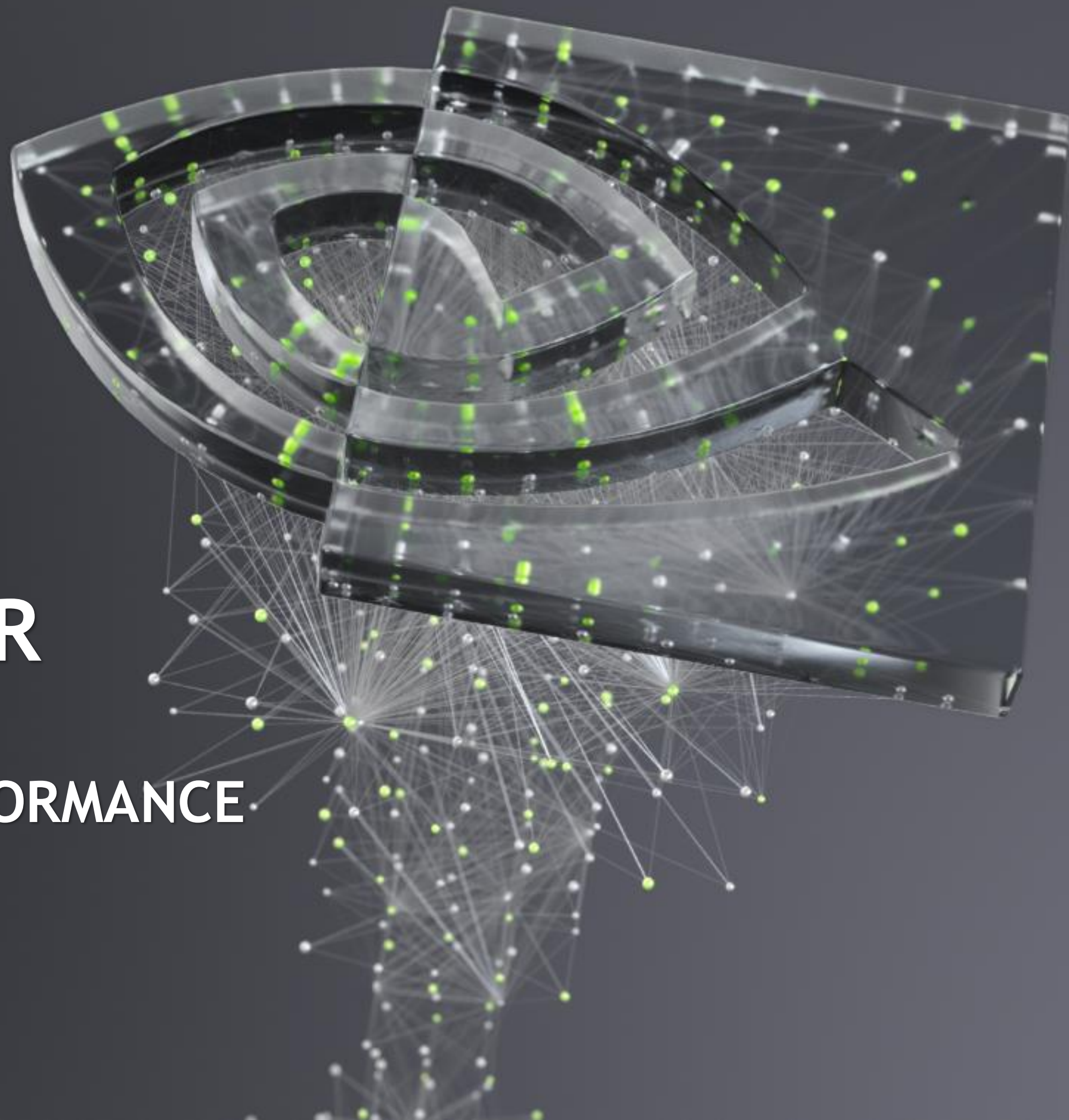




WHAT THE PROFILER IS TELLING YOU HOW TO GET THE MOST PERFORMANCE OUT OF YOUR HARDWARE

Markus Hrywniak, GTC 2020



BEFORE YOU START

The five steps to enlightenment

- Know your application
 - What does it compute? How is it parallelized? What final performance is expected?
- Know your hardware
 - What are the target machines and how many? Machine-specific optimizations okay?
- Know your tools
 - Strengths and weaknesses of each tool? Learn how to use them.
- Know your process
 - Performance optimization is a constant learning process.
- Make it so!

Outline

1. The application
2. Application-level analysis
3. Kernel-level analysis
4. Optimization steps

THE APPLICATION: LATTICE BOLTZMANN METHOD

LBM D2Q37

Reproduce dynamics of fluid by simulating virtual particles which collide and propagate

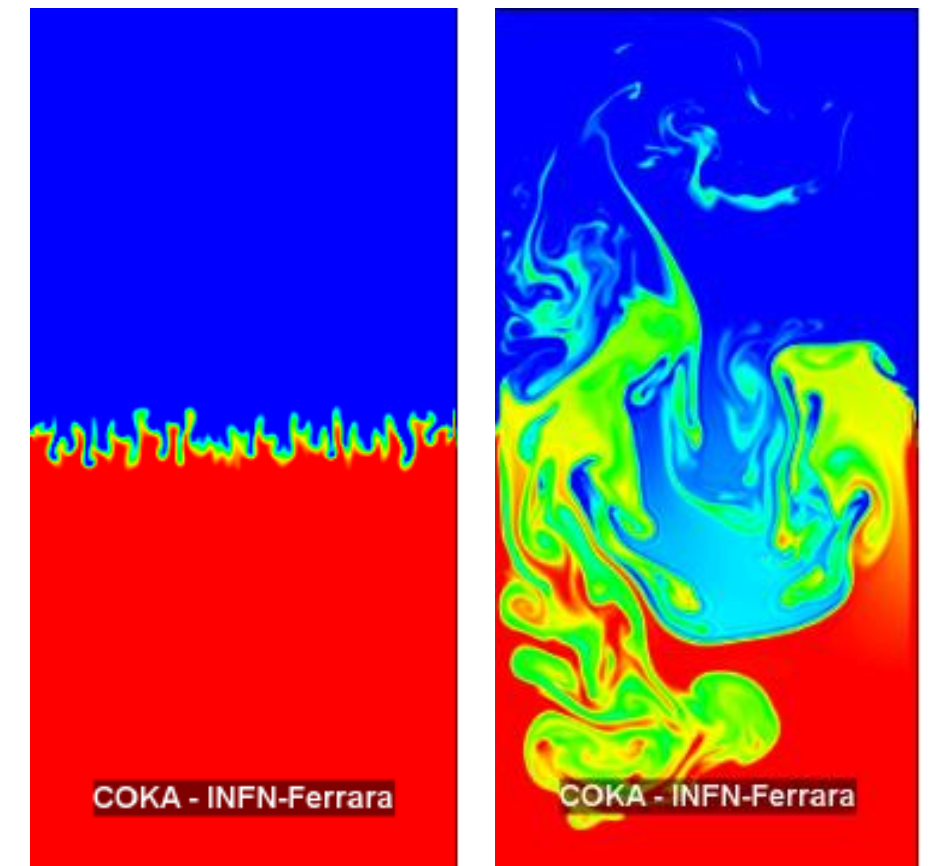
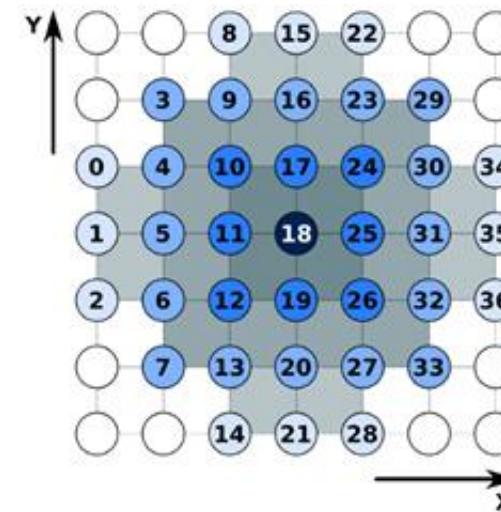
Simulation of large systems requires double precision computation and many GPUs

Application developed at U Rome Tor Vergata/INFN, U Ferrara/INFN, TU Eindhoven

Sebastiano Fabio Schifano, schifano@fe.infn.it

Candidate for next SPEC MPI Accelerator Benchmarks suite.

Variants: CUDA, OpenACC, OpenMP4, MPI.



For this talk: CUDA (+MPI). Walking through/explaining existing optimizations

LBM D2Q37

References - details of different versions

MPI + OpenMP + vector intrinsics using AoS data layout

MPI + OpenACC using SoA data layout and traditional data staging with data regions and data clauses

MPI + CUDA C++ using SoA data layout

OpenCL

Paper comparing these variants has been presented at EUROPAR 2015: „Accelerating Lattice Boltzmann Applications with OpenACC“ - E. Calore, J. Kraus, S. F. Schifano and R. Tripiccione

„Optimization of lattice Boltzmann simulations on heterogeneous computers”

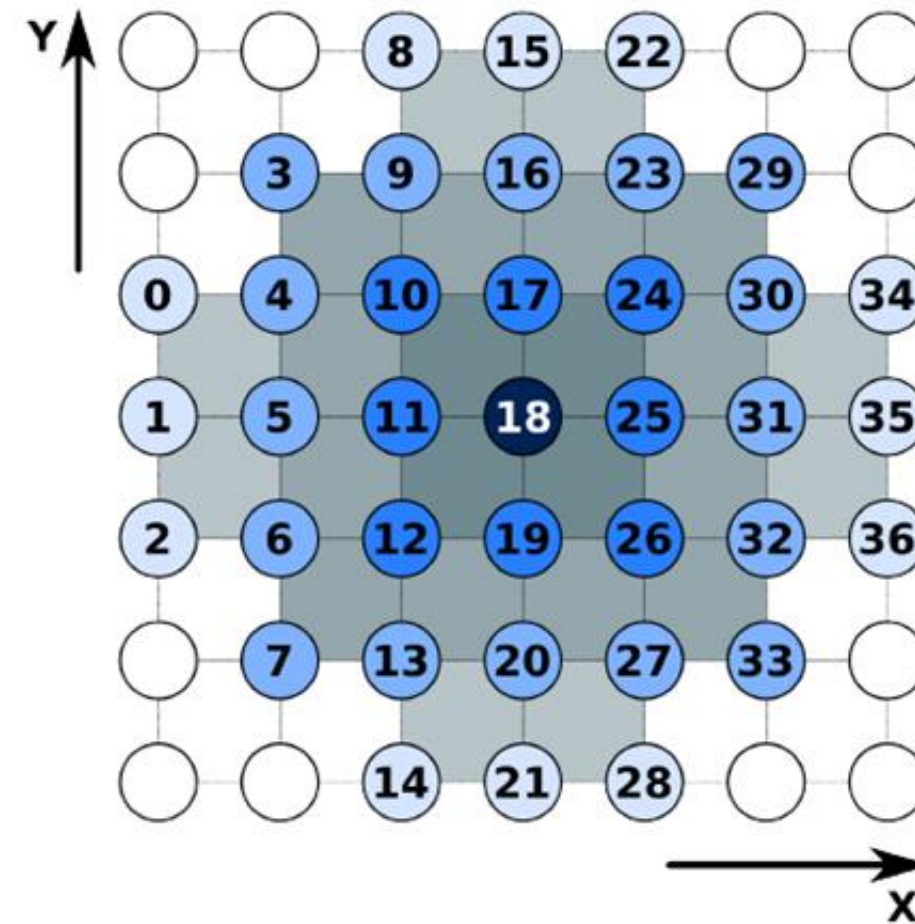
International Journal of High Performance Computing Applications

Volume 33, Issue 1, 1 January 2019, Pages 124-139 - E. Calore, A. Gabbana, S.F. Schifano, R. Tripiccione

KNOW YOUR CODE

Overview of LBM D2Q37 algorithm phases

1. One-time setup: Create and fill data structures
2. Time loop
 1. Propagate
 2. Boundary conditions
 3. Collide
3. Output and finalization



TOOLS WE WILL USE: NSIGHT SUITE

Application-wide profiling (Systems), Kernel-level profiling (Compute)

Instrument with NVIDIA Tools Extension (NVTX):
Automatic or manual

Create (nested) ranges, define macros

Compiler instrumentation

Tracing: CUDA API calls, MPI trace

Sampling, hardware counters

NVTX primer: <https://devblogs.nvidia.com/parallelforall/cuda-pro-tip-generate-custom-application-profile-timelines-nvtx/>

Nsight Systems



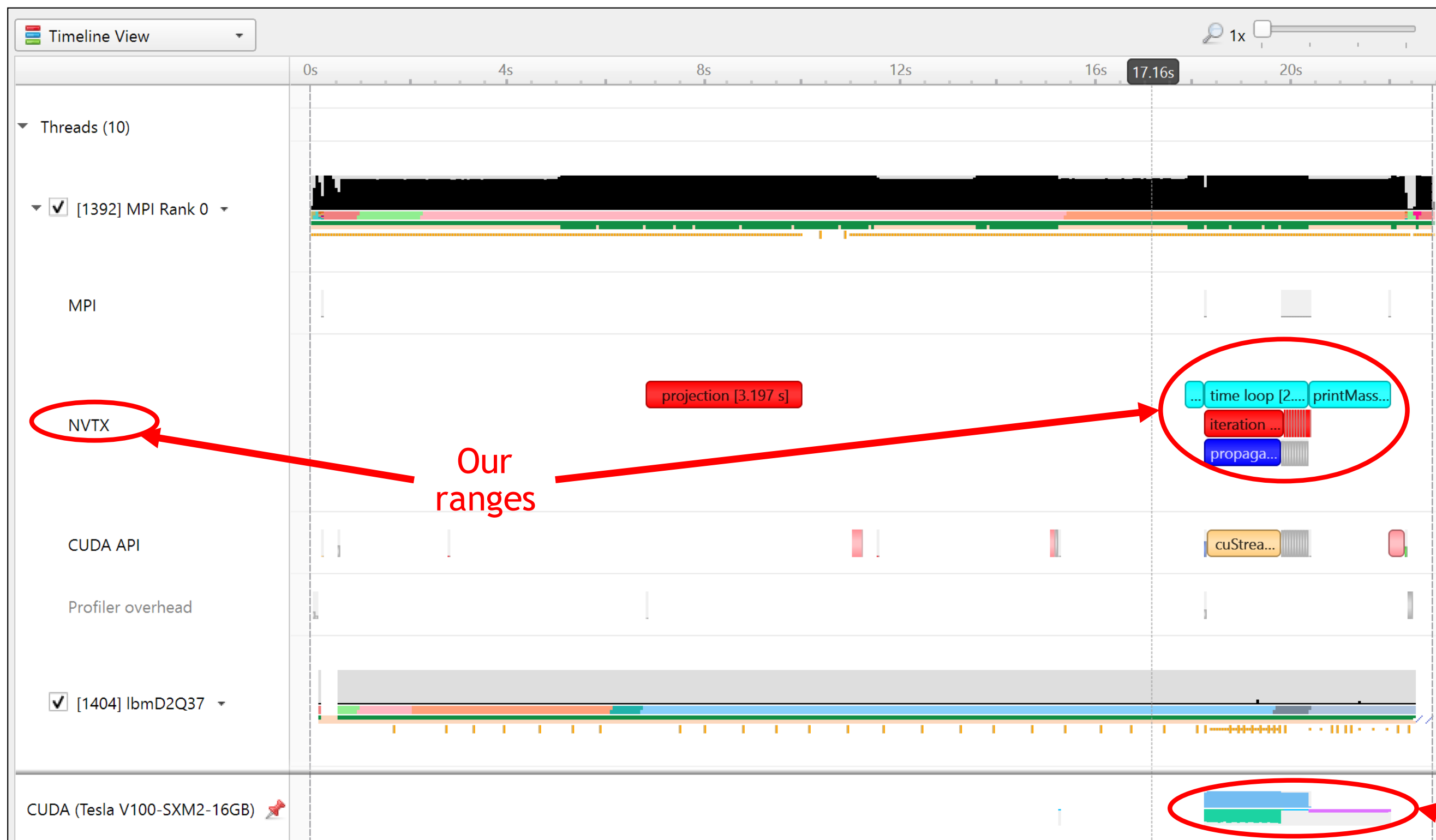
Nsight Compute





A FIRST (I)NSIGHT

Timeline overview in Nsight Systems GUI



Application already ported to GPU
- basic guidelines followed
(coalescing, data movement, SoA)

S7122: [CUDA Optimization Tips, Tricks and Techniques](#) (2017)

A FIRST (I)NSIGHT

Maximum achievable speedup: Amdahl's law

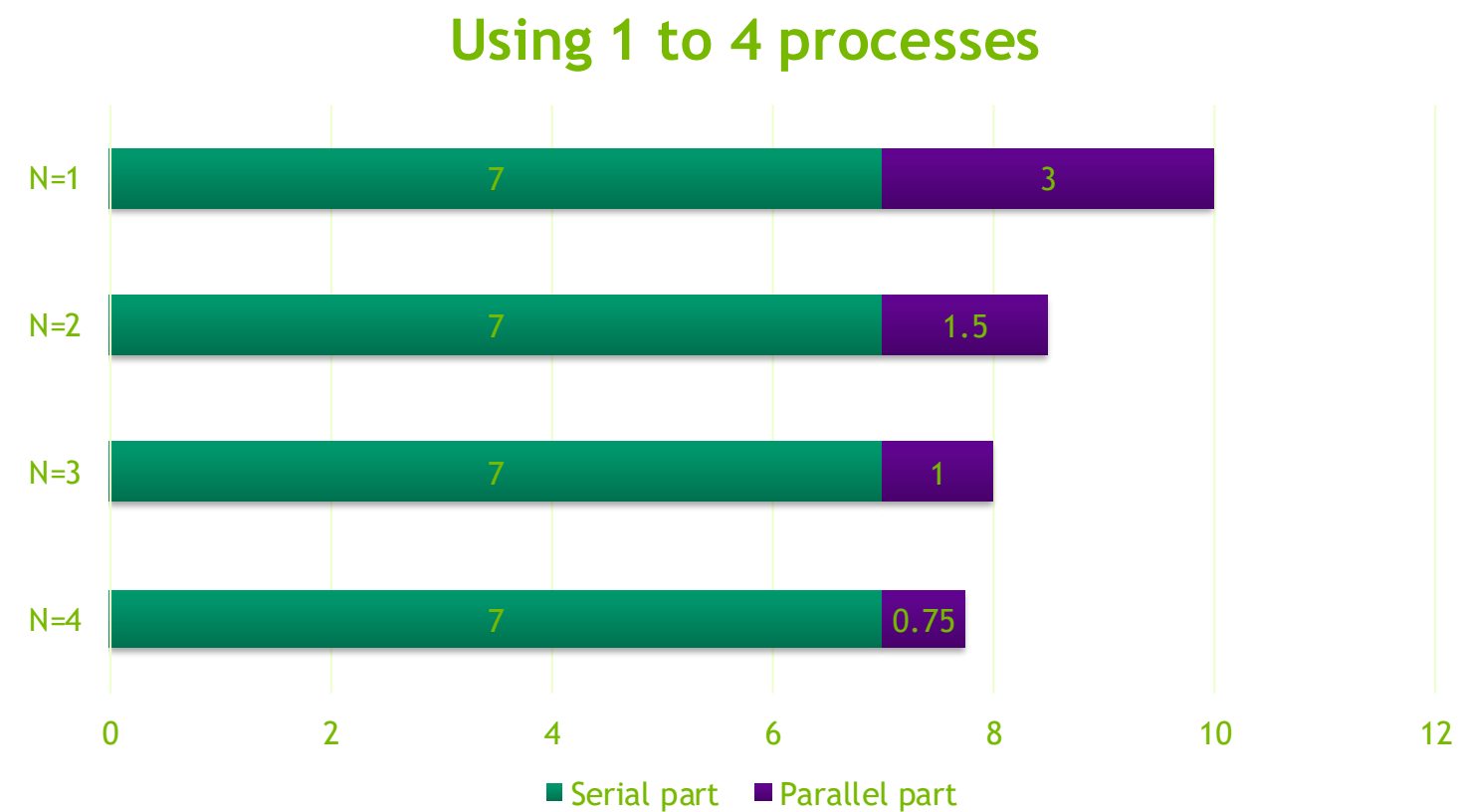
Amdahl's law states overall speedup s given the parallel fraction p of code and number of processes N

$$s = \frac{1}{1 - p + \frac{p}{N}} < \frac{1}{1 - p}$$

Limited by serial fraction, even for $N \rightarrow \infty$

Example for $p = 30\%$

Also valid for per-method speedups



A FIRST (I)NSIGHT

Recording an application timeline

1) Recording, via the GUI (not shown here) or via command line

```
mpirun -np $NP \  
nsys profile -c nvtx -p timeloop -e NSYS_NVTX_PROFILER_REGISTER_ONLY=0 --kill none \  
--trace=cuda,nvtx,mpi \  
--output=my_report.%q{OMPI_COMM_WORLD_RANK}.qdrep ./myApp
```

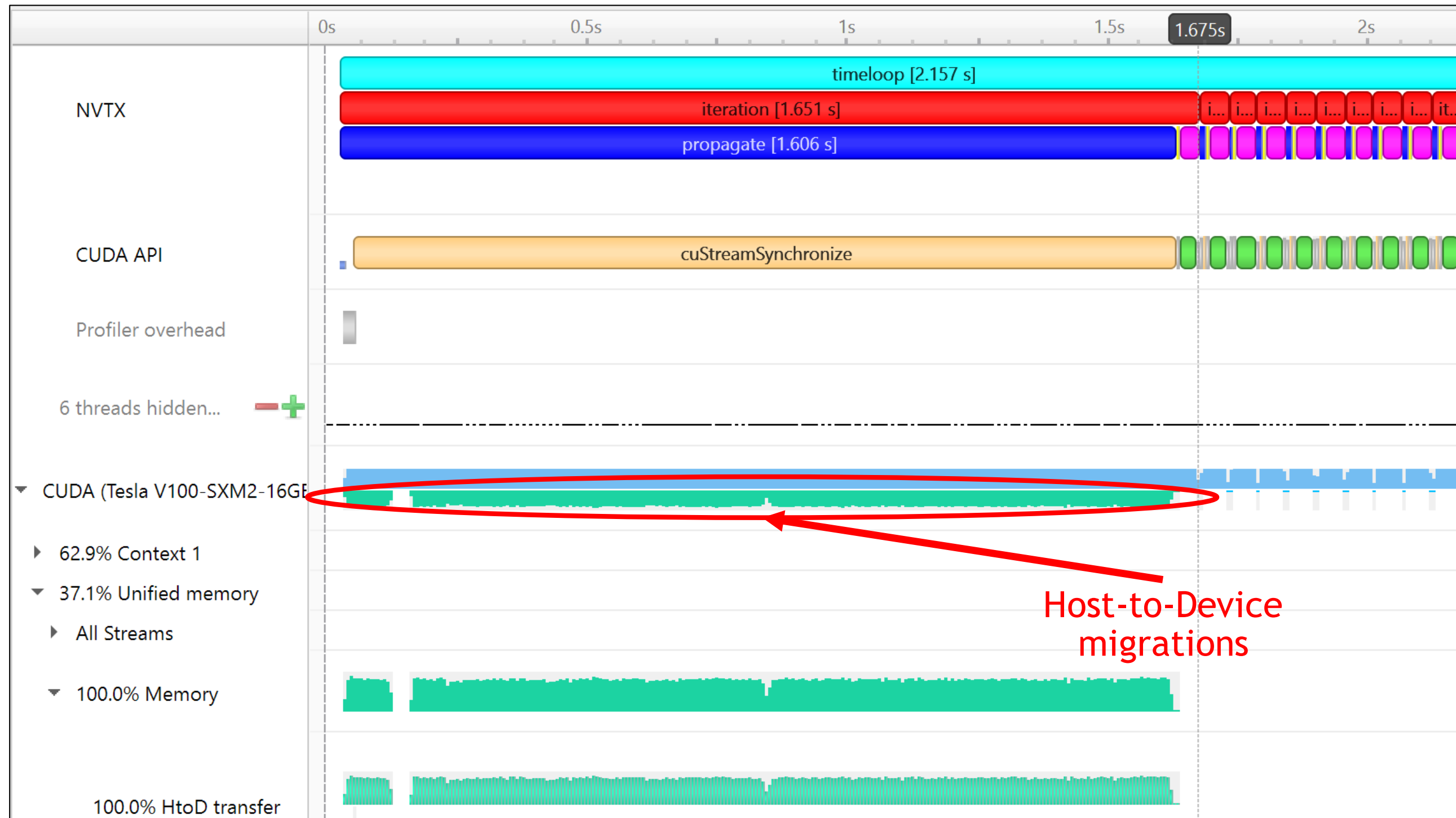
2) Inspect results: Open the report file in the GUI

Also possible to get details on command line (documentation)

See also <https://docs.nvidia.com/nsight-systems/>, "Profiling from the CLI on Linux Devices"

LOOKING CLOSER

Focusing on the time loop



Setup on host, page fault,
transfer to device

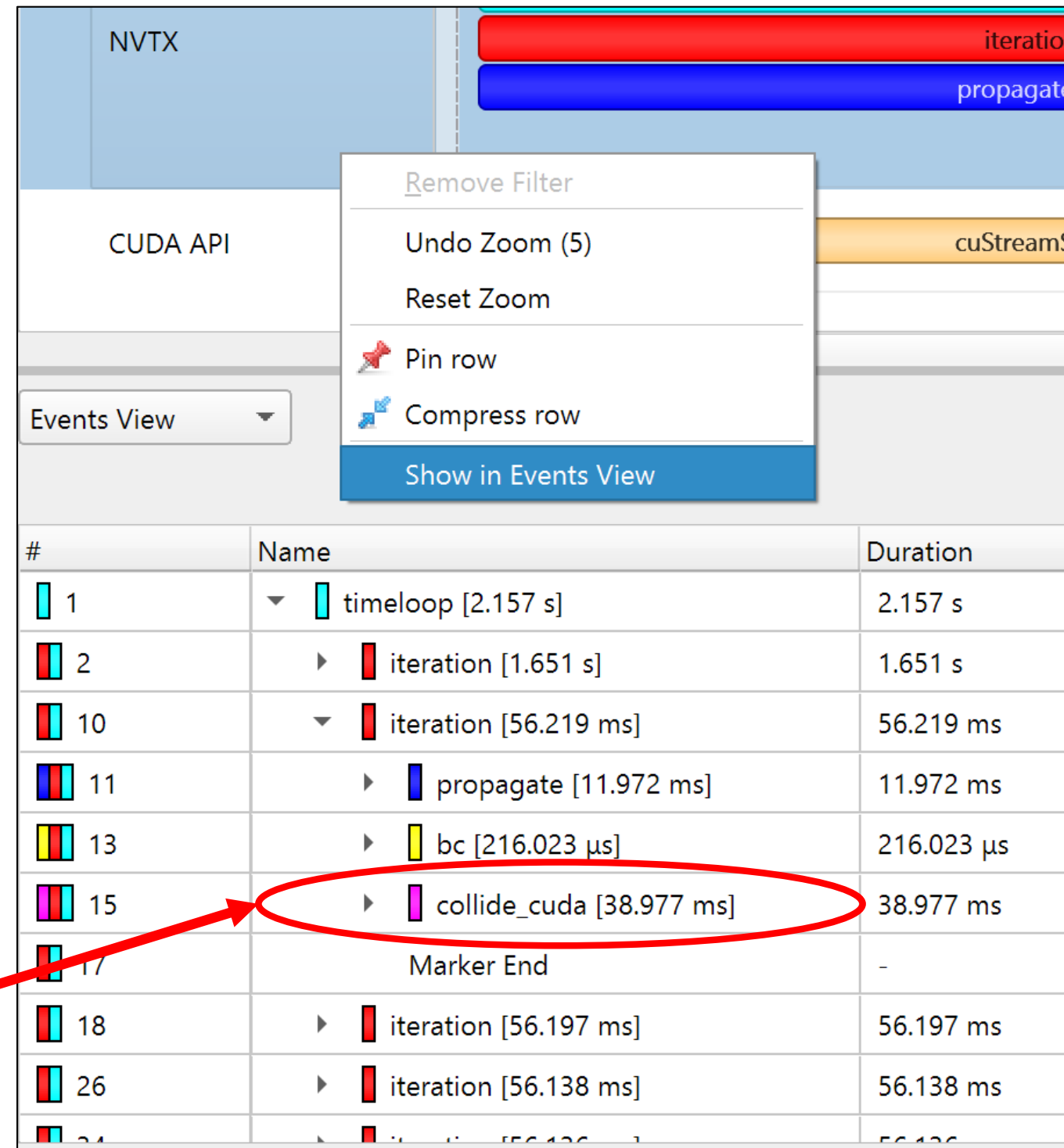
Profiling: Skip long first iteration

Unified Virtual Memory,
Managed Memory

Details in S9727: [Memory Management on Modern GPU Architectures](#) (2019)
and S8430: [Everything You Need to Know About Unified Memory](#) (2018)

LOOKING CLOSER

Focusing on the iteration



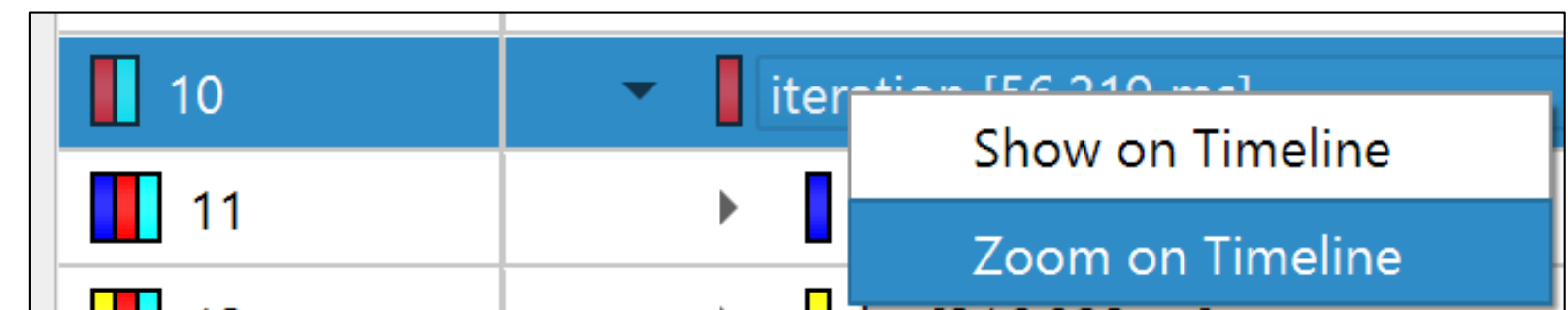
#	Name	Duration
1	timeloop [2.157 s]	2.157 s
2	iteration [1.651 s]	1.651 s
10	iteration [56.219 ms]	56.219 ms
11	propagate [11.972 ms]	11.972 ms
13	bc [216.023 µs]	216.023 µs
15	collide_cuda [38.977 ms]	38.977 ms
17	Marker End	-
18	iteration [56.197 ms]	56.197 ms
26	iteration [56.138 ms]	56.138 ms
34	iteration [56.138 ms]	56.138 ms

Our focus

Zooming in and using Events View for NVTX

Useful for other rows, e.g. CUDA API

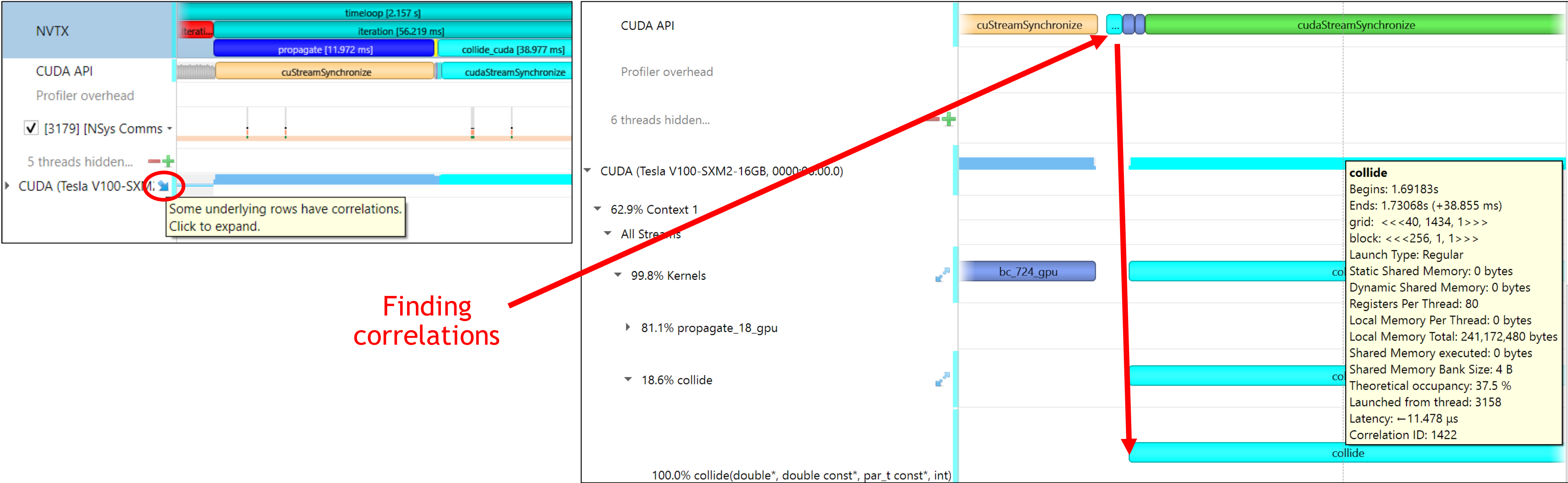
Hierarchy of ranges, use to locate on timeline:



IDENTIFYING INTERESTING REGIONS

How to correlate ranges, API and kernel calls

Basic block NVTX „iteration“. Identify components. Mark kernel in CUDA API row, find kernel launch



Finding correlations



RECAP: HIGH-LEVEL ANALYSIS

Application timeline with Nsight Systems

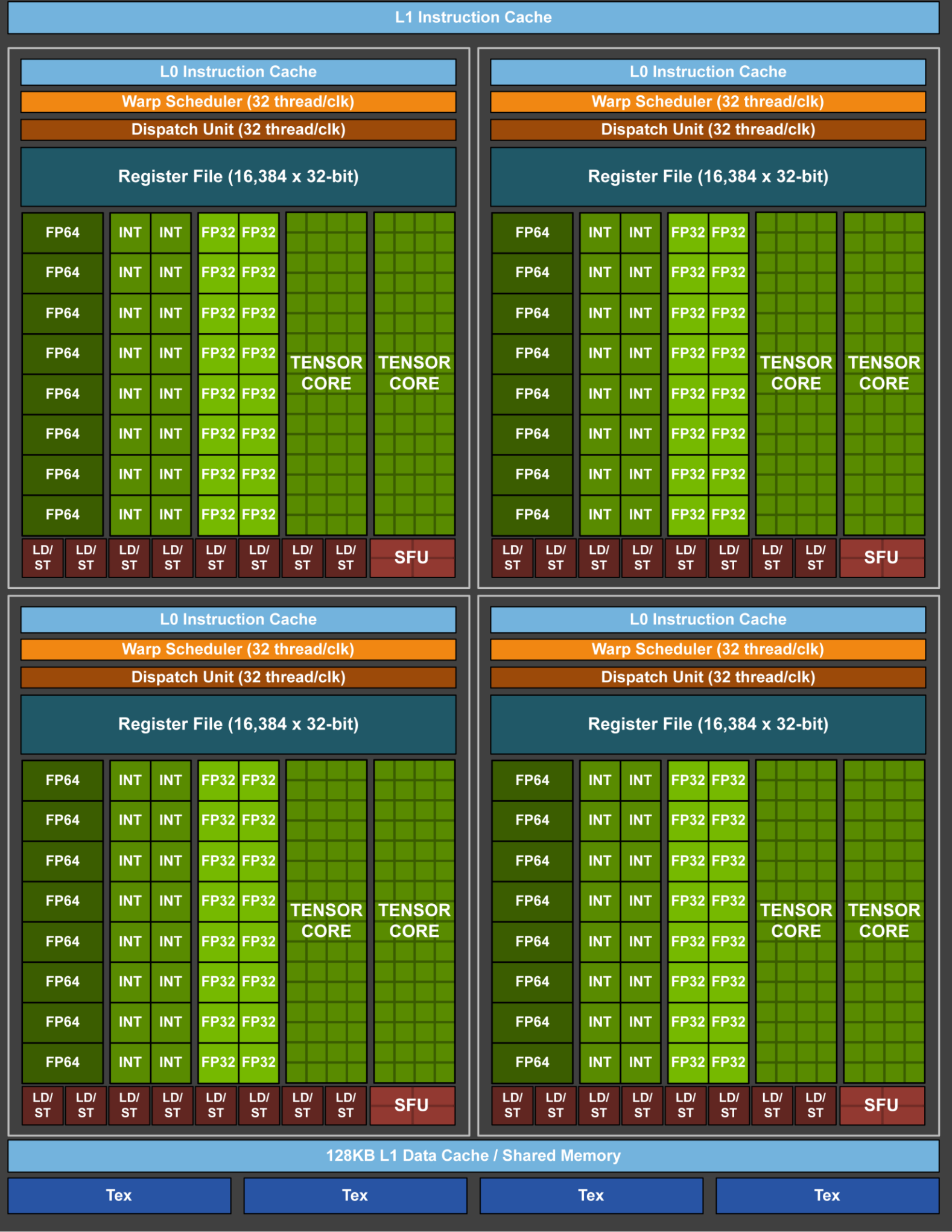
Annotated code

Analyzed regions (NVTX)

Identified first optimization target (Wallclock, Amdahl's law)

Correlated with actual kernel launch

Now: Dive into kernel-level optimization of **collide**



KNOW YOUR HARDWARE

Volta architecture - V100 SM

GV100	
FP32 units	64
FP64 units	32
INT32 units	64
Tensor Cores	8
Register File	256 KB
Unified L1/Shared memory	128 KB
Active Threads	2048

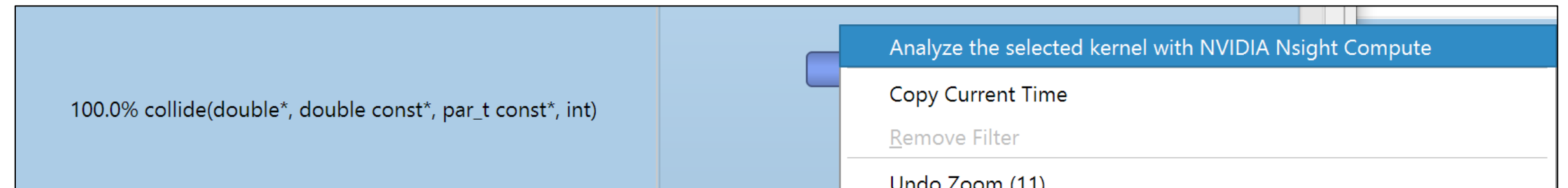
docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#compute-capabilities



DRILLING DOWN ON A KERNEL

Analysis with Nsight Compute

Right-click menu in Nsight Systems,
get command line



Run command line

```
ncu --page details --set full -k collide -s 3 -c 1 -f -o my_report ./lbmD2Q37
```

Important switches for metrics collection, pre-selected sets

Fully customizable, `ncu --help`. Check `--list-metrics` and `--query-metrics`

We use GUI for analysis and load report file

Alternatively, interactively run and analyze directly through GUI

OPTIMIZING „COLLIDE“

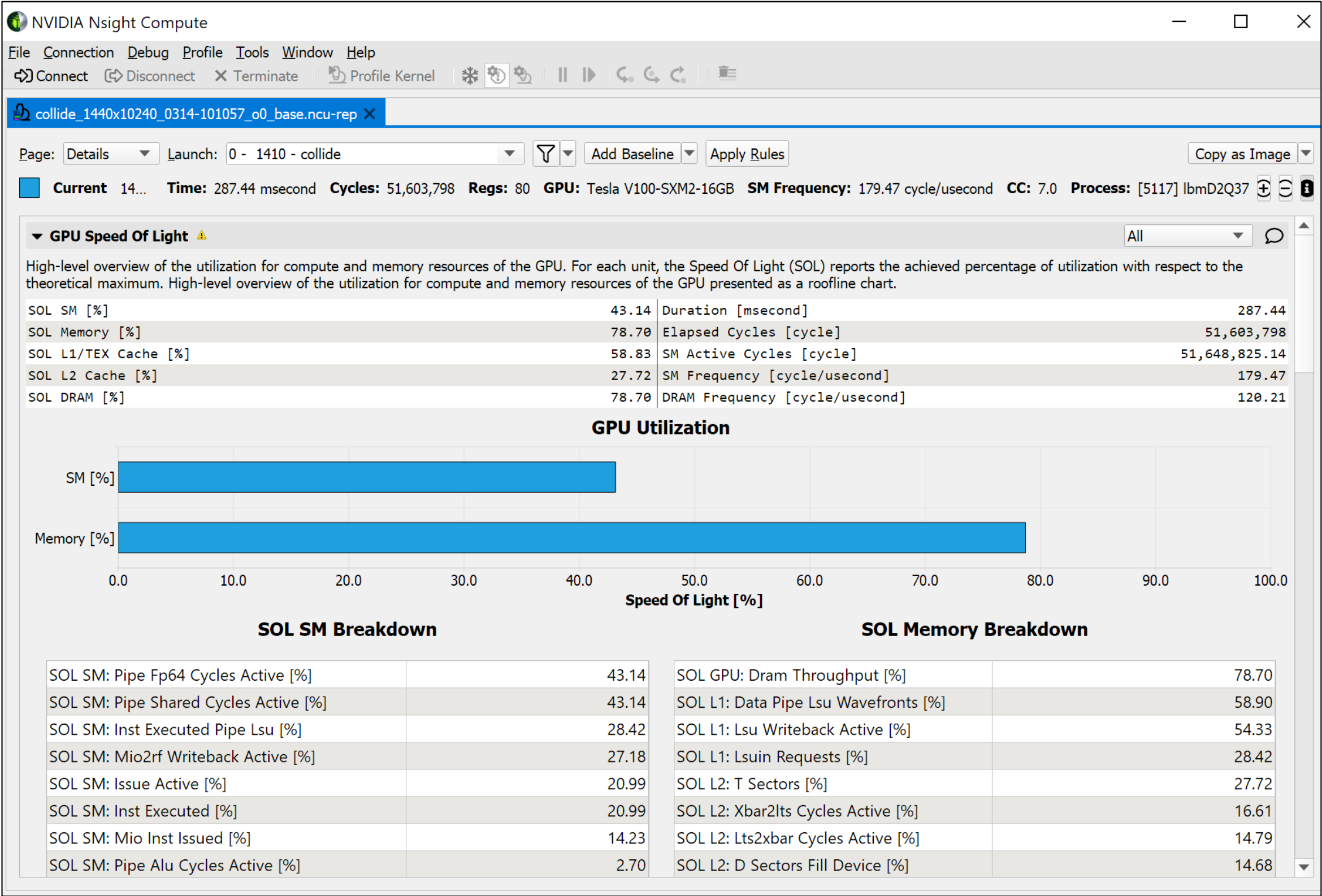
Understand initial limiter

GPU Speed Of Light (SOL)

SOL Breakdown

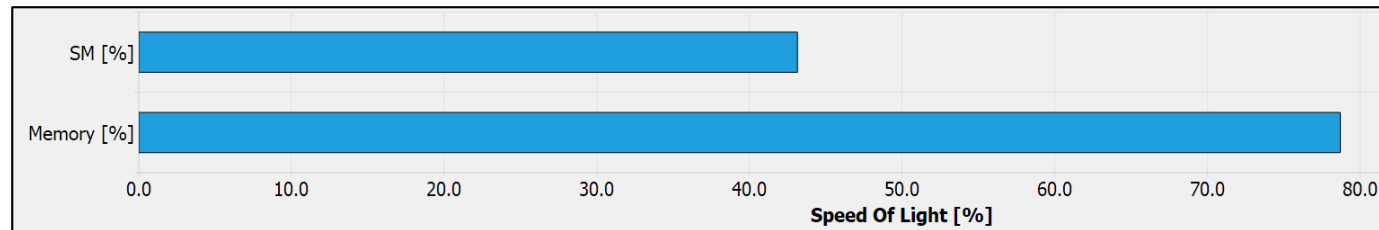
Overall, Memory (Dram) has highest utilization

→ Initial limiter found



KERNEL-LEVEL PROFILING

Performance limiter categories



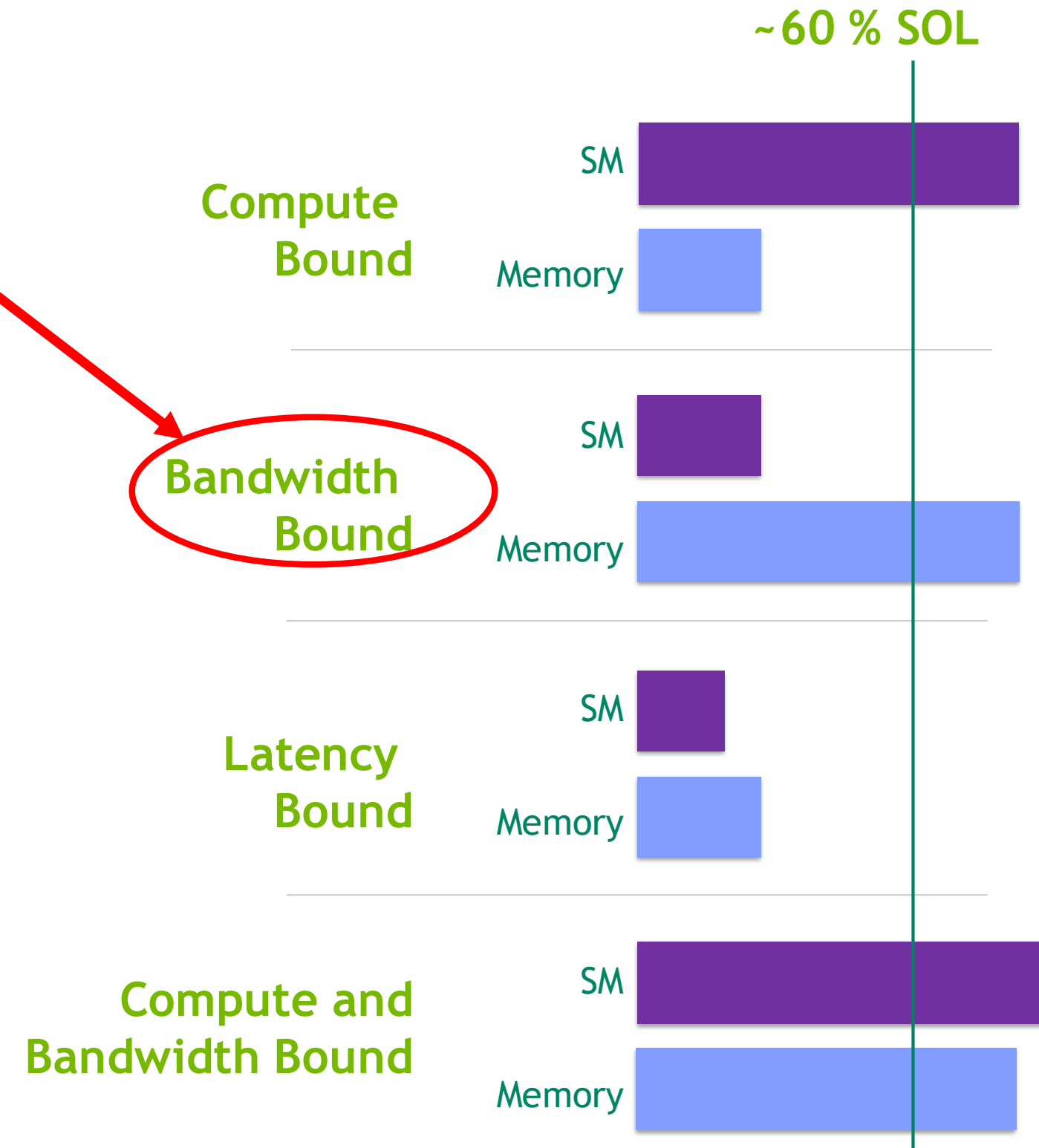
Four possible combinations of high/low...

...memory utilization

...compute utilization

Good? Bad?

→ Depends on problem and its implementation



WHY BANDWIDTH BOUND?

Using the Memory Workload Analysis

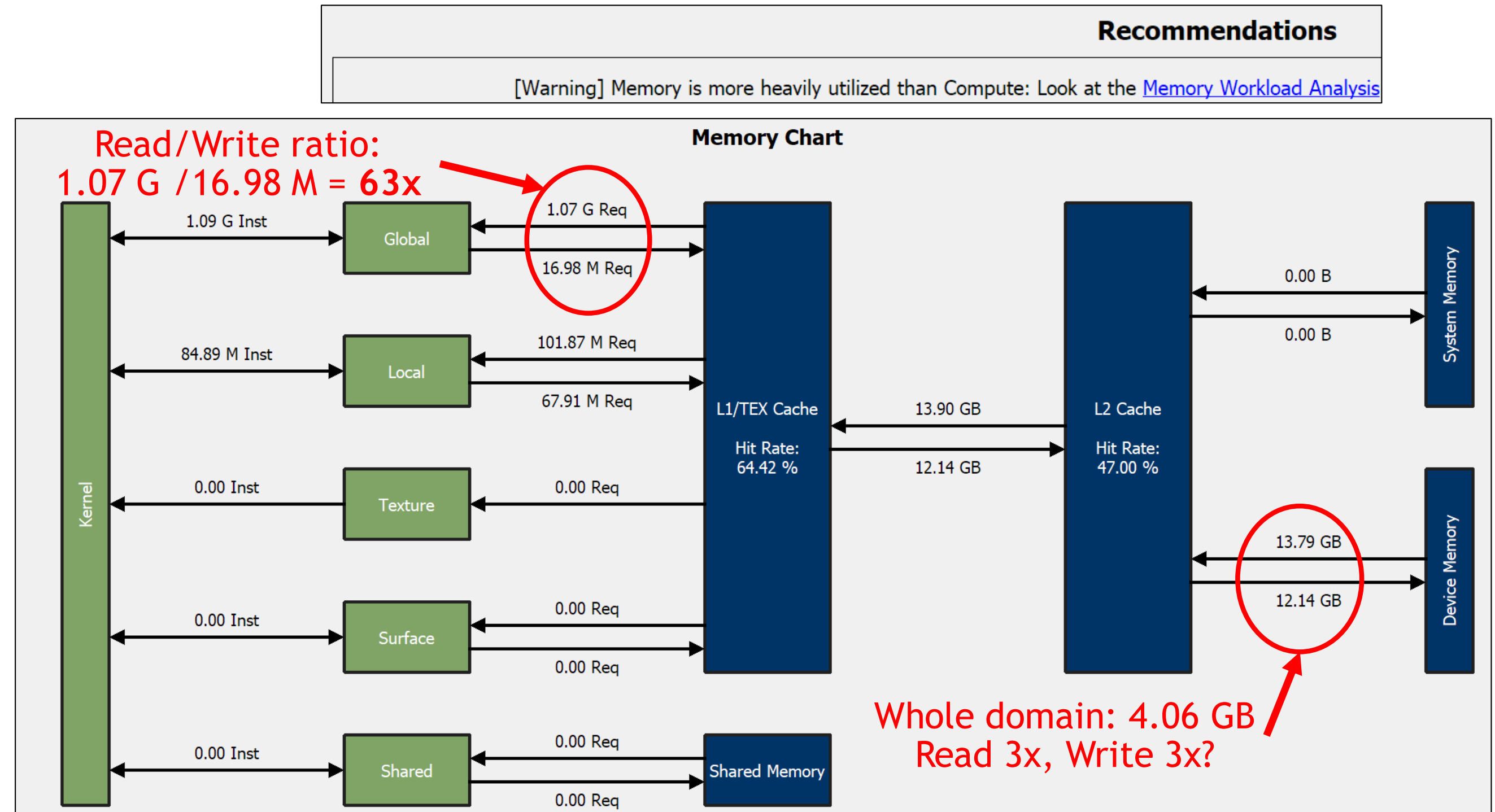
Application requirements:

"Grid size 4.06 GB"

1. Read/Write to „Global“ are not symmetric
2. Device memory traffic > 3x whole grid size

Try to understand (1), why do we read so much?

Revisit (2) later



BUILDING A HYPOTHESIS

Identifying access patterns

Kernel signature:

```
__global__ void collide (  
    data_t * __restrict__ const p_nxt,  
    const data_t * __restrict__ const p_prv,  
    const par_t* param,  
    int npop_  
)
```

Contains loops with this **coalesced** access pattern:

```
for(i = 0; i < npop_; i++) {  
    localPop[i] = p_prv[(i * NX*NY) + idx_cur];  
    popTemp = localPop[i];
```

From global: Read,
whole array to local

```
    rho = rho + popTemp;  
    u    = u + param->cx[i] * popTemp;  
    v    = v + param->cy[i] * popTemp;  
}
```

Read-only
Same \forall threads

Hypothesis: Repeated accesses to `param` cause excessive reads

INITIAL ANALYSIS OF CODE

Determined our first limiter vs. where we want to be

Discovered limiters and expectations

Used profiling tools to figure out what could be better

Required knowledge of our code (read/write ratio)

Used hints Nsight Compute provides

Now: Perform optimization and re-measure

4. Deploy
and Test

1. Assess

- Identify Performance Limiter
- Analyze Profile
- Find Indicators



3. Optimize

2. Parallelize



Build Knowledge

PROGRESS SO FAR

Overview of steps, and why we did them

Version	WCT [ms]	Observation	Next step
Baseline	39	Bandwidth-bound: 63x read/write ratio to global memory.	

COLLIDE: CONSTANT MEMORY

All threads read same location

CUDA best practices guide:

„[...] the cost scales linearly with the number of unique addresses read by all threads within a warp”

“[...] If all threads of a warp access the same location, then constant memory can be as fast as a register access.”

Ideal use case! Implement:

```
__constant__ data_t param_cx[SIZE];  
cudaMemcpyToSymbol(param_cx, param->cx, ...);
```

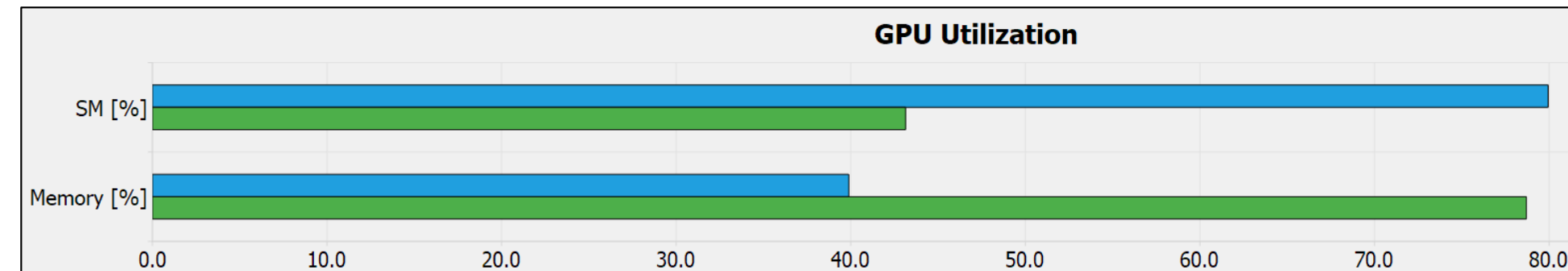
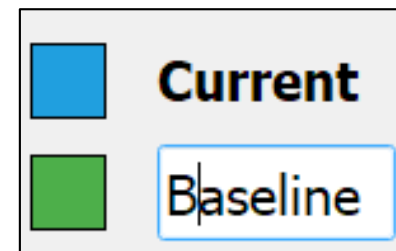
```
__global__ void collide (  
    data_t * __restrict__ p_nxt,  
    const data_t * __restrict__ p_prv,  
    const param_t* param,  
    int npop_  
)
```

COLLIDE: CONSTANT MEMORY

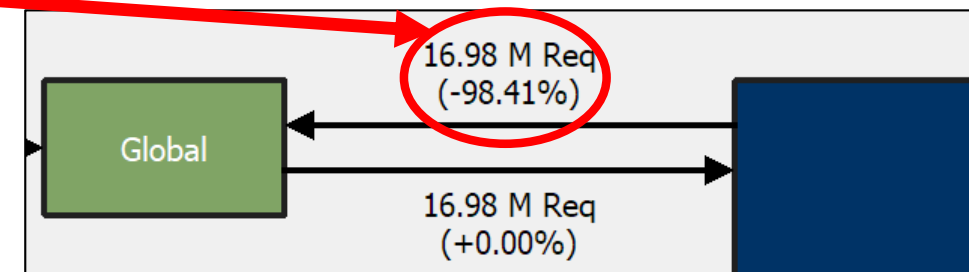
Measuring results

Record new profile, compare with Baseline feature

Add Baseline ▼



Transfers symmetric.
Success!

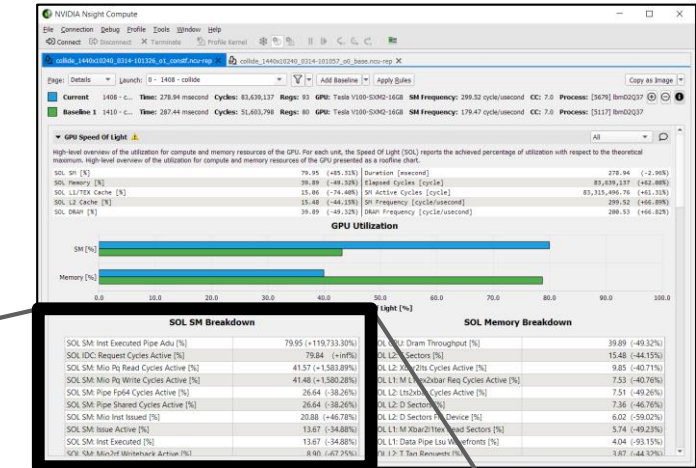


Compute bound (?), but:

Wall time increased. Current: 56 ms, Baseline: 39 ms

GO SLOW TO GO FAST?

What the intermediate slowdown means



Memory no longer bottleneck

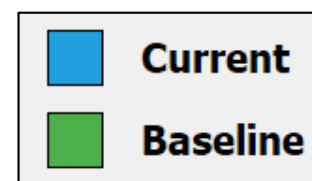
Hypothesis was correct, transfers are symmetric

SM bottleneck \neq Floating point (FP64)

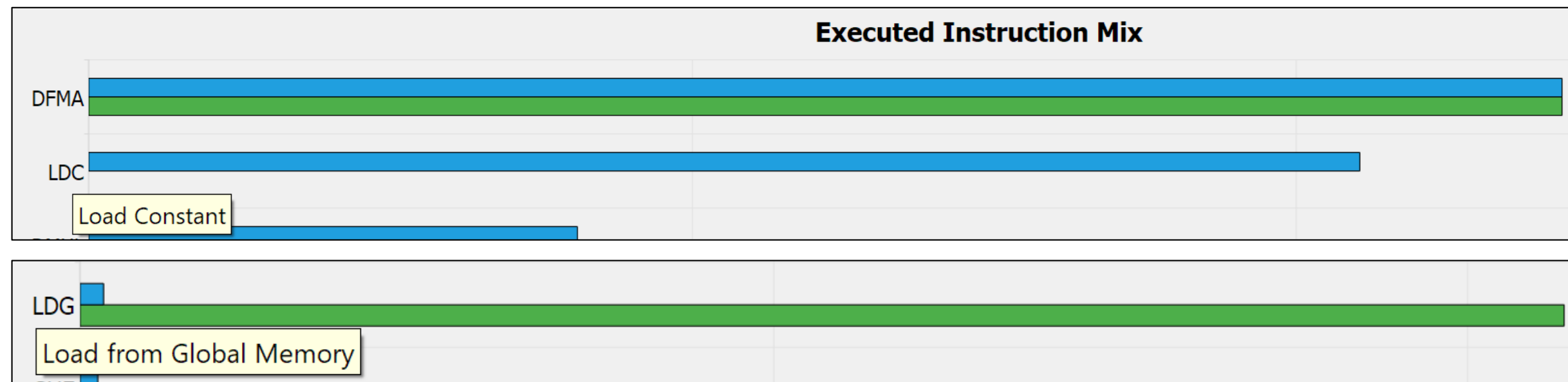
„IDC“ +inf% is suspicious,
„Indexed Constant Cache“

We swapped LDG for extra LDC!

Check „Help > Documentation“ for the
Kernel Profiling Guide



SOL SM Breakdown	
SOL SM: Inst Executed Pipe Adu [%]	79.95 (+119,733.30%)
SOL IDC: Request Cycles Active [%]	79.84 (+inf%)
SOL SM: Mio Pq Read Cycles Active [%]	41.57 (+1,583.89%)
SOL SM: Mio Pq Write Cycles Active [%]	41.48 (+1,580.28%)
SOL SM: Pipe Fp64 Cycles Active [%]	26.64 (-38.26%)



PROGRESS SO FAR

Overview of steps, and why we did them

Version	WCT [ms]	Observation	Next step
Baseline	39	Bandwidth-bound: 63x read/write ratio to global memory.	Identify excessive reads of <code>param</code> as candidate for constant memory
Constant memory	56	Read/write balanced, but slowdown; FP64 not highest utilized. Extra LDCs.	

SOURCE AND SASS

Looking at the compiler output and find the LDCs

colli

Session
Summary
Details
Source
Comments
NVTX
Raw

View: Source and SASS ▾

#	Source	Live Registers
152	for(i = 0; i < npop_; i++) {	31
153	localPop[i] = p_prv[((i) * NX*NY) + idx_cur];	21
154	popTemp = localPop[i];	
155		
156	rho = rho + popTemp;	17
157	u = u + param_cx[i] * popTemp;	23
158	v = v + param_cy[i] * popTemp;	22
159	}	

#	Address	Source	Live Registers	Sample
37	00007f1...	LDC.64 R14, c[0x3][R18+0x128]	15	
38	00007f1...	LDC.64 R16, c[0x3][R18+0x250]	15	
39	00007f1...	CS2R R54, SRZ	14	
40	00007f1...	CS2R R6, SRZ	14	
41	00007f1...	IADD3 R0, R0, 0x1, RZ	12	
42	00007f1... @P1	STL.64 [R1], R8	12	
43	00007f1... @P1	DFMA R4, R8, c[0x3][0x128], RZ	12	
44	00007f1...	STL.64 [R19], R12	12	

Correlation of CUDA source and SASS (GPU assembly). 1:N since compiler can unroll loops, reorder instructions

Calculating $u = 0 + \text{param_cx}[0] * \text{popTemp}$ in iteration $i=0$ DFMA R4, R8, c[0x3][0x128], RZ

Loading constant for loop iteration $i=1$ LDC.64 R14, c[0x3][R18+0x128]

INDEXED constant loads - Can we help statically determine constant index?

UNROLLING LOOPS

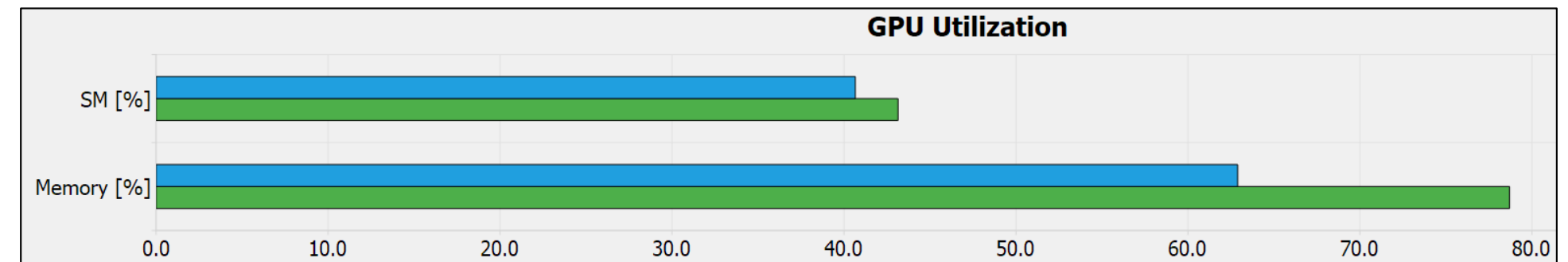
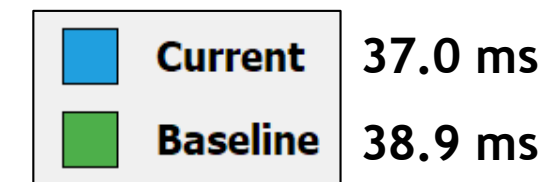
Helping the compiler

Loop bounds statically known - compiler can decide to unroll (and will)

Manually unroll with `#pragma unroll N`

Can do better: Stencil size `NPOP = 37` is fixed

Bandwidth/Latency bound



#	Address	Source
37	00007f1...	LDC.64 R14, c[0x3][R18+0x128]
38	00007f1...	LDC.64 R16, c[0x3][R18+0x250]
39	00007f1...	CS2R R54, SRZ
40	00007f1...	CS2R R6, SRZ
41	00007f1...	IADD3 R0, R0, 0x1, RZ
42	00007f1... @P1	STL.64 [R1], R8
43	00007f1... @P1	DFMA R4, R8, c[0x3][0x128], RZ
44	00007f1...	STL.64 [R19], R12



#	Address	Source
102	00007f9...	DFMA R8, R2, c[0x3][0x128], RZ
103	00007f9...	DADD R24, RZ, R2
104	00007f9...	DFMA R8, R56, c[0x3][0x130], R8
105	00007f9...	DFMA R60, R2, c[0x3][0x250], RZ
106	00007f9...	DFMA R58, R66, c[0x3][0x138], R8

No LDC, the DFMA use constants as if they were registers

STILL MEMORY-LIMITED

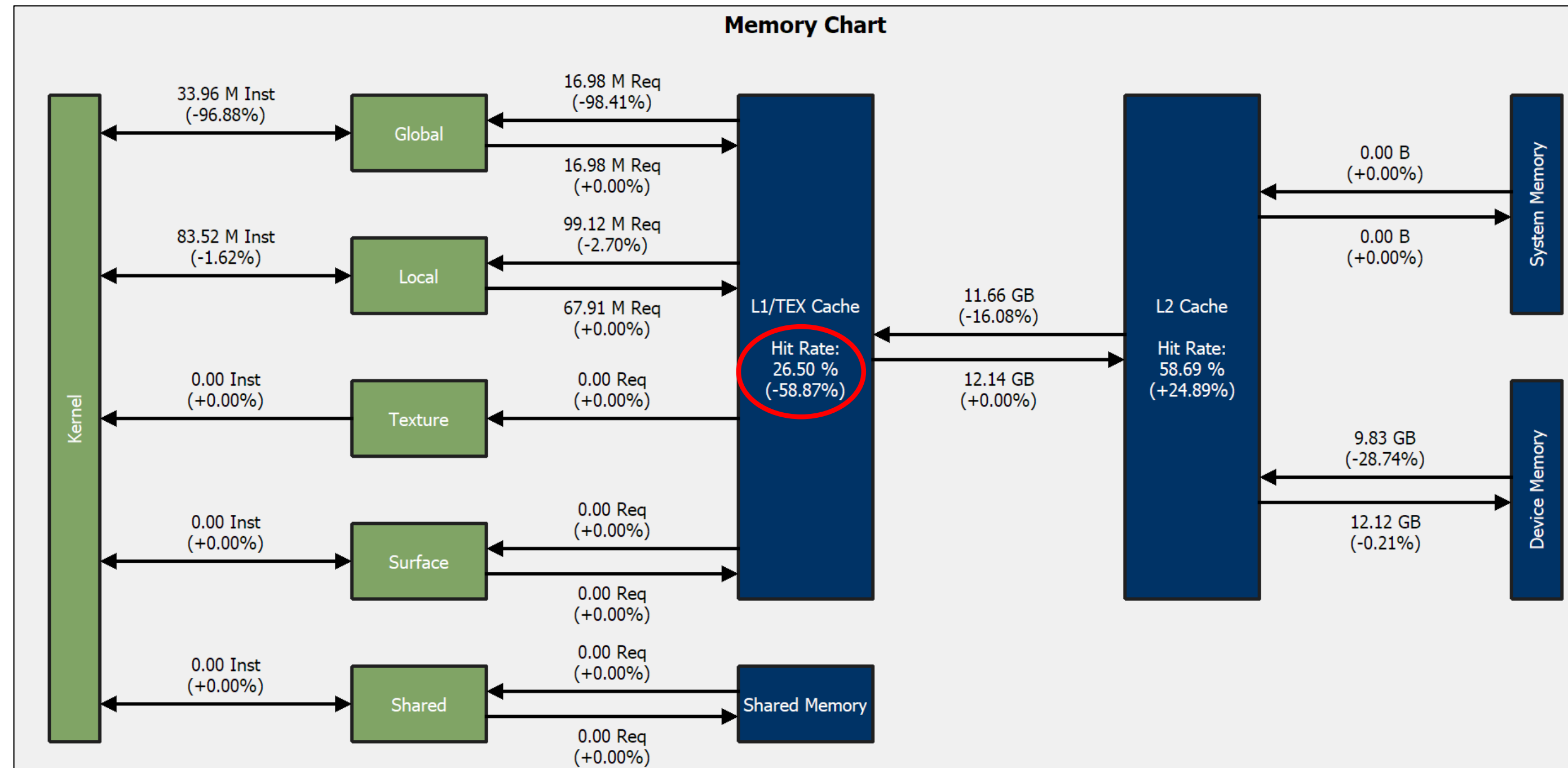
Another look at the Memory Workload

Poor hit rates L1

Size whole dataset / block:

```
stencil_size *  
threads *  
sizeof(double)  
= 37 * 256 * 8  
= 75,776 bytes  
< 128 KBytes
```

Could fit cache on V100



PROGRESS SO FAR

Overview of steps, and why we did them

Version	WCT [ms]	Observation	Next step
Baseline	39	Bandwidth-bound: 63x read/write ratio to global memory.	Identify excessive reads of <code>param</code> as candidate for constant memory
Constant memory	56	Read/write balanced, but slowdown; FP64 not highest utilized. Extra LDCs.	Identify indexed constant loads, eliminate by assisting compiler to statically determine offsets
Unrolling loops	37	Poor L1 cache hit rates. Calculation shows data from 1 block might fit.	

ABOUT OCCUPANCY

Higher may not be better

► Occupancy



Occupancy is the ratio of the number of active warps per multiprocessor to the maximum number of possible active warps. Another way to view occupancy is the percentage of the hardware's ability to process warps that is actively in use. Higher occupancy does not always result in higher performance, however, low occupancy always reduces the ability to hide latencies, resulting in overall performance degradation. Large discrepancies between the theoretical and the achieved occupancy during execution typically indicates highly imbalanced workloads.

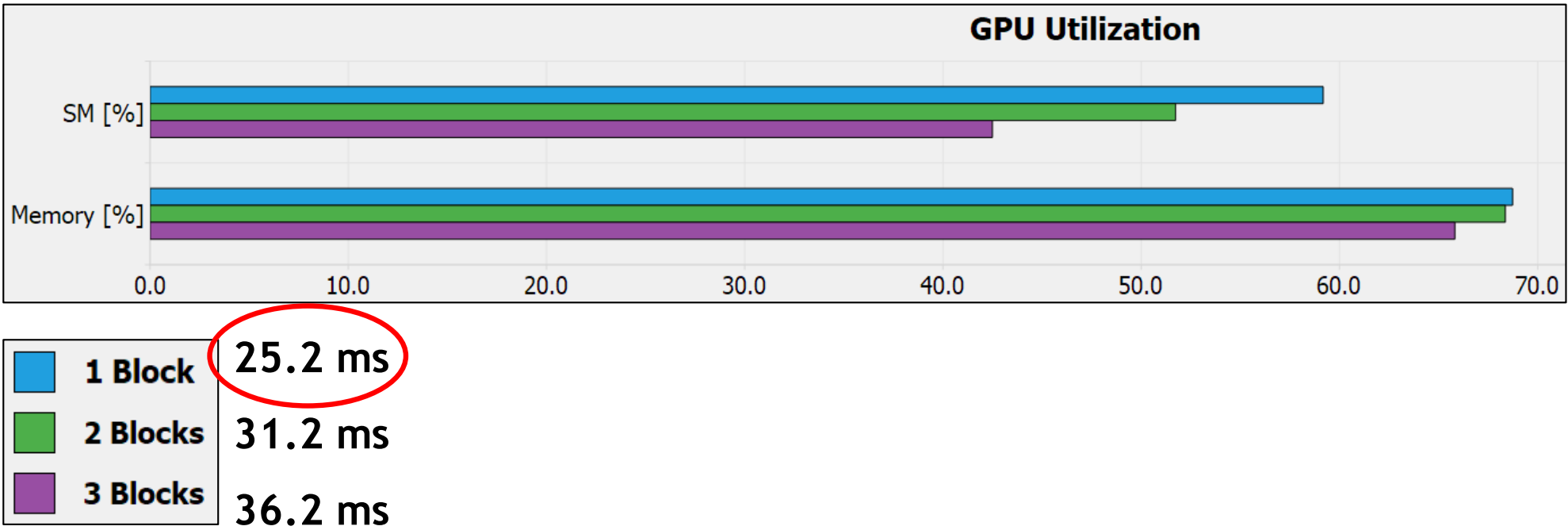
Theoretical Occupancy [%]	37.50	(+0.00%)	Block Limit Registers [block]	3	(+0.00%)
Theoretical Active Warps per SM [warp/cycle]	24	(+0.00%)	Block Limit Shared Mem [block]	32	(+0.00%)
Achieved Occupancy [%]	32.81	(-7.57%)	Block Limit Warps [block]	8	(+0.00%)
Achieved Active Warps Per SM [warp]	21.00	(-7.57%)	Block Limit SM [block]	32	(+0.00%)

Occupancy fairly low, but might be okay if we can hide memory accesses

Lowering occupancy can increase cache reuse

Run experiments with launch bounds from 1..3 blocks

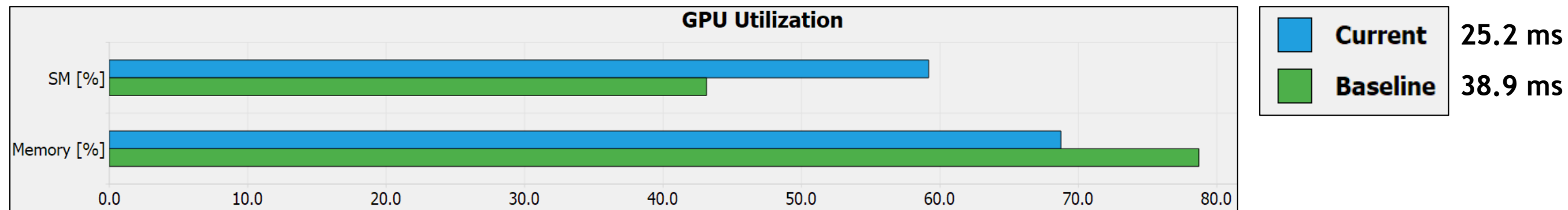
```
__launch_bounds__(256, NUM_BLOCKS)
__global__ void collide (
    data_t * __restrict__ const p_nxt,
    const data_t * __restrict__ const p_prv
)
```



<https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#launch-bounds>

SPEEDUP ACHIEVED

Constant cache + Unrolling + Launch bounds

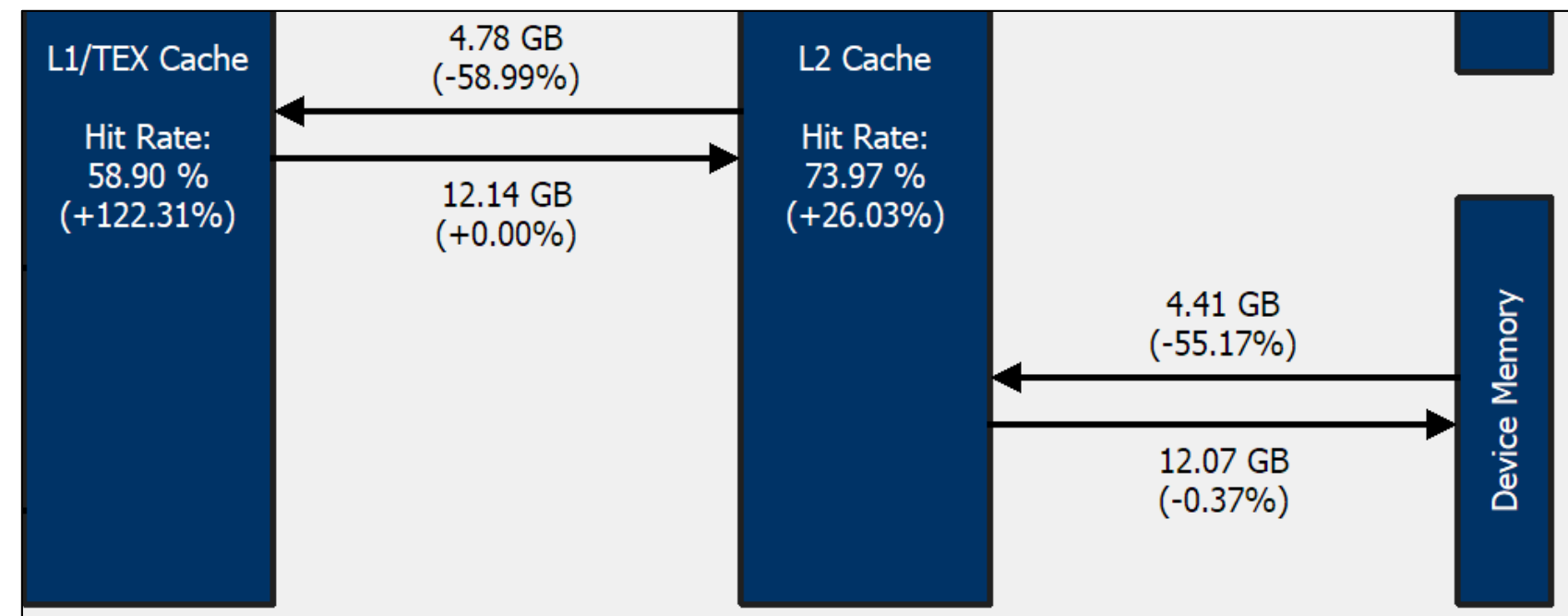


Current version: Only 1 block/SM.

Comparing current 1 block version with previous (unrolled) version:

Less device traffic, better cache hit rates
254 registers vs. 80 registers

Still: Caching not optimal, could achieve 100% L1 for single block. Also: Extra writes.



PROGRESS SO FAR

Overview of steps, and why we did them

Version	WCT [ms]	Observation	Next step
Baseline	39	Bandwidth-bound: 63x read/write ratio to global memory.	Identify excessive reads of <code>param</code> as candidate for constant memory
Constant memory	56	Read/write balanced, but slowdown; FP64 not highest utilized. Extra LDCs.	Identify indexed constant loads, eliminate by assisting compiler to statically determine offsets
Unrolling loops	37	Poor L1 cache hit rates. Calculation shows data from 1 block might fit.	Experimenting with launch bounds, determining if lower occupancy helps caching
Launch bounds, 1 block	25	Still bandwidth-bound. Not all data cached, extra writes.	

OPTIMAL TRANSFERS

Using Shared Memory as manual cache

Size of whole dataset per block = 75,776 bytes < 128 KBytes

Manually caching with shared memory (max. 96 KBytes)

Important: On V100, need to explicitly opt in, default is <= 48 KBytes

```
int maxbytes = 98304; // 96 KB
int usedbytes = 75776;
cudaFuncSetAttribute(collide, cudaFuncAttributeMaxDynamicSharedMemorySize, maxbytes);
collide <<< ..., ..., usedbytes, ...>>> (...);
```

```
// Inside „collide“
extern __shared__ data_t localPopStore[];
data_t *localPop = &localPopStore[tx*NPOP];
```

RESULT OF MANUAL CACHING

Comparing Shared Memory, Launch bounds and Unrolled versions

Data fits "cache"

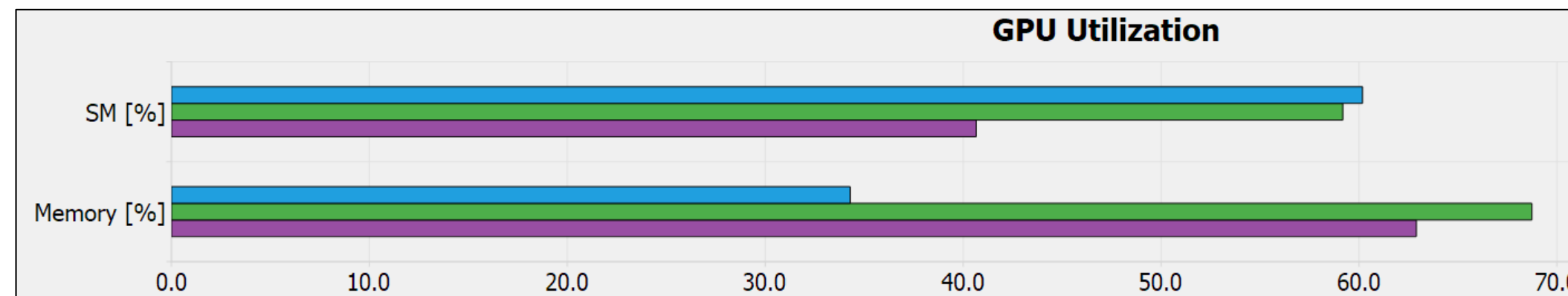
Lower bound on data transfers

Latency/Compute bound

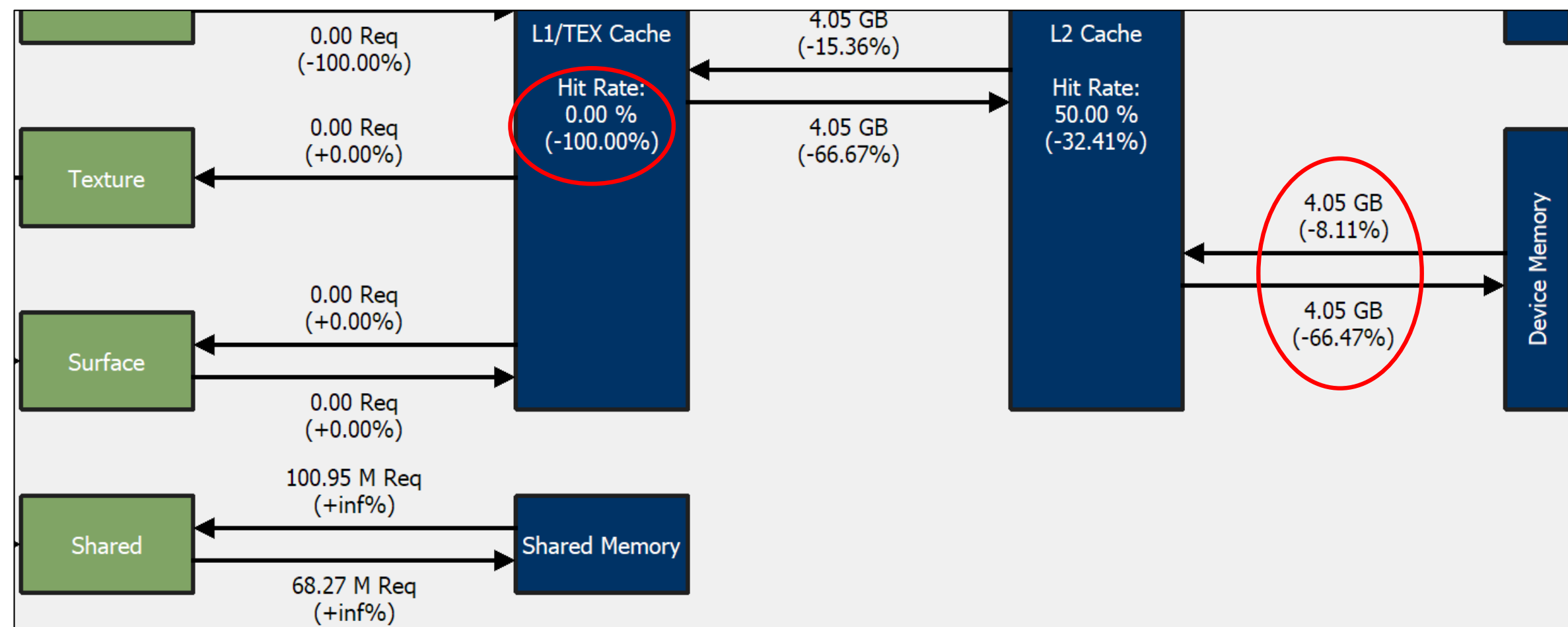
1x read, 1x write

L1 hit rate 0%, nothing left to cache

L2 hit rate 50% due to metric; no reads, all writes "hit"



Current	24.3 ms
LB, 1 Block	25.2 ms
Unrolled	37.0 ms



PROGRESS SO FAR

Overview of steps, and why we did them

Version	WCT [ms]	Observation	Next step
Baseline	39	Bandwidth-bound: 63x read/write ratio to global memory.	Identify excessive reads of <code>param</code> as candidate for constant memory
Constant memory	56	Read/write balanced, but slowdown; FP64 not highest utilized. Extra LDCs.	Identify indexed constant loads, eliminate by assisting compiler to statically determine offsets
Unrolling loops	37	Poor L1 cache hit rates. Calculation shows data from 1 block might fit.	Experimenting with launch bounds, determining if lower occupancy helps caching
Launch bounds, 1 block	25	Still bandwidth-bound. Not all data cached, extra writes.	Using shared memory as explicit cache
Shared memory	24	Achieved minimum data transfers. Latency bound.	

ROOFLINE ANALYSIS

Theoretical vs. Empirical: Approaching the top?

Calculate Arithmetic Intensity:

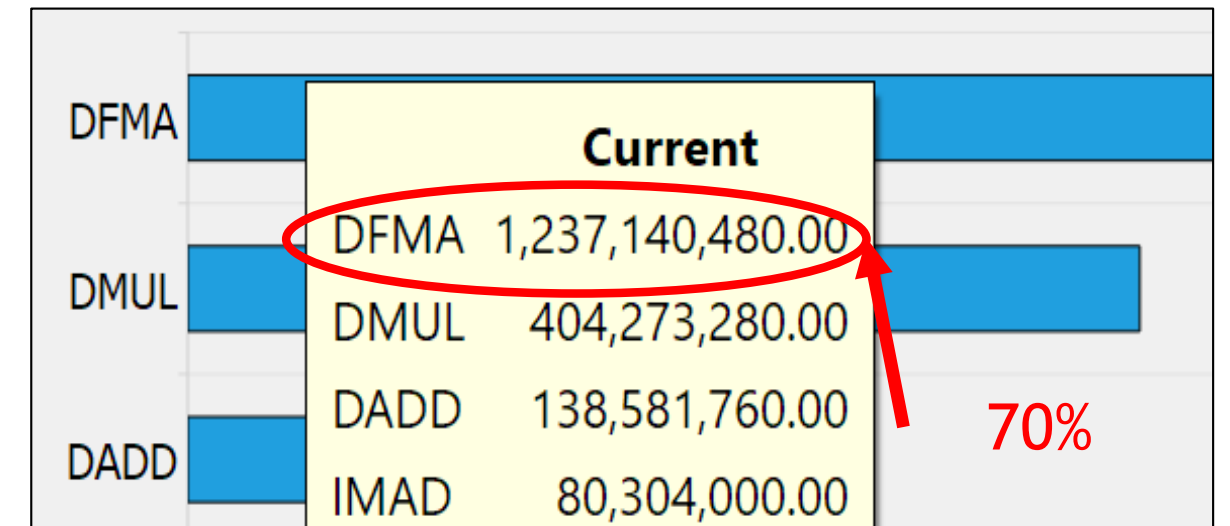
$$AI \left[\frac{\text{FLOP}}{\text{byte}} \right] = \frac{N_{FLOPS}}{\text{bytes transferred}}$$

Manually counting operations, assuming "perfect" data movement:
~12 FLOP/byte

Empirically, measuring DRAM traffic and counting FLOP from metrics:
~11.1 FLOP/byte

Achieved % of device peak FP64 („op_dfma“ peak) for full DFMA: ~51%

Adjusted for our instruction mix (only 70% DFMA): ~73%



Peak PF 64 is for full DFMA

$$p_{FMA} = \frac{\text{DFMA}}{\text{DFMA} + \text{DMUL} + \text{DADD}}$$

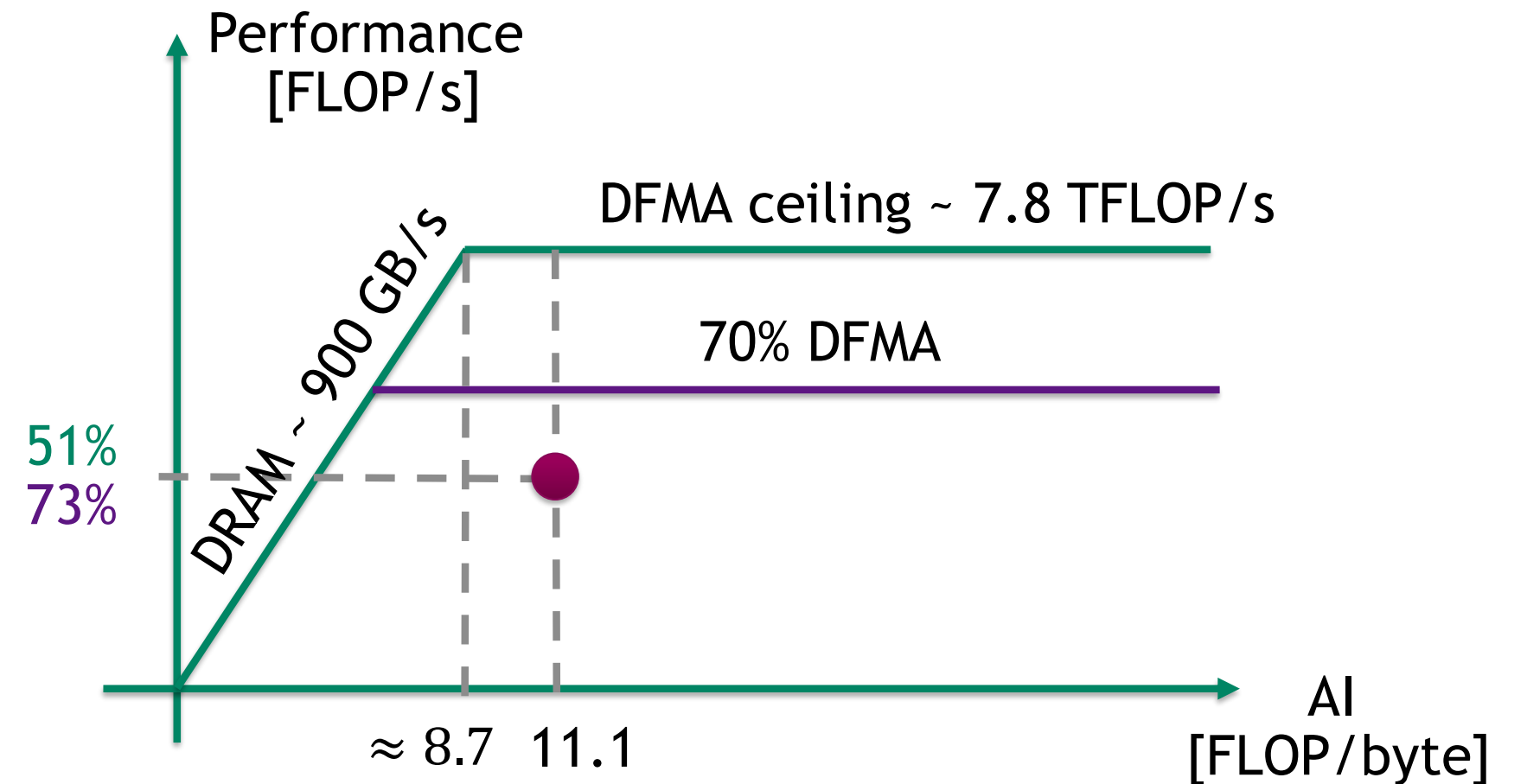
which is 70% for our code

ROOFLINE ANALYSIS

Visualizing the rooflines

Achieved % of device peak FP64 („op_dfma“ peak)
for full DFMA: ~51%

Adjusted for our instruction mix (only 70% DFMA):
~73%



S9624: [Performance Analysis of GPU-Accelerated Applications using the Roofline Model](#) (2019)

GAP TO THE PEAK

Outlook and possible reasons for the gap

Pipeline stalls: Use „Source Counters“ section

Sampling Data (Not Issued)	
Location	Value
collide_cuda_constf-npop-lb-smem.cu:155 (0x7fea3edae1f0 in collide)	26,572
collide_cuda_constf-npop-lb-smem.cu:155 (0x7fea3edaddf0 in collide)	7,404
collide_cuda_constf-npop-lb-smem.cu:382 (0x7fea3edbf520 in collide)	6,864
collide_cuda_constf-npop-lb-smem.cu:155 (0x7fea3edade20 in collide)	6,731
collide_cuda_constf-npop-lb-smem.cu:155 (0x7fea3edaddc0 in collide)	5,589

26,572 (2.30%)

smsp_pcsamp_warps_issue_stalled_long_scoreboard_not_issued: 26,561

smsp_pcsamp_warps_issue_stalled_mio_throttle_not_issued: 10

smsp_pcsamp_warps_issue_stalled_misc_not_issued: 1

Go to corresponding source (and SASS):

#	Source	stall_long_sb	Sampling Data (Not Issued)
154	<code>for(i = 0; i < npop_; i++) {</code>	0	772
155	<code>localPop[i] = p_prv[(i) *</code>	26,905	93,720
156	<code>popTemp = localPop[i];</code>	0	0

„Stall: Long Scoreboard“ is due to memory latency, warp waiting for data from device memory

Next steps: Would need to rethink ordering of memory accesses and computations

SUMMARY OF STEPS

Overview of steps, and why we did them

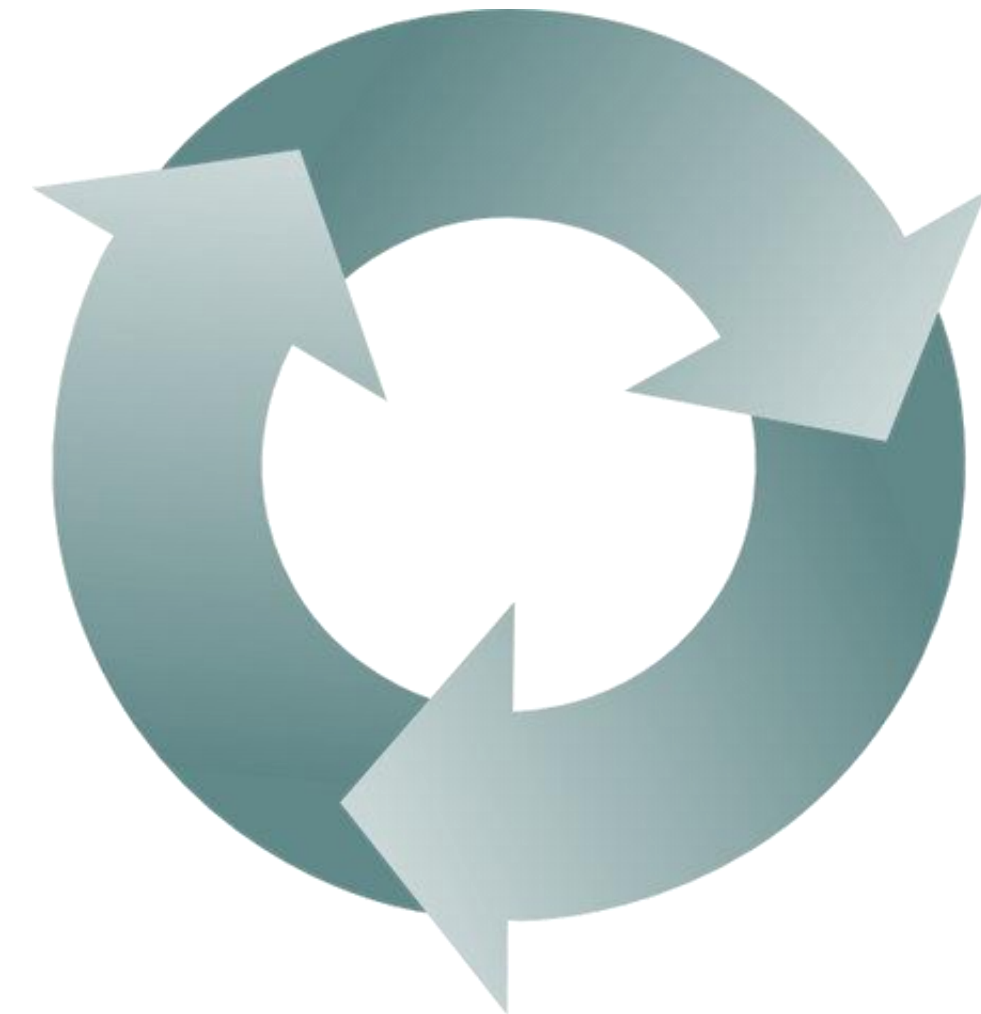
Version	WCT [ms]	Observation	Next step
Baseline	39	Bandwidth-bound: 63x read/write ratio to global memory.	Identify excessive reads of <code>param</code> as candidate for constant memory
Constant memory	56	Read/write balanced, but slowdown; FP64 not highest utilized. Extra LDCs.	Identify indexed constant loads, eliminate by assisting compiler to statically determine offsets
Unrolling loops	37	Poor L1 cache hit rates. Calculation shows data from 1 block might fit.	Experimenting with launch bounds, determining if lower occupancy helps caching
Launch bounds, 1 block	25	Still bandwidth-bound. Not all data cached, extra writes.	Using shared memory as explicit cache
Shared memory	24	Achieved minimum data transfers. Latency bound. Reaching ~73% of expected peak.	Investigate if remaining warp stalls due to memory accesses could be removed

Final speedup: ~ 1.6X

SUMMARY

Performance Optimization is a Constant Learning Process

1. Know your application
2. Know your hardware
3. Know your tools
4. Know your process
 1. Identify the Hotspot
 2. Classify the Performance Limiter
 3. Look for indicators
5. Make it so!



ADDITIONAL REFERENCES

Documentation

<https://docs.nvidia.com/cuda/cuda-c-programming-guide/>

<https://docs.nvidia.com/cuda/cuda-c-best-practices-guide/>

GTC 2020 Sessions:

S21771: Optimizing CUDA Kernels in HPC Simulation and Visualization Codes Using NVIDIA Nsight Compute

S22043: CUDA Developer Tools: Overview and Exciting New Features

S21351: Scaling the Transformer Model Implementation in PyTorch Across Multiple Nodes

THANK YOU!



nvidia