GPU COMPUTING
3 - TOOLS

Mathias Wagner, Lattice Practices 2024

OVERVIEW

CUDA Tools

CUDA-GDB

Extension of GDB, allows debugging of CUDA application

COMPUTE-SANITIZER

Valgrind like tool to check functional correctness

NSIGHT PROFILERS

Timing of Kernel and API calls (Timeline in GUI)

Detailed kernel metrics and events
Guided analysis for performance optimization

DEBUGGING

Debugging Correctness: Best Practices

Before you start

- Crashes are "nice" – the stacktrace often points to the bug
- Prerequisite: Compile flags
  - While developing, always use `-g -lineinfo`
  - Use `-g -G` for manual debugging
  - Specific flags for compilers/languages (e.g. gfortran): `-fcheck=bounds`
- Memory corruption: Out-of-bounds accesses may or may not crash
  - *compute-sanitizer*: Automate finding these errrors
- Other issues: Manual debugging
  - *cuda-gdb*: Command-line debugger, GPU extensions
  - `CUDA_LAUNCH_BLOCKING=1` forces synchronous kernel launches

compute-sanitizer

Functional correctness checking suite for GPU
https://docs.nvidia.com/compute-sanitizer/ComputeSanitizer/

- `compute-sanitizer` is a collection of tools
- `memcheck` (default) tool comparable to Valgrind's memcheck.
- Other tools include
  - `racecheck`: shared memory data access hazard detector
  - `initcheck`: uninitialized device global memory access detector
  - `synccheck`: identify whether a CUDA application is correctly using synchronization primitives
- Main usage: Auto-detect invalid GPU code and shortcut debugging effort
  - Directly pinpoint source code line/addresses, access size
- Leak-checking for device allocations - forgot to call `cudaFree()`?
  - `--leak-check full`
- Filtering and other capabilities. Two commonly useful switches:
  - `--log-file output.log`
    - Separates (potentially verbose) output into separate file
  - `--kernel-regex kns=some_substring`
    - Only checks kernels containing "some_substring"

cuda-gdb

Extends GDB for CUDA applications
https://docs.nvidia.com/cuda/cuda-gdb/index.html

- "Symbolic Debugger" – leaverage debug symbols to correlate execution issues with original source code
- Interactive/manual tool, with useful shortcuts
  - https://docs.nvidia.com/cuda/cuda-gdb/index.html#automatic-error-checking
- Textual, like a shell for debugging – Not the easiest to master, but very powerful, and works everywhere
- Basic workflow for segfaults
  - Crashing app invoked via
    - `./my_app_name my_app_arg another_arg`
  - becomes
    - **`cuda-gdb --args`** `./my_app_name my_app_arg another_arg`
  - Shows you the debugger shell prompt: (cuda-gdb)
    - Launch program with "`run`"
  - Identify the segfault – Done ☺
- Advanced workflow to step through execution, understand program flow, inspect and modify variables,…

cuda-gdb Cheat Sheet

(doubles as a GDB cheat sheet)

- Most commands have abbreviations
  - continue → cont, break → b, info → i, backtrace → bt, ...
  - cuda thread 4 → cu th 4
- Use TAB completion to help you remember command names
- Use help and apropos to avoid a round-trip to the browser (try: `apropos cuda.*api`)

`run`	Begin progam execution under debugger
`backtrace`	Print call stack (e.g. after an exception)
`list`	List source code around current location
`print <var>`	Print contents of <var>, e.g. *"print i"* to print the loop counter *i*
`set var <var>=<value>`	Set value of `<var>` to `<value>`, e.g. *"set var i=42"*

`break 10`
`break foo.cpp:10`
`break my_func`
	Set breakpoint (suspend execution) on: line `10` in current file
	... line `10` in file foo.cpp
	... function my_func in any file

`set cuda api_failures stop`	Break on any CUDA API failures (e.g. launch errors)
`continue / next / step`	Resume execution (after hitting breakpoint) until next: break / line / instruction
`info locals`	Print all local variables in current scope
`info cuda threads`	Print current thread configuration
`cuda thread 15`	Switch focus to thread (here: 15)

The Most Essential Command

In case of segfault, remember the backtrace

- If your app crashes or terminates unexpectedly, the debugger can very often tell you the exact location of the issue
  - Both in CPU and GPU code

```
$ cuda-gdb --args ./gpu-print

(cuda-gdb) run

[...]

CUDA Exception: Warp Illegal Address
The exception was triggered at PC 0xacbc90 (gpu_print.cu:19)

Thread 1 "gpu_print" received signal CUDA_EXCEPTION_14, Warp Illegal Address.
[Switching focus to CUDA kernel 0, grid 1, block (0,0,0), thread (0,0,0), device 0,sm 0,warp
0,lane 0]
0x0000000000acbca0 in print_test<<<(2,1,1),(32,1,1)>>> () at gpu_print.cu:19
19              double x = *(double*)nullptr;
(cuda-gdb) bt # "backtrace"
#0  0x0000000000acbca0 in print_test<<<(2,1,1),(32,1,1)>>> () at gpu_print.cu:19
```

- Backtrace tries to print all stack frames (i.e. function calls) with line information up to the current location
  - Equally useful when manually debugging or using breakpoints
  - Some errors can corrupt the stack, making the backtrace less useful

GPU-Specifics

New commands in cuda-gdb

- GPU-specifics: Setting the *focus*

```
(cuda-gdb) i cuda threads
  BlockIdx ThreadIdx To BlockIdx ThreadIdx Count         Virtual PC     Filename  Line
Kernel 0
*  (0,0,0)   (0,0,0)     (0,0,0)  (31,0,0)    32 0x0000000000acbf90 gpu_print.cu    19
   (1,0,0)   (0,0,0)     (1,0,0)  (31,0,0)    32 0x0000000000acbf60 gpu_print.cu    18
(cuda-gdb) cuda thread
thread (0,0,0)
(cuda-gdb) cuda thread 10
[Switching focus to CUDA kernel 0, grid 1, block (0,0,0), thread (10,0,0), device 0,sm 0,warp 0,lane
10]
19              printf("blockIdx.x = %d, threadIdx.x = %d, i = %d\n", blockIdx.x, threadIdx.x, i);
```

- Focus can be set to specific blocks, SMs, devices, … – **help cuda**
  - Hardware and software abstractions (e.g. blocks vs. SMs)
- Options: Try `(cuda-gdb) set cuda`<ENTER> for a list
  - Two commonly-used options: **api_failures** and **launch_blocking**


NVIDIA NSIGHT SUITE

PERFORMANCE OPTIMIZATION

What exactly is the performance bottleneck?

You might have a feeling where your application spends most of it's time …

… but a more analytic approach might be better

… but keep in mind that you might kill some cats in the process

(Profiling creates overhead)

WHAT DOES A PROFILER DO?

Sampling vs. Instrumentation (very simplified)

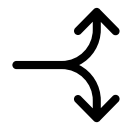
Every ms, take a sample of callstack

```
while (...) {
  do_nothing()
  intense_calculation()
  sleep()
}
```

Samples
0
23
12

(+) Hot spots show up, low overhead

(-) May miss some calls



Instrument function calls, APIs, etc. (automatable)

```
while (...) {
  trace_do_nothing() -> do_nothing()
  trace_intense_calculation() ->  intense_calculation()
  trace_sleep() ->  sleep()
}
```

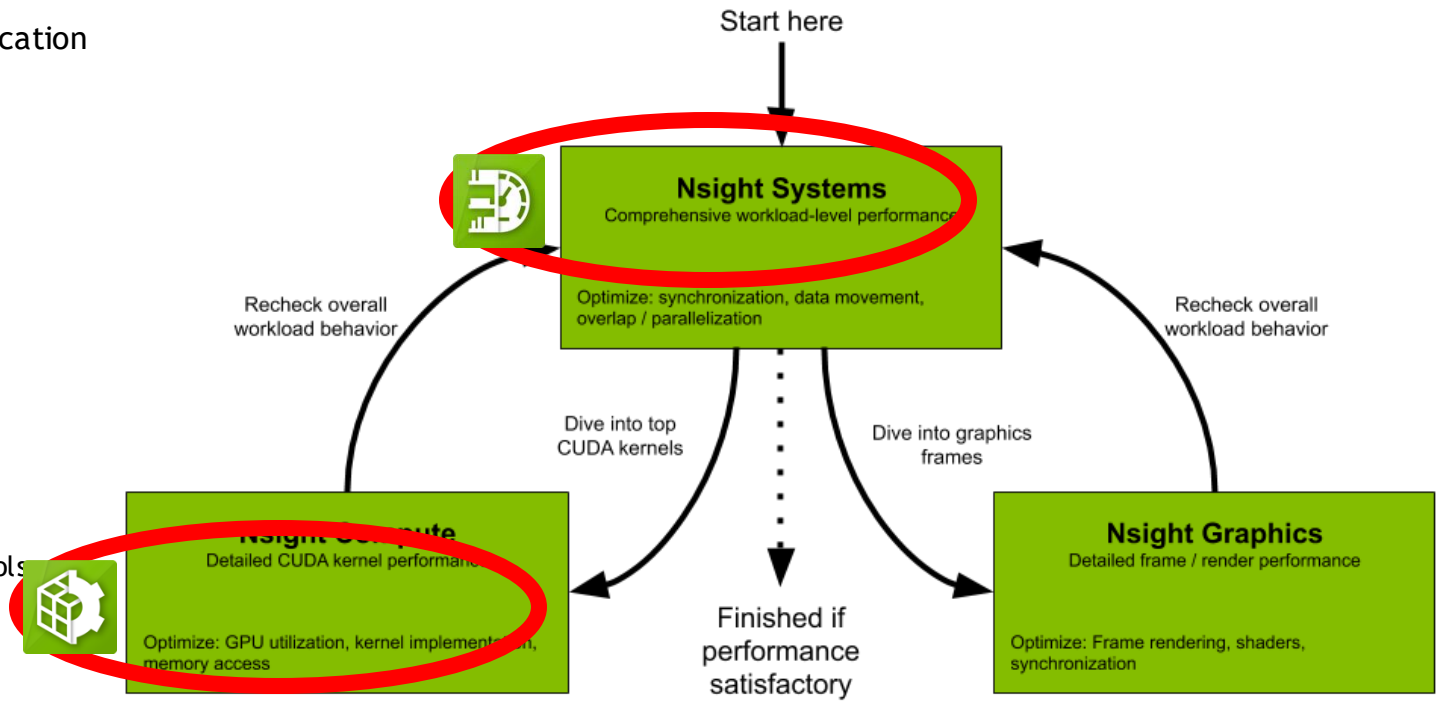
(+) Captures whole program, full call chains

(-) Potentially higher overhead, skew

THE NSIGHT SUITE COMPONENTS

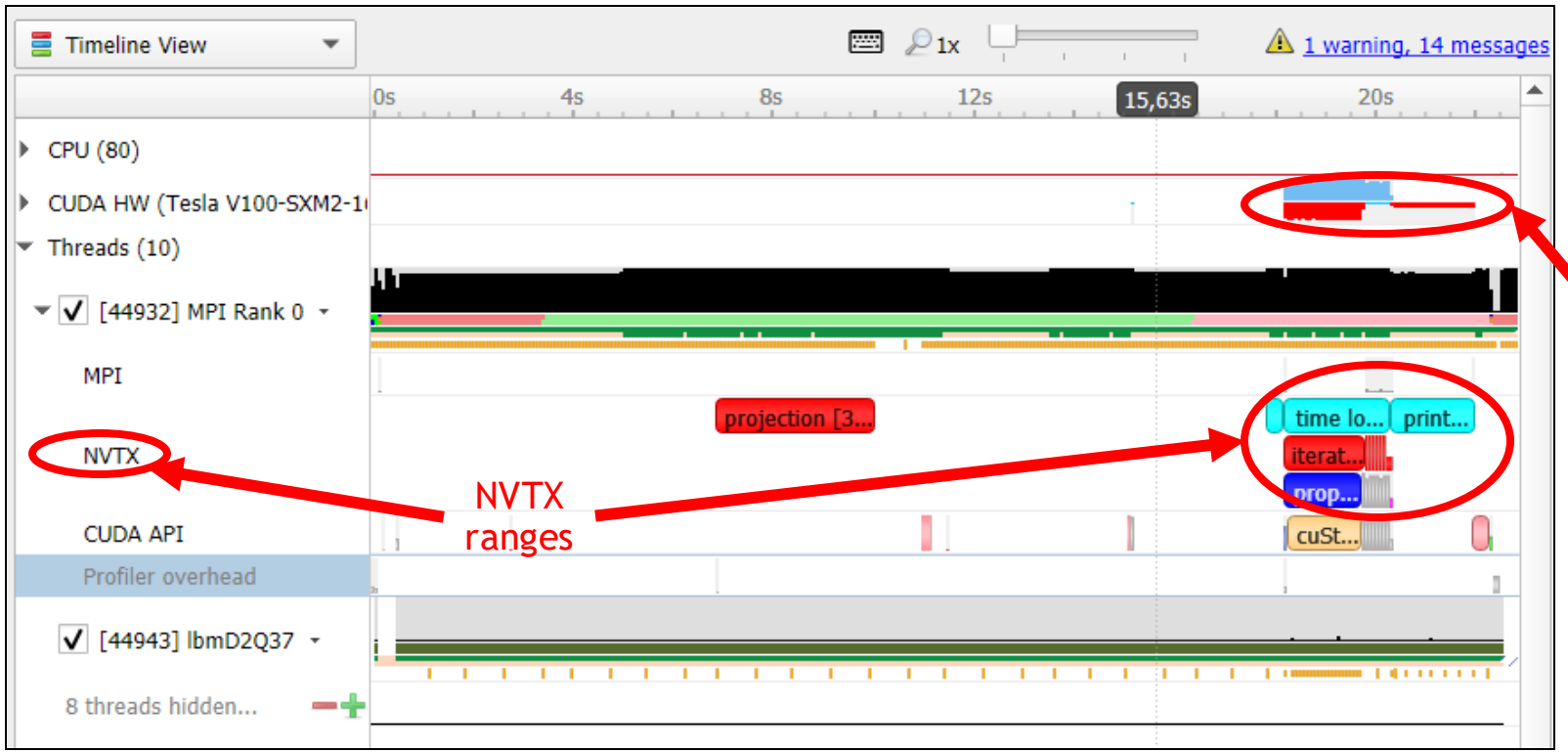
How the pieces fit together

- Nsight **Systems**: Coarse-grained, whole-application
- Nsight **Compute**: Fine-grained, kernel-level
- NVTX: Support and structure across tools
- Main purpose: Performance optimization
  - But at their core, advanced measurement tools



WHETTING YOUR APPETITE

Timeline overview in Nsight Systems GUI



Here: 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

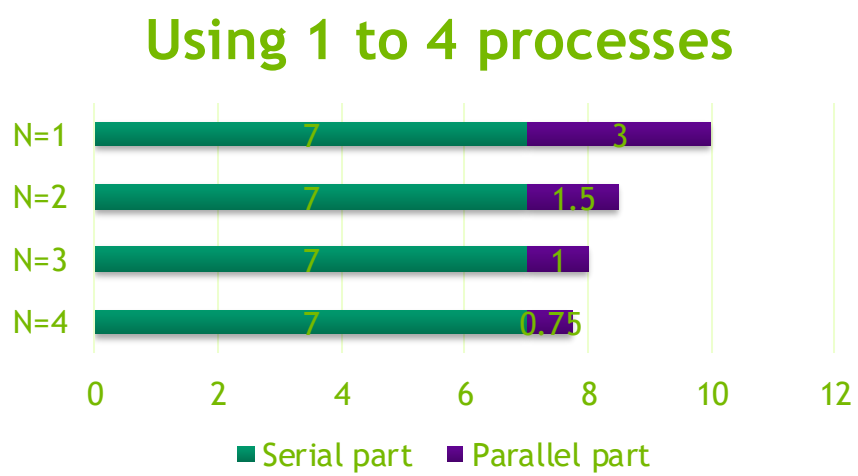
Amdahl‘s law states overall speedup $\boldsymbol{s}$ given the parallel fraction p of code and number of processes N

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

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

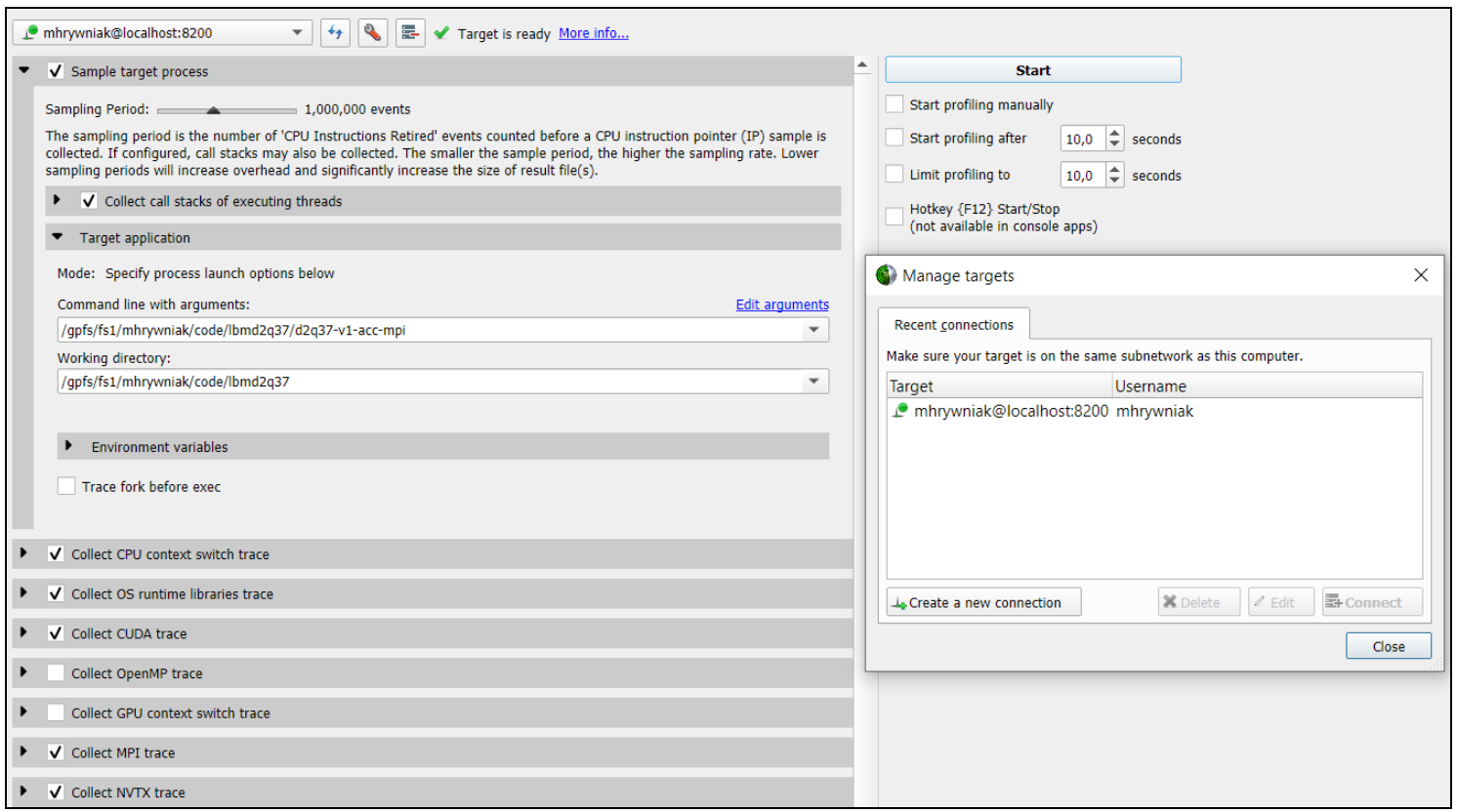
Example for $p = 30\%$

Also valid for per-method speedups



A FIRST (I)NSIGHT

Recording with the GUI



Connect directly

Or use an SSH Tunnel:
`ssh -L 8200:`*`compute-node`*`:22 `*`login-node`*

Select traces to collect

A FIRST (I)NSIGHT

Recording an application timeline

1) We'll use the command line

```
mpirun -np $NP \
nsys profile --trace=cuda,nvtx,mpi \
--output=my_report.%q{OMPI_COMM_WORLD_RANK}.qdrep ./myApp
```

Note: Slurm users, try `srun ... %q{SLURM_PROCID}`

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

Also possible to get details on command line (documentation), `nsys stats --help`

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

USING NSIGHT SYSTEMS

Recording with the CLI

- Use the command line

```
srun nsys profile --trace=cuda,nvtx,mpi --output=my_report.%q{SLURM_PROCID} ./jacobi -niter 10
```

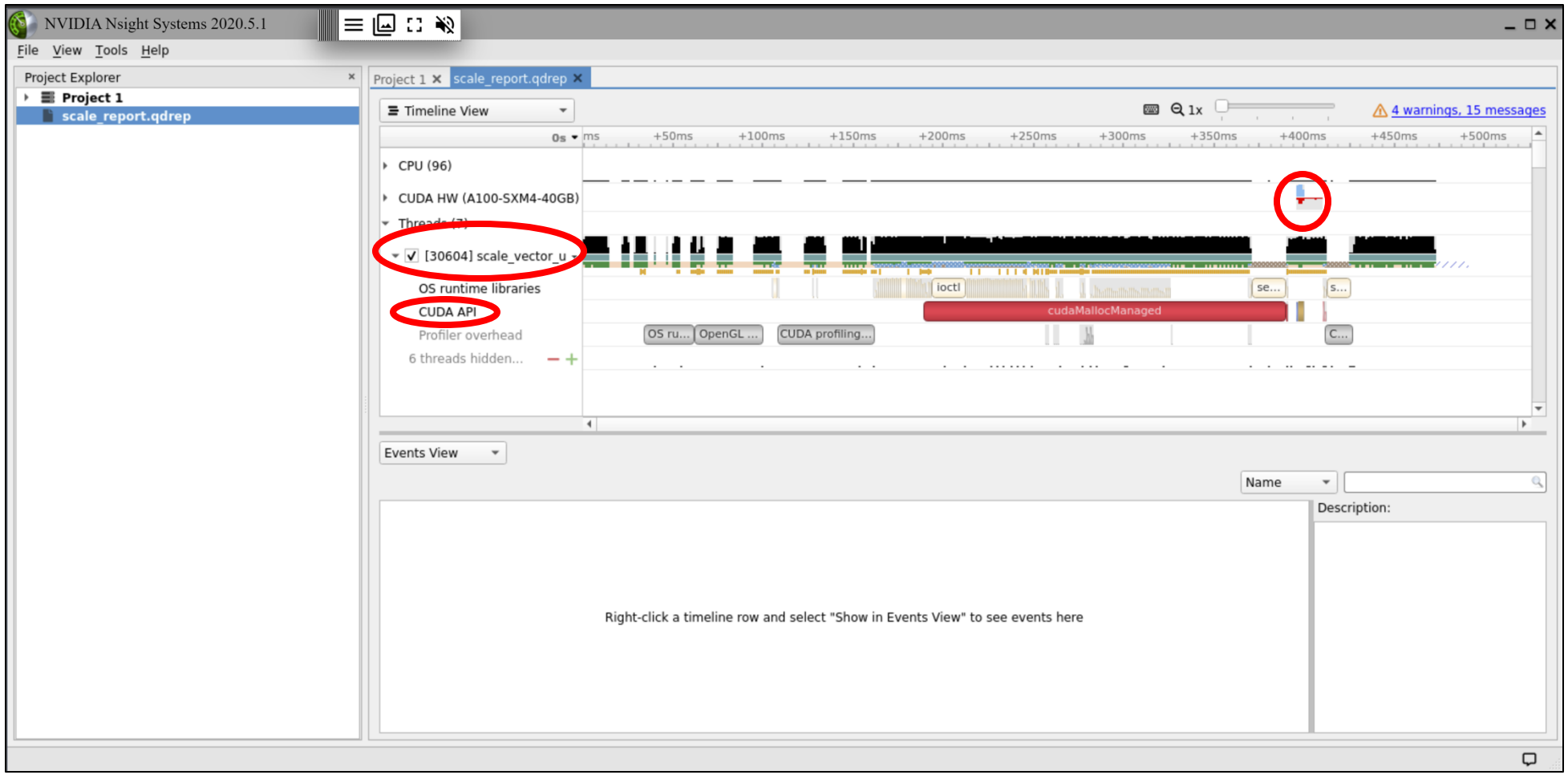
- Inspect results: Open the report file in the GUI
  - Also possible to get details on command line
  - Either add `--stats` to profile command line, or: `nsys stats --help`

- Runs set of reports on command line, customizable (**sqlite** + **Python**):
  - Useful to check validity of profile, identify important kernels

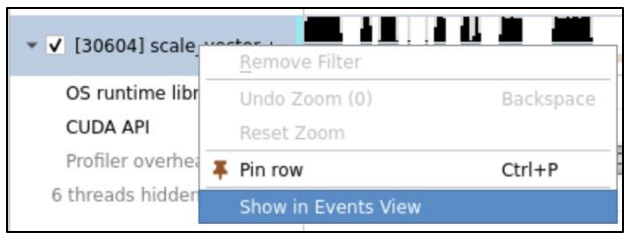
```
Running [.../reports/gpukernsum.py jacobi_metrics_more-nvtx.0.sqlite]...

Time(%)  Total Time (ns)  Instances  Avg (ns)   Med (ns)   Min (ns)  Max (ns)  StdDev (ns)           Name
-------  ---------------  ---------  ---------  ---------  --------  --------  -----------  ---------------------
   99.9         36750359         20  1837518.0  1838466.5    622945   3055044    1245121.7  void jacobi_kernel
    0.1            22816          2    11408.0    11408.0      7520     15296       5498.1  initialize_boundaries
```

LOOKING AT A SIMPLE EXAMPLE



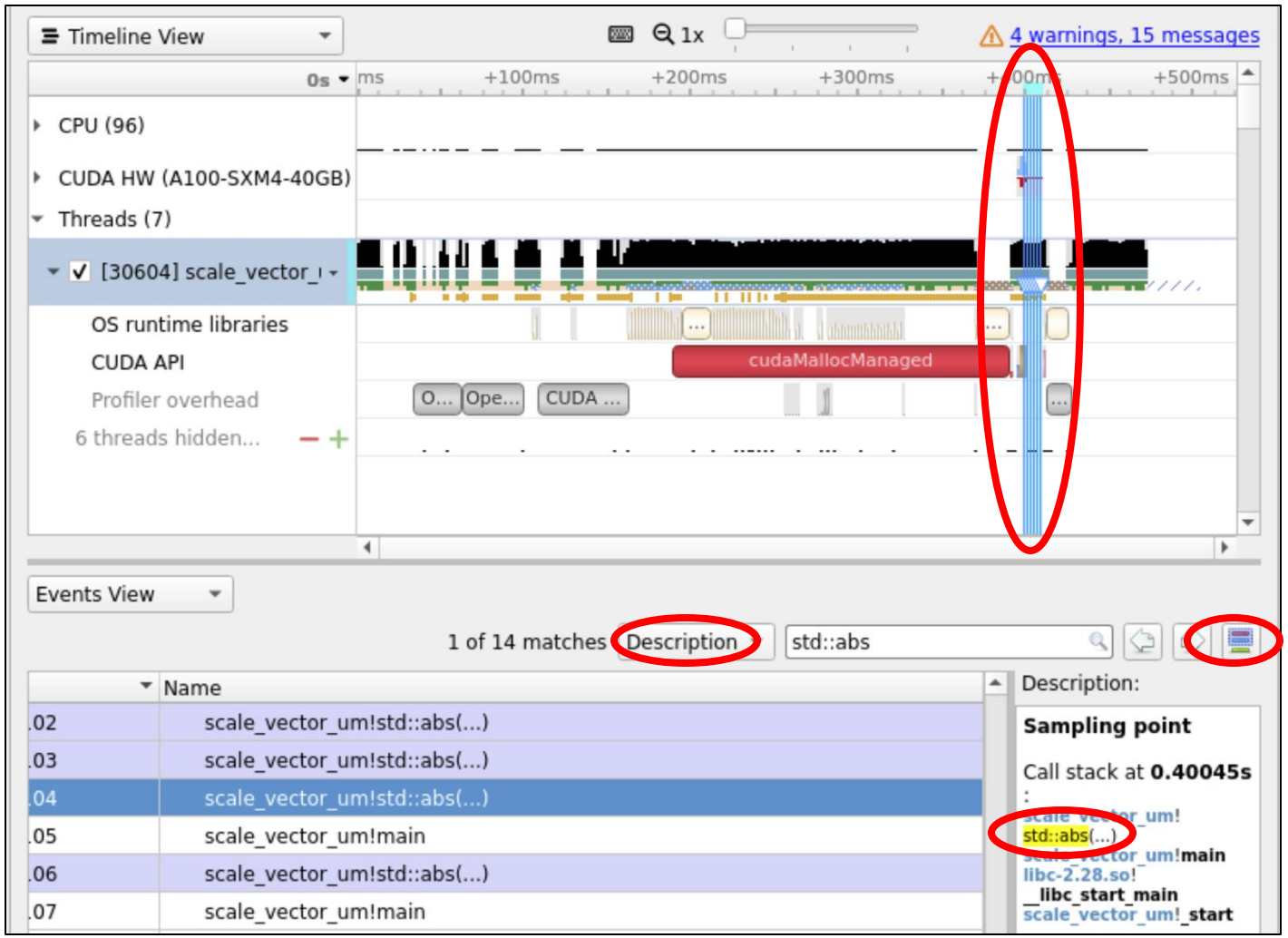
USING CALLSTACK SAMPLES



Events View makes information searchable

„Highlight All“ shows all matches

Can search in description, includes callstack



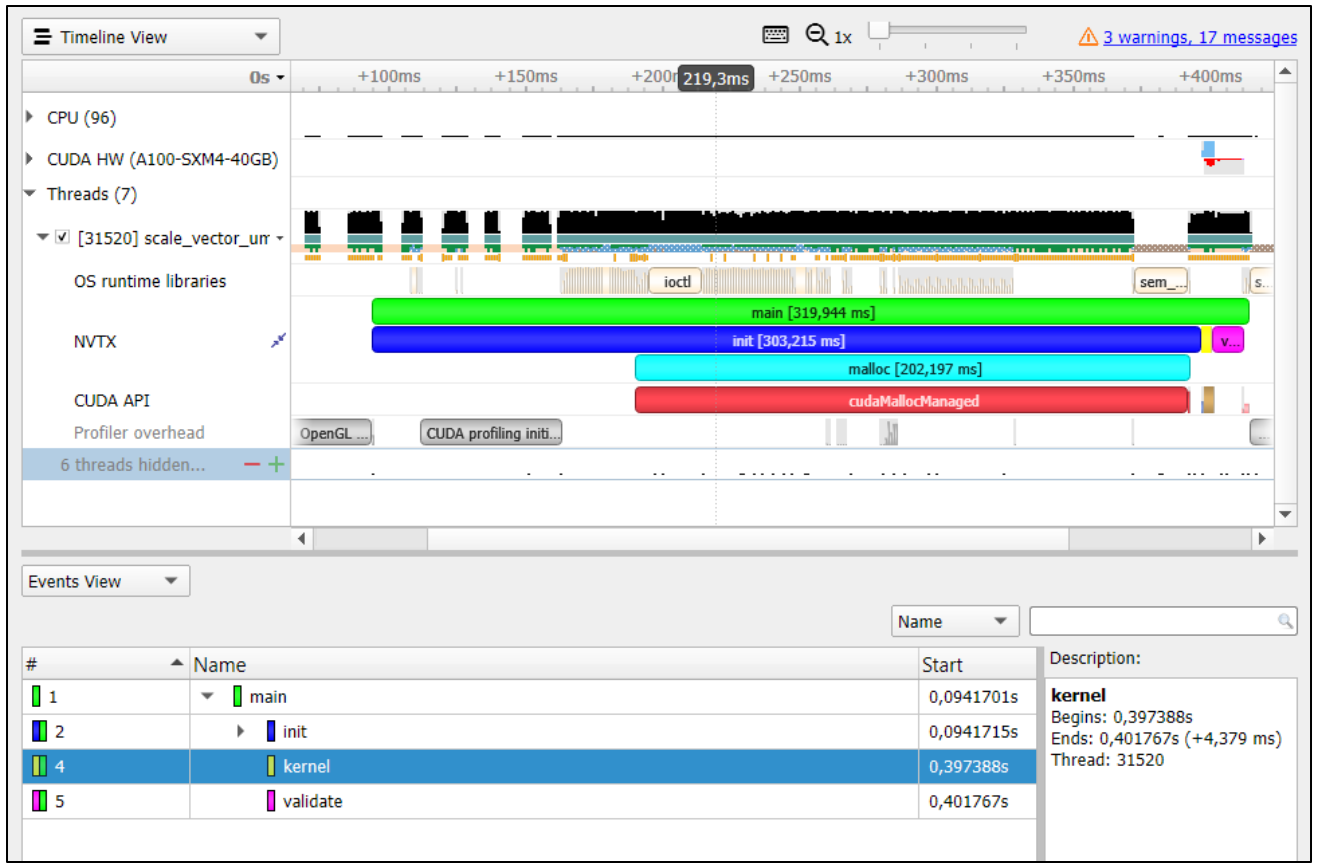
ADDING SOME COLOR

Code annotation with NVTX

Like manual timing, only less work

Nesting, timing

Correlation, filtering



ADDING NVTX

Simple range-based API

```
#include <nvToolsExt3.h>
```

Copy&paste PUSH/POP macros (or module)

```
PUSH(name, color)
```

Sprinkle them strategically through code

NVTX v3 is header-only

Not shown: Advanced usage (domains, ...)

```
int main(int argc, char** argv){
    PUSH("main", 0)
    PUSH("init", 1)
    POP
    PUSH("kernel", 2)

    scale<<<gridDim, blockDim>>>(alpha, a, c, m);

    cudaDeviceSynchronize();
    POP
    PUSH("validate", 3)
```

https://github.com/NVIDIA/NVTX

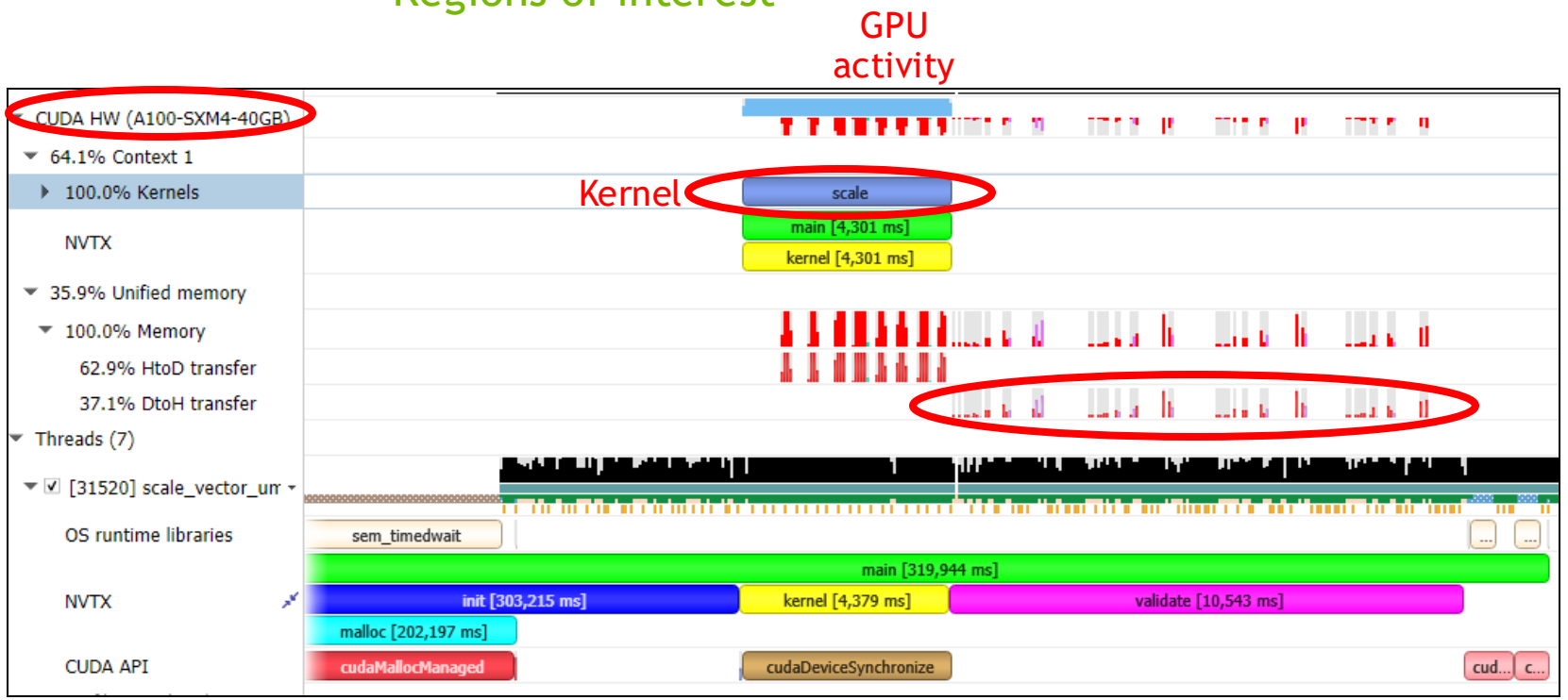
https://developer.nvidia.com/blog/cuda-pro-tip-generate-custom-application-profile-timelines-nvtx/

https://developer.nvidia.com/blog/customize-cuda-fortran-profiling-nvtx/

ZOOMING IN

Regions of interest

- Kernel launch
- UM migrations and page faults
- Use Amdahl‘s law as heuristic



MINIMIZING PROFILE SIZE

Shorter time, smaller files = quicker progress

Only profile what you need – all profilers have some overhead

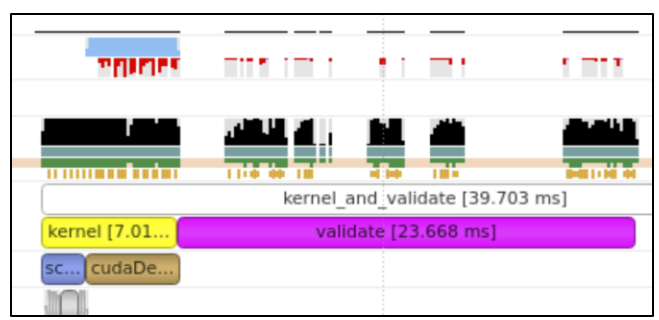
Bonus: lower number of events => smaller file size

Add to nsys command line:

```
--capture-range=nvtx --nvtx-capture=any_nvtx_marker_name \
--env-var=NSYS_NVTX_PROFILER_REGISTER_ONLY=0 --kill none
```

Alternatively: `cudaProfilerStart()` and `–Stop()`

```
--capture-range=cudaProfilerApi
```



OTHER FEATURES

We only covered a small subset

„Traditional“ top-down or bottom-up stack views

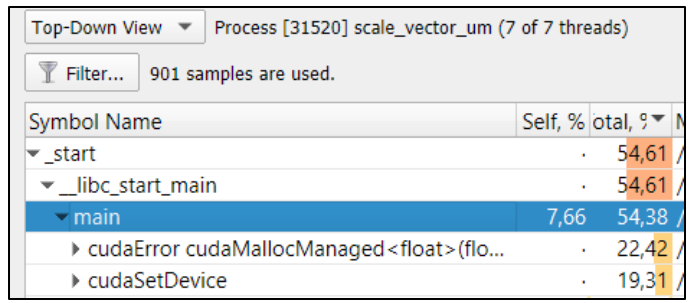
Lots of different traces (MPI, OpenACC, OpenMP, ...)

Data export (csv, sqlite, ...)

Customizable reports via Python scripts

Full guide:

https://docs.nvidia.com/nsight-systems/UserGuide




```
[hrywniak1@jwlogin24 task3]$ nsys stats scale_report.qdrep
Using scale_report.sqlite for SQL queries.
Running [/p/software/juwelsbooster/stages/2020/software/Nsight-Systems/2020.5.1-GCCcore-9.3.0/target-linux-x64/reports/cudaapisum.py scale_report.sqlite]...

 Time(%)  Total Time (ns)  Num Calls   Average   Minimum  Maximum          Name
 -------  ---------------  ---------  ---------  -------  -------  --------------------
    67.5          4704556          1  4704556.0  4704556  4704556  cudaDeviceSynchronize
    32.5          2265468          1  2265468.0  2265468  2265468  cudaLaunchKernel

Running [/p/software/juwelsbooster/stages/2020/software/Nsight-Systems/2020.5.1-GCCcore-9.3.0/target-linux-x64/reports/gpukernsum.py scale_report.sqlite]...

 Time(%)  Total Time (ns)  Instances   Average   Minimum  Maximum               Name
 -------  ---------------  ---------  ---------  -------  -------  --------------------------------
   100.0          4709010          1  4709010.0  4709010  4709010  scale(float, float*, float*, int)

Running [/p/software/juwelsbooster/stages/2020/software/Nsight-Systems/2020.5.1-GCCcore-9.3.0/target-linux-x64/reports/gpumemtimesum.py scale_report.sqlite]...

 Time(%)  Total Time (ns)  Operations  Average  Minimum  Maximum             Operation
 -------  ---------------  ----------  -------  -------  -------  --------------------------------
    65.7          1786518         464   3850.3     3039    32800  [CUDA Unified Memory memcpy HtoD]
    34.3           934091          96   9730.1     2111    53119  [CUDA Unified Memory memcpy DtoH]
```

WHEN TO MOVE ON

Proper tool for the job

Specialized MPI profiling/bottlenecks, load imbalance

Kernel-level profiling -> Nsight Compute

Used later on (get the low-hanging fruit first!)

Use it when you find a hotspot kernel

SUMMARY

How to approach porting your own code

Start with Nsight Systems and record a first profile

Identify roughly some features (use call stacks, code knowledge), add NVTX

- Add and customize traces as needed
- Use capture ranges

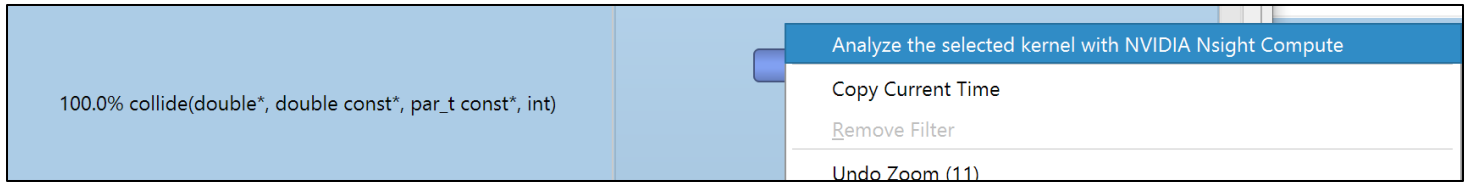
Iteratively eliminate „blank“ spots – is the GPU active?

Switch to more specialized profilers as needed

DRILLING DOWN ON A KERNEL

Analysis with Nsight Compute

Right-click menu in Nsight Systems, get command line



Run command line

```
ncu --page details --import-source true --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`

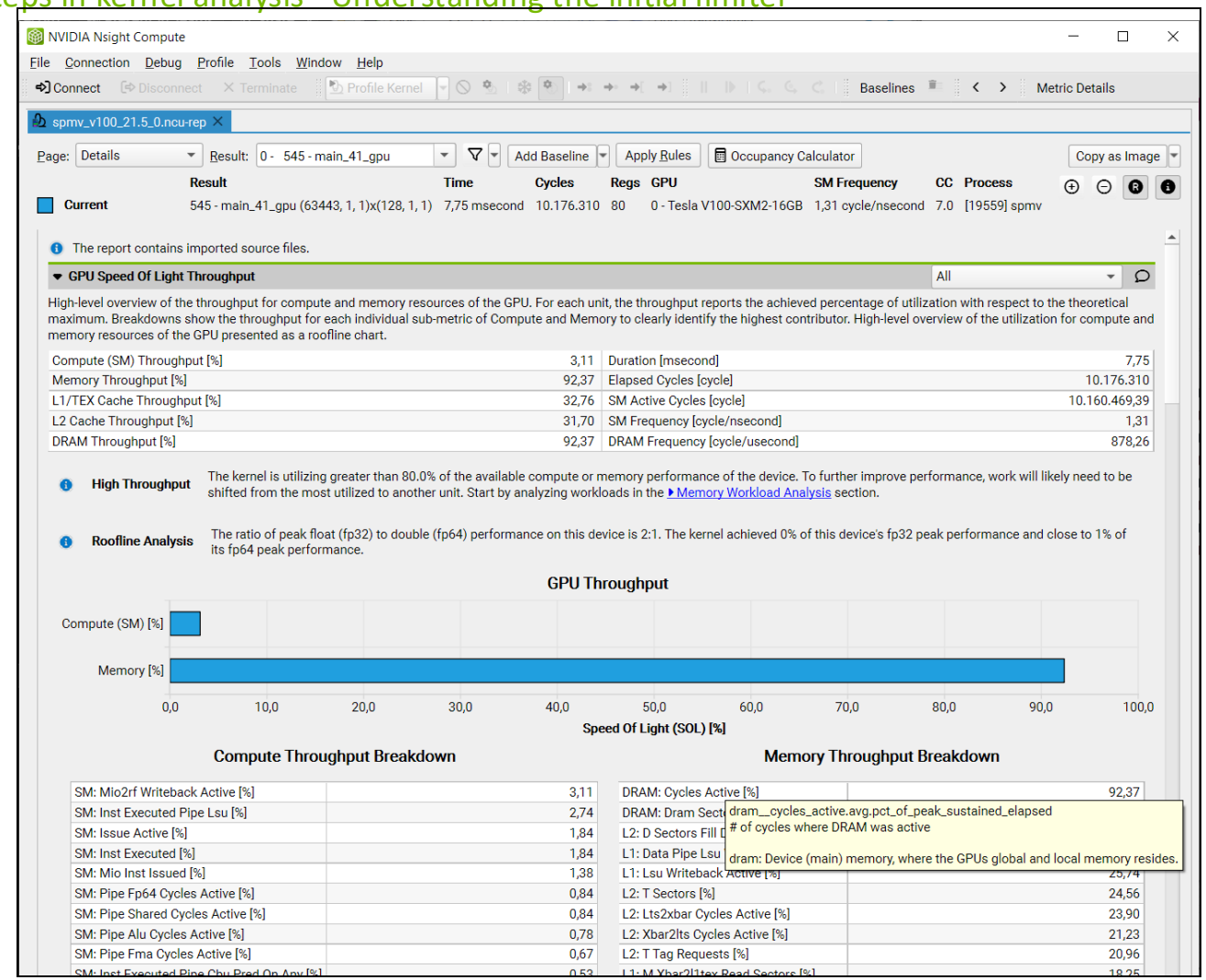
Here: profile with CLI, use GUI for analysis and load report file

Alternatively, interactive analysis of application through GUI. **API Stream** can be very useful.

Nsight Compute GUI

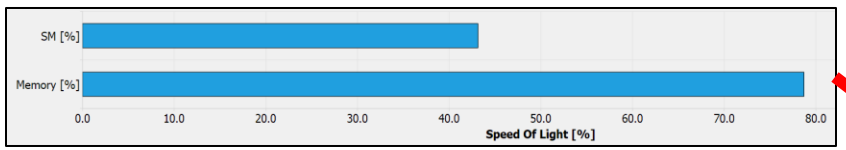
First steps in kernel analysis - Understanding the initial limiter

- GPU "Speed of Light Throughput"
  - SOL = theoretical peak
- "Breakdown" tables
  - DRAM: Cycles Active
- Tooltips
- Rules point to next steps



KERNEL-LEVEL PROFILING

Performance limiter categories



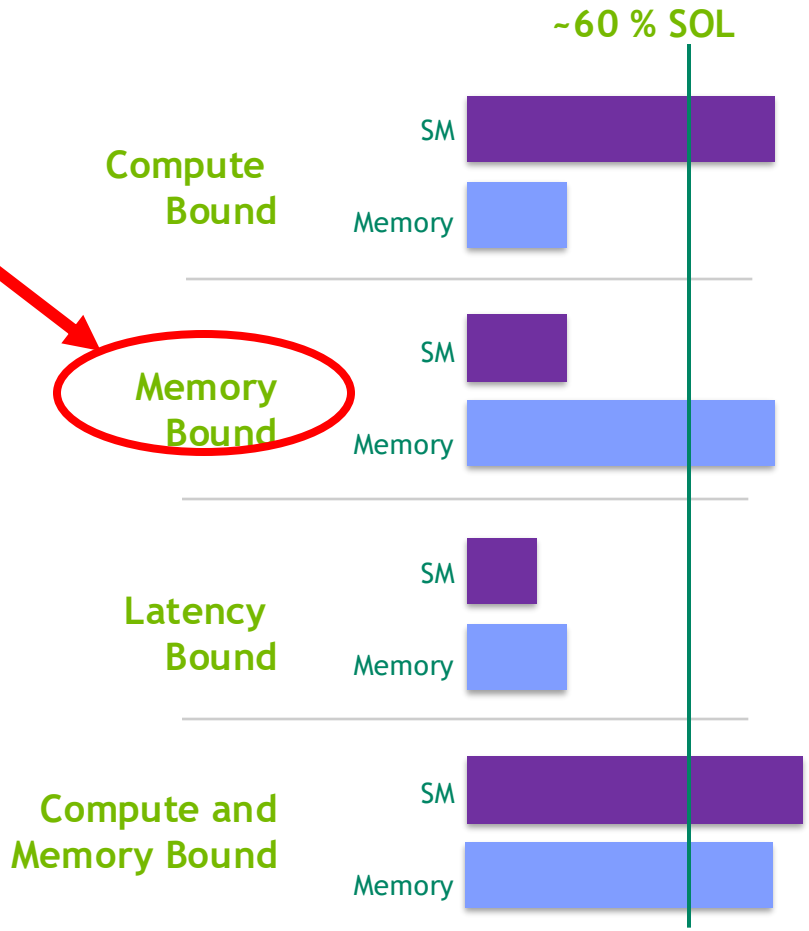
Four possible combinations of high/low...

...memory utilization

...compute utilization

Good? Bad?

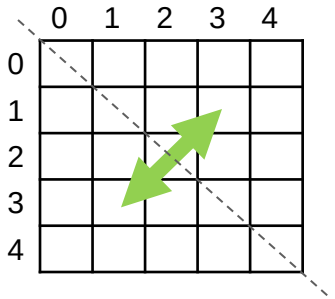
→ Depends on problem and its implementation



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:

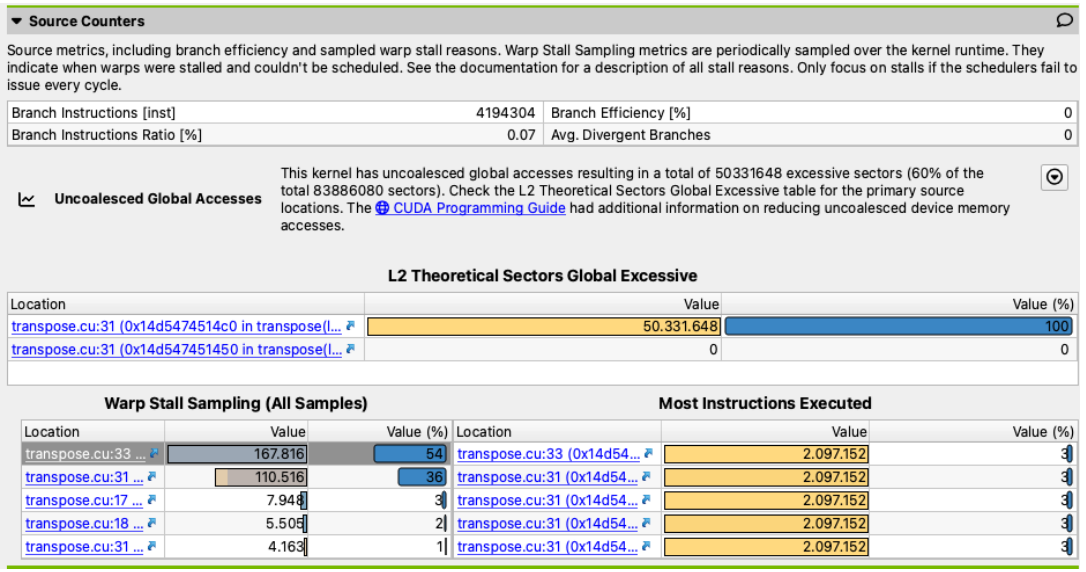
```
__global__ void transpose(
Integer* a_trans,
Integer* a,
Integer n)
{
  Integer row =
        blockIdx.y*blockDim.y+threadIdx.y;
  Integer col =
        blockIdx.x*blockDim.x+threadIdx.x;

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

☹

Row-major ordering

Using Nsight Compute



Source Counters

Source metrics, including branch efficiency and sampled warp stall reasons. Warp Stall Sampling 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]	4194304	Branch Efficiency [%]	0
Branch Instructions Ratio [%]	0.07	Avg. Divergent Branches	0

Uncoalesced Global Accesses

This kernel has uncoalesced global accesses resulting in a total of 50331648 excessive sectors (60% of the total 83886080 sectors). Check the L2 Theoretical Sectors Global Excessive table for the primary source locations. The CUDA Programming Guide had additional information on reducing uncoalesced device memory accesses.

L2 Theoretical Sectors Global Excessive

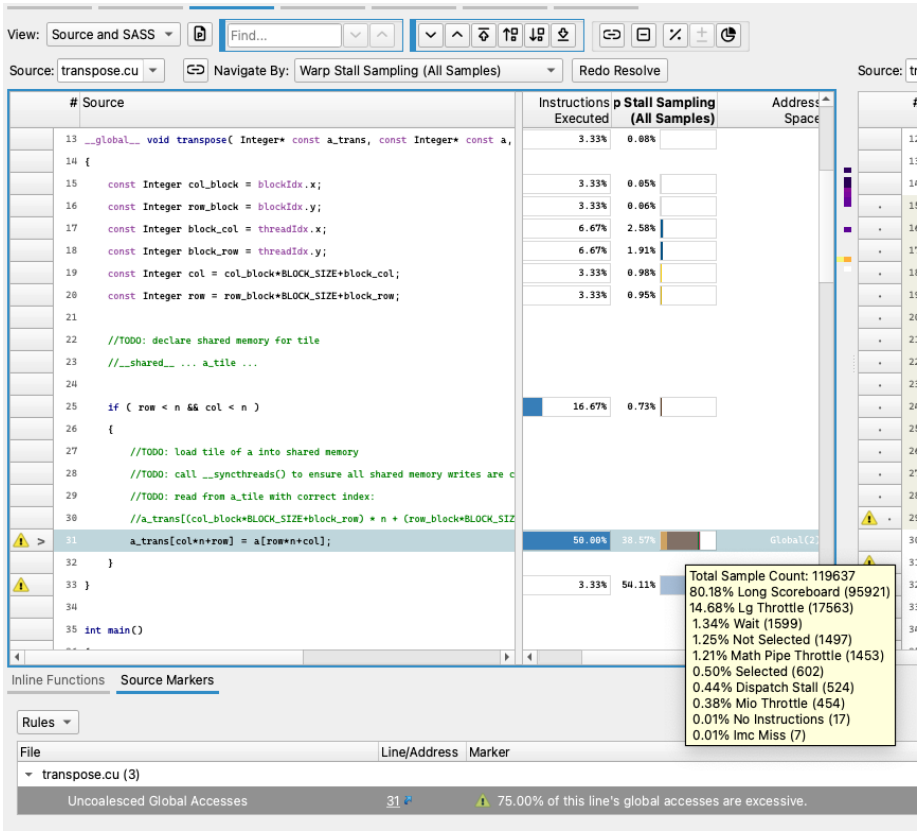
Location	Value	Value (%)
transpose.cu:31 (0x14d5474514c0 in transpose(l...	50.331.648	100
transpose.cu:31 (0x14d547451450 in transpose(l...	0	0

Warp Stall Sampling (All Samples)

Location	Value	Value (%)
transpose.cu:33 ...	167.816	54
transpose.cu:31 ...	110.516	36
transpose.cu:17 ...	7.948	3
transpose.cu:18 ...	5.505	2
transpose.cu:31 ...	4.163	1

Most Instructions Executed

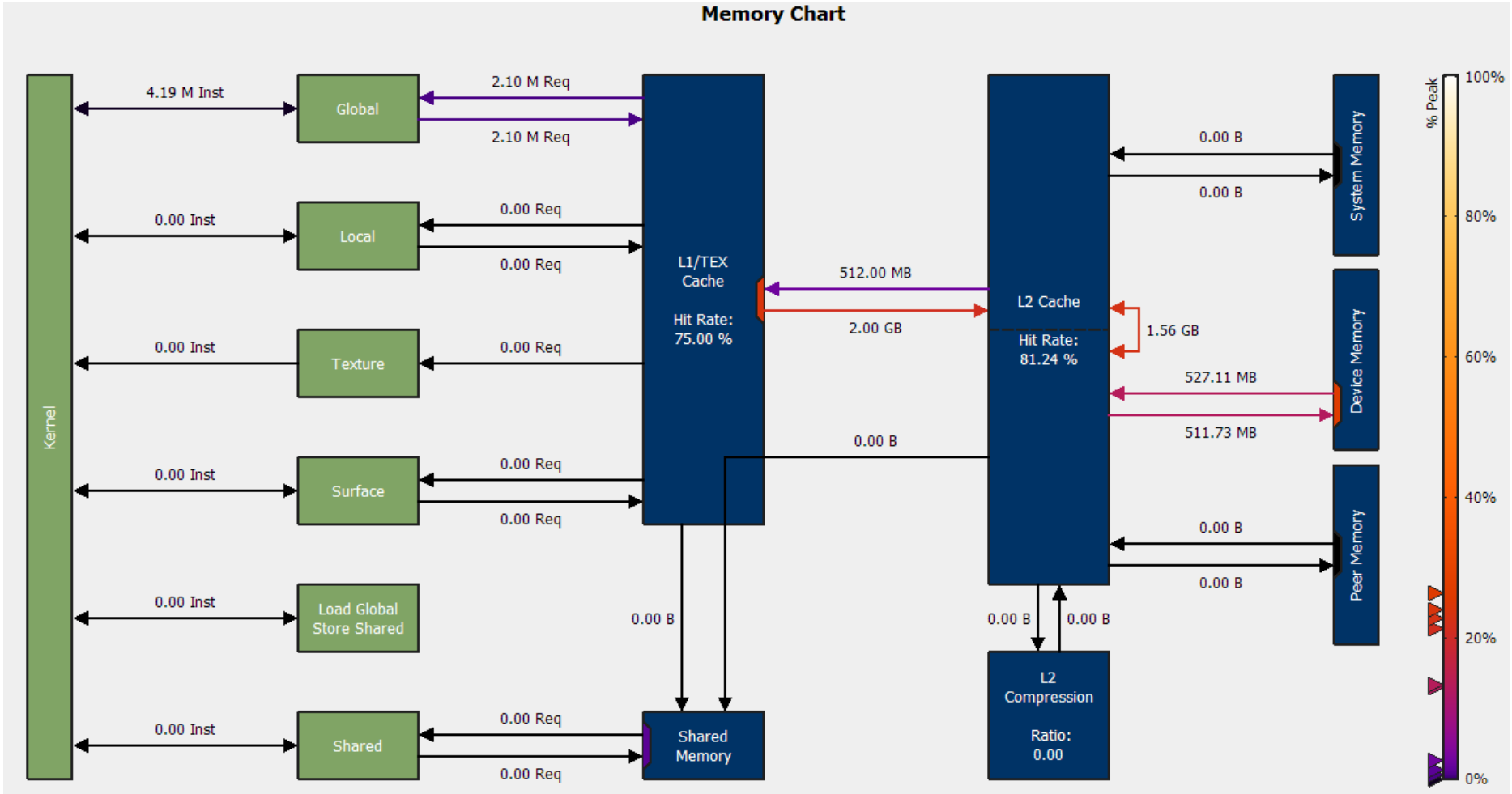
Location	Value	Value (%)
transpose.cu:33 (0x14d54...	2.097.152	3
transpose.cu:31 (0x14d54...	2.097.152	3
transpose.cu:31 (0x14d54...	2.097.152	3
transpose.cu:31 (0x14d54...	2.097.152	3
transpose.cu:31 (0x14d54...	2.097.152	3



„uncoalesced global excesses [...] 60% of the total“

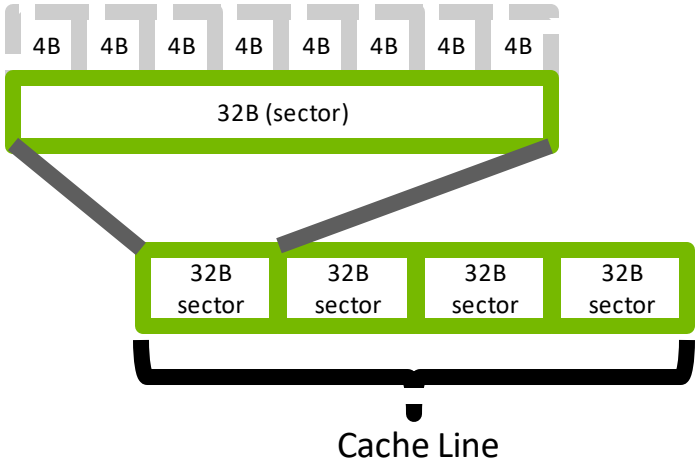
Global, Local, L1, L2?

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
- Data goes from **global** device memory through L2 cache
- Granularity*: Sector for L2, Cache line for L1



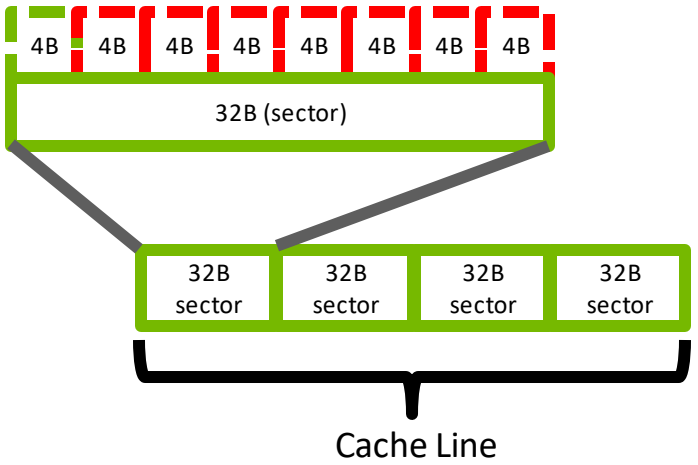
*The full picture:
S32089: Understanding and Optimizing Memory-Bound Kernels with Nsight Compute

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.



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

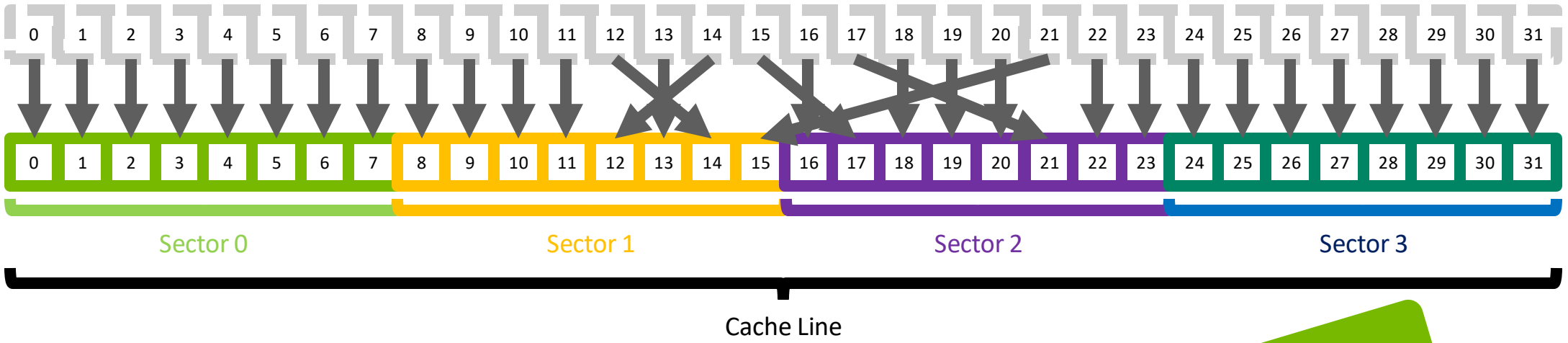
Accessing Global Memory

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

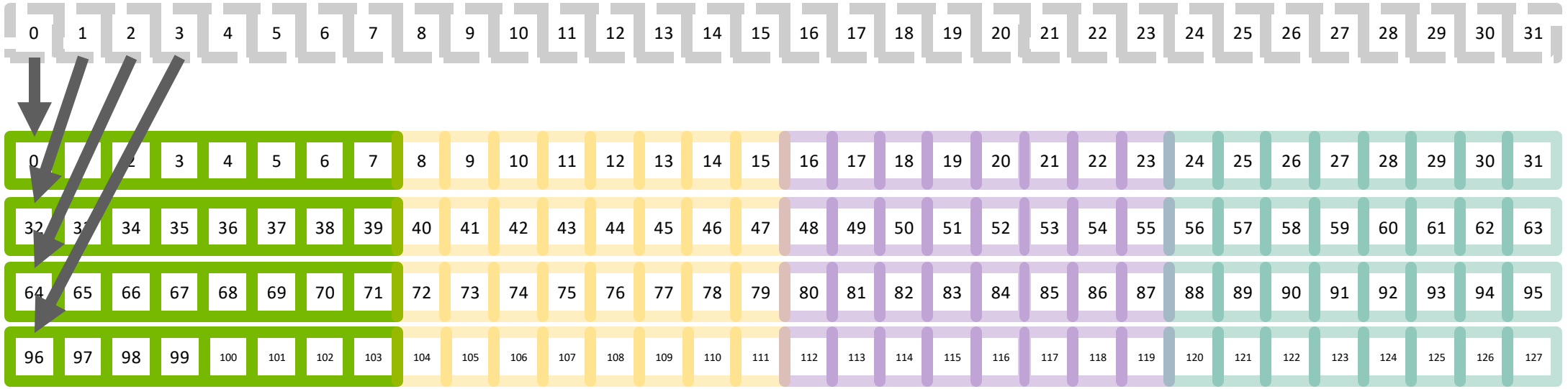
Accessing Global Memory

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%



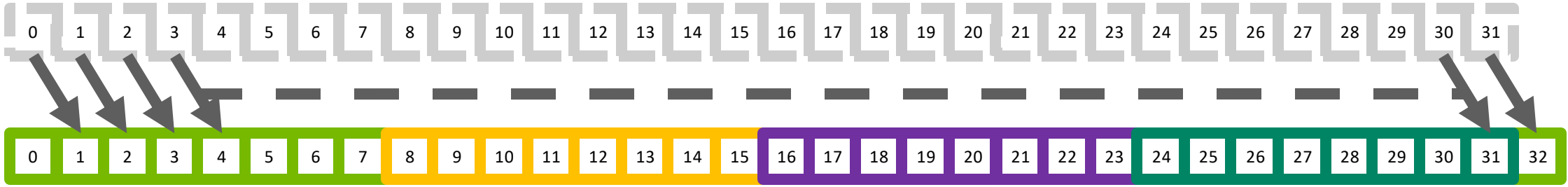
Accessing Global Memory

shifted access

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

All addresses fall within 5 sectors

Bus utilization: 80% = 128B/160B



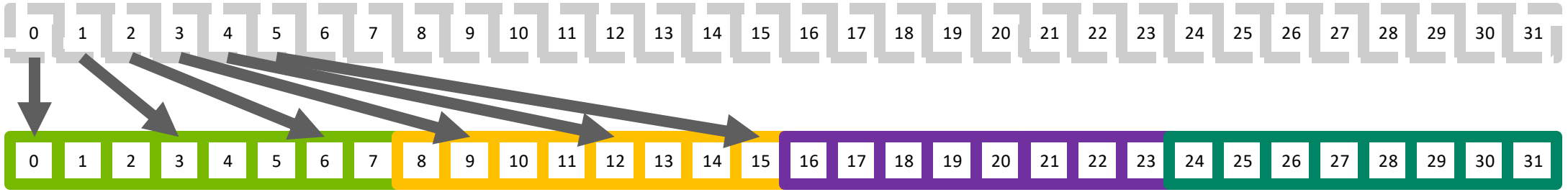
Accessing Global Memory

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?)

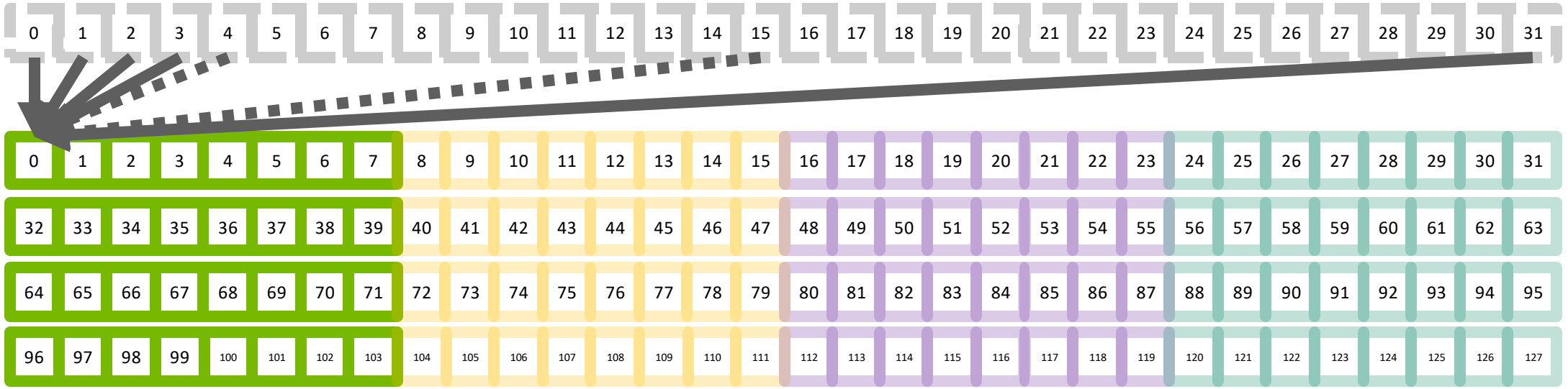
Accessing Global Memory

another “worst case” access pattern?

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

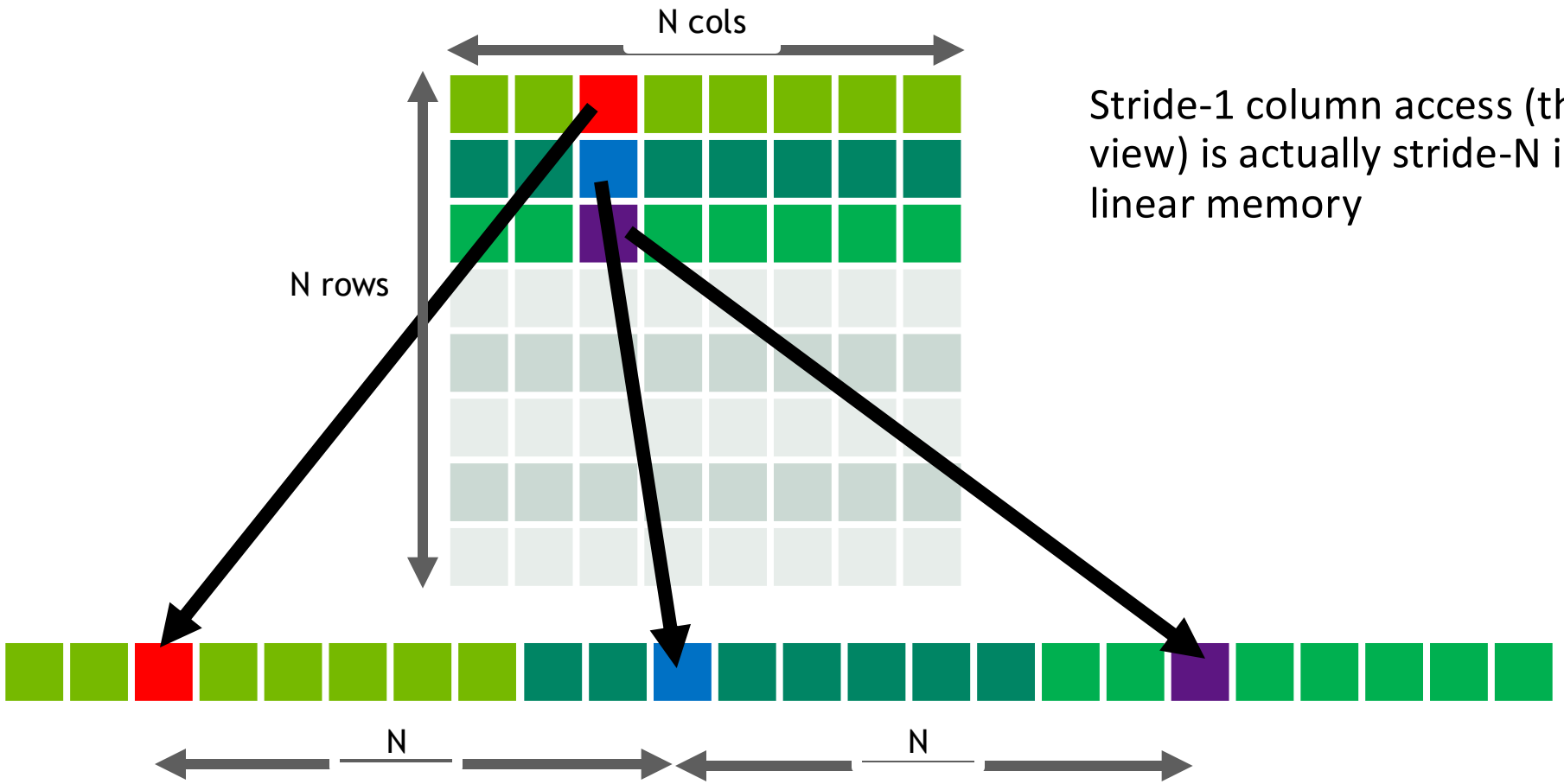
Single address, single sector

Bus utilization: 12.5%



Matrix Transpose

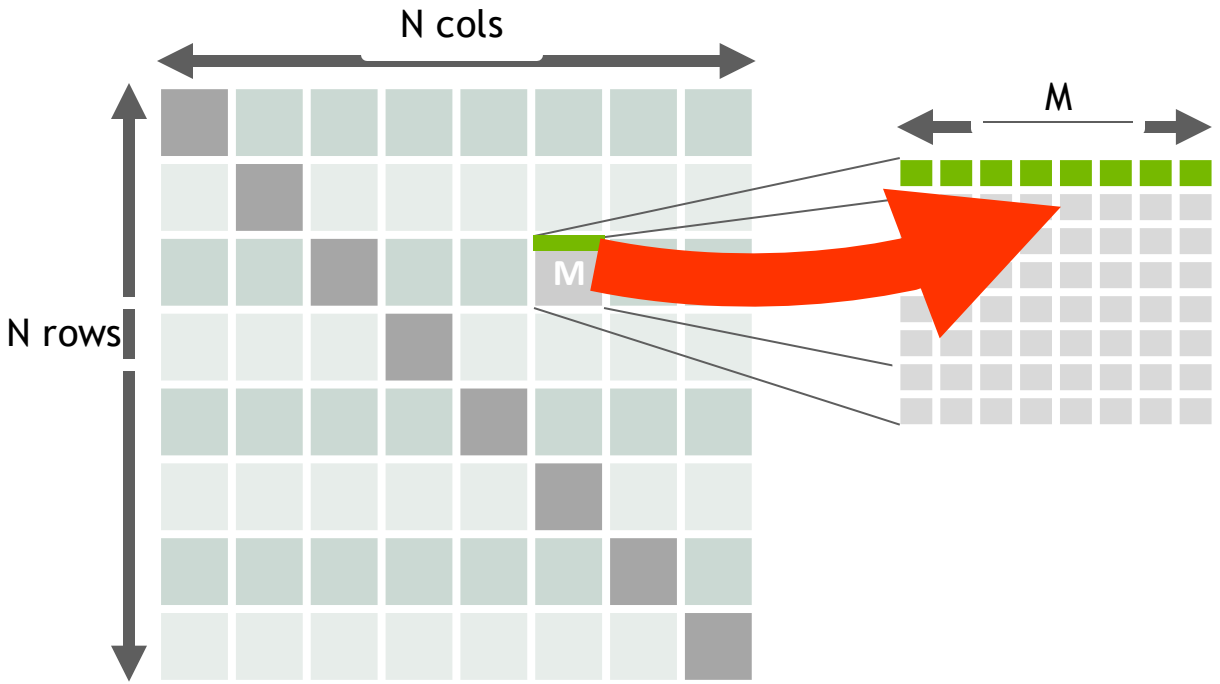
Access pattern



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

Matrix Transpose

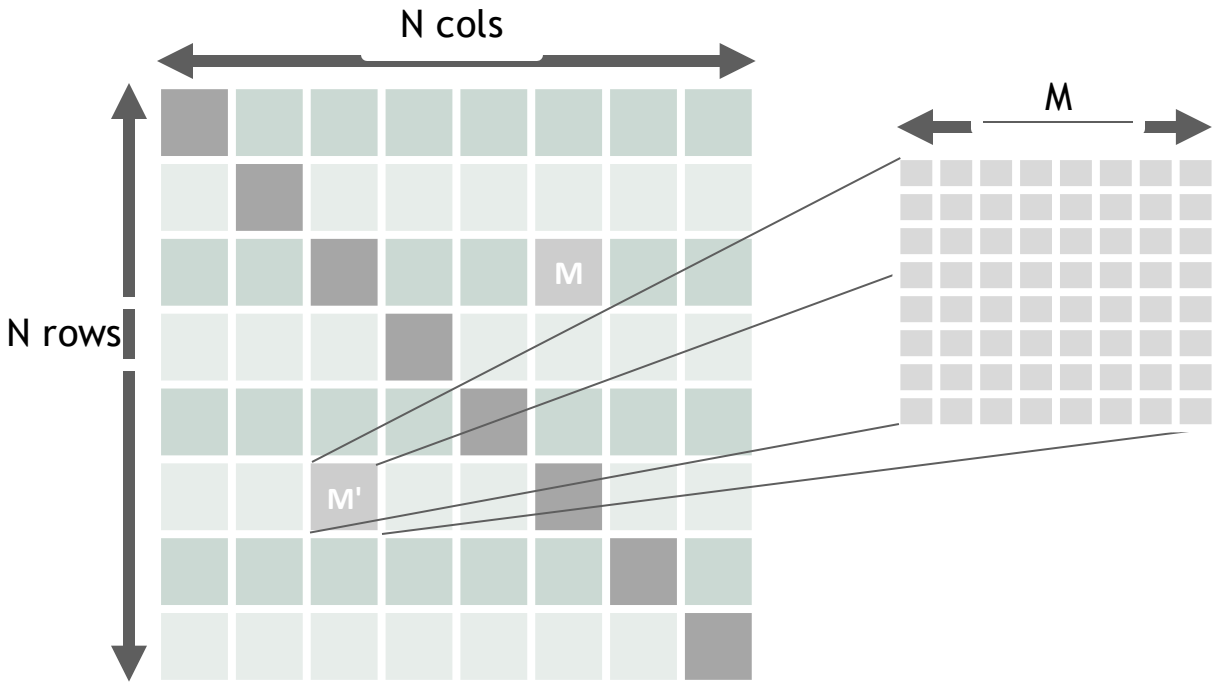
Using shared memory



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

Matrix Transpose

Using shared memory

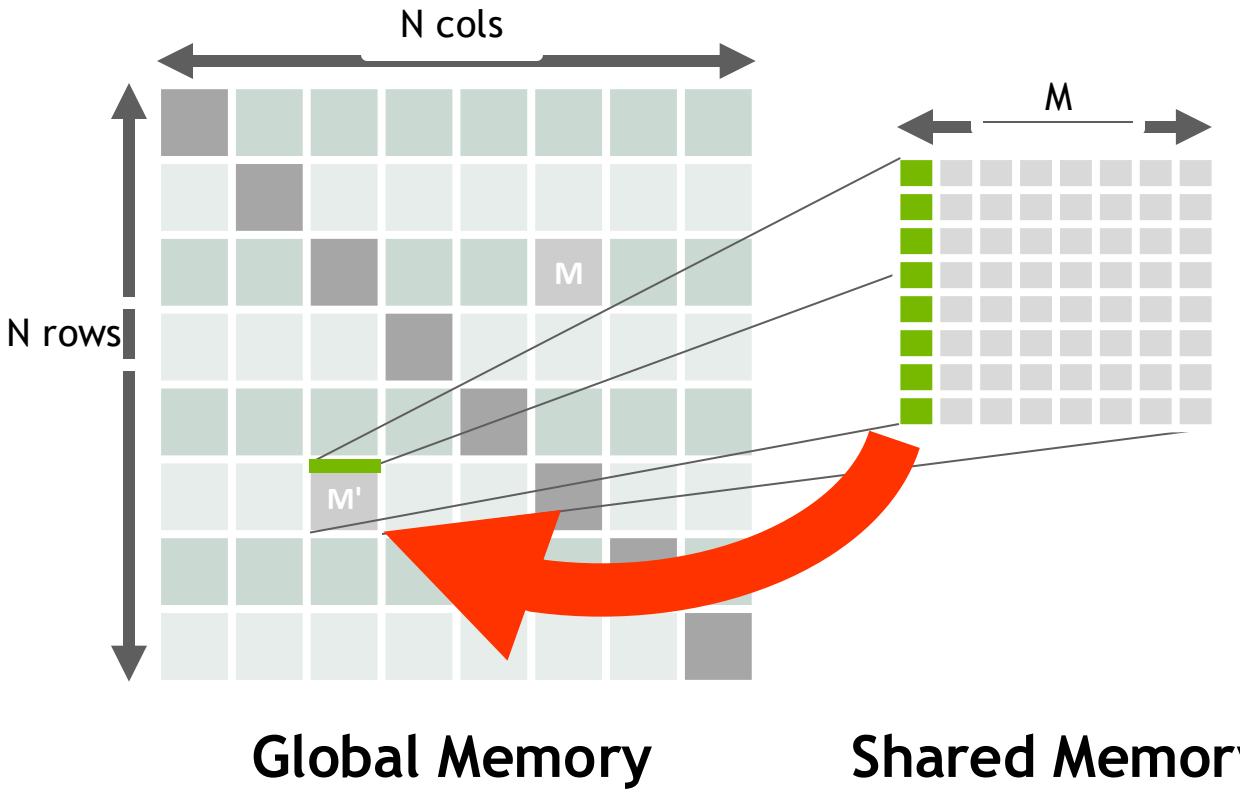


Block row is loaded, fully **coalesced** read

Indexing: Location of block is reflected on the diagonal

Matrix Transpose

Using shared memory



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

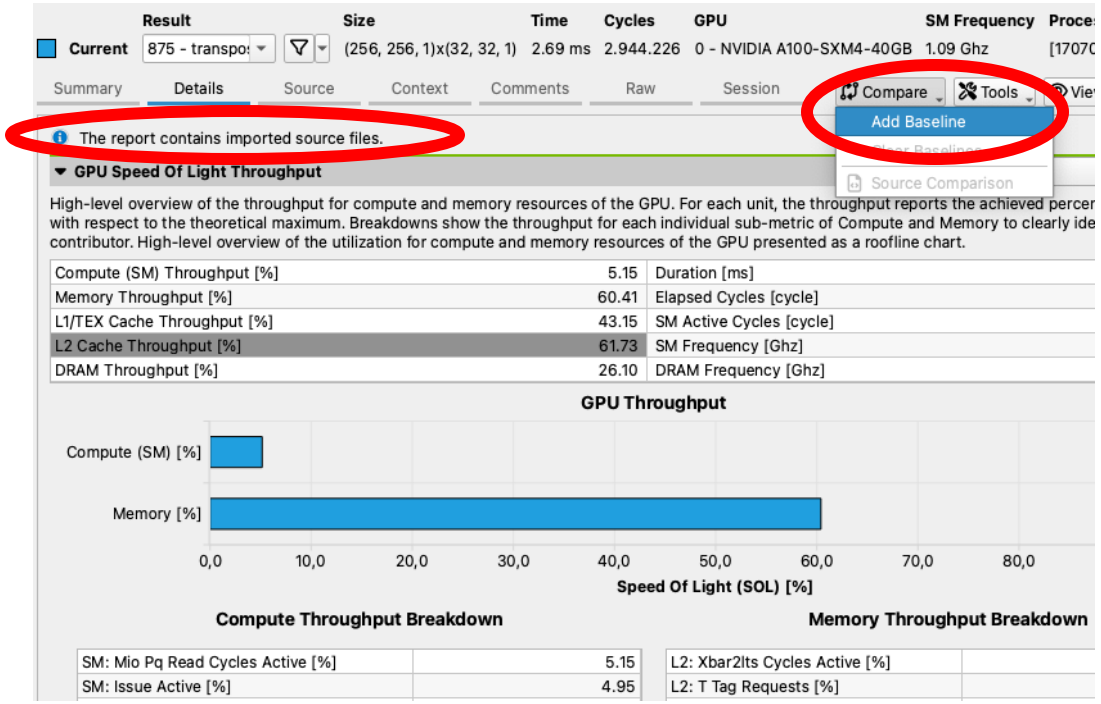
Transposes the block M → M‘

Coalesced write

Analysis with Nsight Compute

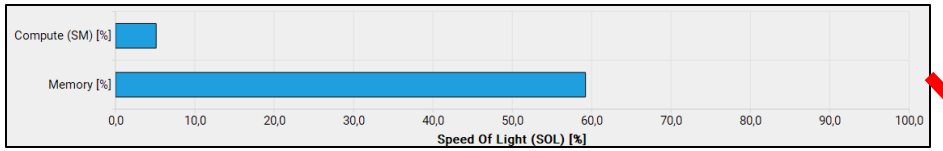
First steps

- 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

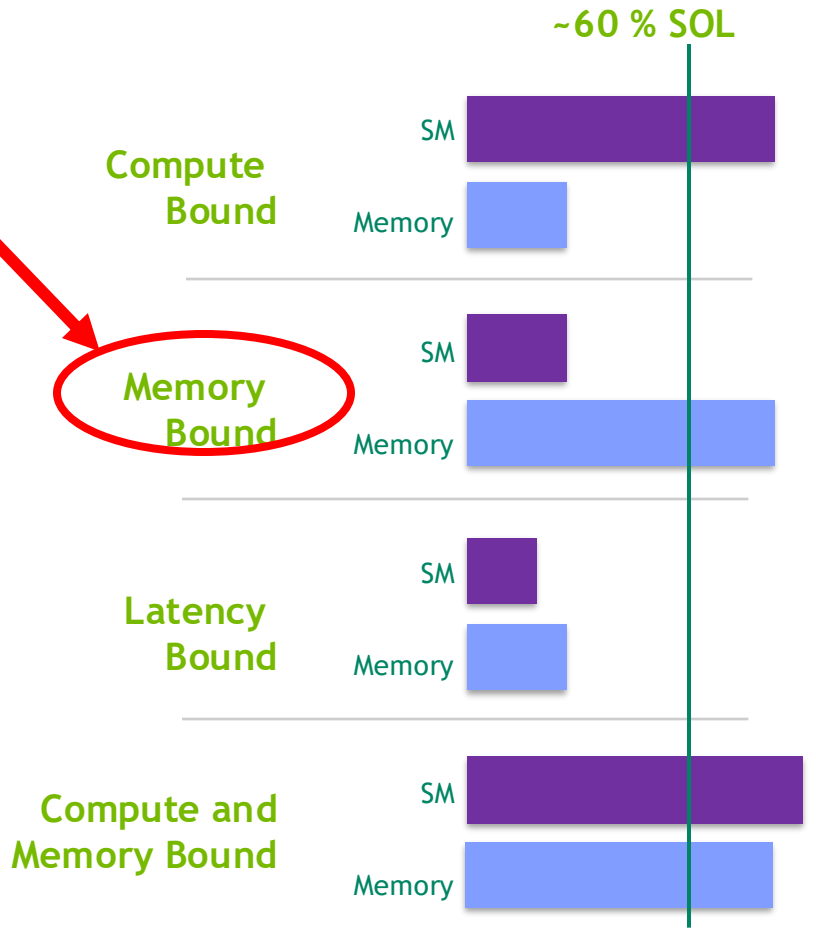
Performance limiter categories



- Four possible combinations of high/low…
  - …memory utilization
  - …compute utilization

- Good? Bad?

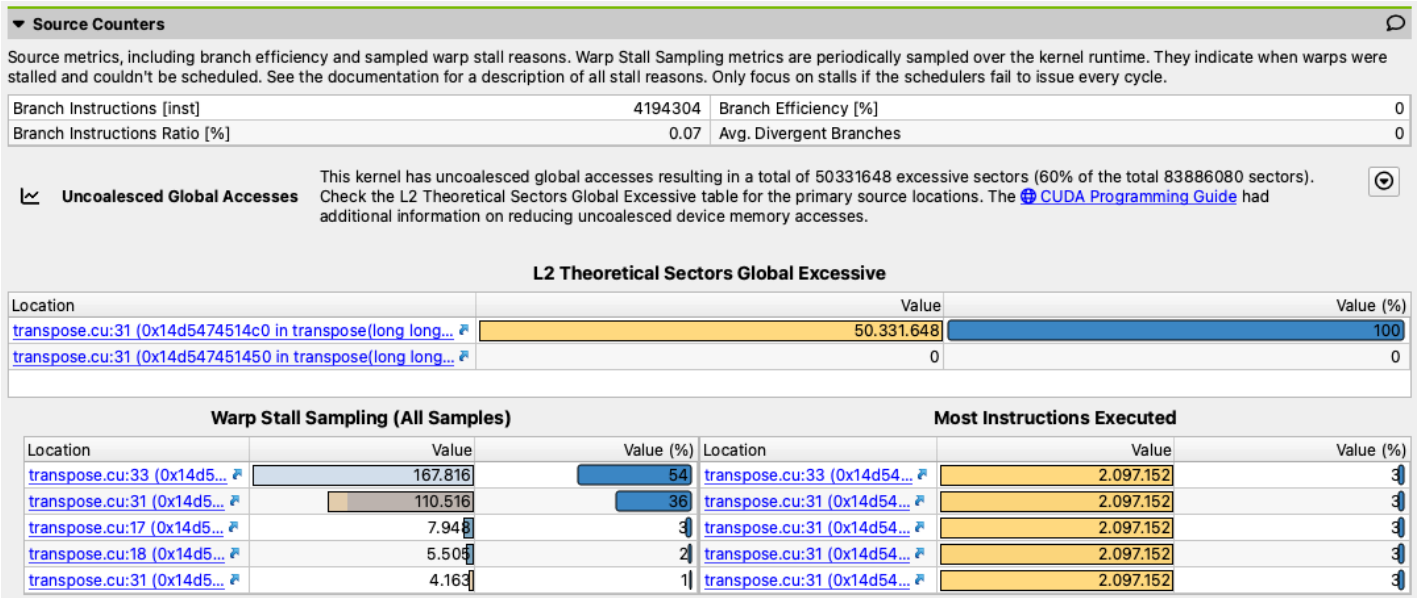
→ Depends on problem and its implementation



Analysis with Nsight Compute

Iterating and comparing

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



Source Counters

Source metrics, including branch efficiency and sampled warp stall reasons. Warp Stall Sampling 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]	4194304	Branch Efficiency [%]	0
Branch Instructions Ratio [%]	0.07	Avg. Divergent Branches	0

Uncoalesced Global Accesses

This kernel has uncoalesced global accesses resulting in a total of 50331648 excessive sectors (60% of the total 83886080 sectors). Check the L2 Theoretical Sectors Global Excessive table for the primary source locations. The CUDA Programming Guide had additional information on reducing uncoalesced device memory accesses.

L2 Theoretical Sectors Global Excessive

Location	Value	Value (%)
transpose.cu:31 (0x14d5474514c0 in transpose(long long...	50.331.648	100
transpose.cu:31 (0x14d547451450 in transpose(long long...	0	0

Warp Stall Sampling (All Samples)

Location	Value	Value (%)
transpose.cu:33 (0x14d5...	167.816	54
transpose.cu:31 (0x14d5...	110.516	36
transpose.cu:17 (0x14d5...	7.948	3
transpose.cu:18 (0x14d5...	5.505	2
transpose.cu:31 (0x14d5...	4.163	1

Most Instructions Executed

Location	Value	Value (%)
transpose.cu:33 (0x14d54...	2.097.152	3
transpose.cu:31 (0x14d54...	2.097.152	3
transpose.cu:31 (0x14d54...	2.097.152	3
transpose.cu:31 (0x14d54...	2.097.152	3
transpose.cu:31 (0x14d54...	2.097.152	3

Roofline Analysis

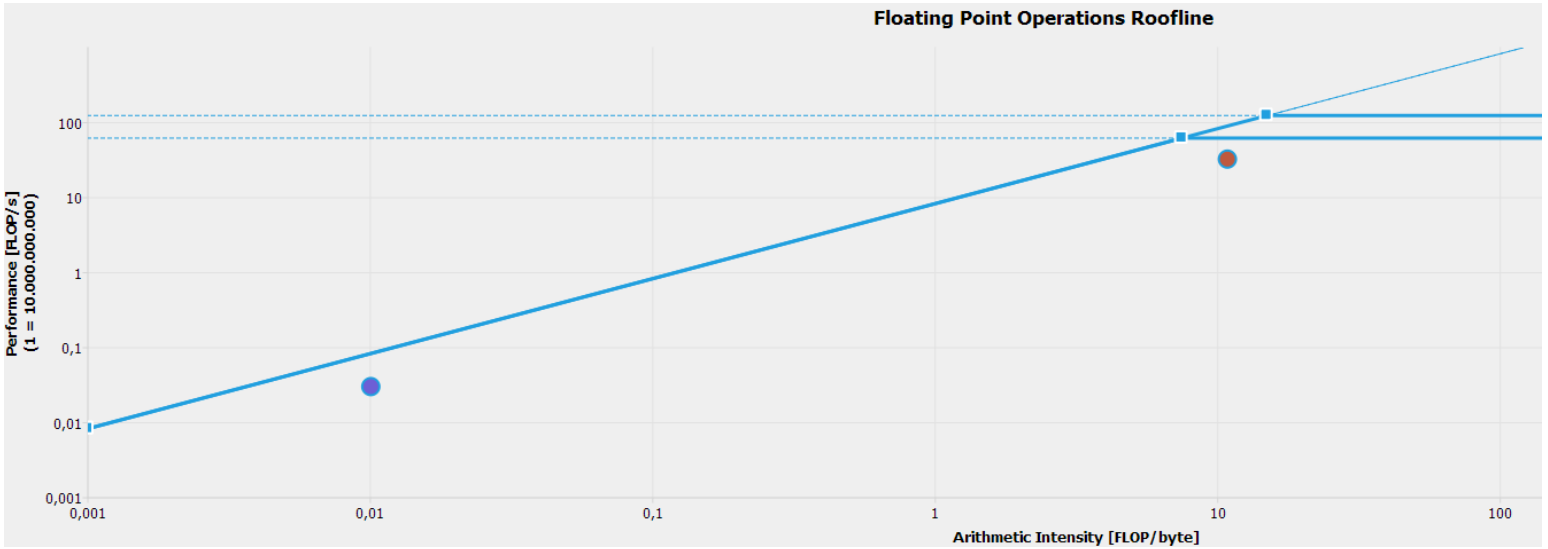
How well is the hardware utilized?

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



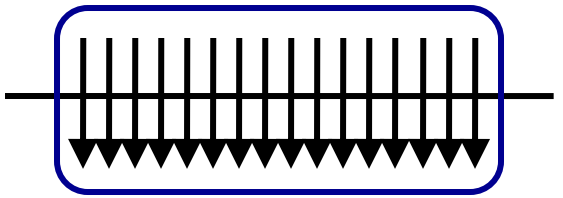
GTC session:
S32062: Performance Tuning CUDA Applications with the Roofline Model

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

Branch Divergence

Recap: Warp execution

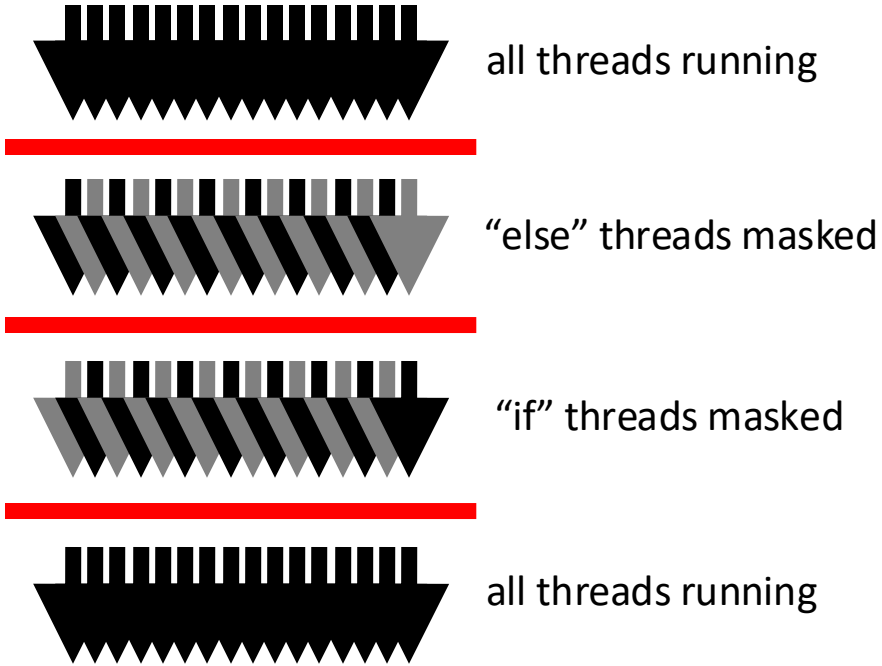
- GPUs use the Single Instruction Multiple Threads (SIMT) execution
  - functionally transparent to the programmer
  - but has performance implications
- warp
  - group of synchronously* executing threads
    (*since Volta: Independent Thread Scheduling)
  - neighbor threads (mostly x dimension)
  -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



Branch Divergence

Within Warp

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



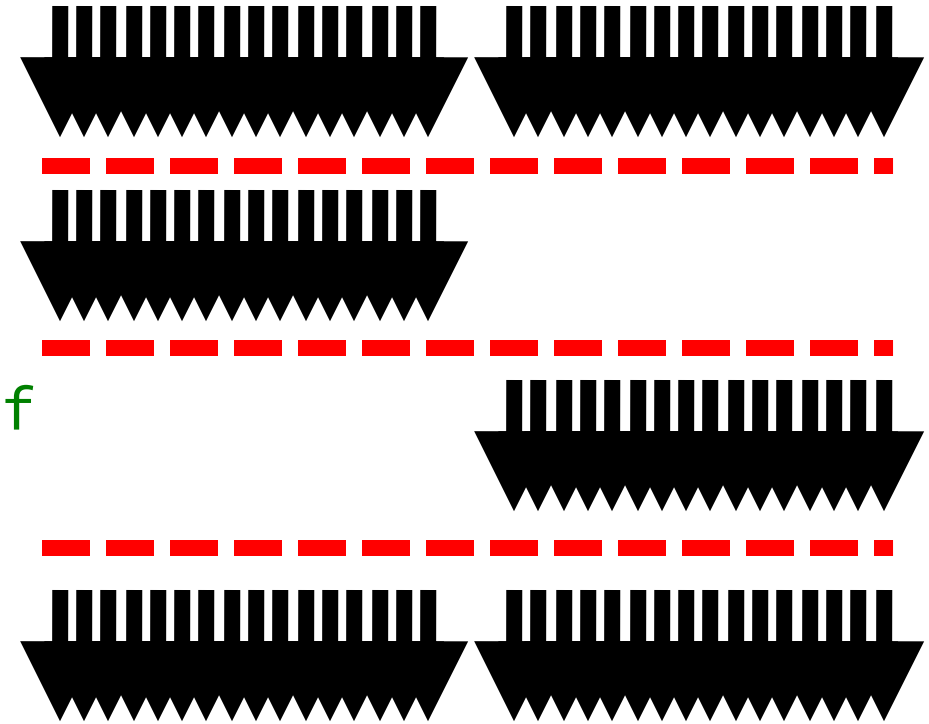
divergence *within* warp → performance penalty

```
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:

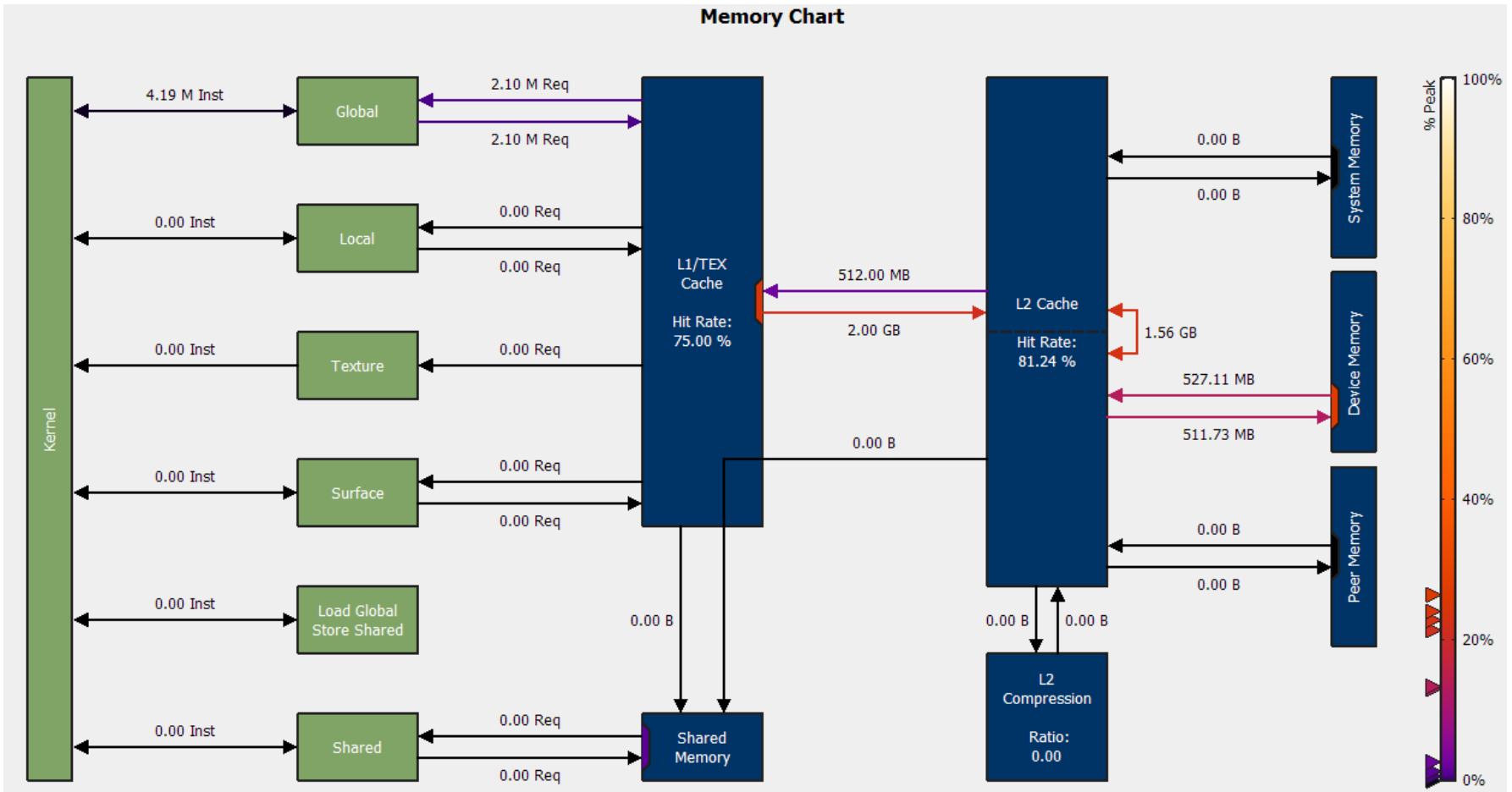
- Usually: Fix your access pattern
- Try to use shared memory (but first, check cache behavior)
- Look for different way of storage or better algorithm

Avoid divergent branches

Use the tools!

GLOBAL, LOCAL, L1, L2?

Understanding the memory hierarchy



MEMORY TRANSACTIONS AND COALESCING

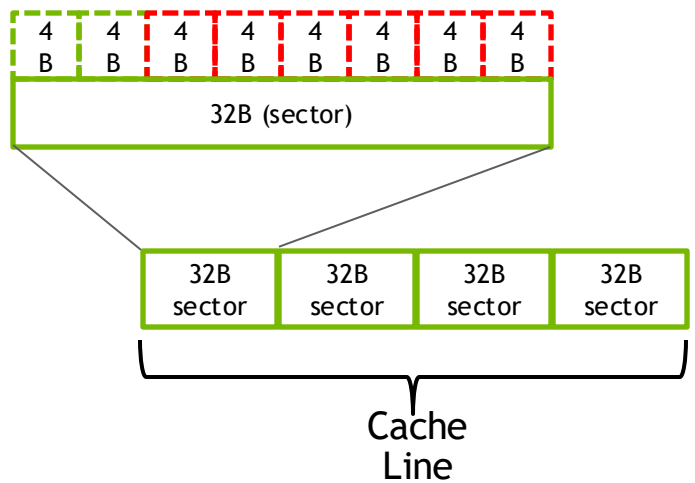
Access to global memory triggers transactions (Device Mem Access)

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

Here: 8 byte datatype



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

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

ANALYSIS WITH NSIGHT COMPUTE

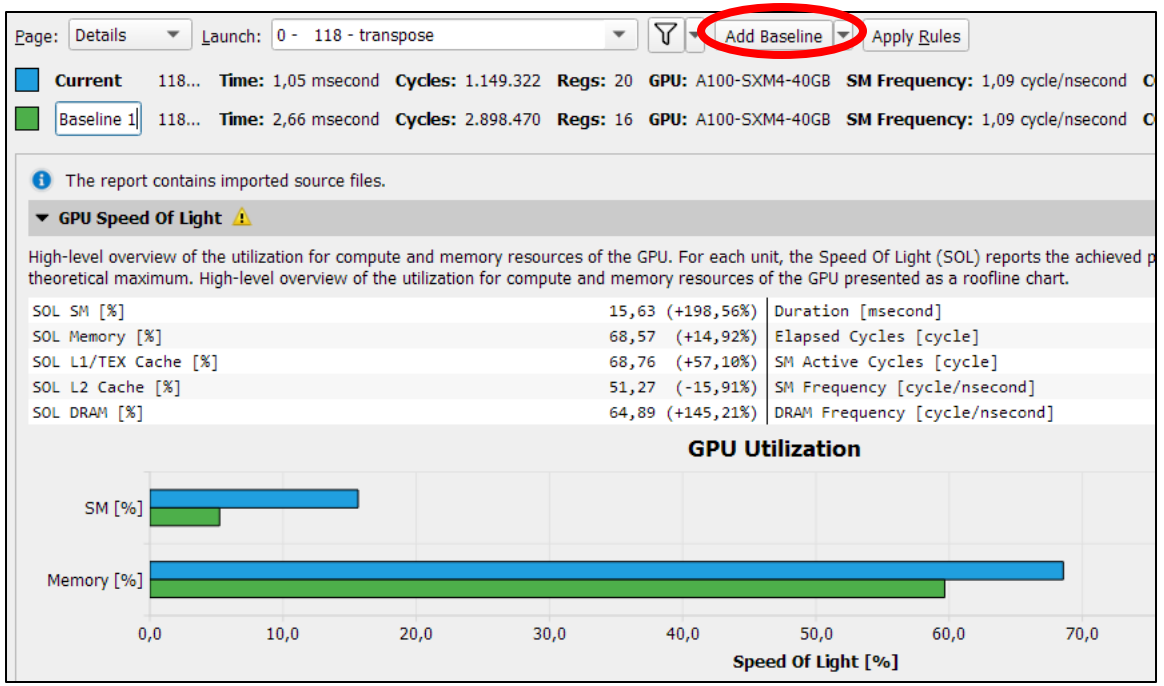
Iterating and comparing

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/...

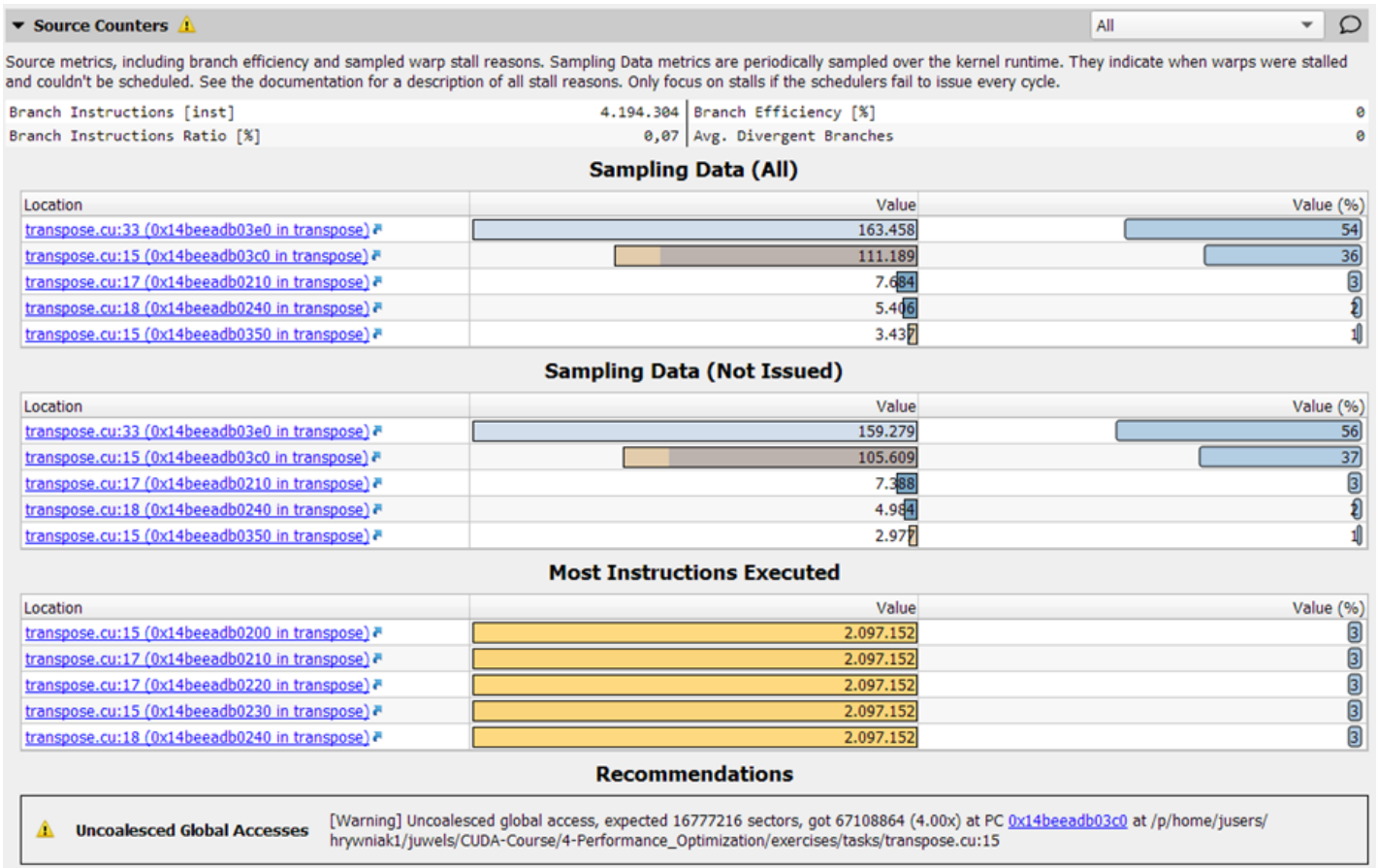


ANALYSIS WITH NSIGHT COMPUTE

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

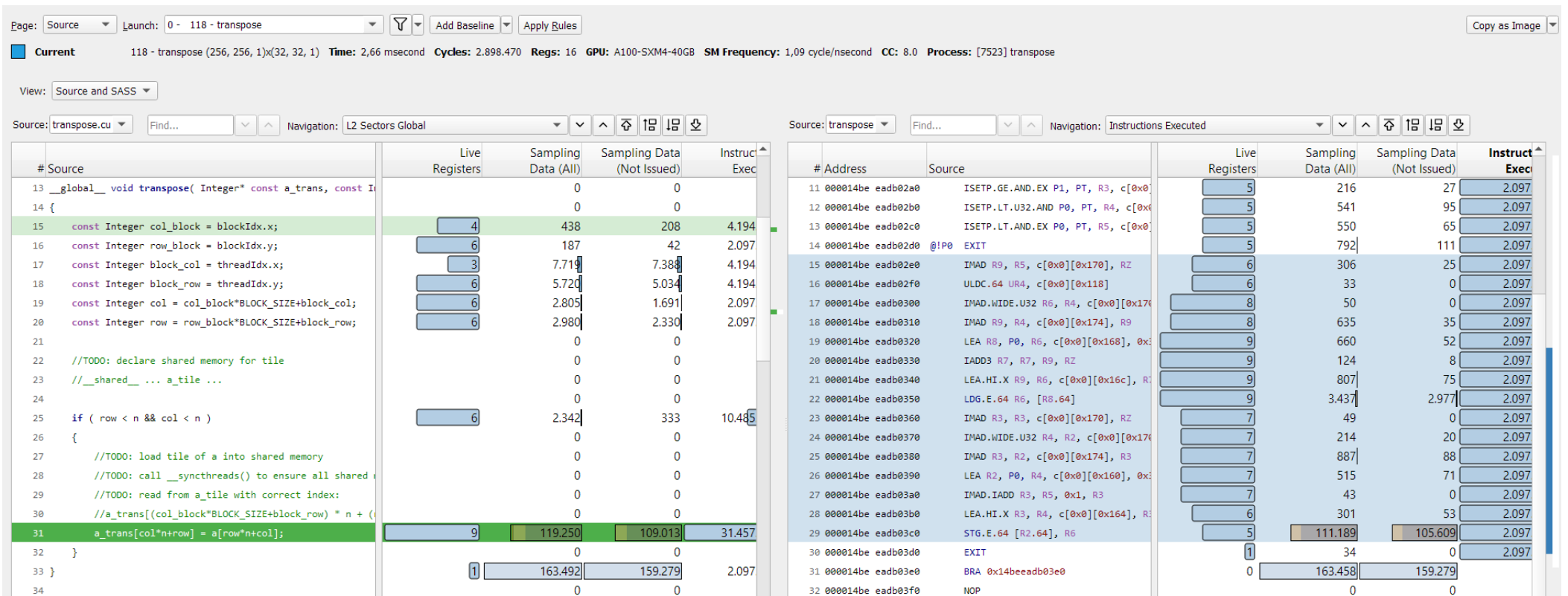
Location	Value	Value (%)
transpose.cu:15 (0x14beeadb0200 in transpose)	2.097.152	3
transpose.cu:17 (0x14beeadb0210 in transpose)	2.097.152	3
transpose.cu:17 (0x14beeadb0220 in transpose)	2.097.152	3
transpose.cu:15 (0x14beeadb0230 in transpose)	2.097.152	3
transpose.cu:18 (0x14beeadb0240 in transpose)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



ROOFLINE ANALYSIS

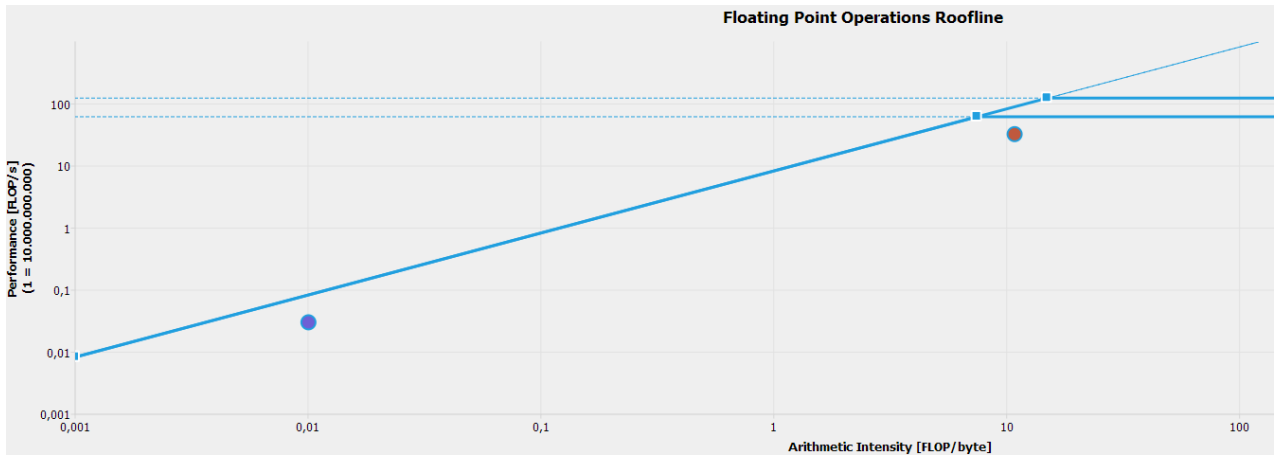
How well is the hardware utilized?

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


MORE DETAILS

CUDA BASICS

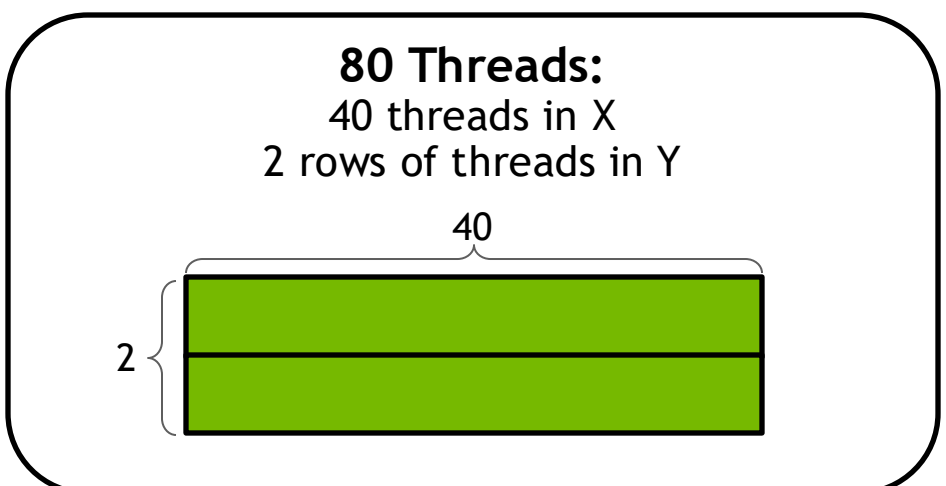
Blocks of threads, warps

- Single Instruction Multiple Threads (SIMT) model
- CUDA hierarchy: **Grid -> Blocks -> Threads**
- One **warp** = 32 threads.
- Why does it matter ?
  Many optimizations based on behavior at the warp level

CUDA BASICS

Mapping threads

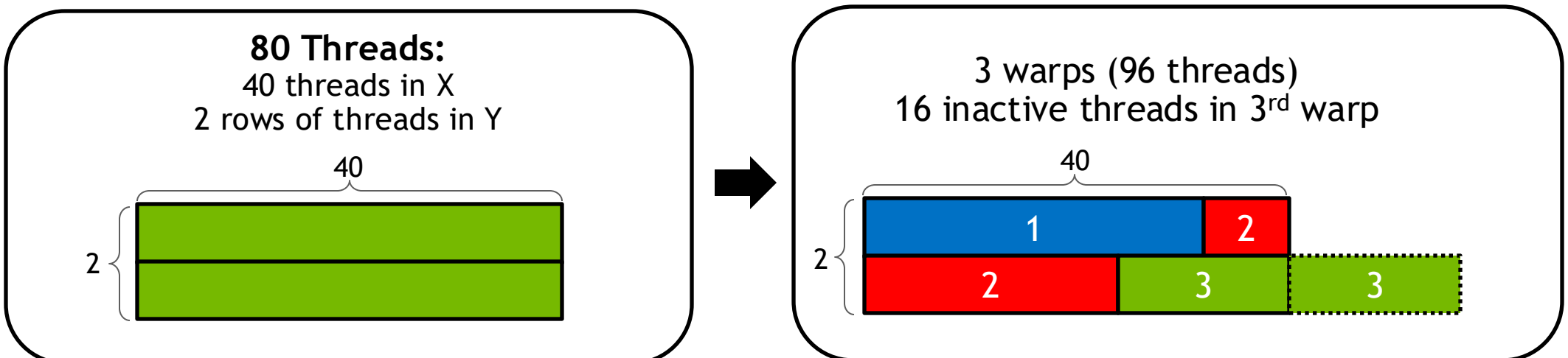
- **Thread blocks can be 1D, 2D, 3D**
  Only for convenience. HW “looks” at threads in 1D
- **Consecutive 32 threads** belong to the same **warp**



CUDA BASICS

Mapping threads

- **Thread blocks can be 1D, 2D, 3D**
  Only for convenience. HW "looks" at threads in 1D
- **Consecutive 32 threads** belong to the same **warp**



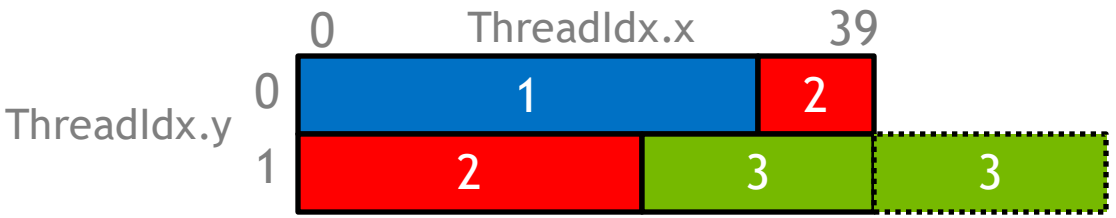
CUDA BASICS

Control Flow

- Different warps can execute different code
  No impact on performance
  Each warp maintains its own Program Counter

- Different code path inside the same warp ?
  Threads that don't participate are masked out,
  but the whole warp executes both sides of the branch

CONTROL FLOW



Instructions, time

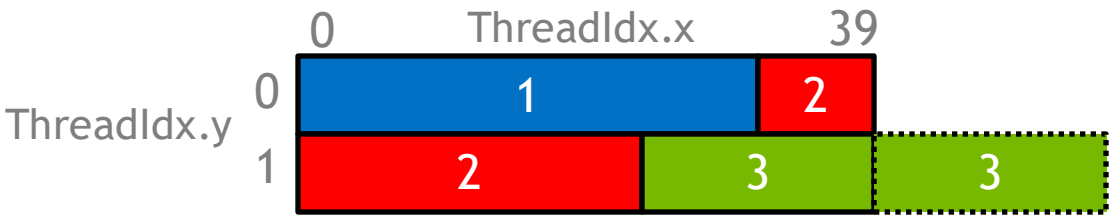
```
A;
if(threadIdx.y==0)
  B;
else
  C;
D;
```

Warp 1: 0 … 31

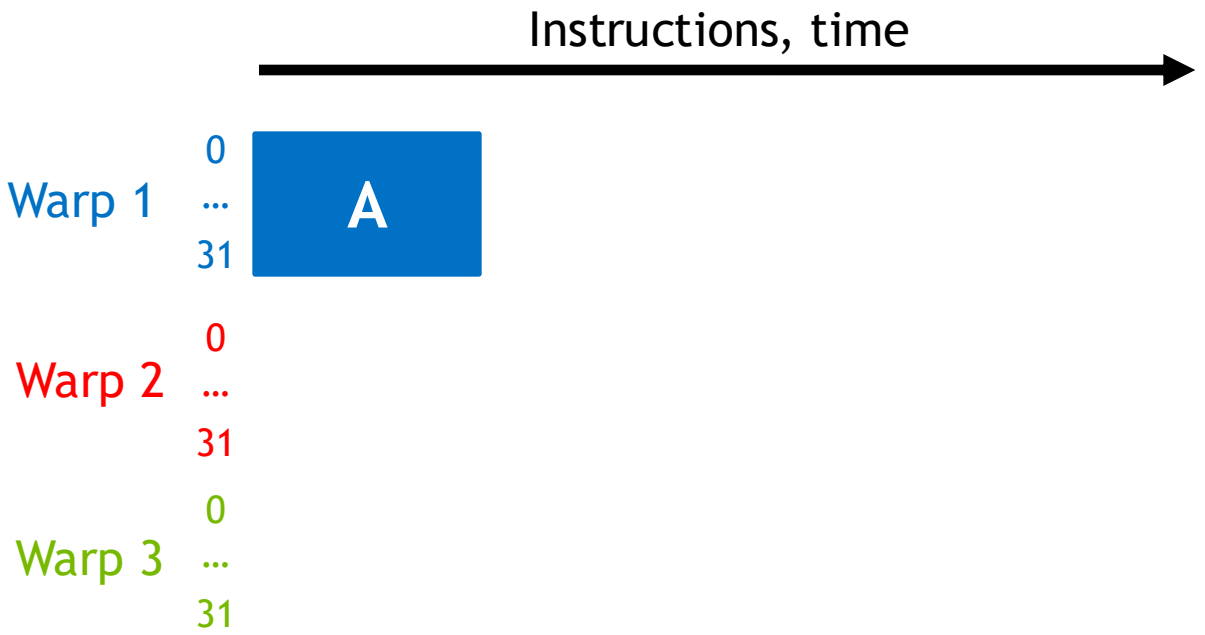
Warp 2: 0 … 31

Warp 3: 0 … 31

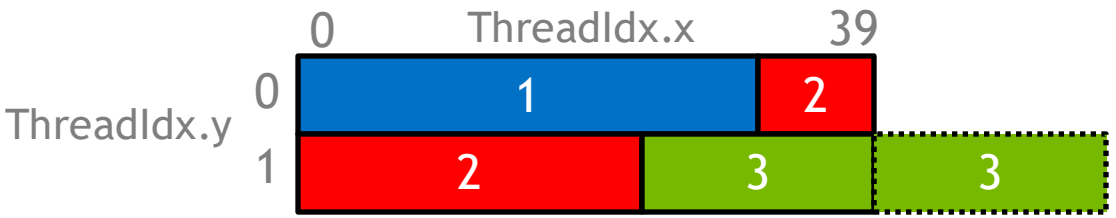
CONTROL FLOW



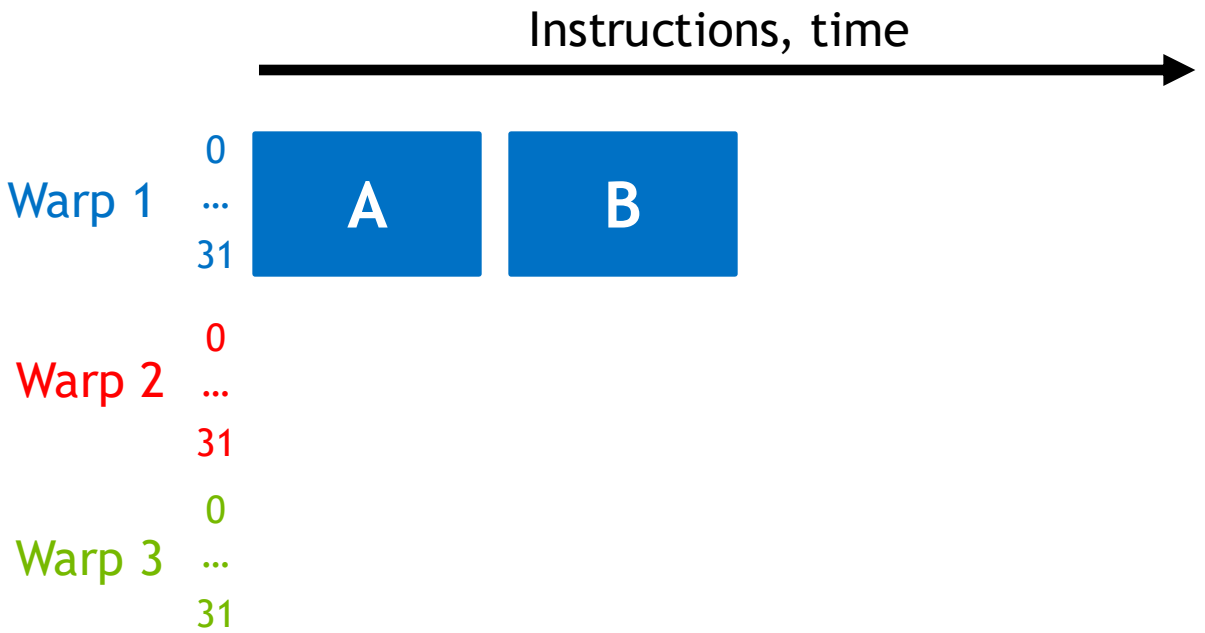

```
A;
if(threadIdx.y==0)
  B;
else
  C;
D;
```



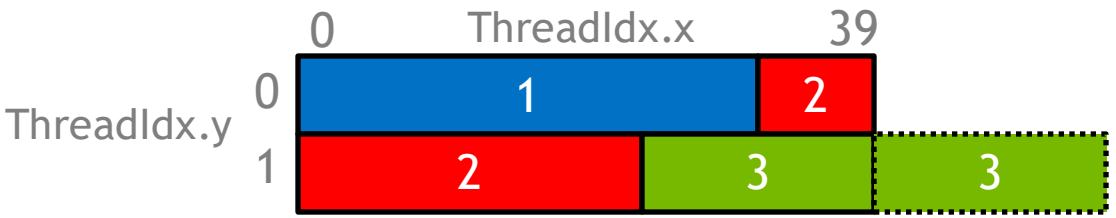
CONTROL FLOW



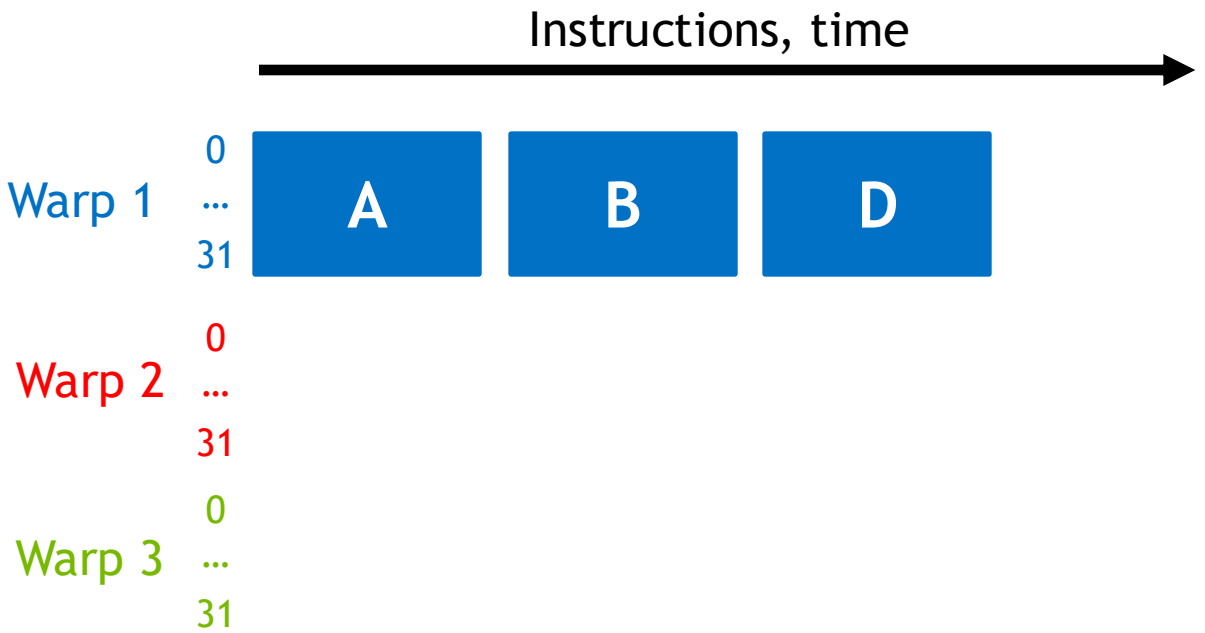

```
A;
if(threadIdx.y==0)
  B;
else
  C;
D;
```



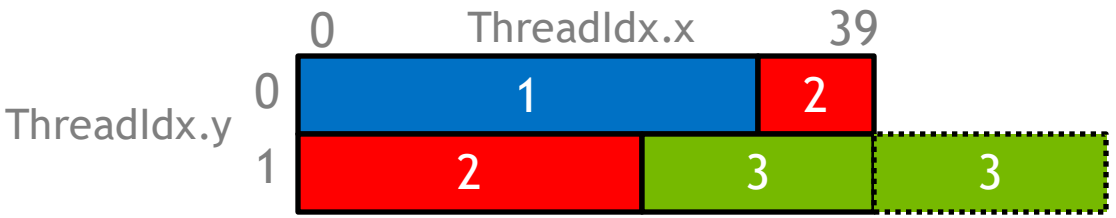
CONTROL FLOW



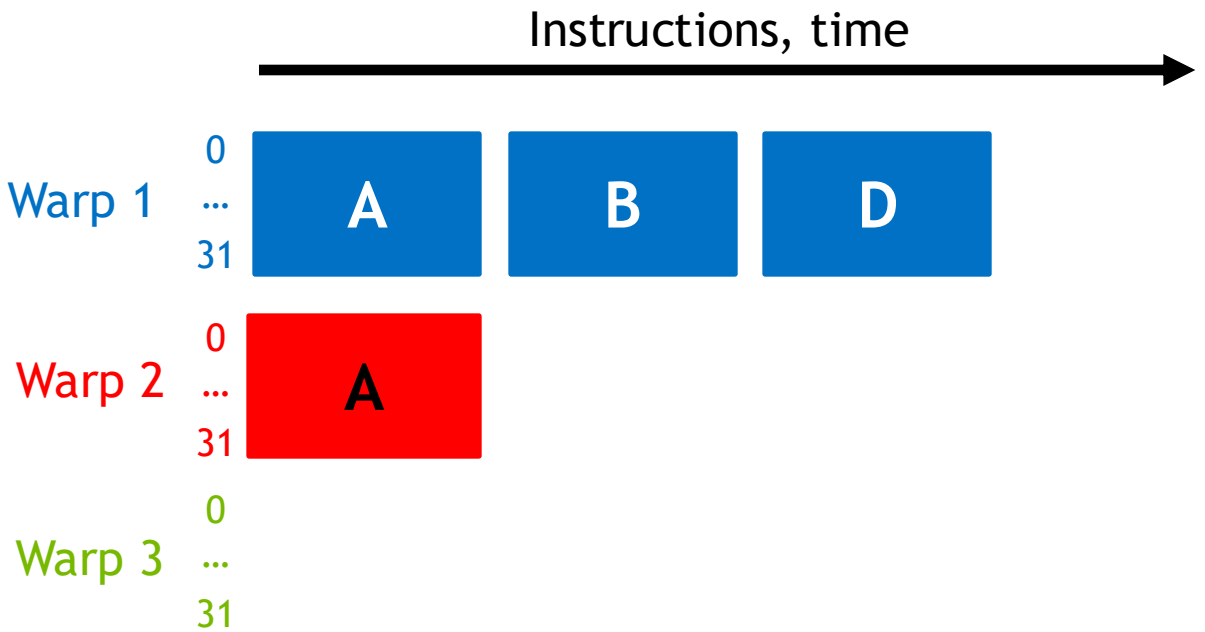

```
A;
if(threadIdx.y==0)
  B;
else
  C;
D;
```



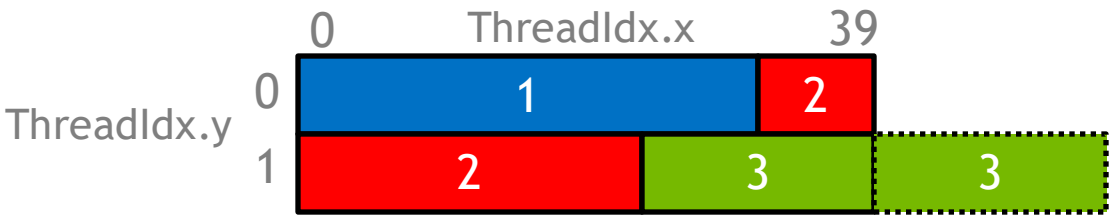
CONTROL FLOW



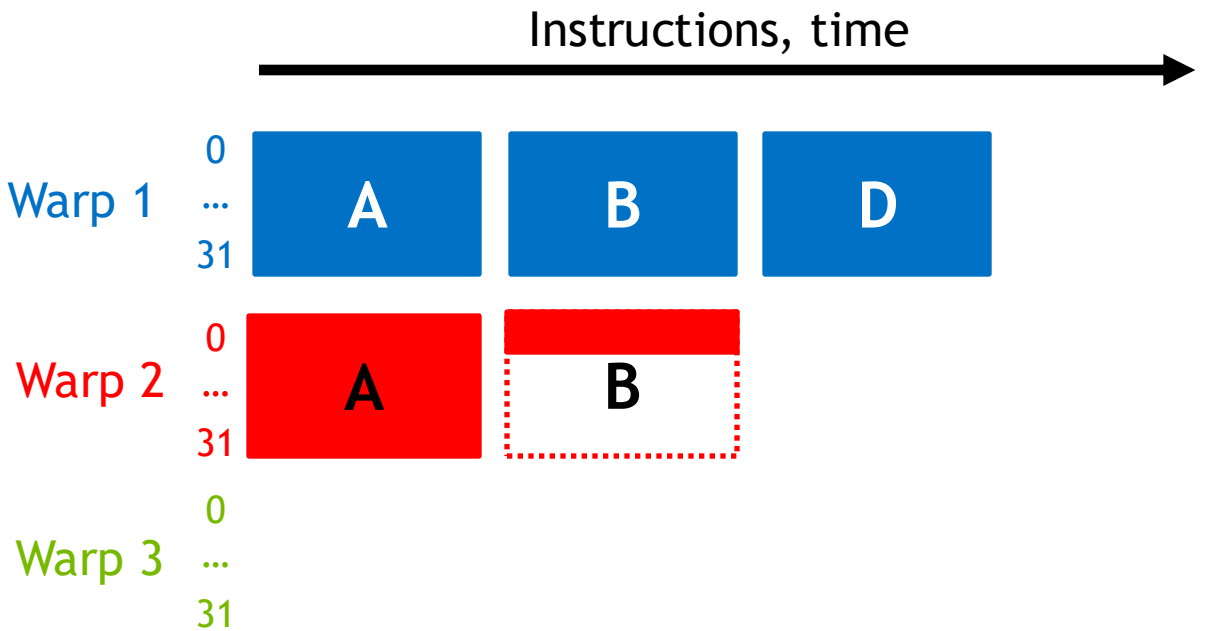

```
A;
if(threadIdx.y==0)
  B;
else
  C;
D;
```



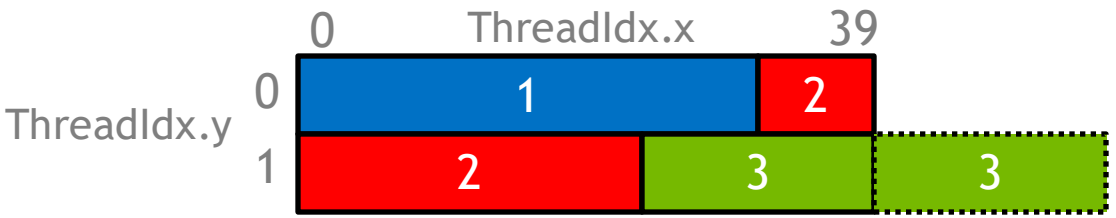
CONTROL FLOW



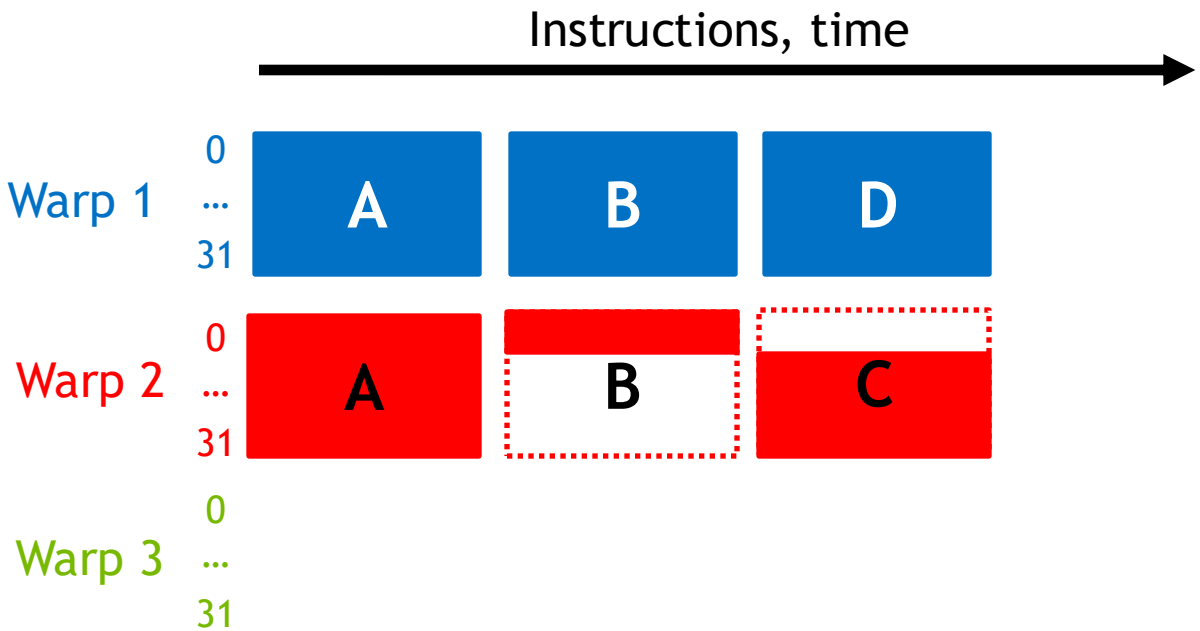

```
A;
if(threadIdx.y==0)
  B;
else
  C;
D;
```



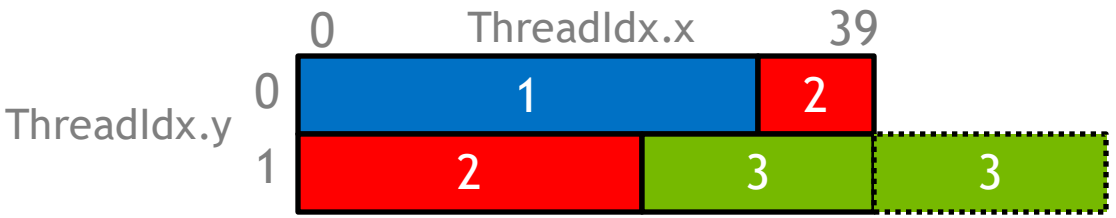
CONTROL FLOW



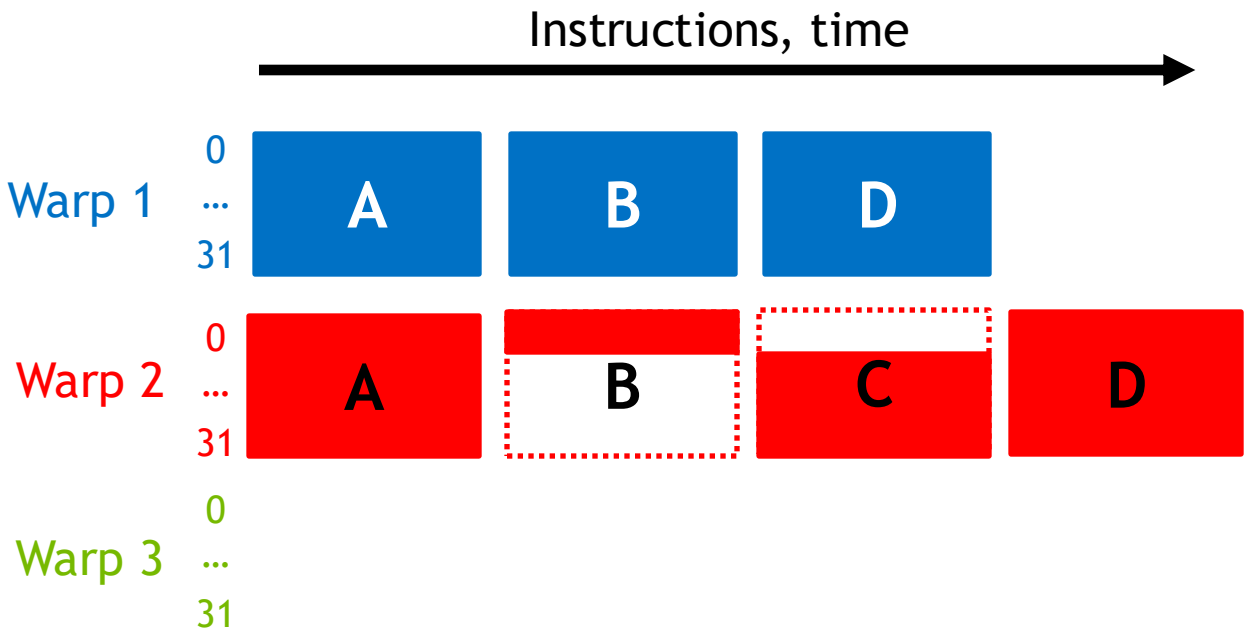

```
A;
if(threadIdx.y==0)
  B;
else
  C;
D;
```



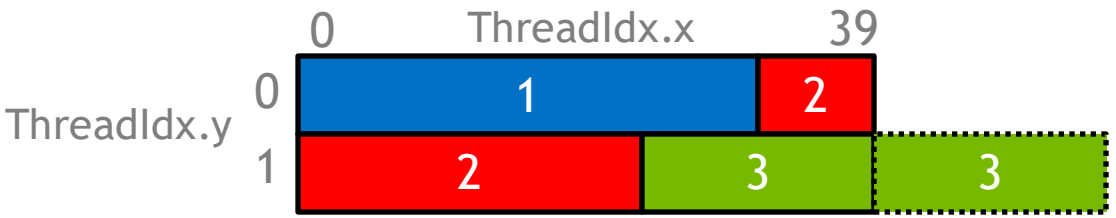
CONTROL FLOW



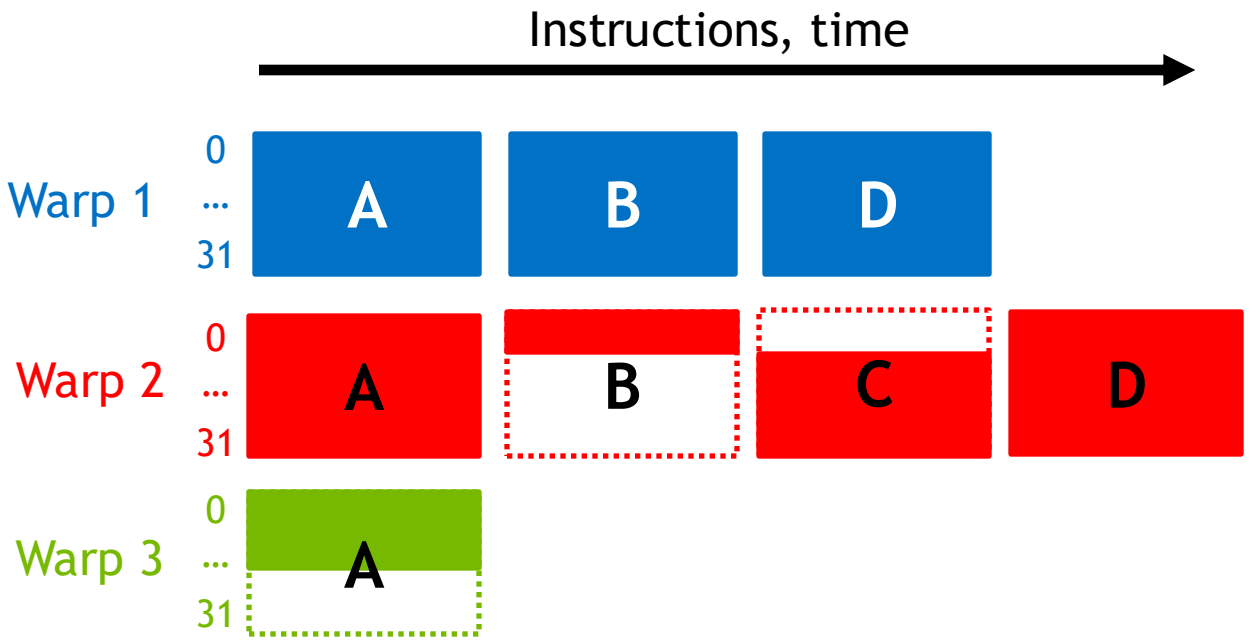

```
A;
if(threadIdx.y==0)
  B;
else
  C;
D;
```



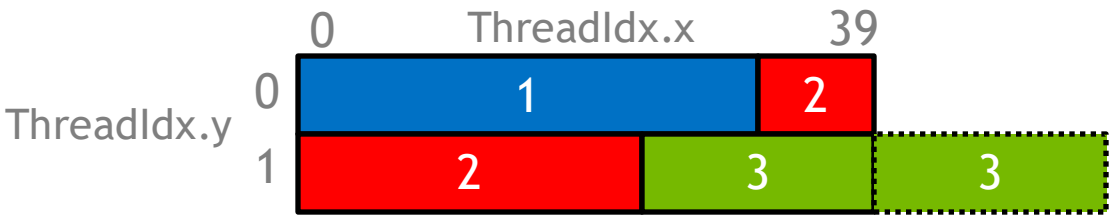
CONTROL FLOW



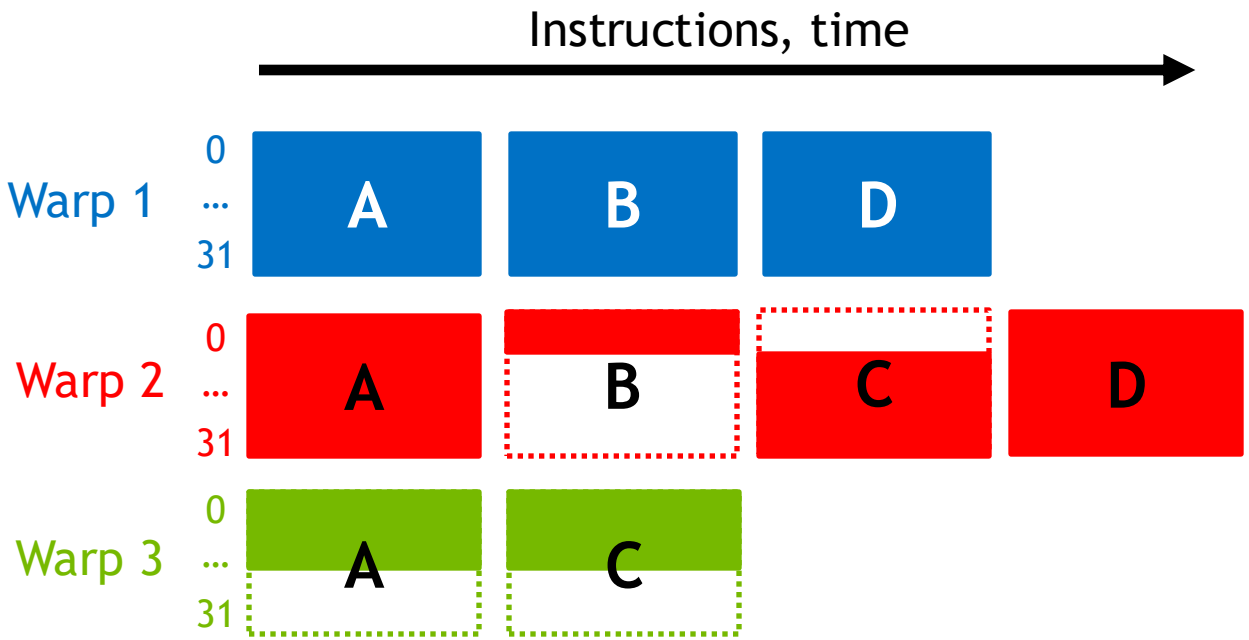

```
A;
if(threadIdx.y==0)
  B;
else
  C;
D;
```



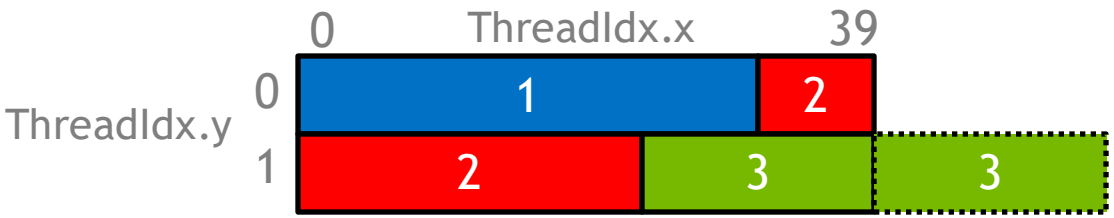
CONTROL FLOW



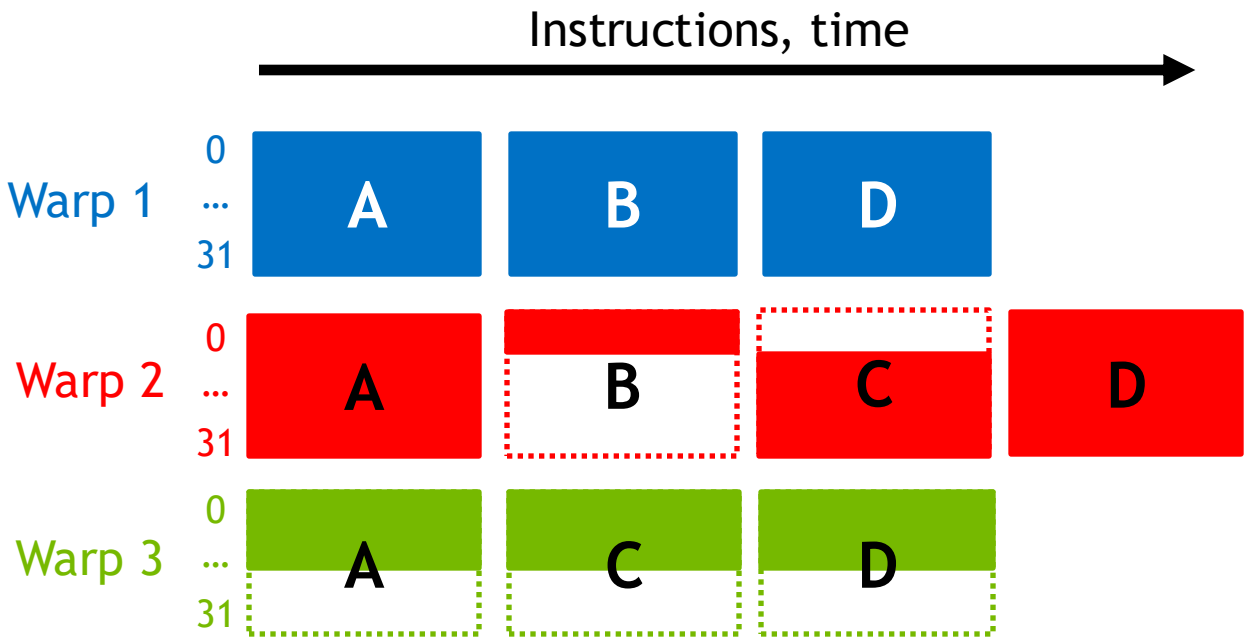

```
A;
if(threadIdx.y==0)
  B;
else
  C;
D;
```



CONTROL FLOW




```
A;
if(threadIdx.y==0)
  B;
else
  C;
D;
```



CONTROL FLOW

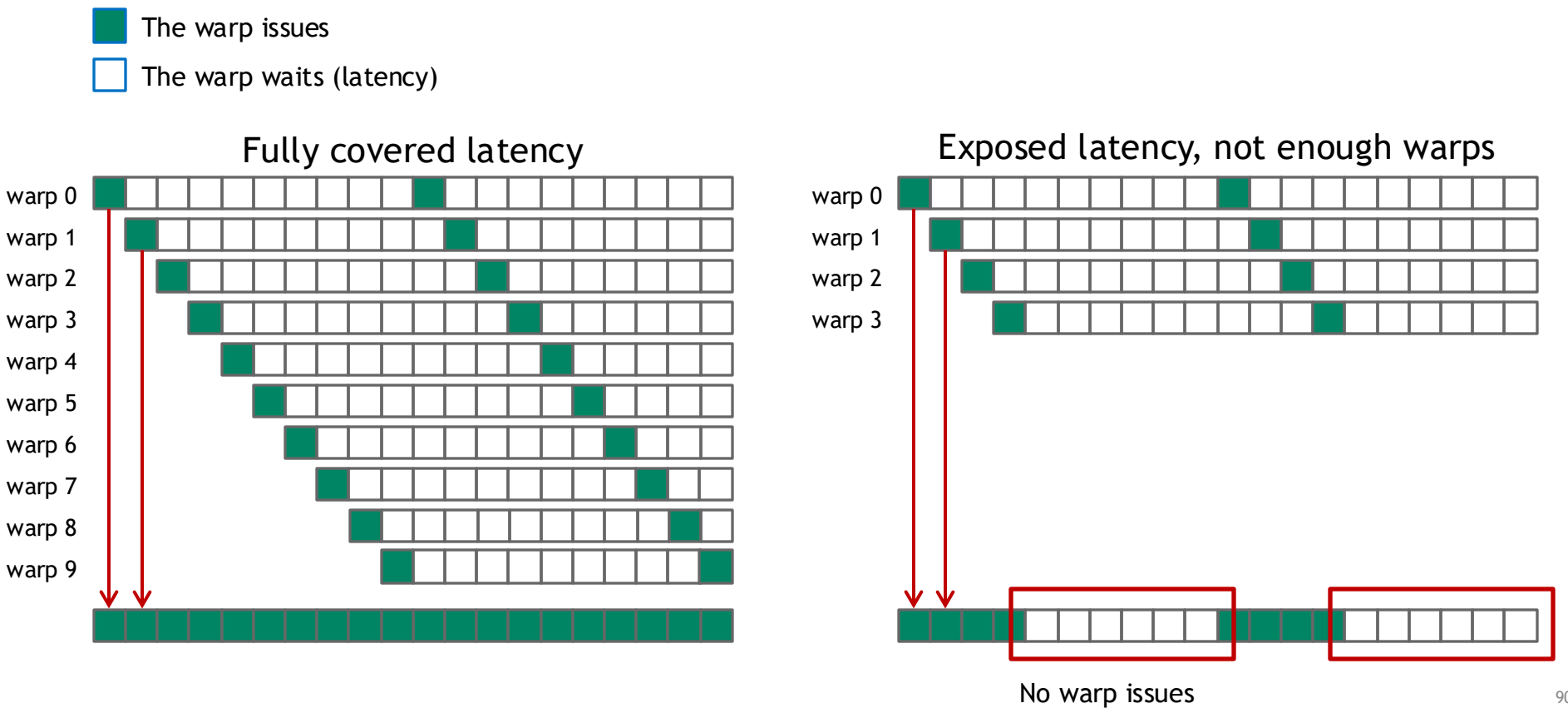
Takeaways

- Minimize thread divergence inside a warp
- Divergence between warps is fine
- Maximize “useful” cycles for each thread

HIDING LATENCY

LATENCY

GPUs cover latencies by having a lot of work in flight



SM RESOURCES

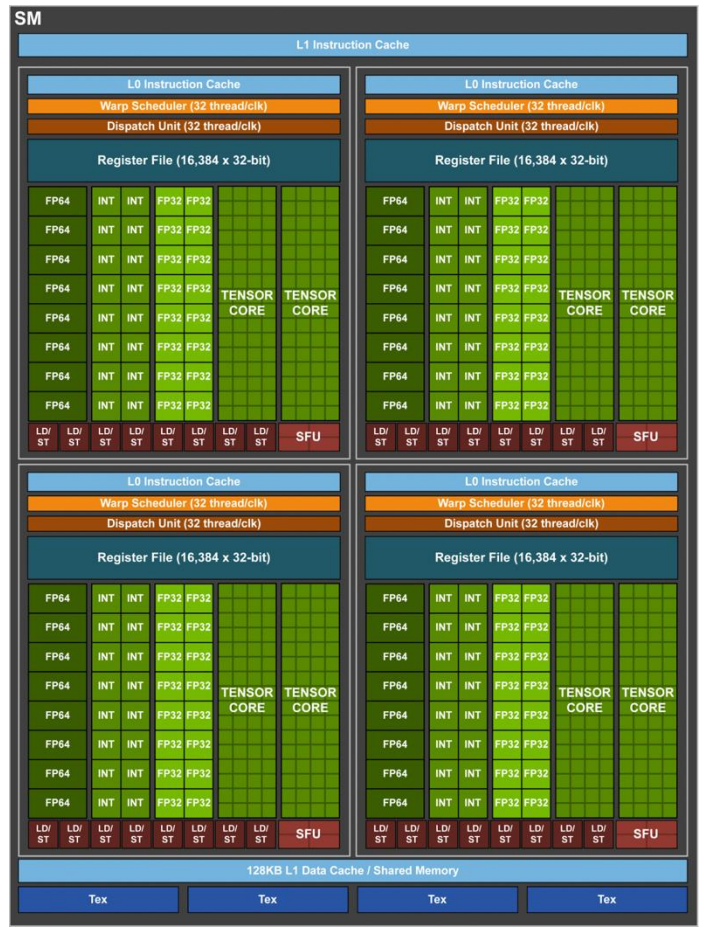
Each thread block needs:

- **Registers** (#registers/thread x #threads)
- **Shared memory** (0 ~ 96 KB)

Volta limits per SM:

- 256KB Registers
- 96KB Shared memory
- 2048 threads max (64 warps)
- 32 thread blocks max

Can schedule any resident warp without context switch



OCCUPANCY

$$\textbf{Occupancy} = \frac{\text{Achieved number of threads per SM}}{\text{Maximum number of threads per SM}}$$

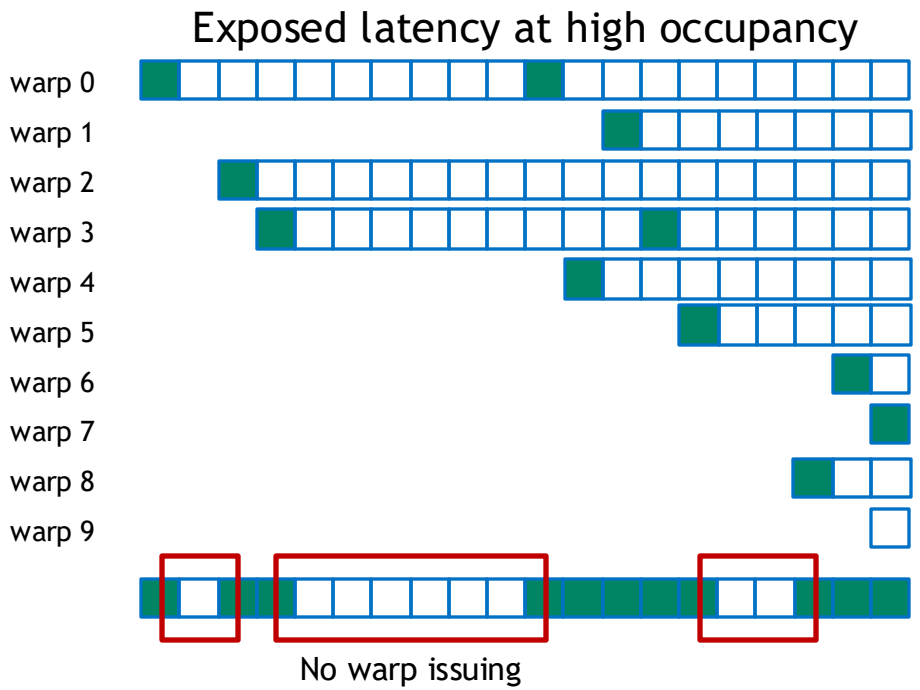
(Use the occupancy calculator XLS in CUDA Toolkit)

Higher occupancy can help to hide latency!
SM has more warp candidates to schedule while other warps are waiting for instructions to complete

Achieved occupancy vs **theoretical occupancy**
Need to run enough thread blocks to fill all the SMs!

LATENCY AT HIGH OCCUPANCY

Many active warps but with high latency instructions



INCREASING IN-FLIGHT INSTRUCTIONS

Ways to improve parallelism:

- **Expose enough parallelism**
  have O(10 x number of CUDA cores) threads
- **Improve occupancy**
  More threads -> more instructions
- **Improve instruction parallelism (ILP)**
  More **independent instructions** per thread

ADDITIONAL REFERENCES

GTC ‘21 talk from Nsys team: https://www.nvidia.com/en-us/on-demand/session/gtcspring21-s31566

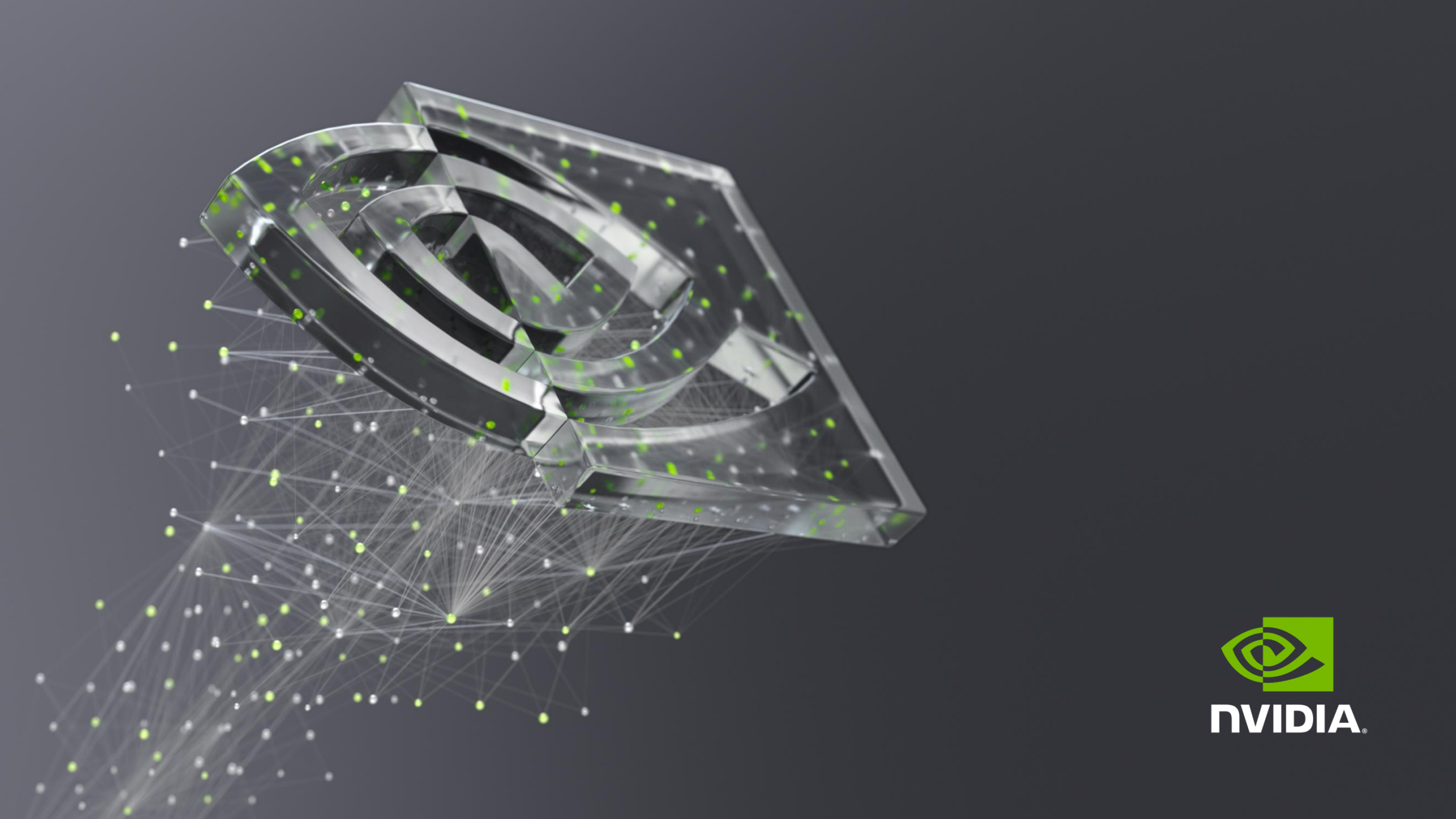
Self-paced lab GTC lab by the Nsight team: https://github.com/NVIDIA/nsight-training

https://developer.nvidia.com/nsight-systems and https://developer.nvidia.com/nsight-compute
+ usage tips, videos on these pages

More explanation on „Long Scoreboard Stall“ and other warp states:
https://docs.nvidia.com/nsight-compute/ProfilingGuide/index.html#statistical-sampler

Nsight Compute is heavily customizable via Sections/Rules: https://docs.nvidia.com/nsight-compute/CustomizationGuide/index.html

For really advanced users:
https://docs.nvidia.com/nsight-compute/CustomizationGuide/index.html#report-file-format


NVIDIA