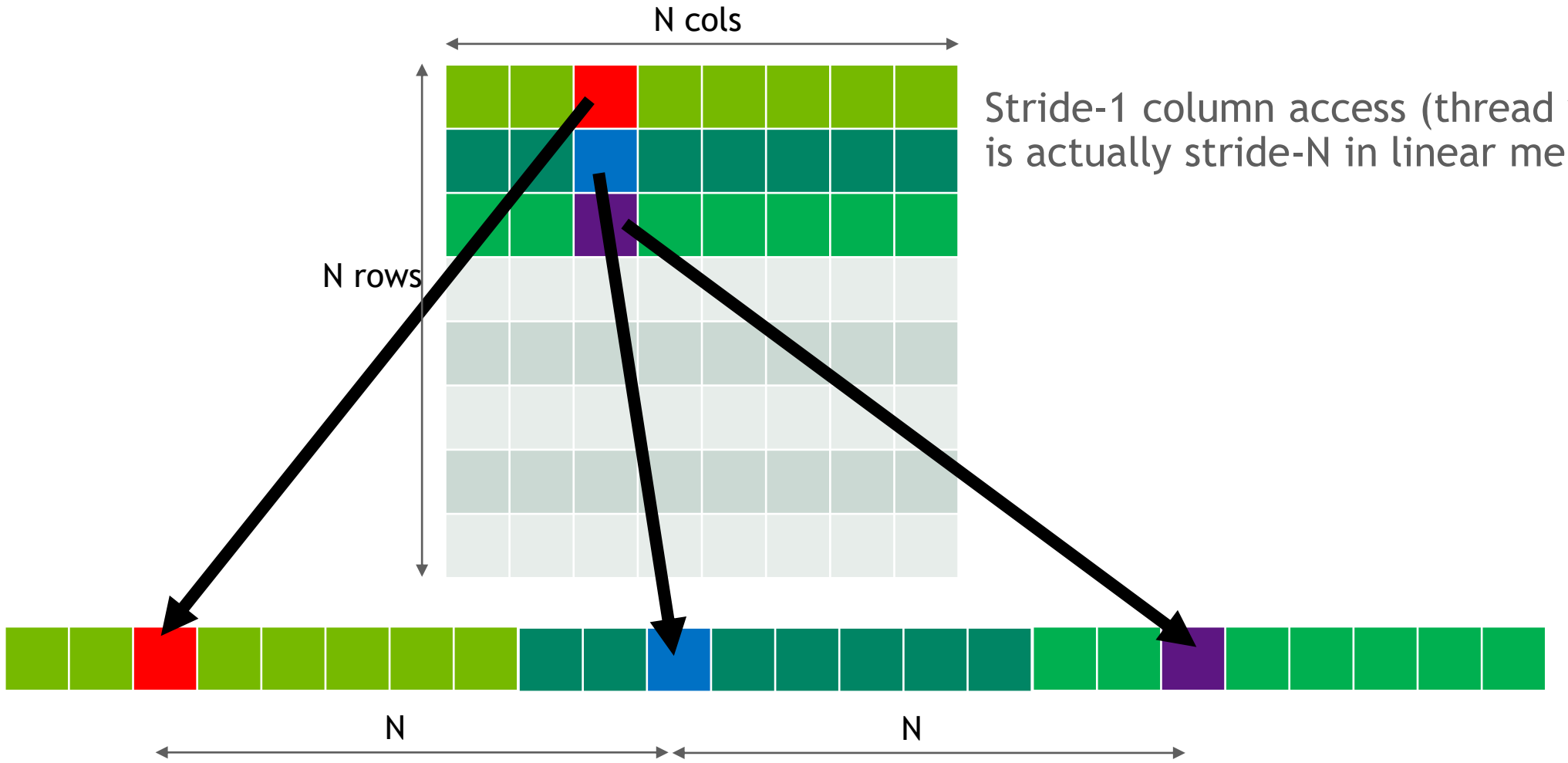
Single address, single sector

Bus utilization: 12.5%



MATRIX TRANSPOSE

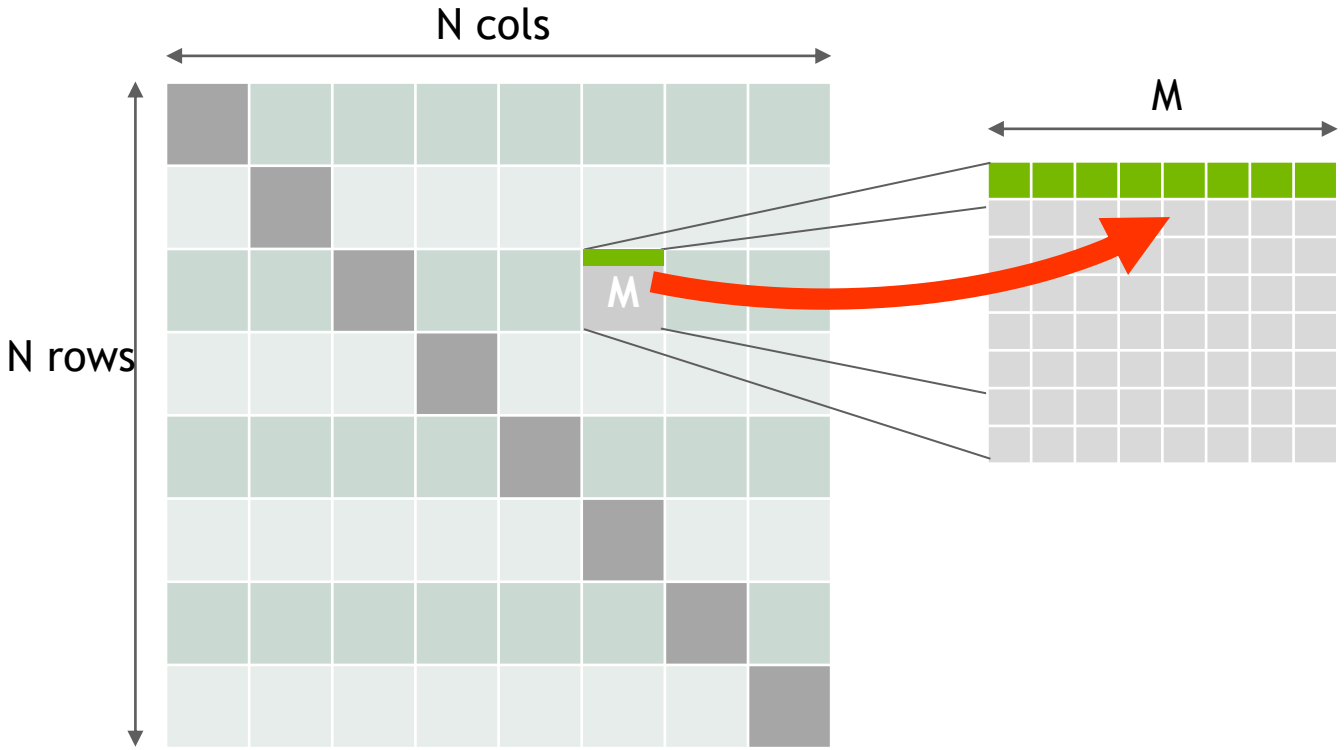
Access pattern



Stride-1 column access (thread view)
is actually stride-N in linear memory

MATRIX TRANSPOSE

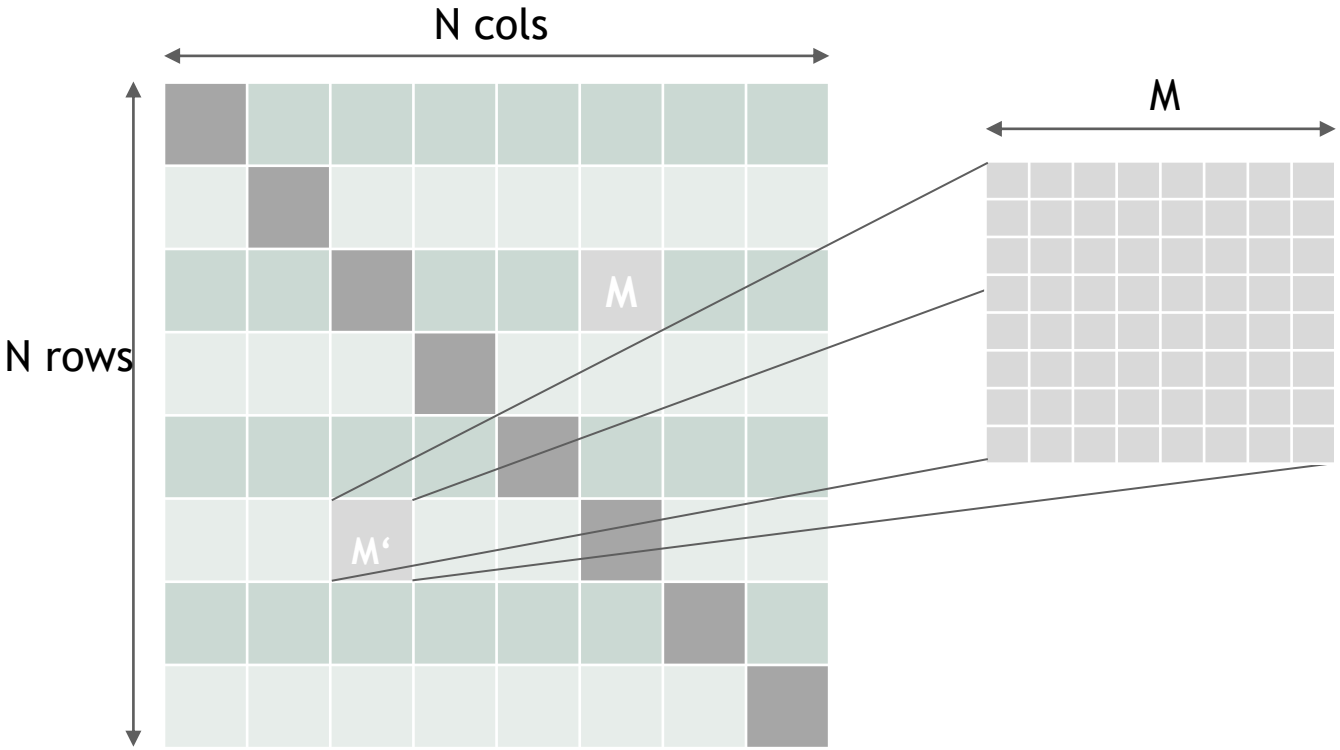
Using shared memory



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

MATRIX TRANSPOSE

Using shared memory

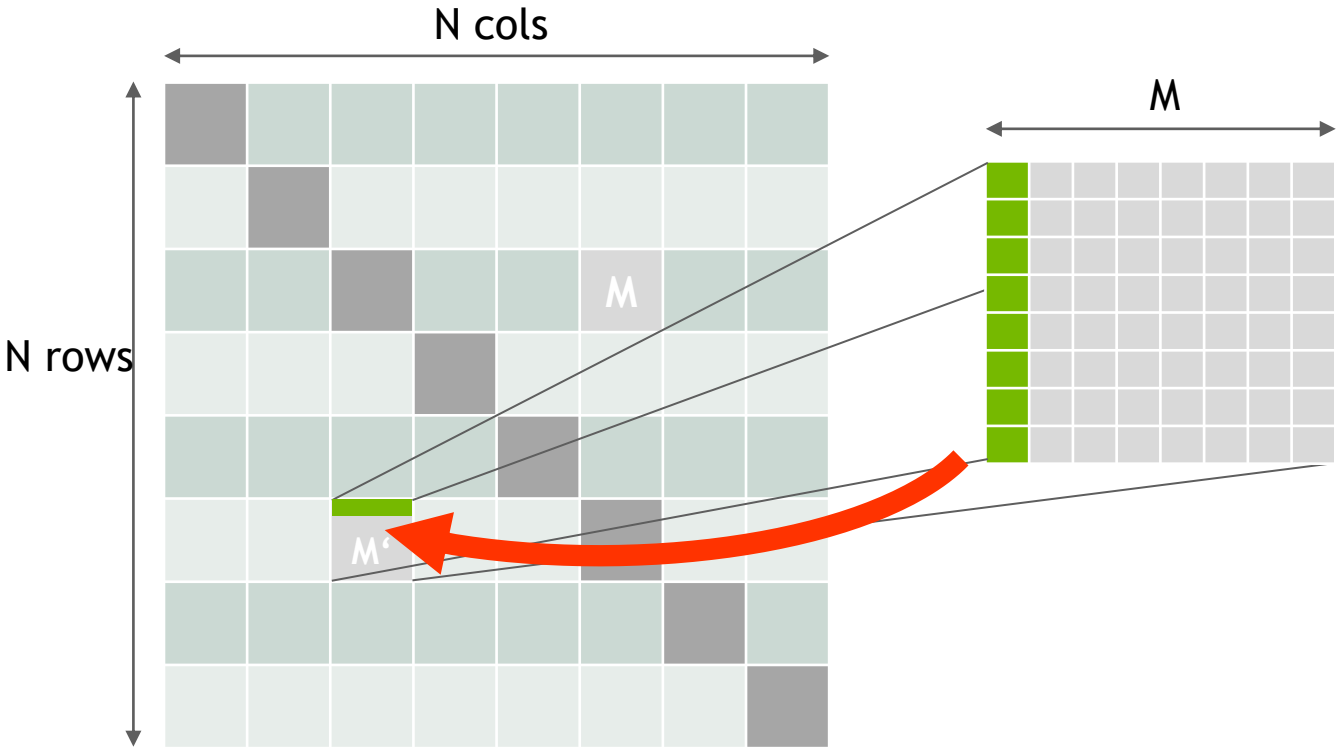


Block row is loaded, fully **coalesced** read

Indexing: Location of block is reflected on the diagonal

MATRIX TRANSPOSE

Using shared memory



Global Memory **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 → M‘

Coalesced write

ANALYSIS WITH NSIGHT COMPUTE

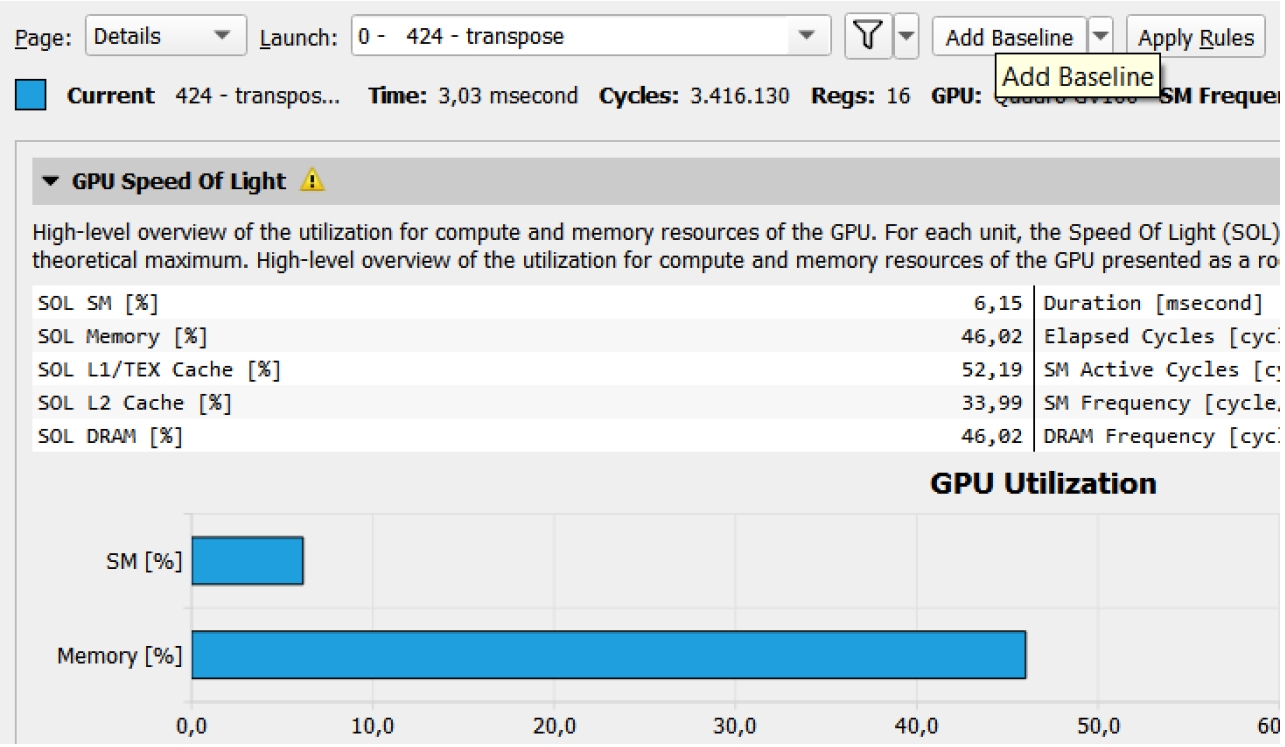
Iterating and comparing

Add baseline for comparison

Check the „Breakdown“ tables and how they change

Look at the other sections, warp state statistics

Tooltips on mouseover over metrics/names/...

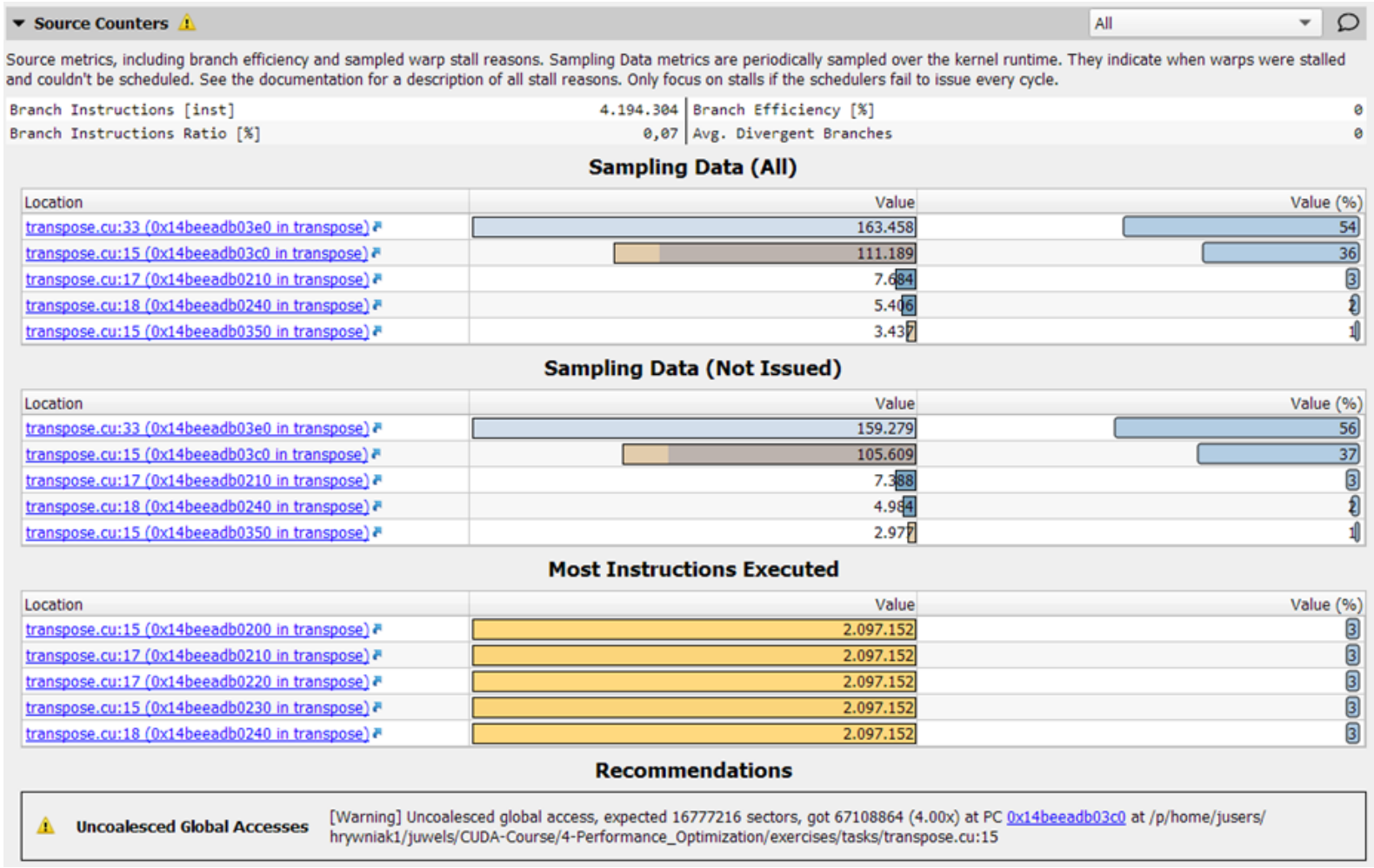


ANALYSIS WITH NSIGHT COMPUTE

Iterating and comparing

Check the Source Counters section (also on CLI)

Links will take you to Source/SASS view



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 [%]	0
Branch Instructions Ratio [%]	0,07	Avg. Divergent Branches	0

Sampling Data (All)

Location	Value	Value (%)
transpose.cu:33 (0x14beeadb03e0 in transpose)	163.458	54
transpose.cu:15 (0x14beeadb03c0 in transpose)	111.189	36
transpose.cu:17 (0x14beeadb0210 in transpose)	7.684	3
transpose.cu:18 (0x14beeadb0240 in transpose)	5.406	2
transpose.cu:15 (0x14beeadb0350 in transpose)	3.437	1

Sampling Data (Not Issued)

Location	Value	Value (%)
transpose.cu:33 (0x14beeadb03e0 in transpose)	159.279	56
transpose.cu:15 (0x14beeadb03c0 in transpose)	105.609	37
transpose.cu:17 (0x14beeadb0210 in transpose)	7.388	3
transpose.cu:18 (0x14beeadb0240 in transpose)	4.984	2
transpose.cu:15 (0x14beeadb0350 in transpose)	2.977	1

Most Instructions Executed

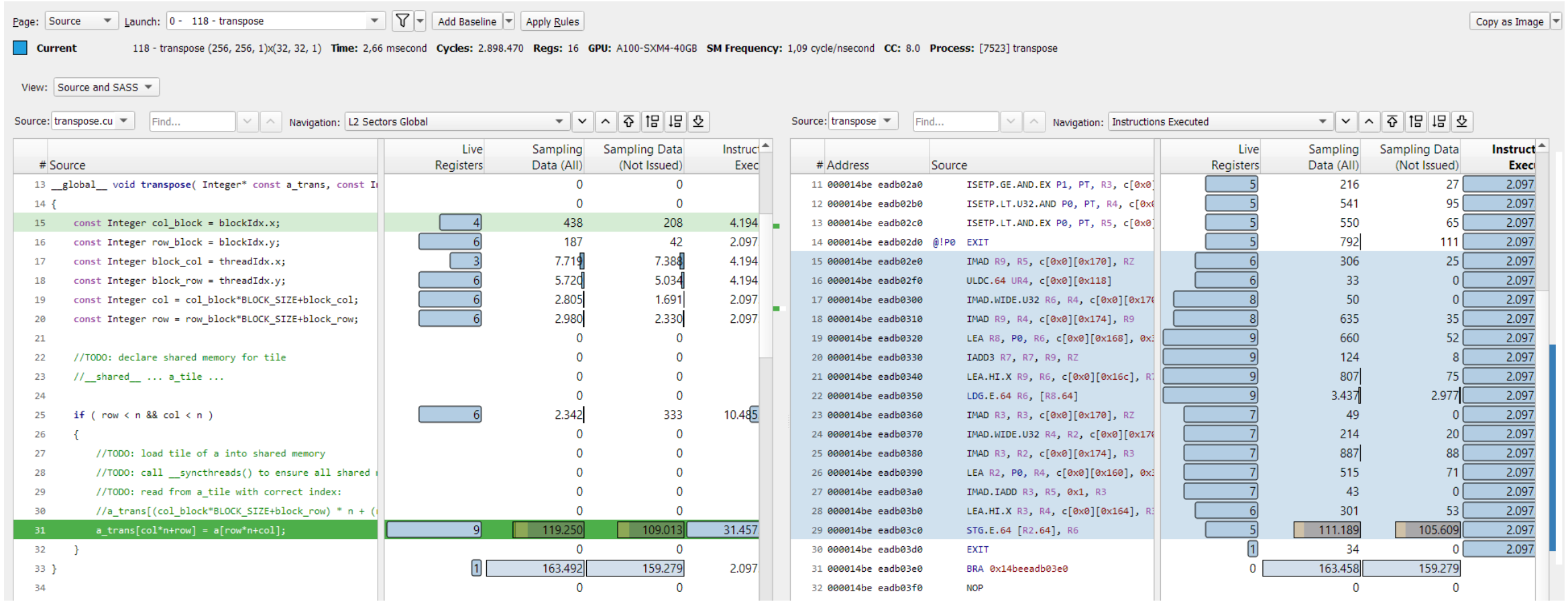
Location	Value	Value (%)
transpose.cu:15 (0x14beeadb0200 in transpose)	2.097.152	3
transpose.cu:17 (0x14beeadb0210 in transpose)	2.097.152	3
transpose.cu:17 (0x14beeadb0220 in transpose)	2.097.152	3
transpose.cu:15 (0x14beeadb0230 in transpose)	2.097.152	3
transpose.cu:18 (0x14beeadb0240 in transpose)	2.097.152	3

Recommendations

⚠ **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

ANALYSIS WITH NSIGHT COMPUTE

Iterating and comparing



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

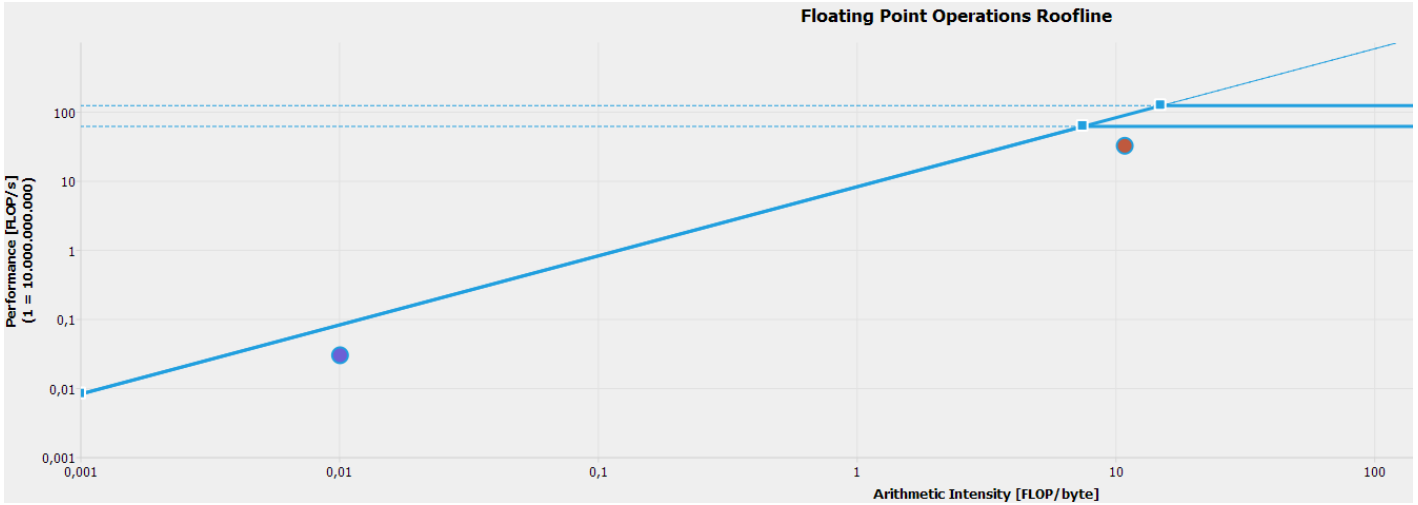
How well is the hardware utilized?

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

Counting flops and transferred bytes → AI, x-axis

Measuring achieved performance →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

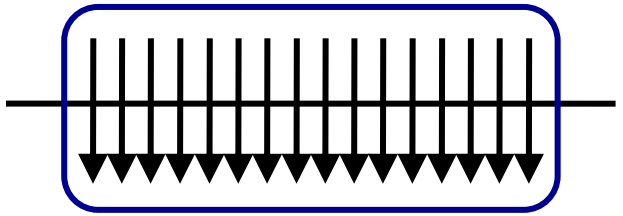
warp

group of synchronously* executing threads

(*since Volta: Independent Thread Scheduling)

neighbor threads (mostly x dimension)

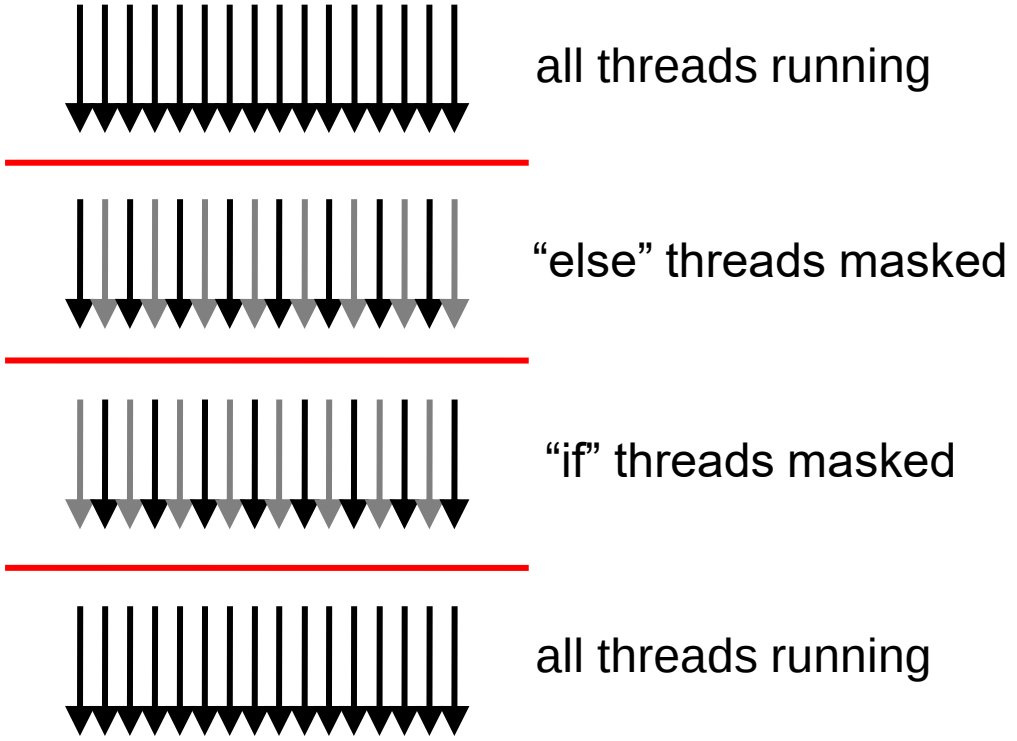
basic unit of scheduling



BRANCH DIVERGENCE

Within Warp

```
// ...
if (condition) {
  // do smth
  // ...
} else {
  // do smth else
  // ...
}
// ...
```



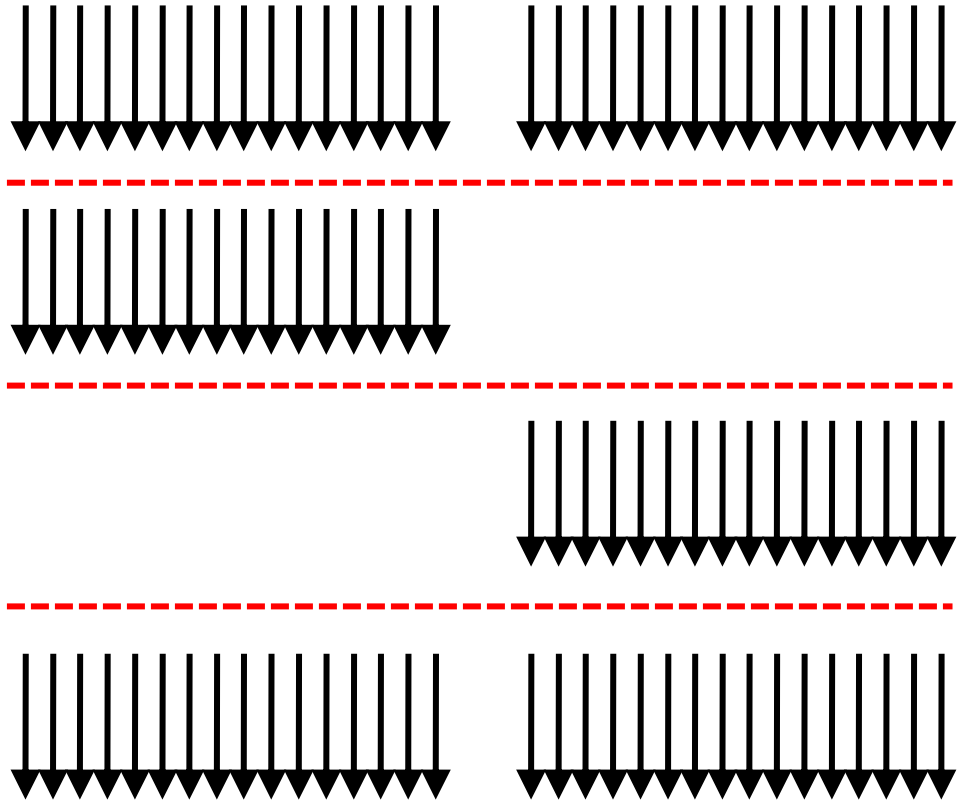
divergence *within* warp → performance penalty

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

BRANCH DIVERGENCE

Between Warps

```
// ...
if (condition) {
   // do smth
   // ...
} else {
   // do smth else
   // ...
}
// ...
```



divergence *between* warps → no penalty

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

CONCLUSION

To achieve coalesced global memory access:

- Try to use shared memory
- Look for different way of storage or better algorithm

Avoid divergent branches

Use the tools!