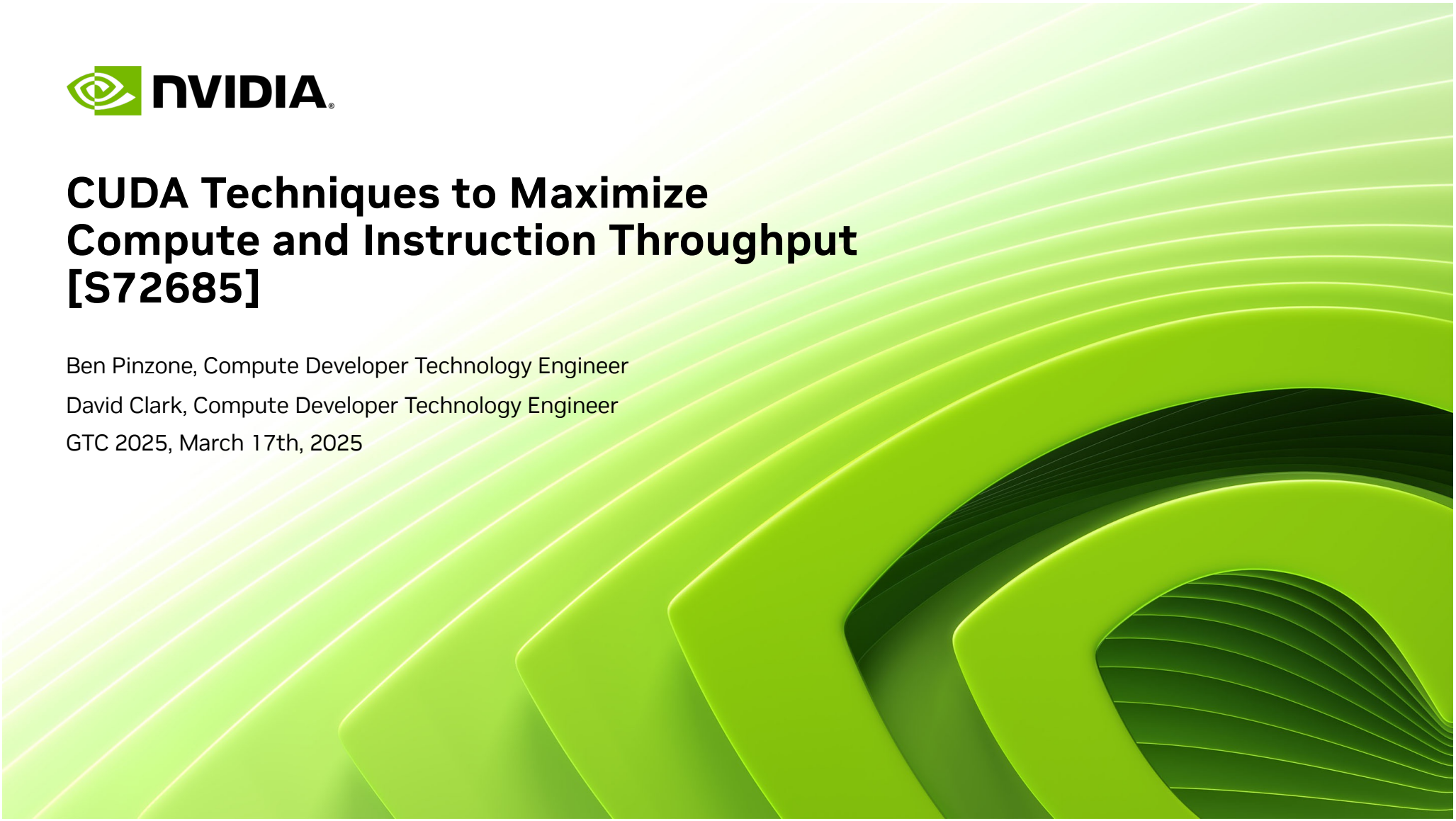


CUDA Techniques to Maximize Compute and Instruction Throughput [S72685]

Ben Pinzone, Compute Developer Technology Engineer

David Clark, Compute Developer Technology Engineer

GTC 2025, March 17th, 2025

Goals

- Refresh on GPU programming concepts.
- Briefly illustrate where these concepts are seen in profiling tools. (Nsight Compute)
  - **You'll see many screenshots of Nsight Compute – Nvidia's CUDA kernel profiling tool.**
- Outline general optimization strategies, using specific examples.
  - These are ideas, not hard rules! Your mileage may vary.
  - Code changes of non-trivial complexity should be motivated by profiler.

- Some points will be quick or simply reference other resources. Just to raise awareness.

Agenda

- GPU thread hierarchy, SIMT, and Warp divergence.
- Warp scheduler and Kernel profiling at a glance.
- Latency hiding and increasing instruction throughput.
- Reducing instruction count and making throughput useful.
- Tensor Core Summary

GPU thread hierarchy, SIMT, Warp divergence

GPU Overview

NVIDIA H100 SXM

Streaming Multiprocessor (SM)

Hopper architecture

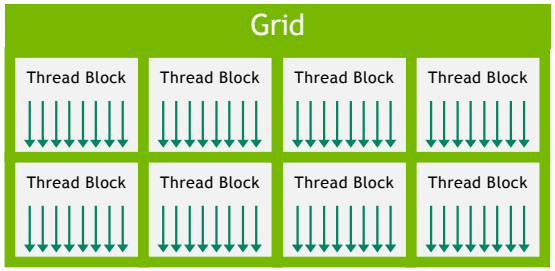
SM has 4 sub-partitions

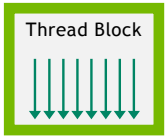
- 128 FP32 units
- 64 FP64 units
- 64 INT32 units
- 4 mixed-precision Tensor Cores
- 16 special function units (transcendentals)
- 4 warp schedulers

- 32 LD/ST units
- 64K 32-bit registers
- 256 KiB unified L1 data cache and shared memory
- Tensor Memory Accelerator (TMA)

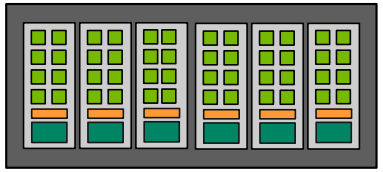
Thread Hierarchy: Grid & Blocks

CUDA/Software

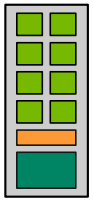




Hardware



Device

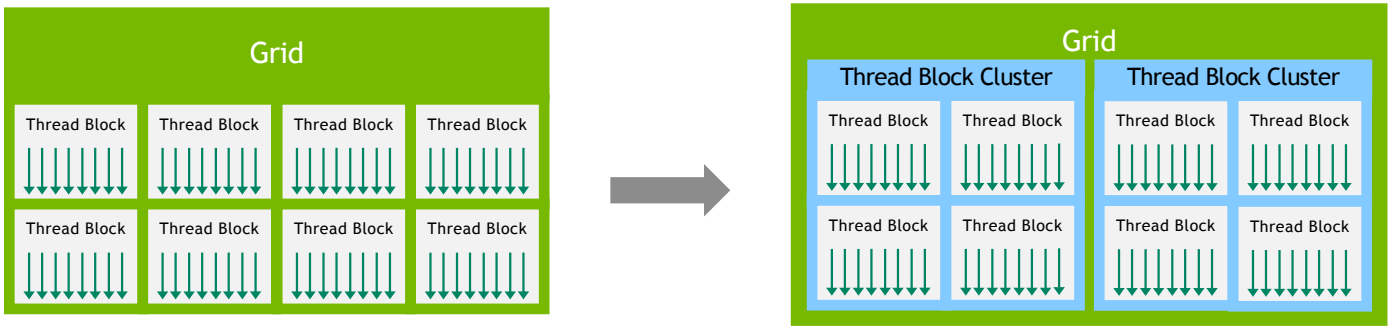


SM

- A CUDA kernel is a launched grid of thread blocks, which are completely independent.
- Thread blocks are executed on SMs.
  - Several blocks can reside on an SM concurrently.
  - Blocks do not migrate.
  - Each block can be scheduled on any of the available SMs, in any order, concurrently or in series.

Thread Hierarchy: Clusters

- Starting with Hopper GPUs, CUDA introduced an optional level in the thread hierarchy called **Thread Block Clusters**.
- Thread blocks in a cluster are guaranteed to be concurrently scheduled and enable efficient cooperation and data sharing for threads across multiple SMs.

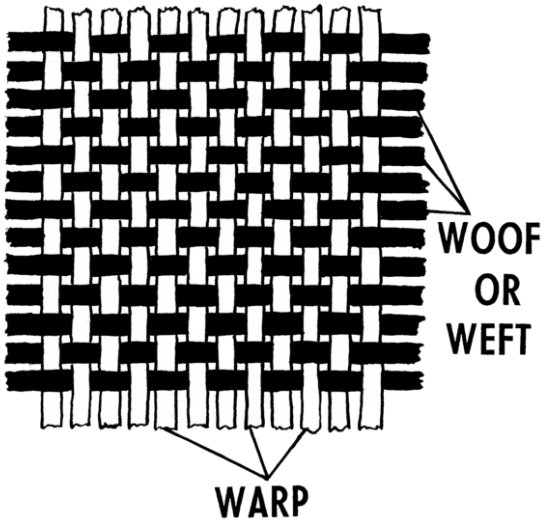


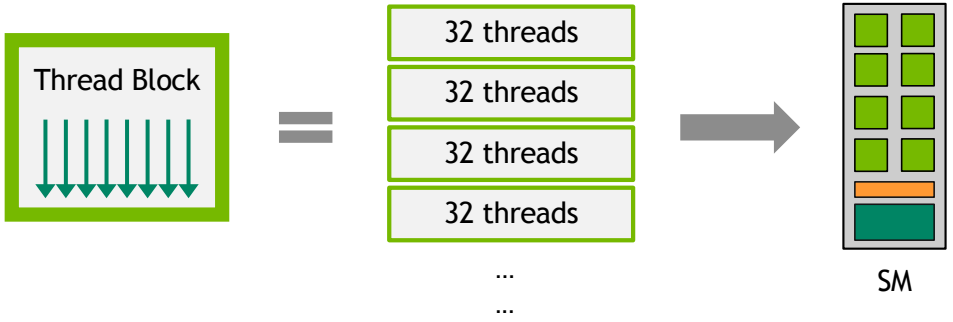
Thread Hierarchy: Warps

"Weaving, the first parallel thread technology"

–CUDA C++ Programming Guide

- 1 warp = 32 threads
- At runtime, a block is divided into warps for SIMT execution.
- Threads in a warp have consecutive thread IDs.
- The total number of warps in a block is defined as:
  - Ceil(threads per block / warp size)

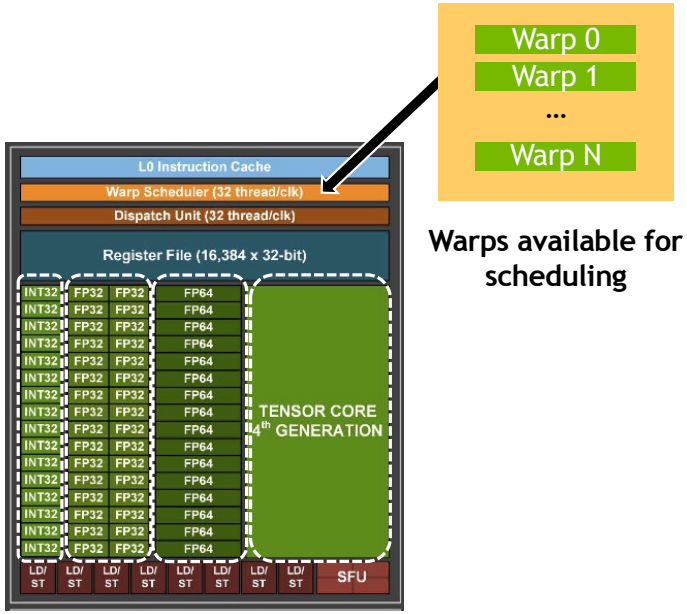




SIMT Architecture

Single-Instruction, Multiple-Thread

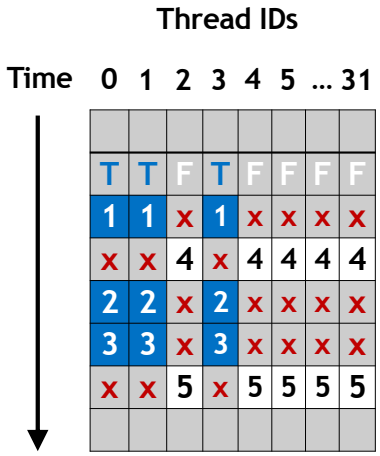
- Every thread has its own program counter.
- SIMT = SIMD + Program counters.
- You get SIMD execution while retaining the benefit of reasoning at the scalar / thread level.
  - (Rather than explicitly programming vectors.)
- So, what happens to SIMD HW if PCs are different?...



SIMT Architecture

Warp divergence

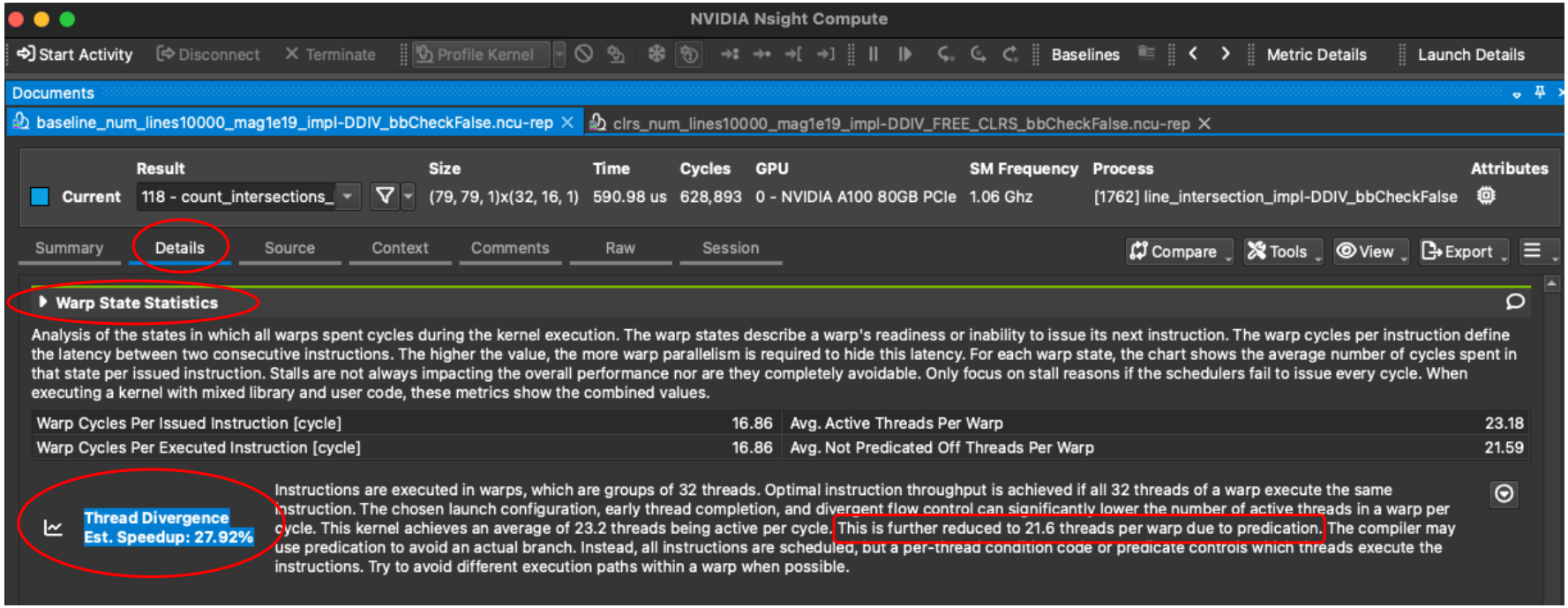
- If threads in a warp **diverge** (e.g., via a conditional branch), the warp **executes separately every branch** path.
- **Full efficiency** is realized when all **32 threads** of a warp agree on their execution path.
  - Aka they are **converged**.
- Every X is a missed opportunity to have the HW produce something useful.



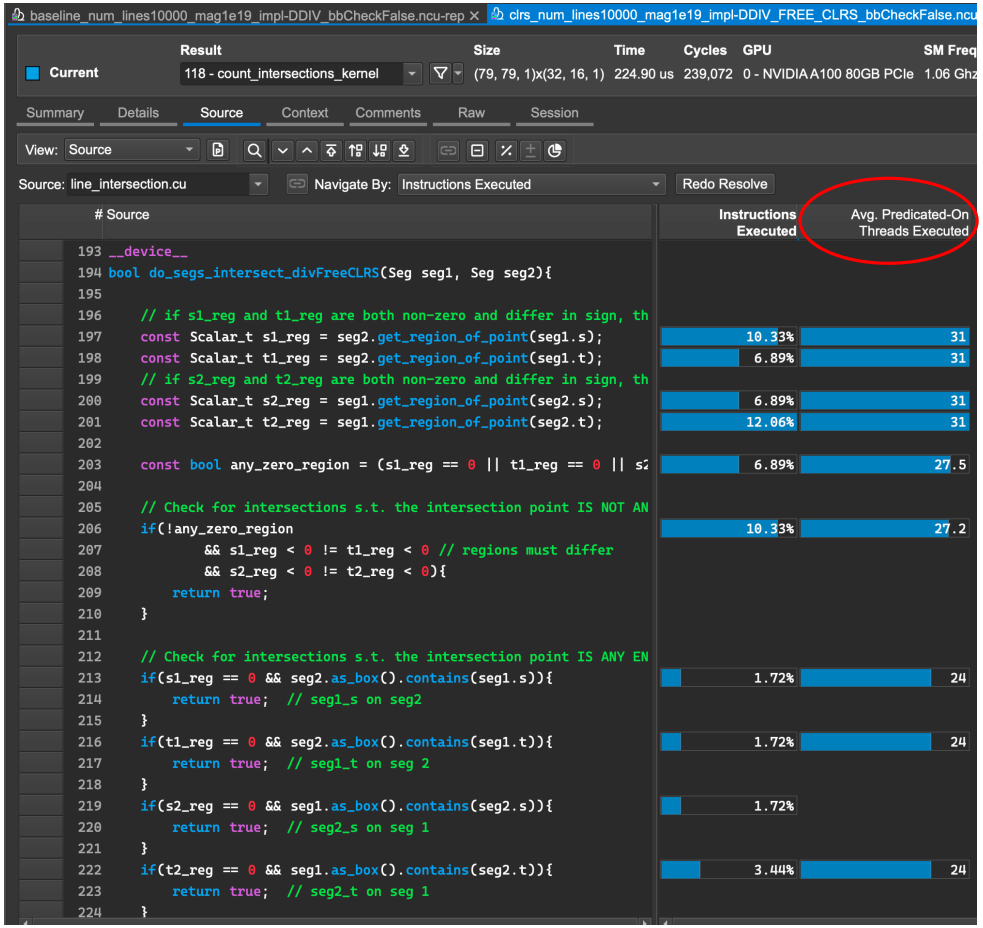
Code snippet

```
if (/*cond*/) {
  instruction 1
  instruction 2
  instruction 3
} else {
  instruction 4
  instruction 5
}
```

Where is Warp Divergence in NCU?



Where is Warp Divergence in NCU?



Consider divergence at your hot spot, specifically.

(Rather than averages)

Source page: Columns/metrics of interest:

- **Avg. Predicated-On Threads Executed**
  - At this instruction, how converged is my warp on average?
- **Divergent Branches**
  - Number of times branch target differed
- **Coming Soon: Derivative Avg. Predicated-On Threads Executed**
  - Helps characterize divergence slightly differently.
  - Consider a piece of code that:
    - At top level, diverges severely, but only once. Stays diverged.
    - Meanwhile in lower level code, you frequently diverge and reconverge, although less severely.
    - Now, which metric characterizes what?
      - Derivative metric: In this case, has higher value for the top level divergence.
      - Divergent Branches metric: In this case, has higher value for the lower level code.

Tips for reducing warp divergence

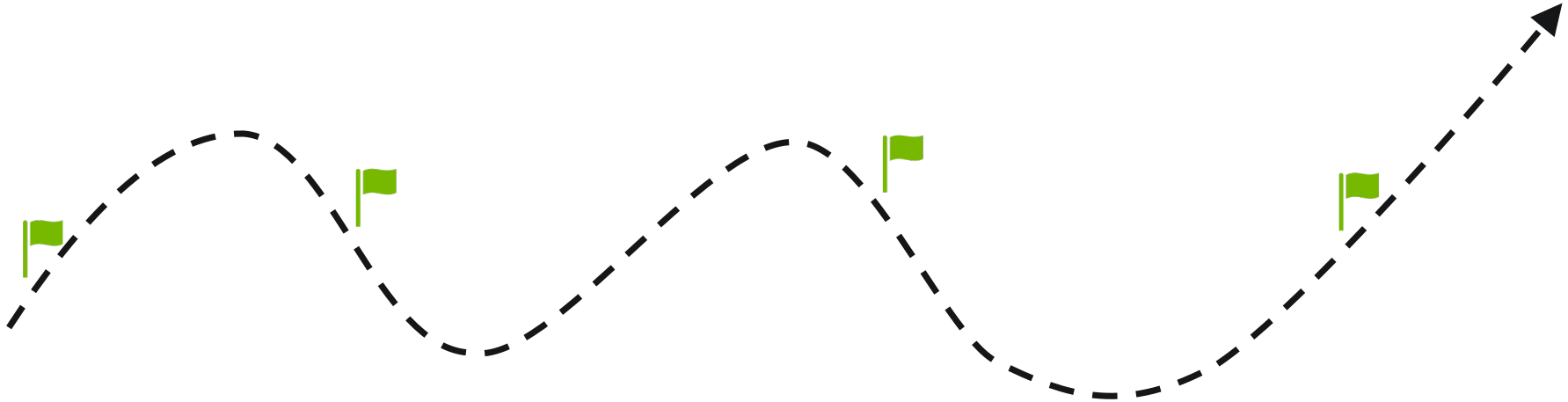
- Common causes for warp divergence and possible solutions.
  - Per thread work **is different**.
    - Possible solution: Queue and bin/sort the work.
  - Per thread work **is discovered at different times.**
    - Possible solution: Queue the work.
  - Per thread work **ends at different times.**
    - Possible solution: Split into multiple kernels.
  - Warning: When solving, avoid introducing too much overhead in remapping work items to threads!
- Implement conceptual divergence via varying data, instead of varying control flow.
- Consider algorithmic / higher order changes.

Work Queueing in shared memory

Imagine this problem: You're looking for buried treasure on the beach.

You have a metal detector and an excavating machine, but it takes a significant amount of time to switch between using the two tools

Instead of immediately switching to digging after detecting metal, mark the beach whenever you find something and only switch to excavating once there is enough locations to amortize the cost of switching

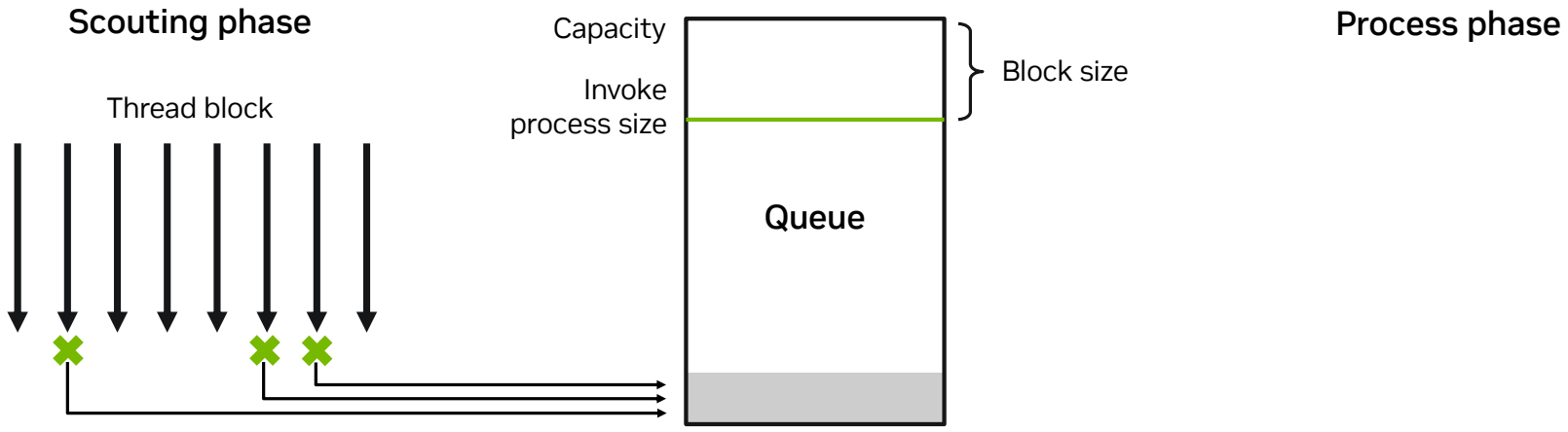


Work Queueing in shared memory

There are workloads like this where an expensive computational calculation has a lightweight check to guard against it. A naïve implementation may suffer from high divergence as not all threads will have work that passes the check.

Solution:

- When a thread finds a place to deep dive, add it to the queue, and move on.
- Occasionally all threads work simultaneously to clear the queue.
- Note: Threads that are finished scouting will stick around to help clear the queue.

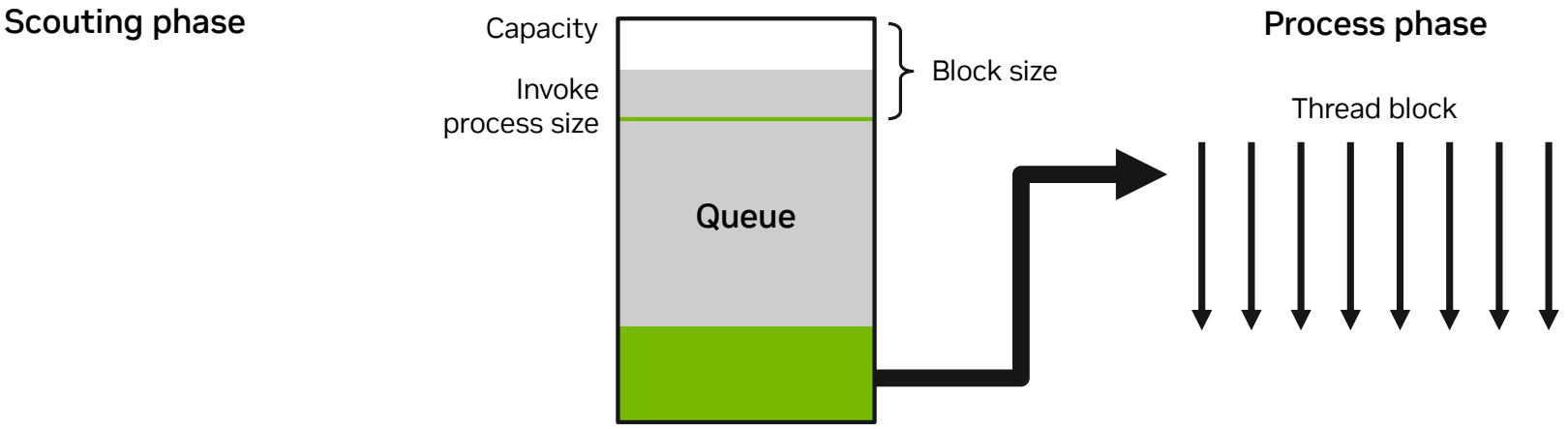


Work Queueing in shared memory

There are workloads like this where an expensive computational calculation has a lightweight check to guard against it. A naïve implementation may suffer from high divergence as not all threads will have work that passes the check.

Solution:

- When a thread finds a place to deep dive, add it to the queue, and move on.
- Occasionally all threads work simultaneously to clear the queue.
- Note: Threads that are finished scouting will stick around to help clear the queue.



Work Queueing in shared memory

At the start of every scouting iteration: As needed, process the queue to ensure every thread is able to insert if it finds something.

```
3   constexpr int k_block_size = 64; // keep sync overhead low.
4
5   constexpr int k_capacity_factor = 3; // somewhat arbitrary
6   constexpr int k_queue_capacity = k_block_size * k_capacity_factor;
7
8   // Process the queue whenever it breaches this size.
9   // Hence always leave enough room for every thread to possibly insert.
10  constexpr int k_queue_process_size = k_block_size * (k_capacity_factor - 1);
11
12  using Work_item_t = int;
13
14  // REQUIRES: 1D blocks. All threads to call, as it calls syncthreads.
15  // EFFECTS: threads will be synchronized at exit.
16  __device__
17  void process_queue(Work_item_t (&block_queue) [k_queue_capacity], int &block_queue_size){
18      // assert: block_queue_size ⩽ k_queue_capacity
19
20      for(int queue_idx = threadIdx.x; queue_idx < block_queue_size; queue_idx += blockDim.x){
21          const Work_item_t work_item = block_queue[queue_idx];
22          // perform deep dive on work item...
23          // .........
24      }
25      __syncthreads();
26
27      if(threadIdx.x == 0){
28          block_queue_size = 0;
29      }
30      __syncthreads();
31  }
```

Work Queueing in shared memory – A simple flavor

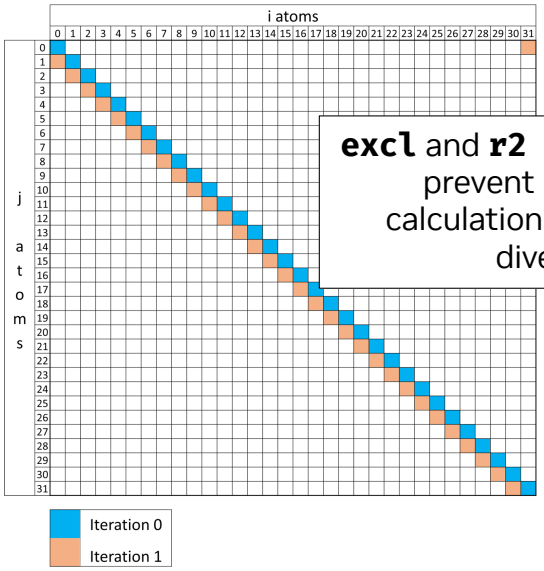
```
__global__ void scout_and_dive_kernel(){
    __shared__ int block_num_threads_finished_scouting;
    __shared__ Work_item_t block_queue[k_queue_capacity];
    __shared__ int block_queue_size;

    // elided initialization code ...
    const auto & is_thread_finished_scouting = []() { return false; /*your condition here*/};

    while(block_num_threads_finished_scouting < k_block_size){
        // invariant: all threads are here. Block is sync'd

        // make sure there is enough room in the queue for everyone to insert.
        if(block_queue_size ⩾ k_queue_process_size){
            process_queue(block_queue, block_queue_size);
        }

        if(!is_thread_finished_scouting()){
            // perform scouting work ...

            const bool found_dive = false; // your condition.
            if(found_dive){
                const Work_item_t work_idx = 0; // your work id mechanism.
                const auto queue_write_dst = atomicAdd(&block_queue_size, 1);
                // assert: queue_write_dst < k_queue_capacity
                block_queue[queue_write_dst] = work_idx;
            }

            // advance to next piece of scouting work
            if(!is_thread_finished_scouting()){
                // advance to next scouting location
            }
            else {
                atomicAdd(&block_num_threads_finished_scouting, 1);
            }
        }
        __syncthreads(); // see block_num_threads_finished_scouting.
    } // while any thread is scouting.

    // flush queue at end ...
}
```

Some initialization code has been elided for brevity.

Full source available on Accelerated Computing Hub.

Spatial sorting in molecular dynamics to reduce divergence

- NAMD computes pairwise interactions between atoms within a cutoff distance using 32x32 tiles to create SIMT friendly work units and increase arithmetic intensity
- Cutoff radius introduces sparsity within tiles which introduces warp divergence



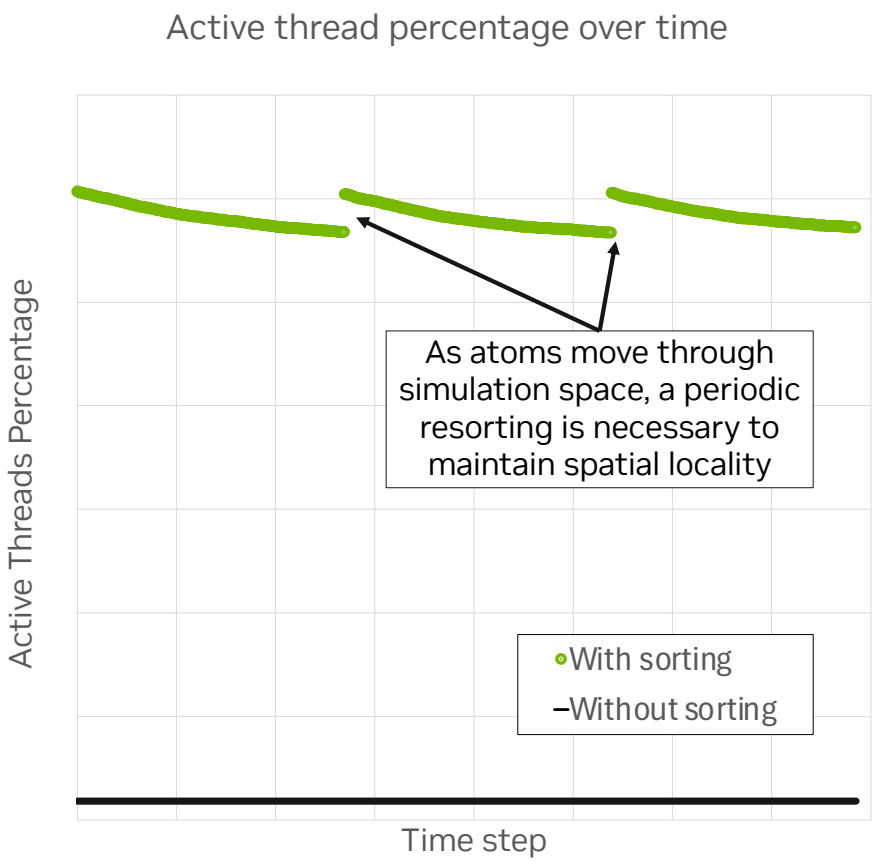
`excl` and **`r2 < cutoff`** checks prevent unnecessary calculations but introduce divergence

#	Source	Avg. Threads Executed	Avg. Predicated-On Threads Executed
253	`#pragma unroll 4`		
254	`for (int t = t_start; t < WARPSIZE; t++) {`	28.7	26.8
255	`if ((excl & 1)) {`	32	27.1
256	`const int j = ((t + wid) & (WARPSIZE-1));`	13.3	13.3
257	`xyzq_j = s_xyzq[iwarp][j];`	13.3	13.3
258	`float dx = xyzq_j.x - xyzq_i.x;`	13.3	13.3
259	`float dy = xyzq_j.y - xyzq_i.y;`	13.3	13.3
260	`float dz = xyzq_j.z - xyzq_i.z;`	13.3	13.3
261			
262	`float r2 = dx*dx + dy*dy + dz*dz;`	13.3	13.3
263	`if (r2 < cutoff2) {`	13.3	7.5
264	`float rinv = rsqrtf(r2);`	12.7	12.7
265	`float f, fSlow;`		
266			
267	`fSlow = xyzq_i.w * xyzq_j.w;`	12.7	12.7
268			
269	`const float rinv2 = rinv * rinv;`	12.7	12.7
270	`const float rinv3 = rinv * rinv2;`	12.7	12.7
271	`const float rinv6 = rinv3 * rinv3;`	12.7	12.7
272	`const float rinv8 = rinv6 * rinv2;`	12.7	12.7
273			

NAMD was developed by the Theoretical and Computational Biophysics Group in the Beckman Institute for Advanced Science and Technology at the University of Illinois at Urbana-Champaign. http://www.ks.uiuc.edu/Research/namd/

Spatial sorting in molecular dynamics to reduce divergence

- It is important for atoms close together in memory to be close together in simulation space
- NAMD uses recursive bisection scheme to sort atoms within spatial decompositions, other molecular dynamic codes use space filling curves
- Spatial sorting increases number of active threads per warp by over 50%



Optimization Tip

Pre-processing steps can reduce warp divergence caused by data dependencies

Reducing warp divergence:
Implement Conceptual divergence via dataflow

The following code which assigns a value to x will result in divergence or predication

```
227    float x = 0.0f;
228    if (isA) {
229      x = valA;
230    } else if (isB) {
231      x = valB;
232    }
```

The following code will have the same effect but avoid divergence by treating the Booleans as scale factors.

```
235    float x = (isA) * valA +
236              (isB) * valB;
237
```

Reducing warp divergence: Implement Conceptual divergence via dataflow

Greatly obfuscates code but eliminates divergence. Again, only go here if profiler demands it.

```
3   if (numInPoints == 1 || numInPoints == 3) {
4
5       if ((numInPoints == 1 && leftBotIn)
6           || (numInPoints == 3 && !leftBotIn)) {
7           edgePt1X -= pixelMinX;
8           edgePt2X -= pixelMinX;
9           edgePt1Y -= pixelMinY;
10          edgePt2Y -= pixelMinY;
11      }
12      else if ((numInPoints == 1 && leftTopIn)
13          || (numInPoints == 3 && !leftTopIn)) {
14          edgePt1X -= pixelMinX;
15          edgePt2X -= pixelMinX;
16          edgePt1Y -= pixelMaxY;
17          edgePt2Y -= pixelMaxY;
18      }
19      else if ((numInPoints == 1 && rightBotIn)
20          || (numInPoints == 3 && !rightBotIn)) {
21          edgePt1X -= pixelMaxX;
22          edgePt2X -= pixelMaxX;
23          edgePt1Y -= pixelMinY;
24          edgePt2Y -= pixelMinY;
25      }
26      else if ((numInPoints == 1 && rightTopIn)
27          || (numInPoints == 3 && !rightTopIn)) {
28          edgePt1X -= pixelMaxX;
29          edgePt2X -= pixelMaxX;
30          edgePt1Y -= pixelMaxY;
31          edgePt2Y -= pixelMaxY;
32      }
33  }
34
```

```
33  if (numInPoints == 1 || numInPoints == 3) {
34
35      if(numInPoints == 3){
36          leftBotIn = !leftBotIn;
37          leftTopIn = !leftTopIn;
38          rightBotIn = !rightBotIn;
39          rightTopIn = !rightTopIn;
40      }
41
42      // Exactly one of <someCorner>In is true! Predicate math.
43      // <someCorner>In variables will be implicitly cast from
44      // bool to int all over the place here.
45
46      // Will be pixelMinX, pixelMaxX, or 0
47      const auto edgePt1X_subtrahend =
48          (leftBotIn + leftTopIn)  * pixelMinX
49        + (rightBotIn + rightTopIn) * pixelMaxX;
50
51      const auto edgePt1Y_subtrahend =
52          (leftBotIn + rightBotIn) * pixelMinY
53        + (leftTopIn  + rightTopIn) * pixelMaxY;
54
55      const auto edgePt2X_subtrahend = edgePt1X_subtrahend;
56      const auto edgePt2Y_subtrahend = edgePt1Y_subtrahend;
57
58      edgePt1X -= edgePt1X_subtrahend;
59      edgePt2X -= edgePt2X_subtrahend;
60      edgePt1Y -= edgePt1Y_subtrahend;
61      edgePt2Y -= edgePt2Y_subtrahend;
62  }
63
```

Bonus: Dynamic block dimensions

NOT thread divergence. More like ~"block divergence"

For every polygon in collection

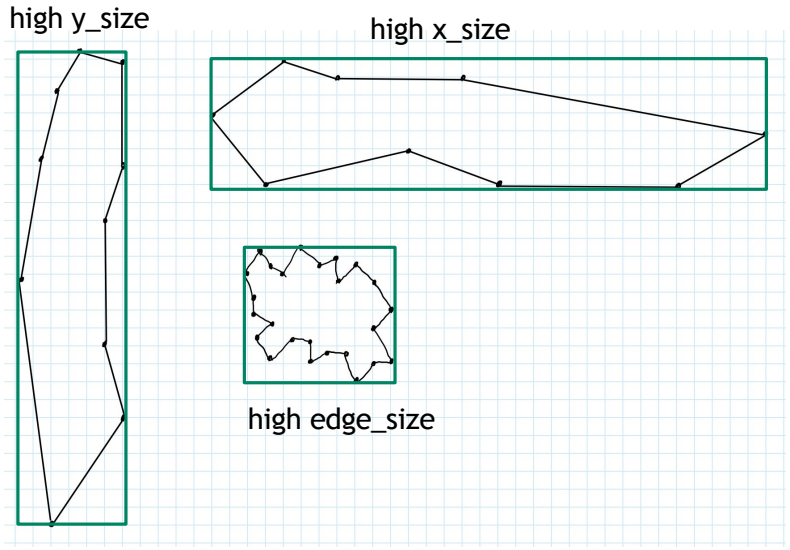
For every pixel under polygon

For every edge in polygon

<Do something>

Parallelism scheme: One block per polygon. 3D blocks: (pixel_x, pixel_y, poly_edge)

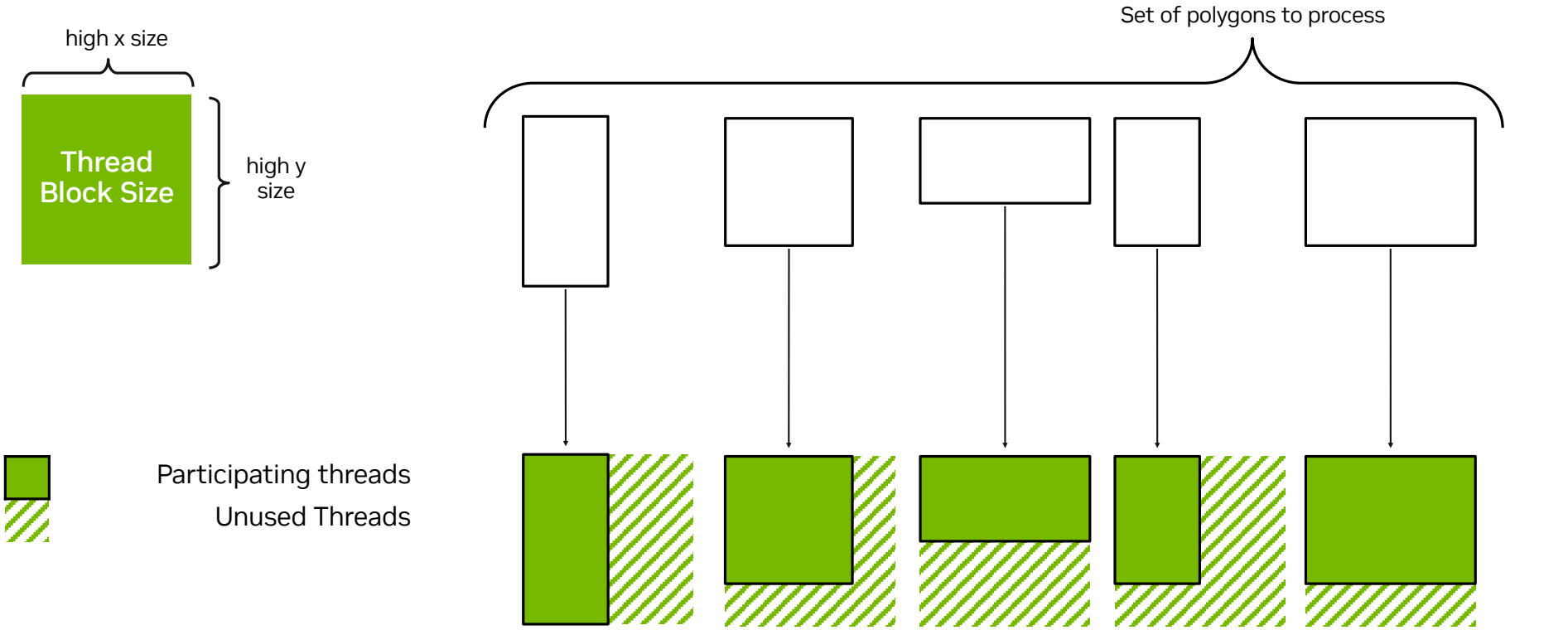
```
3   // gridDim is 1D. One block per polygon.
4   // blockDim is 3D. Sized arbitrarily.
5   const auto polygon_idx = blockIdx.x;
6
7   // fetch characteristics of polygon.
8   const auto x_size = ...;
9   const auto y_size = ...;
10  const auto edges_size = ...;
11
12  for (int x_idx = threadIdx.x; x_idx < x_size; x_idx += blockDim.x) {
13      for (int y_idx = threadIdx.y; y_idx < y_size; y_idx += blockDim.y) {
14          for (int edge_idx = threadIdx.z; edge_idx < edges_size; edge_idx += blockDim.z) {
15              // work...
16          }
17      }
18  }
```



Bonus: Dynamic block dimensions

NOT thread divergence. More like ~"block divergence"

Problem: Threads are used inefficiently when polygon characteristics vary widely.



Bonus: Dynamic block dimensions

NOT thread divergence. More like ~"block divergence"

```
24  // Dynamic block size:
25  // More efficient use of threads when polygon characteristics vary widely.
26
27  // gridDim is 1D. One block per polygon.
28  // blockDim is 1D. Size is a power of 2.
29  const auto polygon_idx = blockIdx.x;
30
31  // fetch characteristics of polygon.
32  const auto x_size = ...;
33  const auto y_size = ...;
34  const auto edges_size = ...;
35
36  // Determine virtual blockDim. Assume concrete block size is 1D and a power of 2.
37  dim3 vBlockDim(1, 1, 1);
38  const auto threads_allocated = 1;
39
40  // first fill X
41  while(vBlockDim.x < x_size && threads_allocated < blockDim.x) {
42      vBlockDim.x *= 2;
43      threads_allocated *= 2;
44  }
45  // then fill Y
46  while(vBlockDim.y < y_size && threads_allocated < blockDim.x) {
47      vBlockDim.y *= 2;
48      threads_allocated *= 2;
49  }
50  // then fill E
51  while(vBlockDim.z < edges_size && threads_allocated < blockDim.x) {
52      vBlockDim.z *= 2;
53      threads_allocated *= 2;
54  }
```

```
55  // Determine virtual threadIdx
56  dim3 vThreadIdx;
57  vThreadIdx.x = threadIdx.x % vBlockDim.x;
58  vThreadIdx.y = (threadIdx.x / vBlockDim.x) % vBlockDim.y;
59  vThreadIdx.z = (threadIdx.x / (vBlockDim.x * vBlockDim.y));
60
61  // work
62  for (int x_idx = vThreadIdx.x; x_idx < x_size; x_idx += vBlockDim.x) {
63      for (int y_idx = vThreadIdx.y; y_idx < y_size; y_idx += vBlockDim.y) {
64          for (int edge_idx = vThreadIdx.z; edge_idx < edges_size; edge_idx += vBlockDim.z) {
65              // work ...
66          }
67      }
68  }
```

Now, each block dynamically distributes its threads among the loops in a useful manner.

Warp scheduling and Kernel profiling at a glance

Building a mental model for instruction issue and stalling

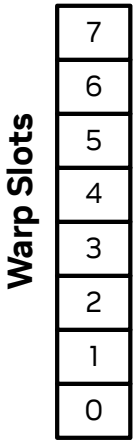
Warp Scheduling

Hopper SM

- 4 warp schedulers per SM.
- Each scheduler manages a pool of warps.
  - Hopper: 16 warp slots per scheduler.
- Each clock cycle, each scheduler can issue an instruction for 1 warp.

WARP SCHEDULER STATISTICS

Mental Model for Profiling

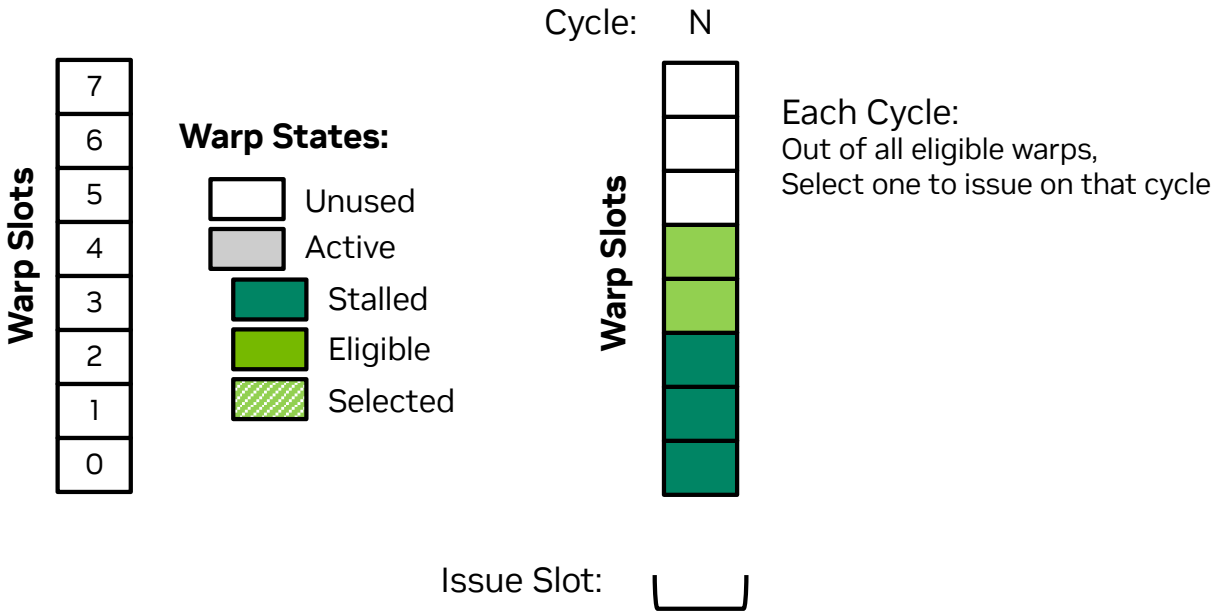


Warp States:

- Unused
- Active – Warp is resident on processor.
- Stalled- Warp is waiting for previous instructions to finish; for input data of next instruction to be produced.
- Eligible- All data/etc the warp needs to execute the next instruction is ready.
- Selected – Eligible & selected by scheduler to issue instruction this cycle.

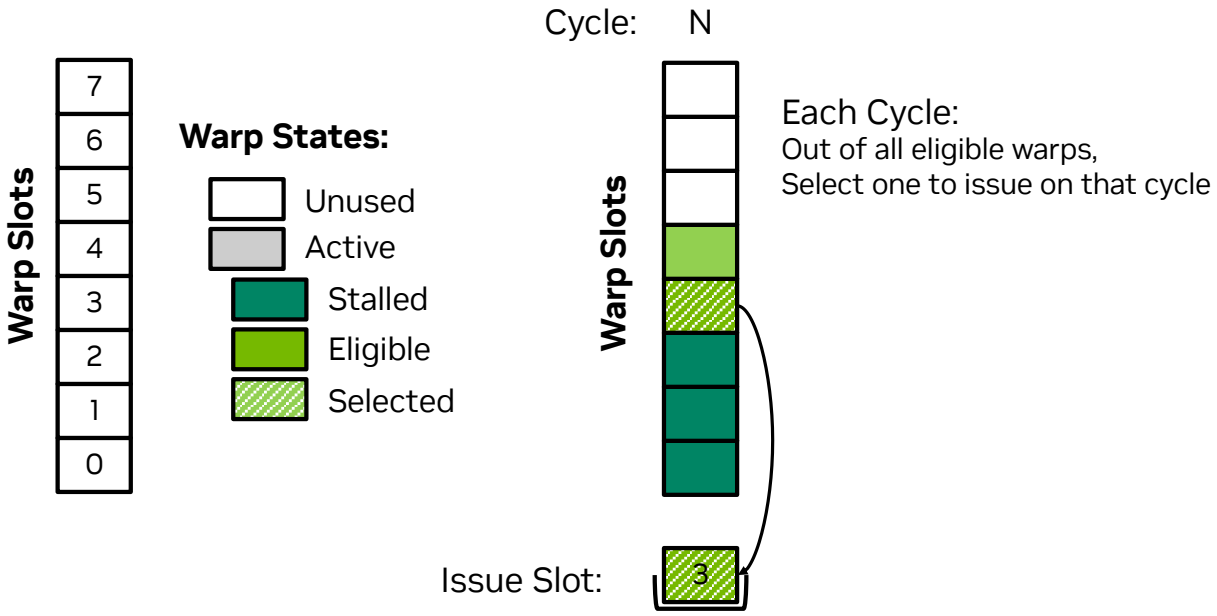
WARP SCHEDULER STATISTICS

Mental Model for Profiling



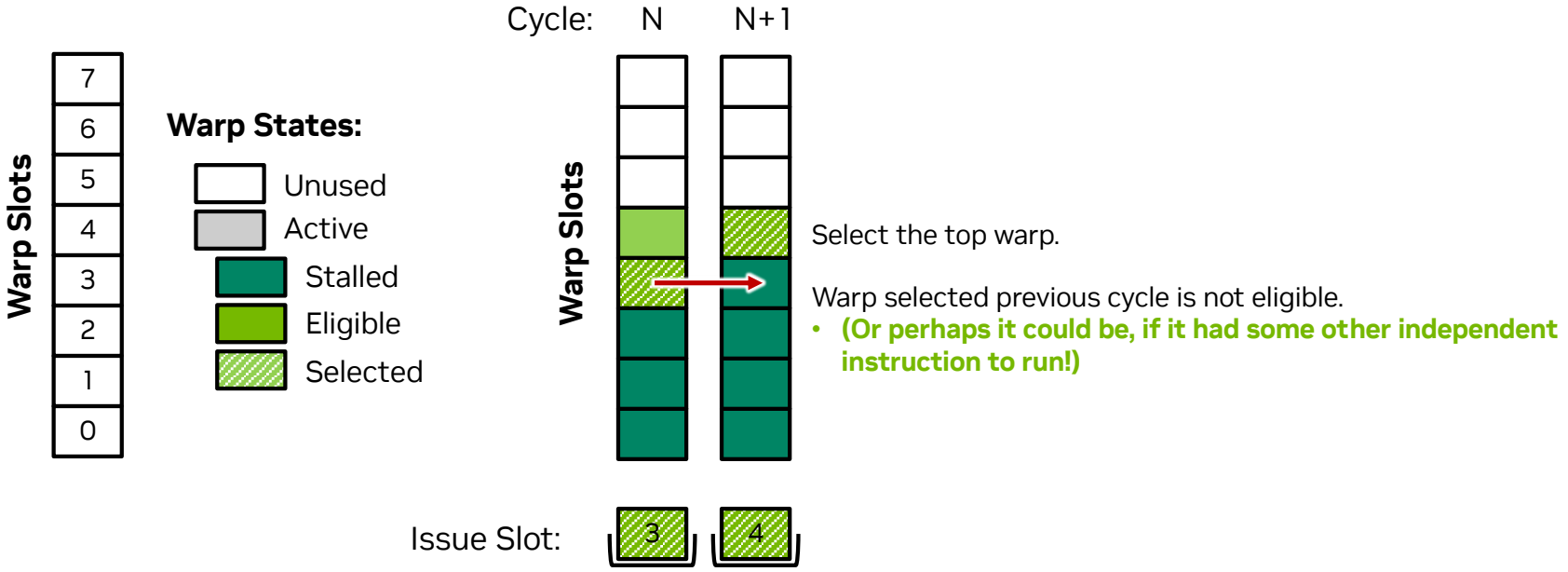
WARP SCHEDULER STATISTICS

Mental Model for Profiling



WARP SCHEDULER STATISTICS

Mental Model for Profiling



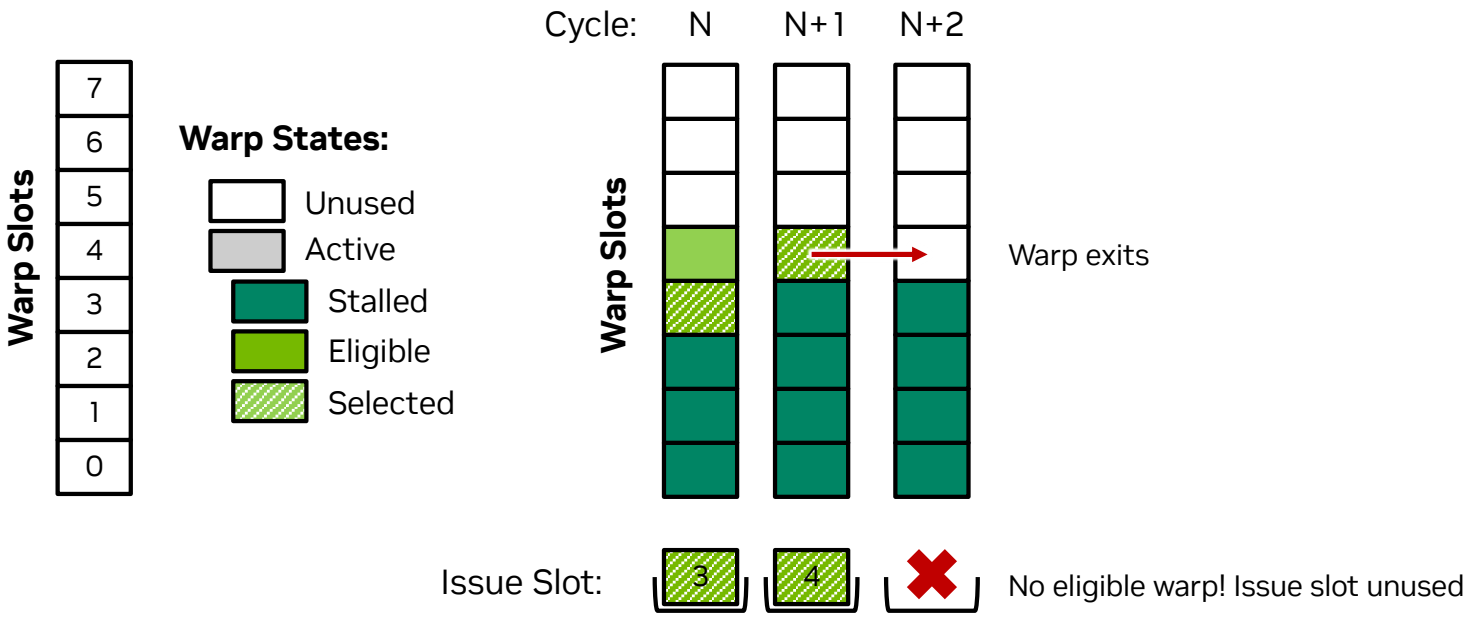
Select the top warp.

Warp selected previous cycle is not eligible.

- **(Or perhaps it could be, if it had some other independent instruction to run!)**

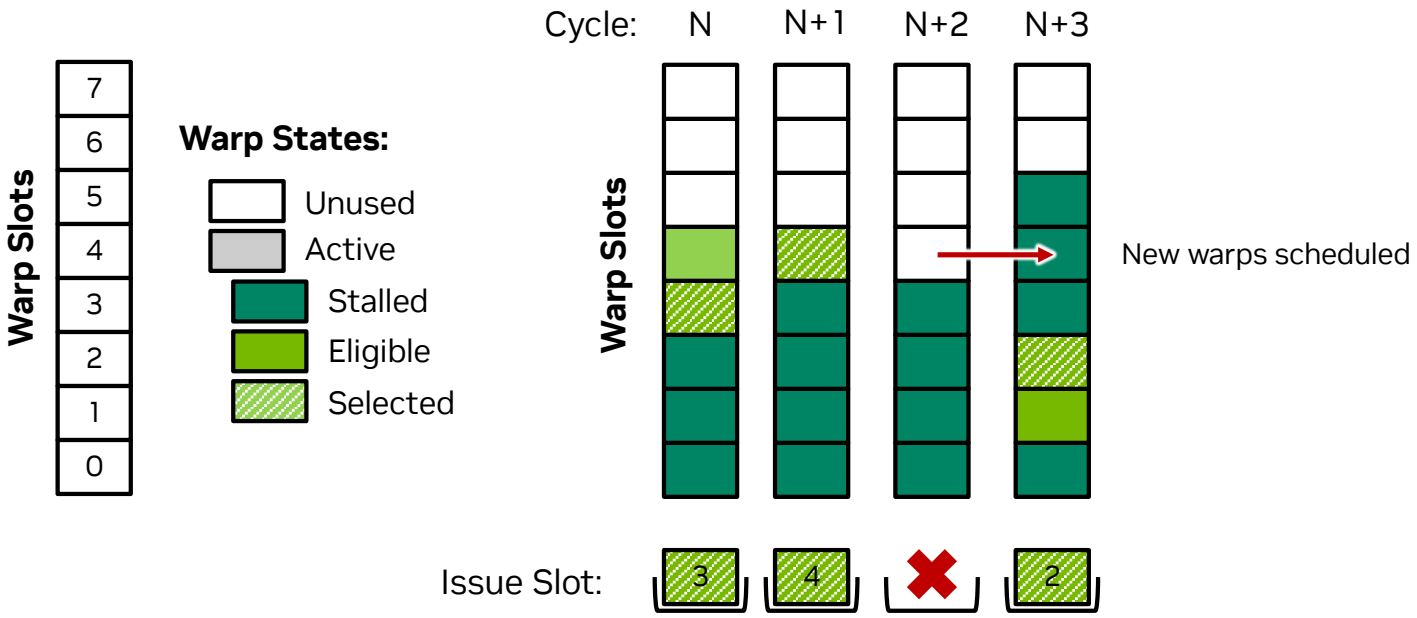
WARP SCHEDULER STATISTICS

Mental Model for Profiling



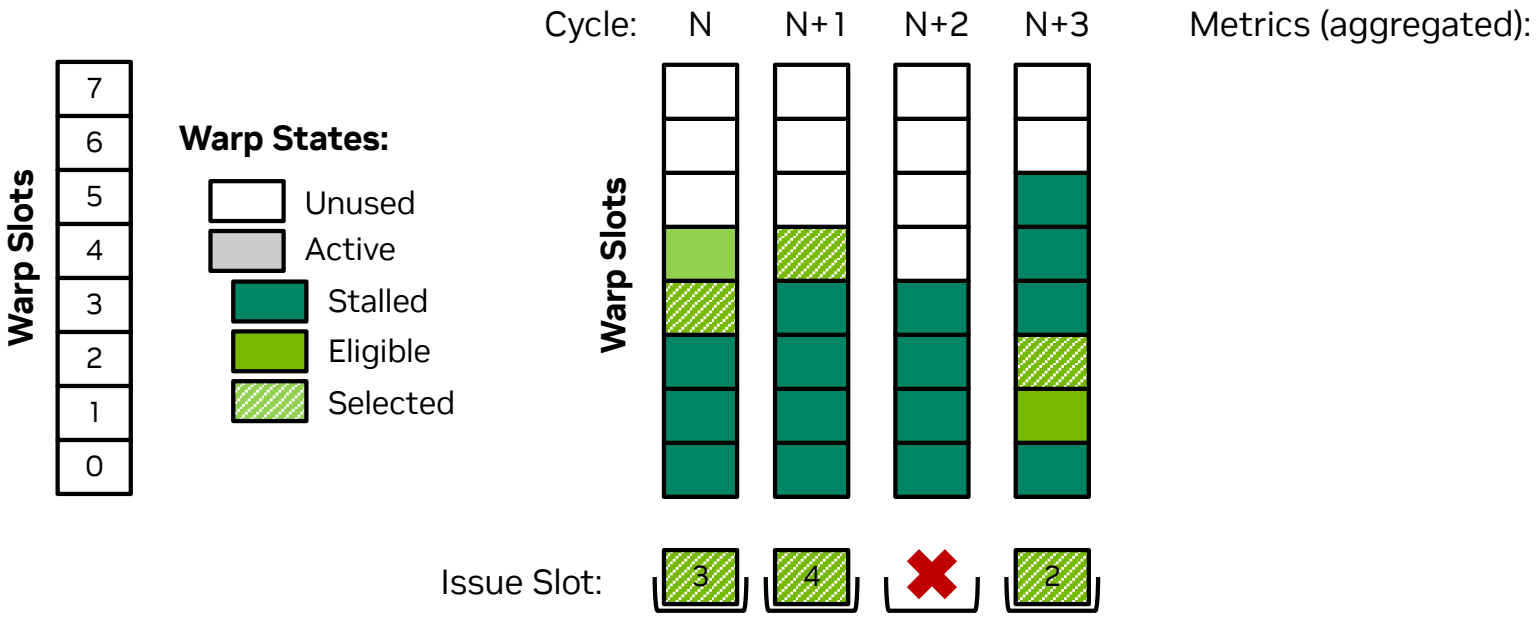
WARP SCHEDULER STATISTICS

Mental Model for Profiling



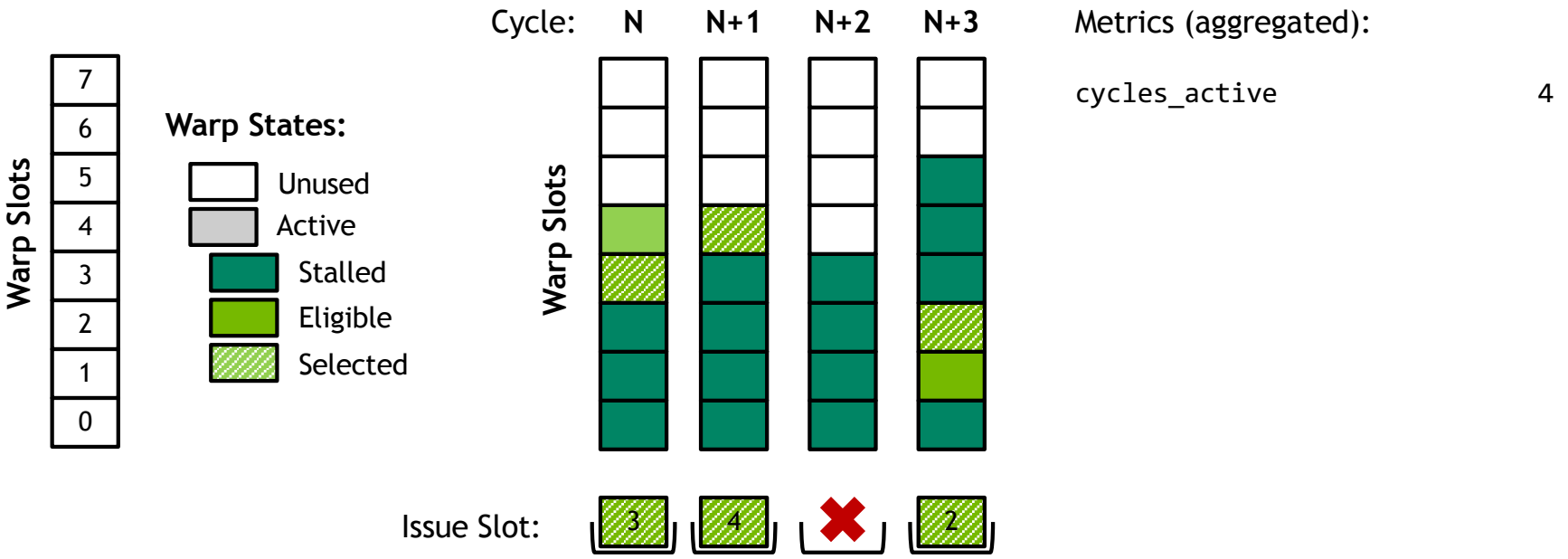
WARP SCHEDULER STATISTICS

Mental Model for Profiling



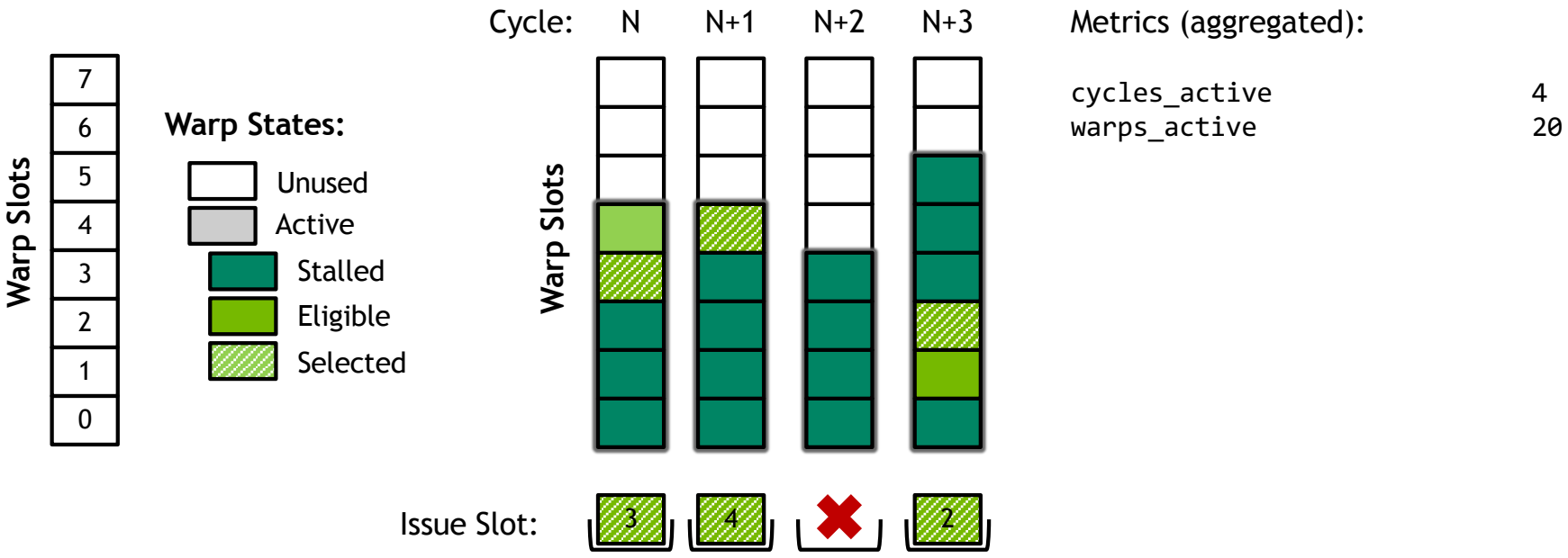
WARP SCHEDULER STATISTICS

Mental Model for Profiling



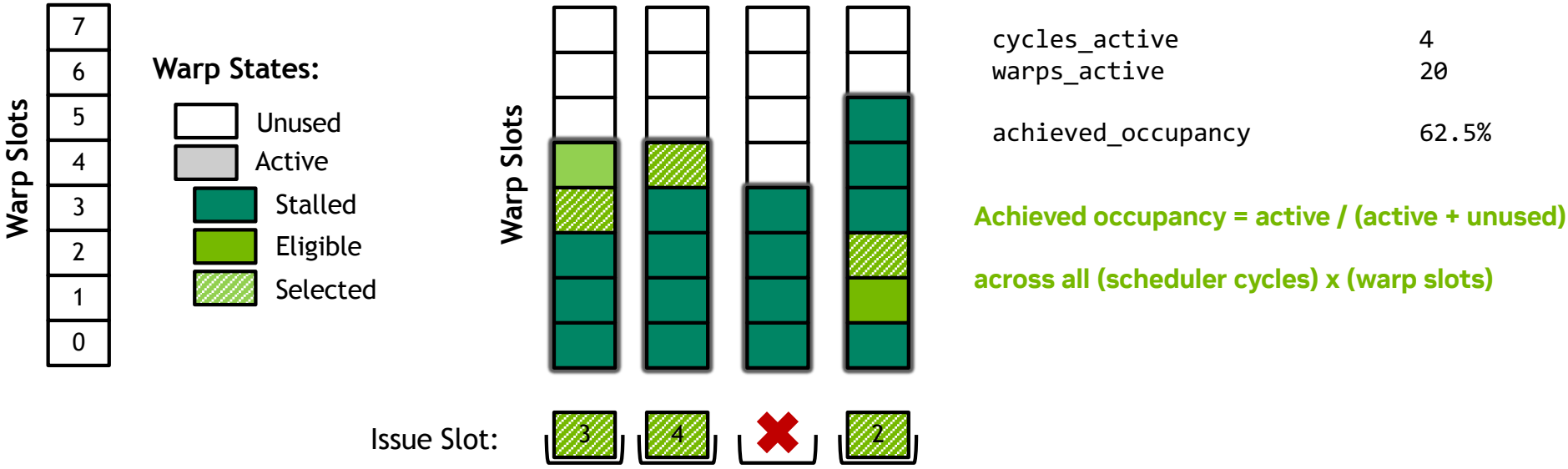
WARP SCHEDULER STATISTICS

Mental Model for Profiling



WARP SCHEDULER STATISTICS

Mental Model for Profiling



Metrics (aggregated):

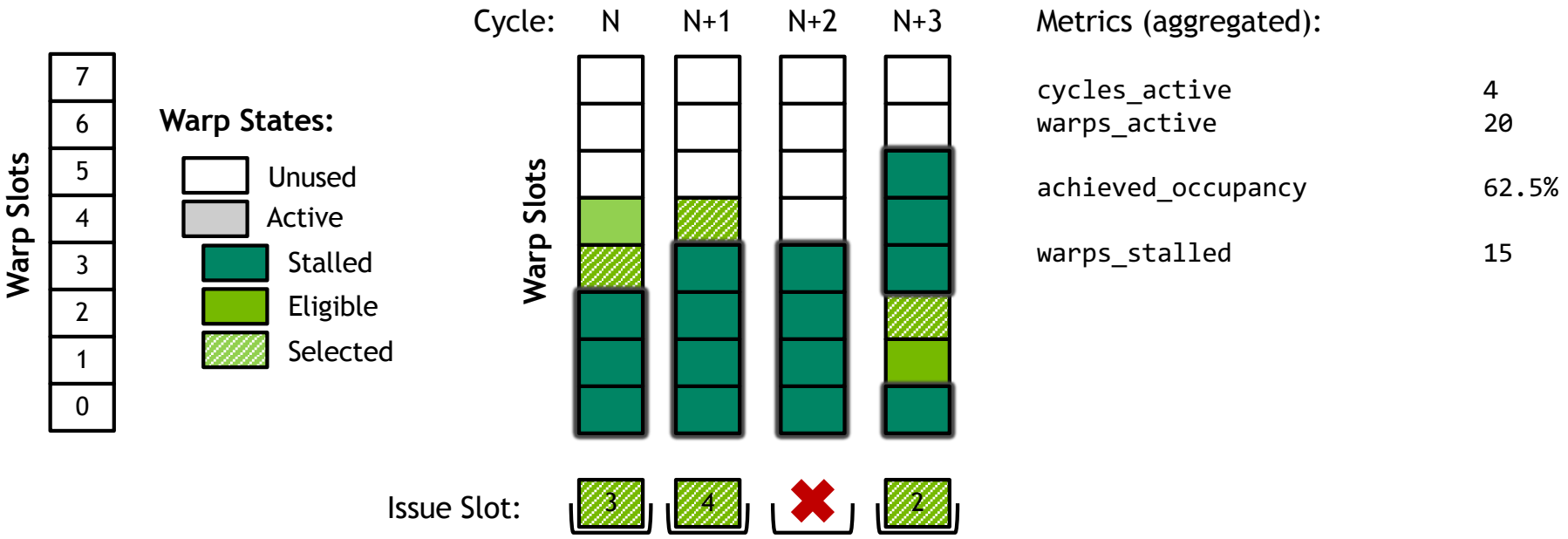
`cycles_active`	4
`warps_active`	20
`achieved_occupancy`	62.5%

Achieved occupancy = active / (active + unused)

across all (scheduler cycles) x (warp slots)

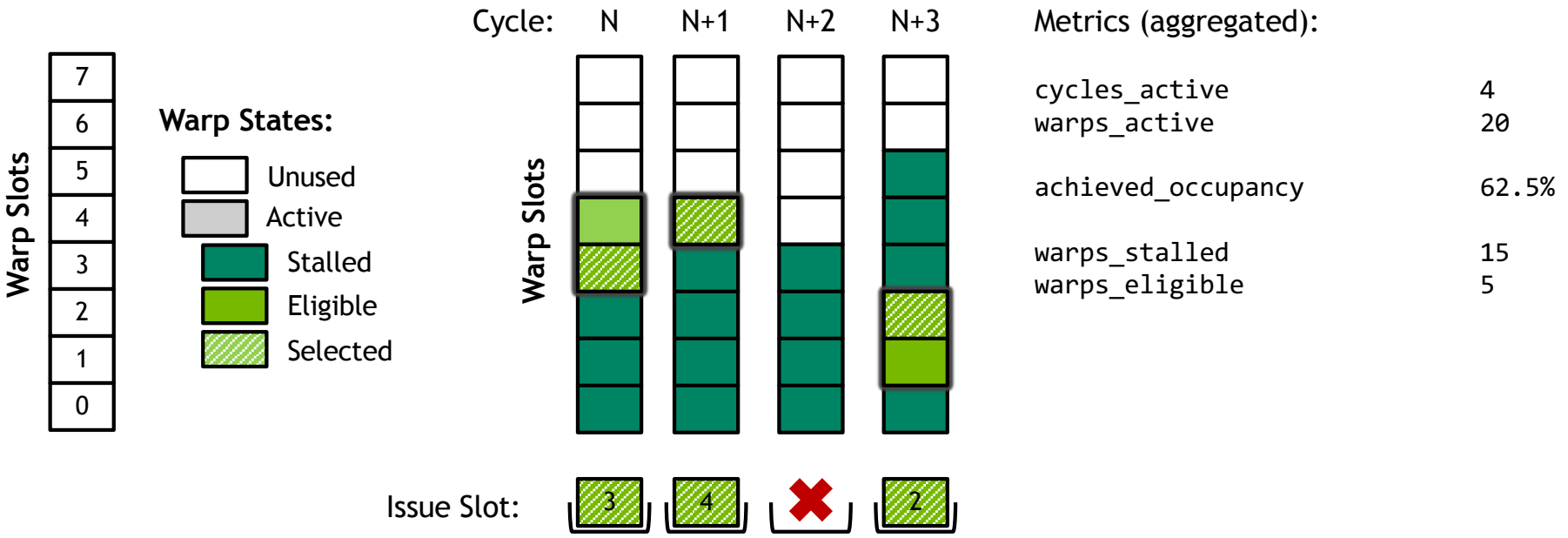
WARP SCHEDULER STATISTICS

Mental Model for Profiling



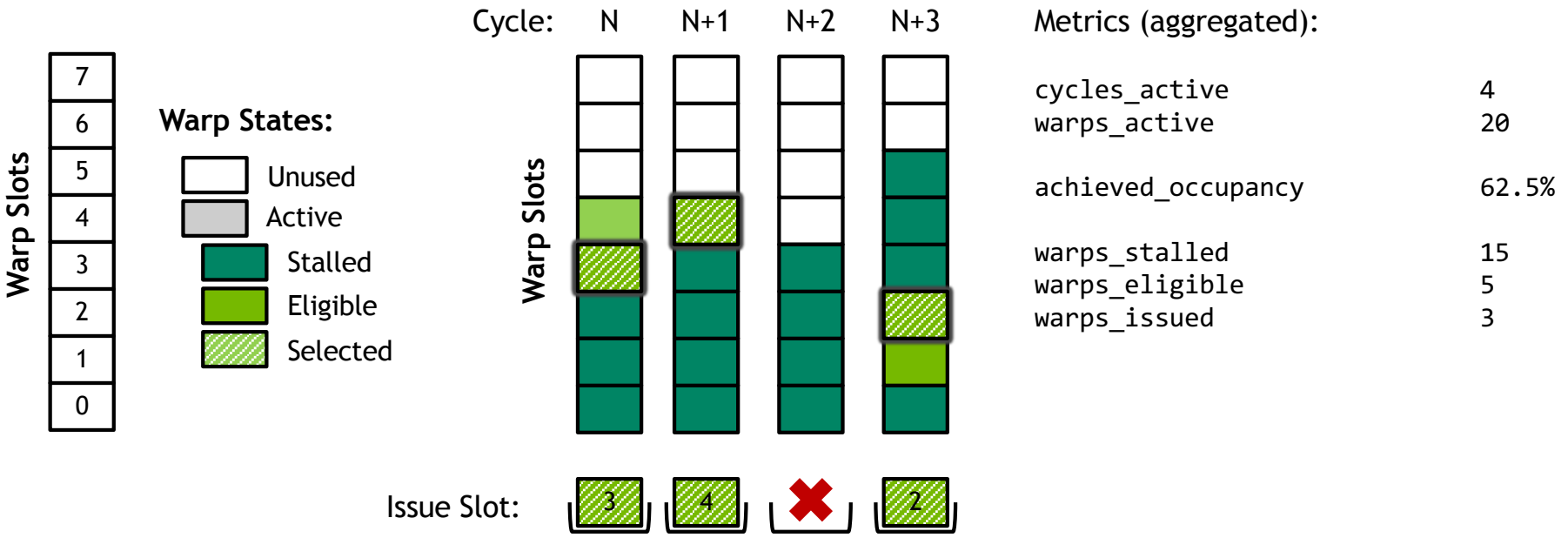
WARP SCHEDULER STATISTICS

Mental Model for Profiling



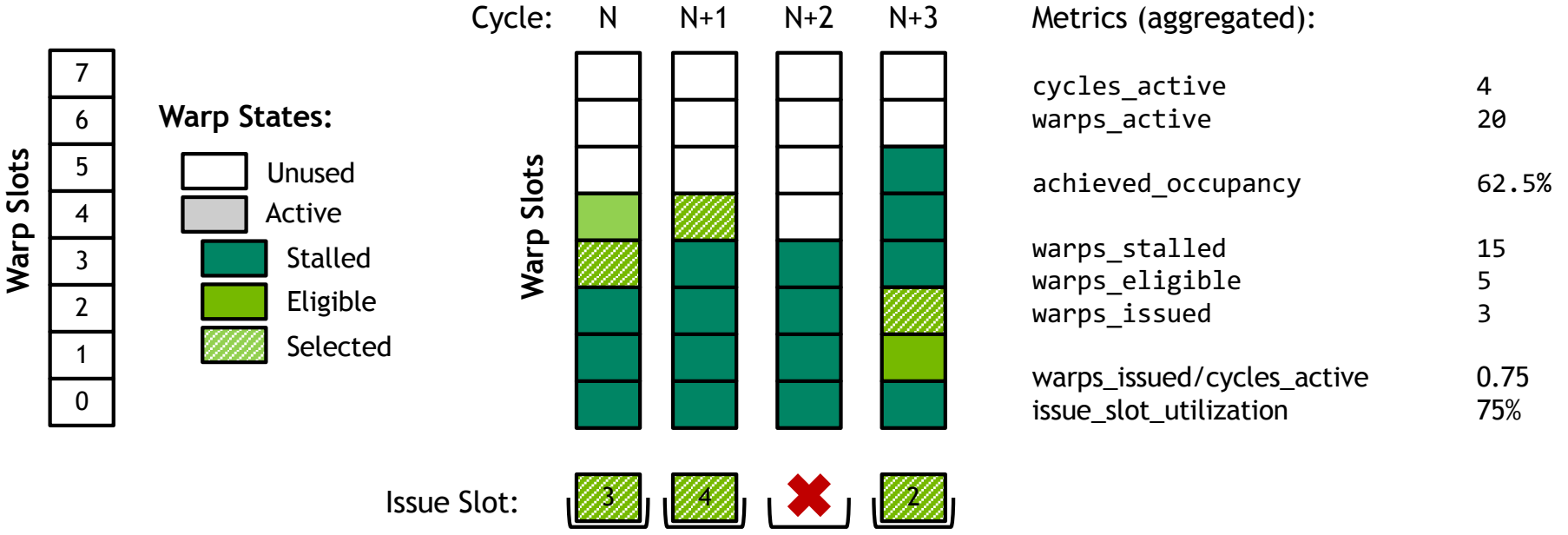
WARP SCHEDULER STATISTICS

Mental Model for Profiling



WARP SCHEDULER STATISTICS

Mental Model for Profiling



Metrics (aggregated):

cycles_active	4
warps_active	20
achieved_occupancy	62.5%
warps_stalled	15
warps_eligible	5
warps_issued	3
warps_issued/cycles_active	0.75
issue_slot_utilization	75%

Latency bound kernel

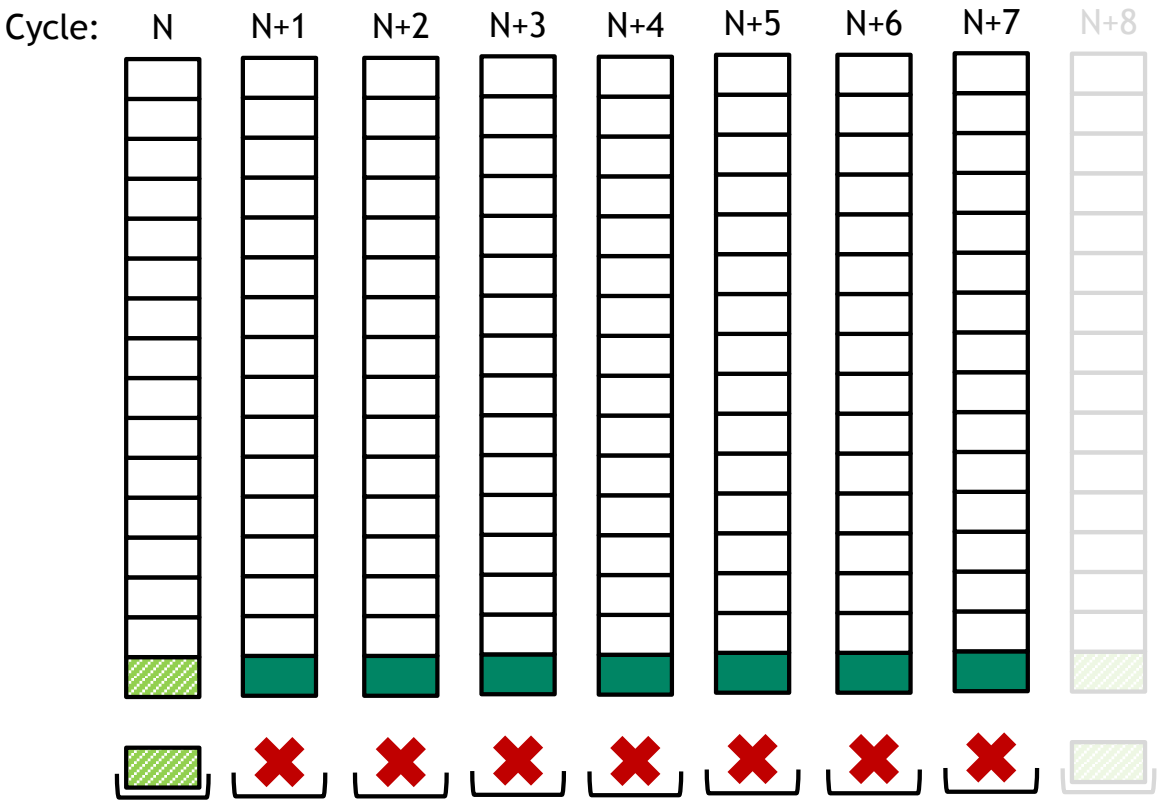
One active warp per scheduler

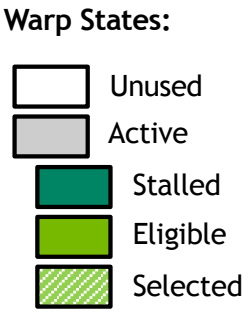
Each thread accumulates PI 1000 times using double-precision

```
__global__
void kernel_A(double* A, int K){

  double result = 0.0;
  for(int j = 0; j < K; ++j) {
    result += 3.14;
  }

  int i = blockIdx.x * blockDim.x + threadIdx.x;
  A[i] = result;
}

…
kernel_A<<<1, 32>>>(d_f64, 1000);
cudaDeviceSynchronize();
```

Latency bound kernel

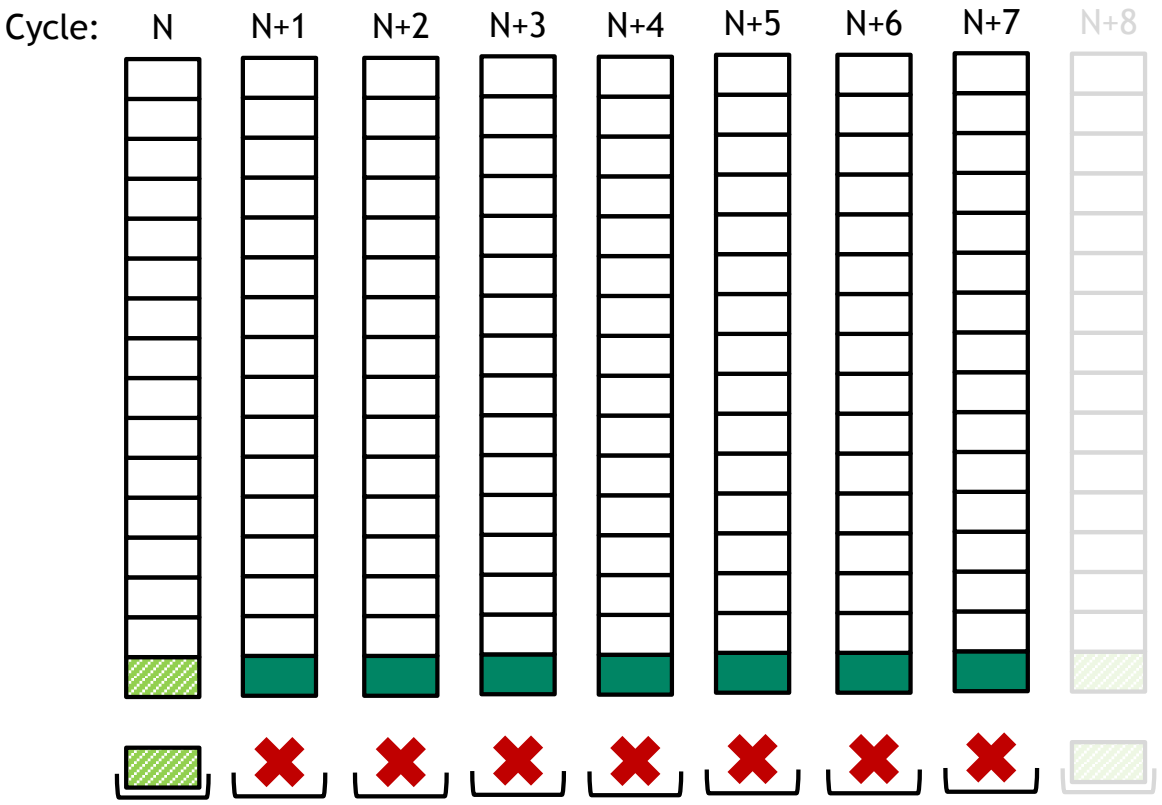
One active warp per scheduler

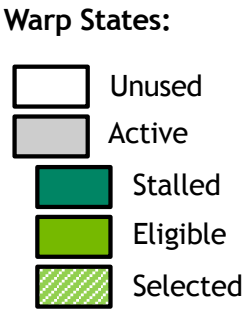




Latency bound kernel

One active warp per scheduler





Metrics (theoretical; every 8 cycles):

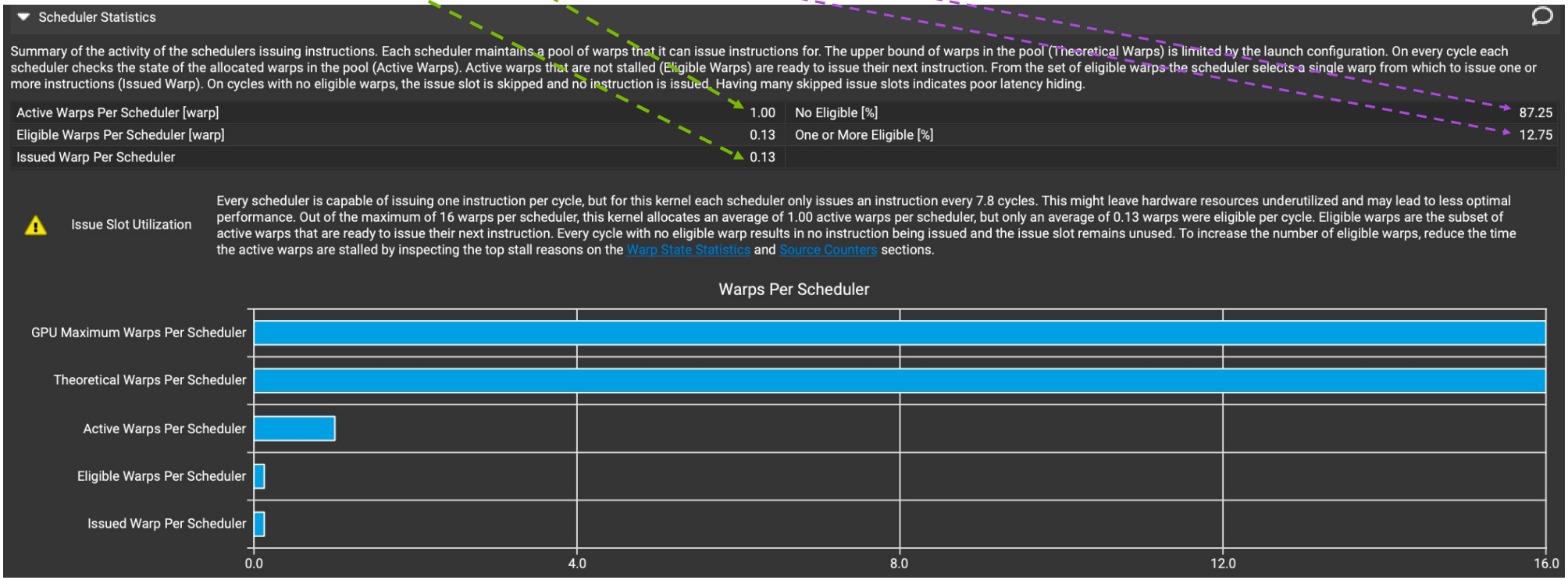
warps_active	8	1
warps_stalled	7	7/8 = 0.87
warps_eligible	1	1/8 = 0.125
warps_selected	1	1/8 = 0.125

Latency bound kernel: Scheduler statistics

One warp per scheduler

Metrics (from report; rounded):

```
warps_active                1.00
warps_stalled               0.87
warps_eligible              0.13
warps_selected              0.13
```



Note: If you looked at Warp State Statistics you'd see "Stall Wait"- waiting on fixed latency execution dependency. Launch more warps/blocks!

Bottom line: The two best things about GPUs.

1. SIMD is great.

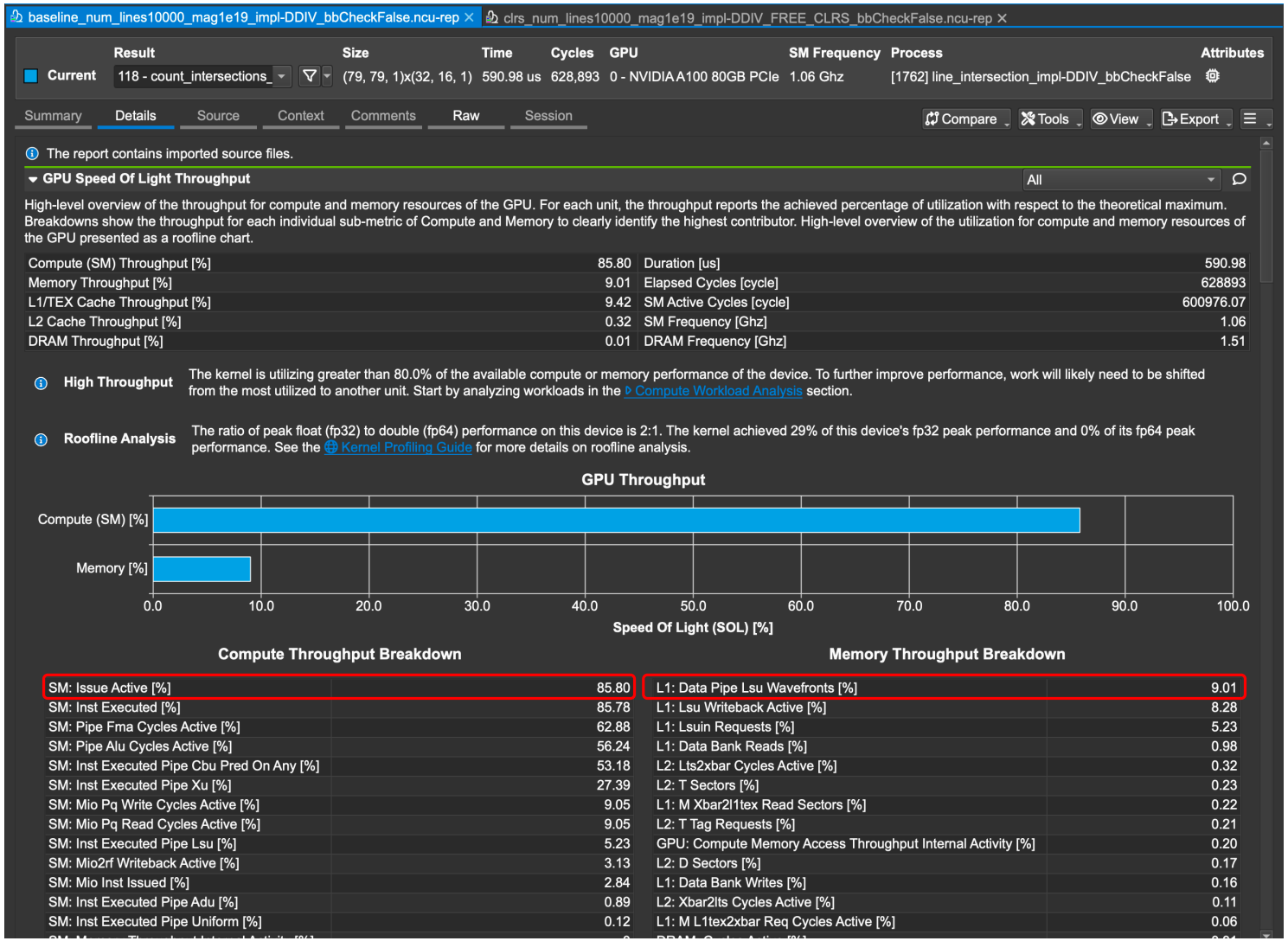
But wait, there's more!

2. Hyperthreading to the extreme.
   - Warps time-slice usage of all processor pipeline resources, to hide each other's latency and keep hardware resources busy.
   - "Context switch" between warps is "free" from SW perspective.
     - Context is always resident on processor.
     - Switch is implemented in hardware.

Kernel profiling at a glance

- Keep code as simple as possible, until the profiler tells you otherwise.
- Profiler should drive your decisions: No guessing or premature optimization development.
- **When are you done optimizing?**
  - Practically: Amdahl's law. When the code is no longer a bottleneck compared to the rest of the system.
  - **Theoretically: When all work performed is absolutely necessary, and the hardware units responsible for that work are fully utilized / occupied.**
    - At that point, look at different algorithms, or talk to HW / compiler teams.

Kernel profiling at a glance



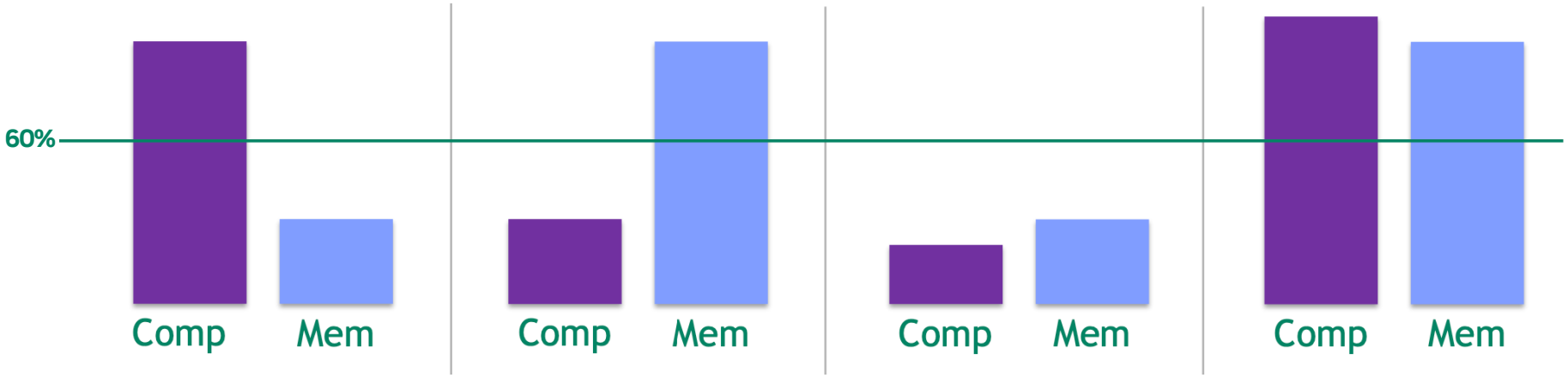
GPU Speed of Light Throughput:

For each the SM and Memory System:

- The resource with highest utilization rises to the top and is reported as the blue bar.

Kernel profiling at a glance

Performance Limiter Categories: Your CUDA kernel will be in 1 of 4 states



Compute Bound

Memory Bound

Latency Bound

Any sort of latency,
NOT just memory!

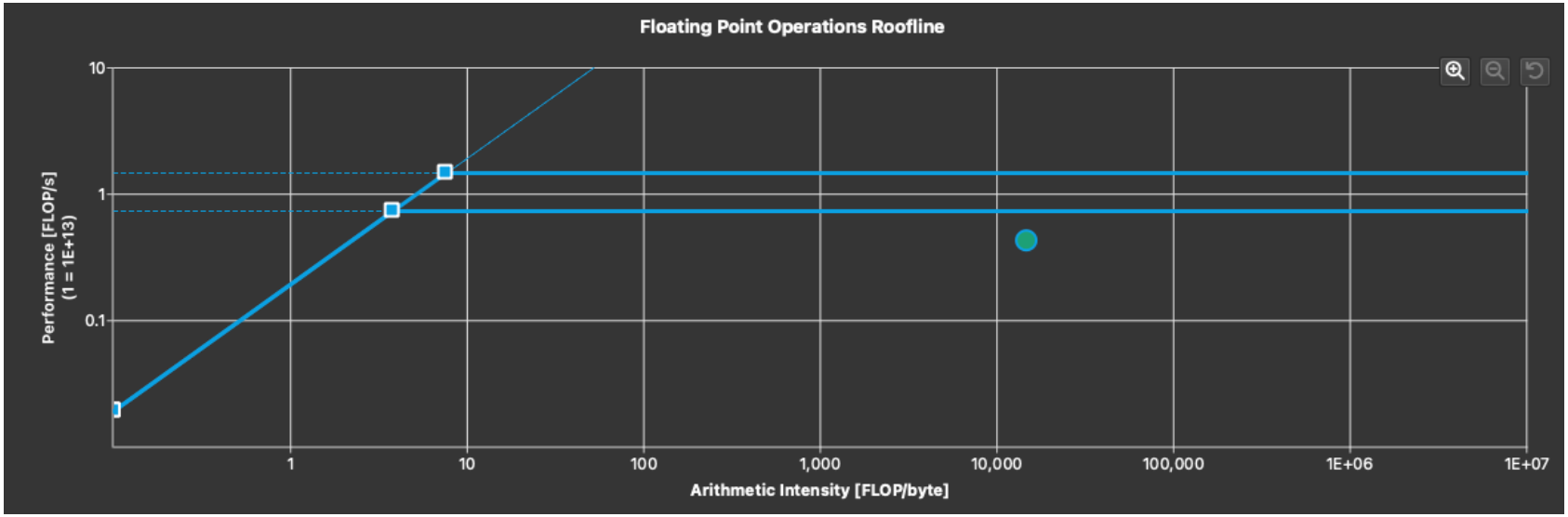
Comp & Mem Bound

Or perhaps, "divergence bound"

Kernel profiling at a glance

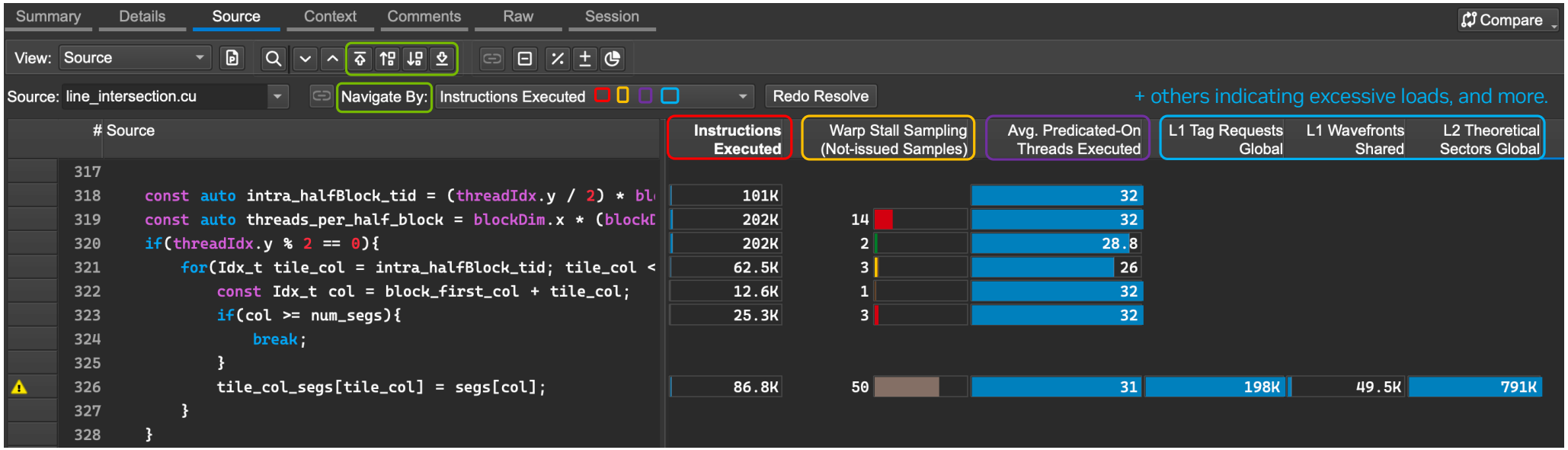
Roofline Model

- Characterizes the ratio of compute to memory operations.
- For awareness- there is plenty of great content out there on this type of analysis.



NCU Source Page: Given your bounding type, what column should you look at?

To show various columns in UI: Right click any column -> "Column chooser"



Navigate the source page. For each line of source code: Each column is a hardware counter / statistic. Can navigate by highest values.

Compute bound- Find the instructions that execute the most.

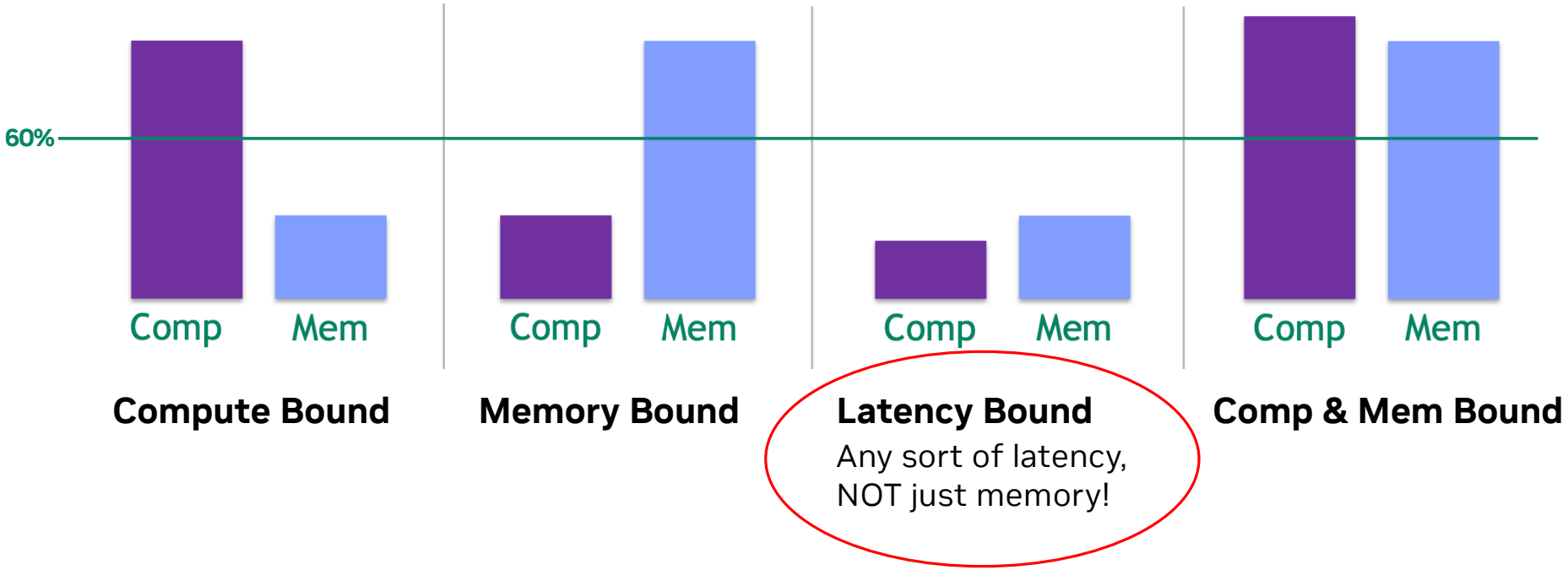
Latency bound- Find the instructions that most frequently cannot be issued yet - we are waiting for previous instructions to complete.

Memory bound- Find the instructions putting the most load on the memory system. (Which column depends on which subsystem is loaded)

Divergence bound – Beware how many threads are converged at your hot spot.

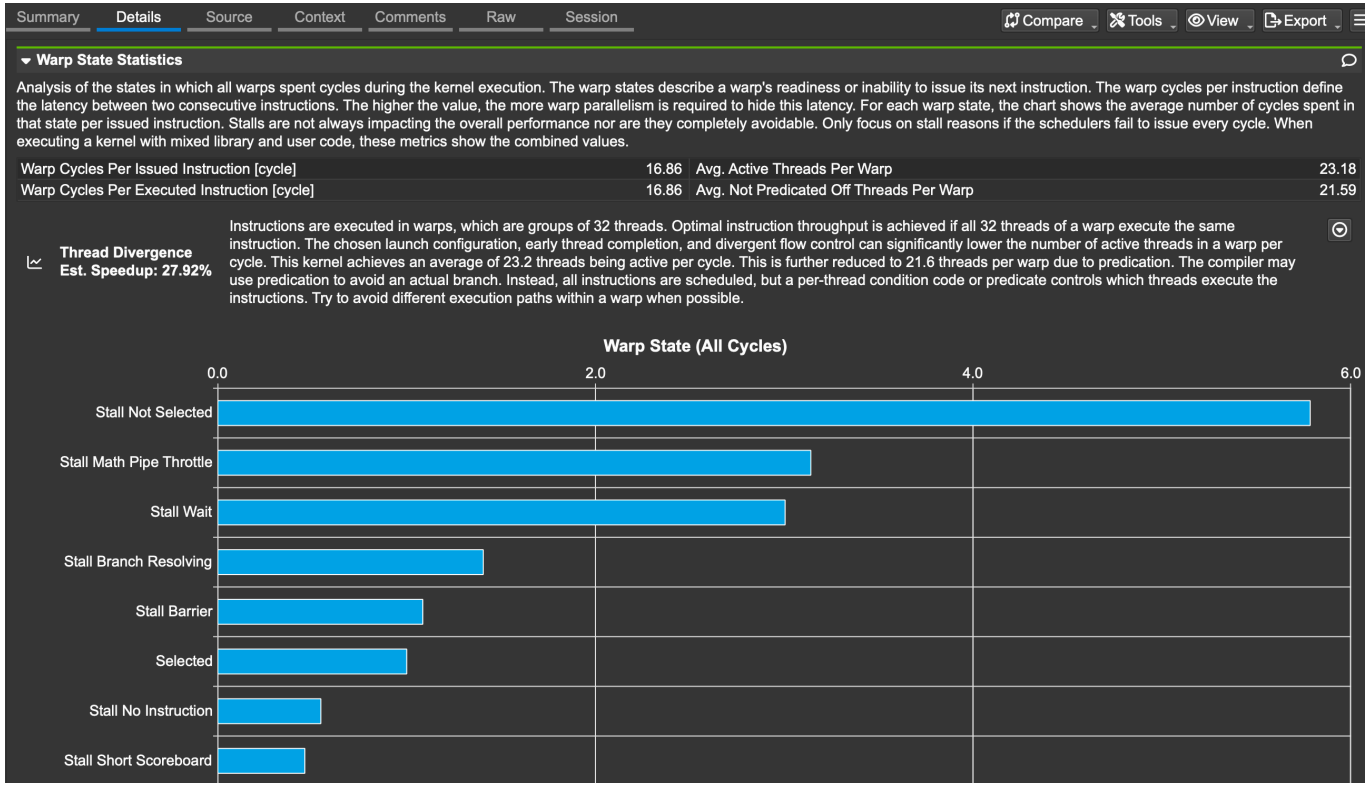
Latency hiding / Increasing instruction throughput

Latency hiding / Increasing instruction throughput



- Most time is spent waiting for instructions to finish, and hardware resources are underutilized.
  - **Sometimes we talk about a "bubble of idleness" in a compute pipeline. In this case, the pipeline is mostly bubbles!**
- Need more instructions in flight at once to hide instruction latency and increase hardware utilization.
- Goal: Increase hardware busyness. Issue instructions more often!
  - However, as we'll see later, busy does not always mean useful!

Reasons for stalling.



Types of Warp Stall Reasons (docs)

- **Wait**– Waiting for an instruction of compile-time-known latency.
- **Scoreboard** – Waiting for an instruction of runtime-determined latency.
  - Long SB – typically global memory
  - Short SB – typically shared memory
- **Throttle** – Waiting for the **queue** of a hardware resource to have free space.
- **Branch resolving** – Waiting for branch / PC bookkeeping
- **Barrier** – Waiting for other threads synchronize.

Stall avoidance strategies

Whichever instruction is causing the stall, can you:

- Do it less? Or not at all?
- Otherwise, increase their concurrency?
- Share the result among threads?
- Kick it off sooner? (Prefetch)
- Put more instructions between the issue and use?
- Fetch vs recompute?
- Use a lower latency instruction?
- Utilize a different memory space?

Illustrate concept of register / software pipelining:

- Must break register dependency.
  - Compute must not reference the register that was just loaded.

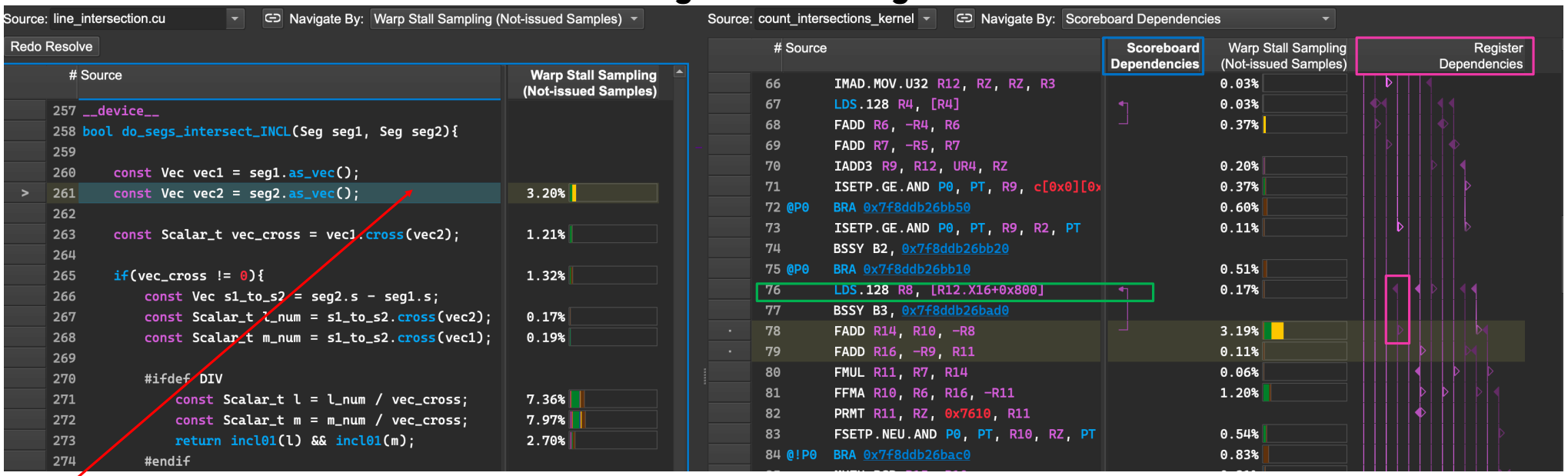
Advanced APIs available

- Blog "Boosting Application Performance with GPU Memory Prefetching"
  - __pipeline_memcpy_async, __pipeline_commit(), etc.

```
// Sketch without register pipelining

auto result = 0;
for(…)
  auto working_set = load(…)
  // Stalling to load working_set...

  working_set.compute()
  // Compute lots with working_set...

  result += working_set
```

```
// Sketch of register pipelining with
// prefetch distance = 1

auto result = 0;
auto prefetched = load(…)
// Stall only on first iteration...

for (…. )
  auto working_set = prefetched
  prefetched = load(...)

  working_set.compute()

  // Compute + Mem IO now concurrent...

  result += working_set
  // hopefully by the time we loop again,
  // prefetch has had enough time to
  // come back.
```

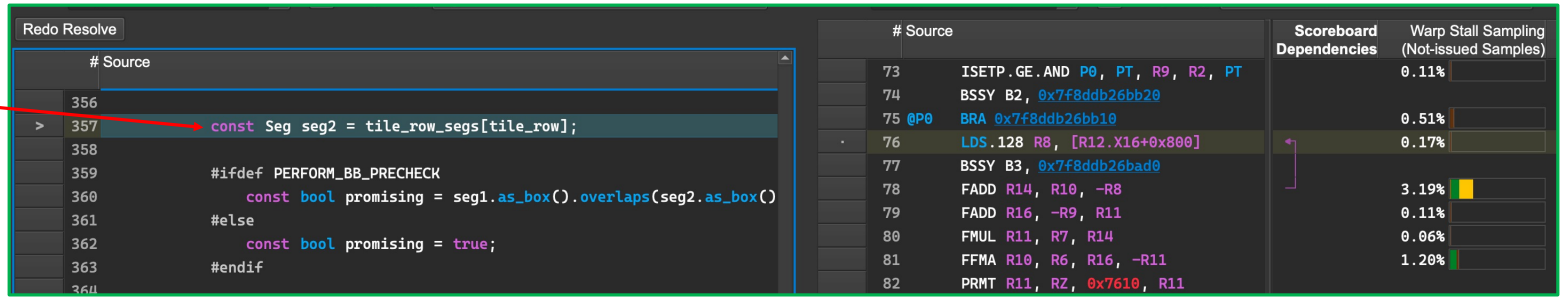
NCU Scoreboard Tracing: Find the origin of scoreboard stalls



Stalling on line 261,

Because waiting for load on 357

Then, clicking on the load instruction...



Click on the **pointed to load instruction** to see the origin of the stall.
Coming soon: Trace dependencies with **Scoreboard Dependencies** rather than **Register Dependencies**

NVIDIA

Barriers (briefly)

- A location in code for threads to stop and wait for each other before moving on.
- Facilities collaboration through memory, etc.
- `__syncthreads()`
  - Syncs entire thread block. Required to be called by all threads in the block. Cannot be called within conditionals unless they evaluate identically across thread block. Otherwise, its undefined behavior.
- Cooperative Groups Sync
  - Syncs entire group defined by user. Permitted to be called by only some threads and in divergent branches

```
104  __global__ void kernel(Args args) {
105    if (threadIdx.z % 2 == 0) {
106      doWork();
107      __syncthreads();
108    } else {
109      doOtherWork();
110      __syncthreads();
111    }
112  }
```

Undefined Behavior!

```
114  __global__ void kernel(Args args) {
115    if (threadIdx.y % 2 == 0) {
116      doWork();
117      __syncthreads();
118      doOtherWork();
119    }
120  }
```

Allowed as non-participating threads have exited

```
122  __global__ void kernel(Args args) {
123    auto block = this_thread_block();
124    if (threadIdx.y % 2 == 0) {
125      doWork();
126      block.sync();
127    } else {
128      doOtherWork();
129      block.sync();
130    }
131  }
```

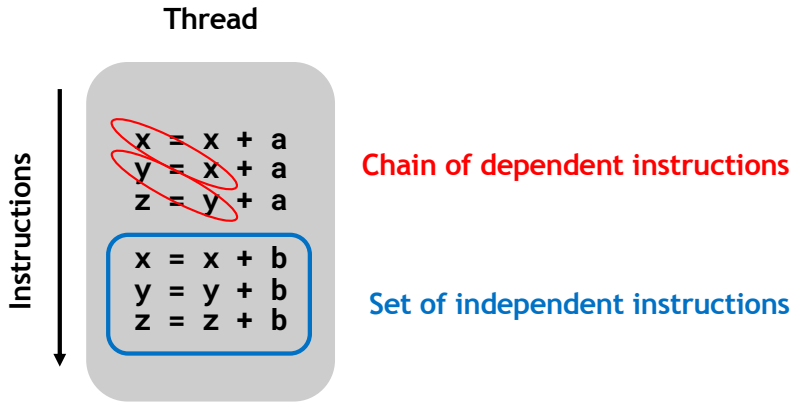
Defined behavior with cg barrier

Hiding Latencies

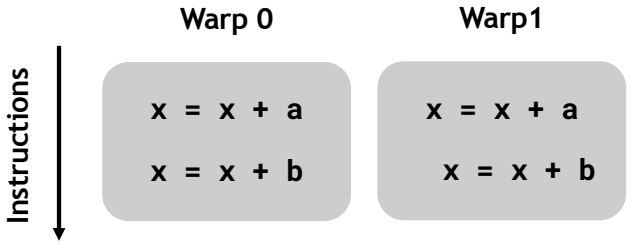
Increasing in-flight instructions

- Two ways to increase in-flight instructions:

1. Improve Instruction Level Parallelism (ILP)
   - More **independent work (instructions) per thread**.



2. Improve Occupancy. (AKA Thread Level Parallelism, TLP)
   - Describes how many warps can run concurrently given HW resource constraints.
   - More concurrently active warps = more in-flight instructions.

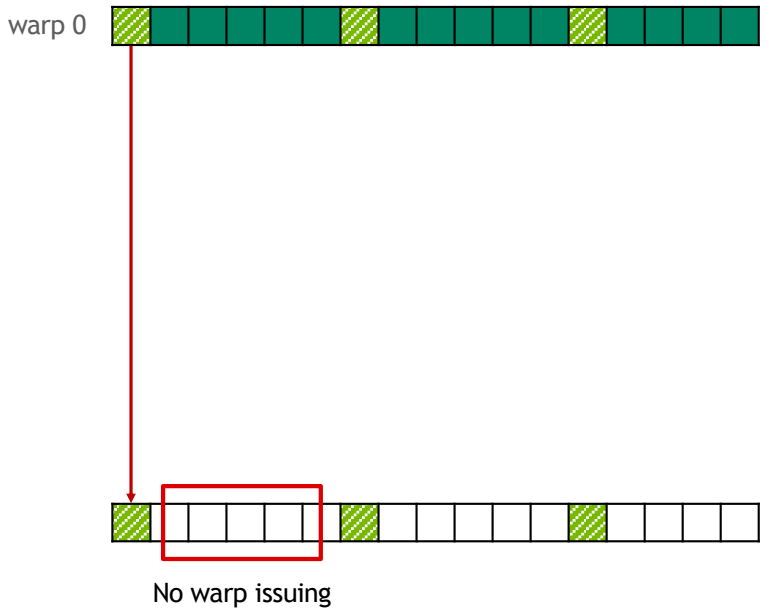


Latency bound kernel: What improvement would look like.

GPUs cover latency by having lots of work in flight


```
173  __global__ void kernel_A(float* A, int K)
174  {
175    float result = 0.0f;
176    int i = blockIdx.x * blockDim.x + threadIdx.x;
177
178    for (int j = 0; j < K; ++j) {
179      result += 3.14f;
180    }
181    A[i] = result;
182  }
183
184  ...
185  kernel_A<<<1, 32>>>(d_f32, 1000);
186  cudaDeviceSynchronize();
```

Assume FP32 add has a latency of 6 cycles



Latency bound kernel: What improvement would look like.

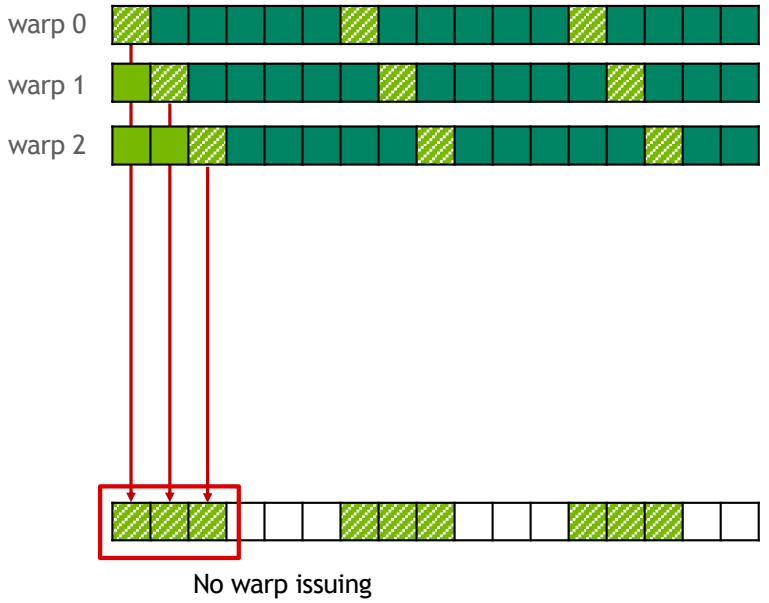
GPUs cover latency by having lots of work in flight

Warp States:

- Unused
- Active
- Stalled
- Eligible
- Selected

Increasing number of warps hides latencies

```
190  __global__ void kernel_A(float* A, int K)
191  {
192    float result = 0.0f;
193    int i = blockIdx.x * blockDim.x + threadIdx.x;
194
195    for (int j = 0; j < K; ++j) {
196      result += 3.14f;
197    }
198    A[i] = result;
199  }
200
201  ...
202  kernel_A<<<3, 32>>>(d_f32, 1000);
203  cudaDeviceSynchronize();
```



Little's Law applied here describes how many instructions we need in flight at once to avoid exposing latency.

Latency bound kernel: What improvement would look like.

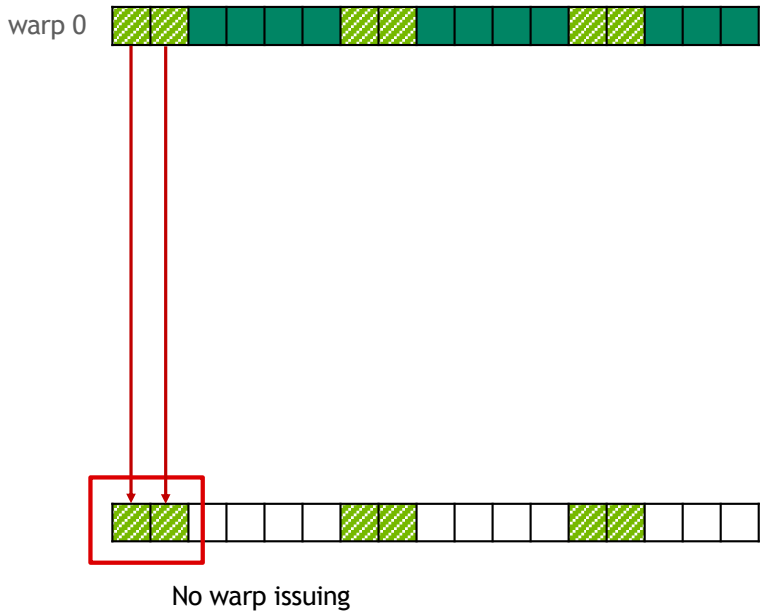
GPUs cover latency by having lots of work in flight

Warp States:

- Unused
- Active
- Stalled
- Eligible
- Selected

Increasing instruction level parallelism hide latency

```
208  __global__ void kernel_A(float2* A, int K)
209  {
210    float2 result = make_float2(0.0f, 0.0f);
211    int i = blockIdx.x * blockDim.x + threadIdx.x;
212
213    for (int j = 0; j < K; ++j) {
214      result.x += 3.14f;
215      result.y += 3.14f;
216    }
217    A[i] = result;
218  }
219
220  ...
221  kernel_A<<<1, 32>>>(d_f32, 1000);
222  cudaDeviceSynchronize();
```



No warp issuing

Bottom Line on Stalling

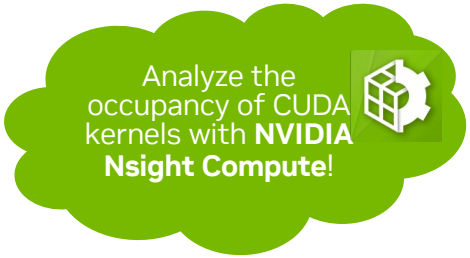
1. If SM or Memory System resources are busy, don't worry about stalls or unused issue slots.
   - Issuing more frequently won't help! Resources are already busy.
2. Otherwise, you are latency bound. Provide HW with more concurrent work. Try to:
   - Issue more frequently.
   - Stall less frequently.
   - Busy yourself with something else during the stall.
   - Decrease duration of stall. (use lower latency instruction)

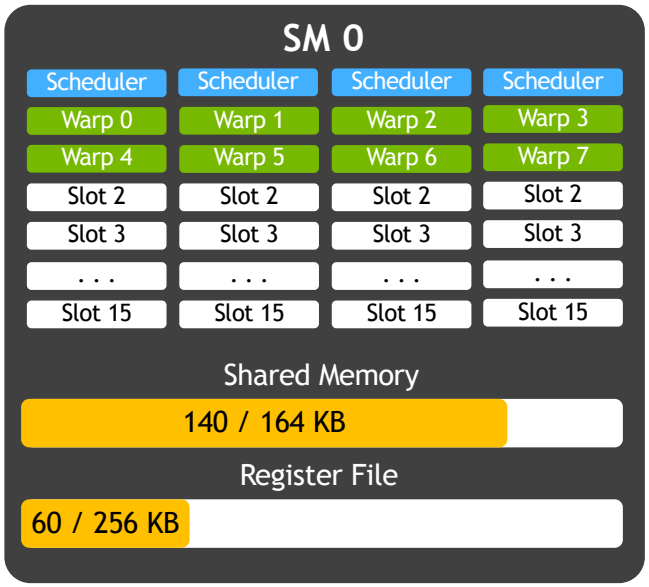
Occupancy: What is it? How is it determined?

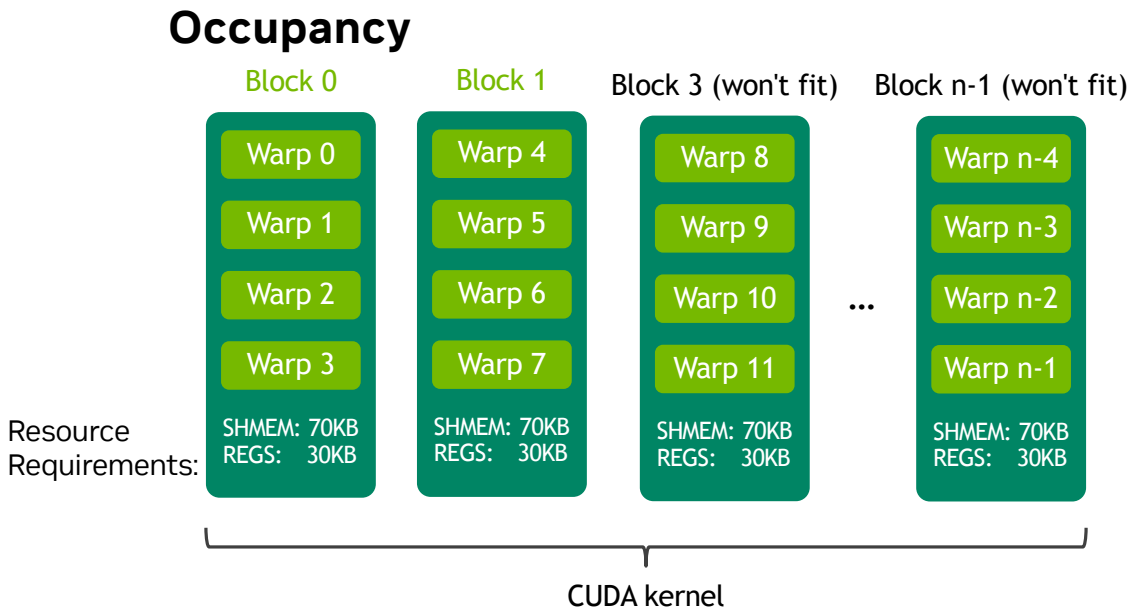
- There is a maximum number of warps which can be concurrently active on an SM.
  - **Device** (depends on compute capability of the GPU)
  - **Achievable** (depends on kernel implementation + compiler)
  - **Achieved** (depends mostly on the grid size or workload behavior. Such as workload imbalance.)

$$Occupancy = \frac{Achievable\ \#\ active\ warps\ per\ SM}{Device\ \#\ active\ warps\ per\ SM}$$

- Achievable Occupancy of a CUDA kernel will be limited by at least one of several factors.
  - SM resource assignment:
    - For example: Shared memory and Registers must be partitioned among threads.
    - Block size
  - Other hardware factors:
    - For example: Max blocks per SM, Max warps per SM etc.
    - Outlined in Table 23 Technical Specifications per Compute Capability







Card dealing analogy: Blocks (cards) have certain resource requirements (shmem, registers).

- Given the block's size and requirements, deal out blocks to SMs while the SM has sufficient resources to service them concurrently.

Above case: Given that block size =128 threads = 4 warps, the requested shmem, regs, and various HW limitations:

- The first resource to run dry is shared memory.
- Achieved occupancy of 8 warps per SM

Next, we could increase occupancy in one of two ways:

- Reprogram kernel to use less shmem per block.
- Decrease block size to 3 warps = 96 threads. (Notice the 24KB of wasted shmem)
  - By dealing smaller/finer blocks, each (new) block needs 70 * (¾) KB = 52.5 KB. Hence, we can deal floor(164 / 52.5) = 3 blocks.
  - Now achieved 9 warps per SM. Small effort and improvement.

Occupancy Limiters

Registers

- Register usage: compile with `--ptxas-options=-v`
  - Reports registers per thread
- The maximum number of registers per thread can be set manually:
  - At compile time on a per-file basis using the `--maxrregcount` flag of nvcc
  - Per-kernel using the `__launch_bounds__` or `__maxnreg__` qualifiers
- Hopper has 64K (65536) registers per SM
  - Allocated in fixed-size chunks of 256 registers

- **Example:**
  - Kernel uses 63 registers per thread
  - Registers per warp = 63 * 32 = 2016
  - Registers allocated per warp = 2048
  - Achievable active warps per SM = 65536 / 2048 = 32
  - Occupancy = 32 / 64 * 100 = 50%
    - Hopper supports up to 64 warps per SM

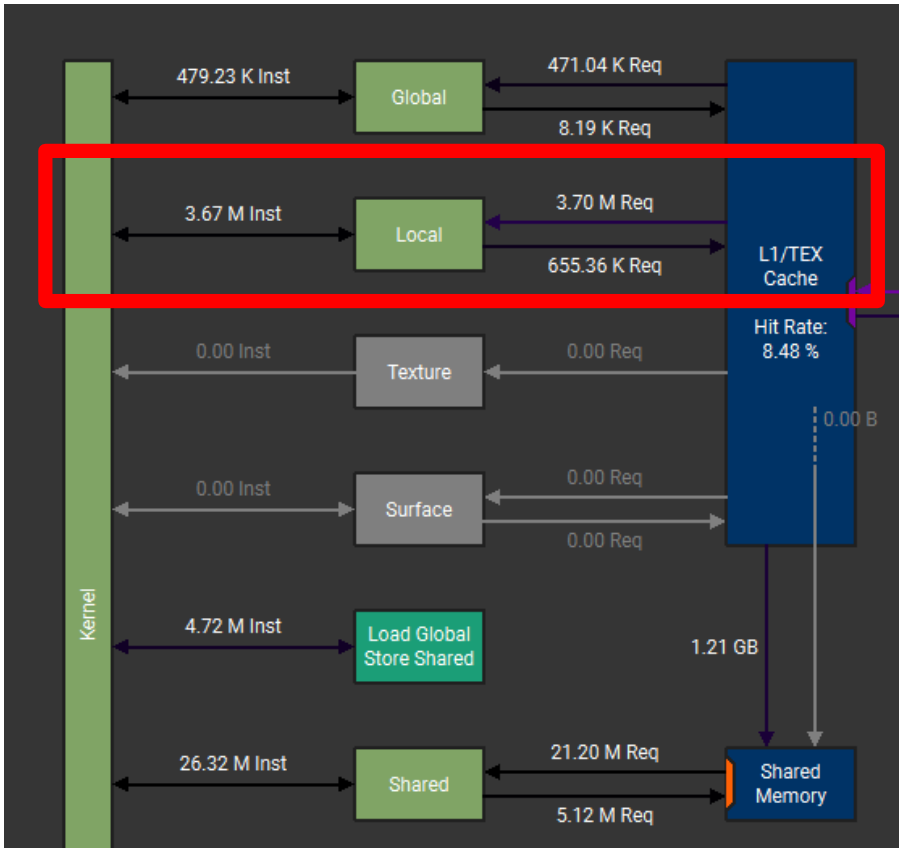
Exceeding the register limit

- If the compiler needs more registers for a kernel than is allowed by the device or a user-specified limit, via launch bounds or maxrregcount, it will spill into local memory
  - Local memory is a thread-private storage space located in device memory and cached in L1 and L2. The compiler can use local memory for other reasons

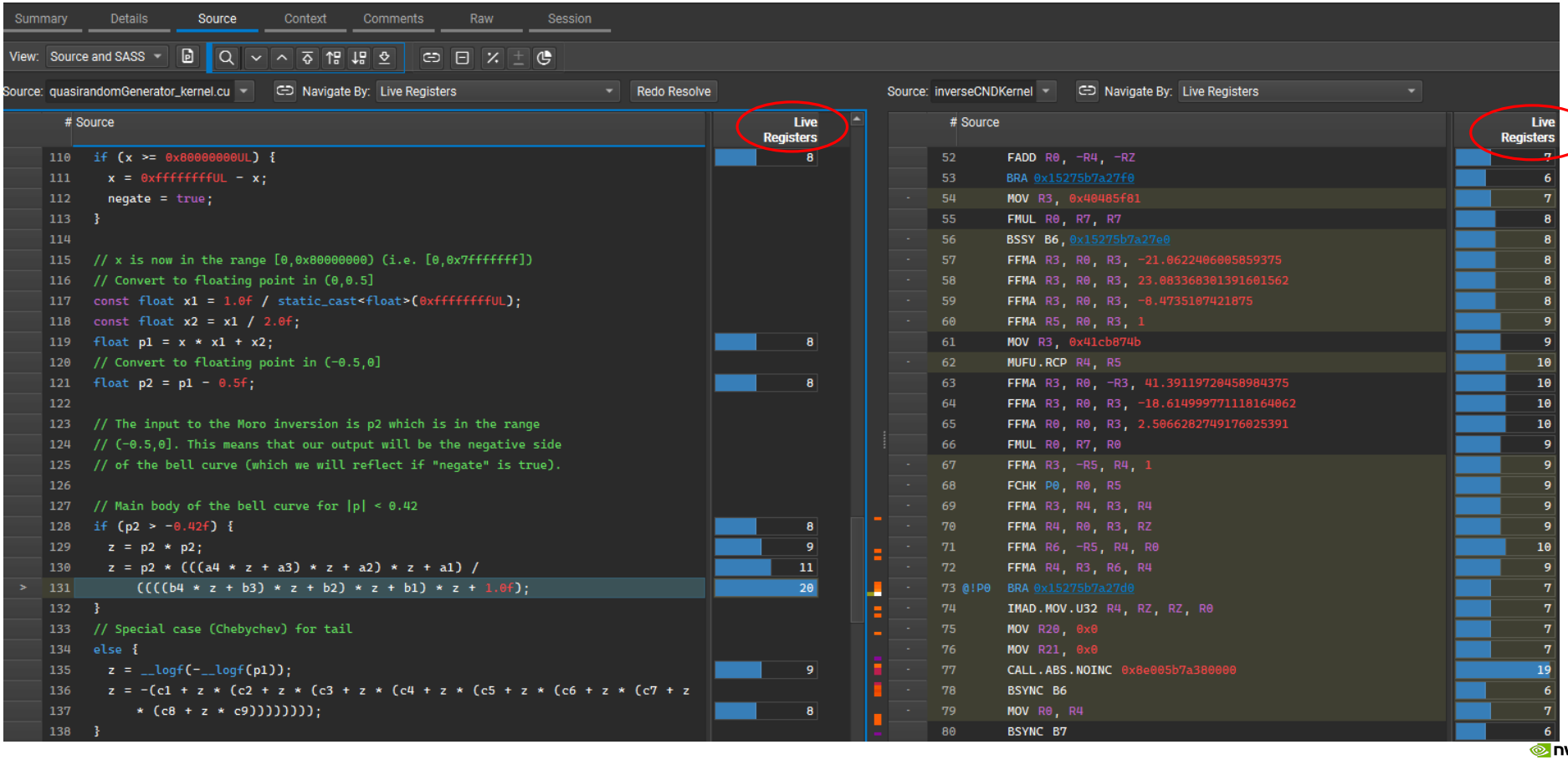
Optimization Tip

Limited local memory usage can be beneficial to performance

If data stays in L1, access will be fast and improve occupancy; however, it warrants further investigation in most cases.



Where is Register Pressure in NCU?



Tips for reducing register pressure

Optimization Tip

`__forceinline__` to avoid function call overheads and the ABI

- The `__forceinline__` function qualifier will force the compiler to inline a device function
  - This avoids the cost of a function call and can reduce register pressure by avoiding the ABI
  - Inlining long functions which are called in many places can lead to instruction cache thrashing

- The `#pragma unroll` directive can be used to control the unrolling of loops
  - Loop unrolling can decrease flow control overhead but may also increase register pressure and lead to instruction cache thrashing
  - Factors can be tuned to find the right balance which may vary depending on GPU generation

Optimization Tip

Tune loop unroll factors

Optimization Tip

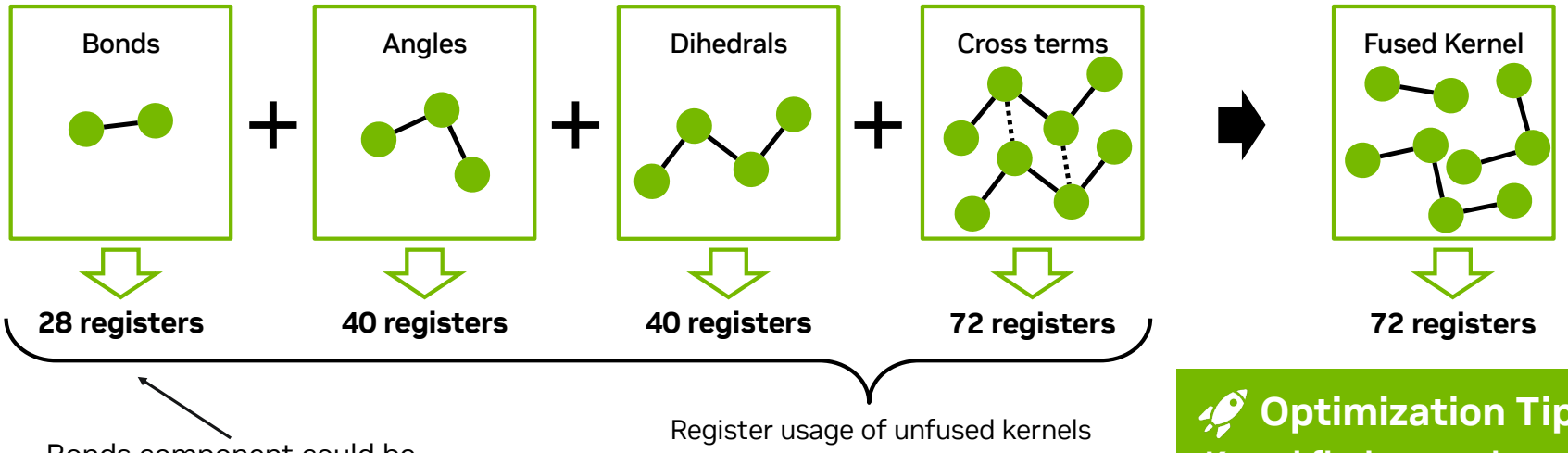
Avoid 64-bit types whenever possible

- 64-bit values will consume two registers, so it is best to use 32-bit types whenever possible

Register usage is determined by high-water mark of a kernel

Kernel fusion is a great optimization strategy but having a kernel do too much can have consequences

- Most biological molecular dynamic codes will compute the forces between predetermined sets of atoms. This component of the force is only a small fraction of the overall work, and the kernels are very short for systems of scientific interest
- Due to their short runtime, NAMD fuses the calculations for the different types of sets into a single kernel. The more complex groupings take more registers and dictate the register usage of the combined kernel



Optimization Tip

Kernel fission can decrease register usage and improve occupancy

Occupancy Limiters

Thread block size

- Thread block size is a multiple of warp size (32).
  - Even if you request fewer threads, HW rounds up.
- Each thread block can have a maximum size of 1024.
- Each SM can have up to 64 warps, 32 blocks and 2048 threads (Hopper).

Block Size	Active threads per SM	Active Warps per SM	Active Warps per Block	Active Blocks per SM	Occupancy (%)
32	1024	32	1	32	**50**
64	2048	64	2	32	100
768	1536	48	24	2	**75**
1024	2048	64	32	2	100

What Occupancy Do I Need?

General guidelines

Rule of thumb: Try to maximize occupancy.

But some algorithms will run better at low occupancy.

More registers and shared memory can allow higher data reuse, higher ILP, higher performance.

Low Occupancy

Fewer threads per SM.

More resources per thread.

Enough instruction-level parallelism or GPU will starve!

High Occupancy

More threads per SM.

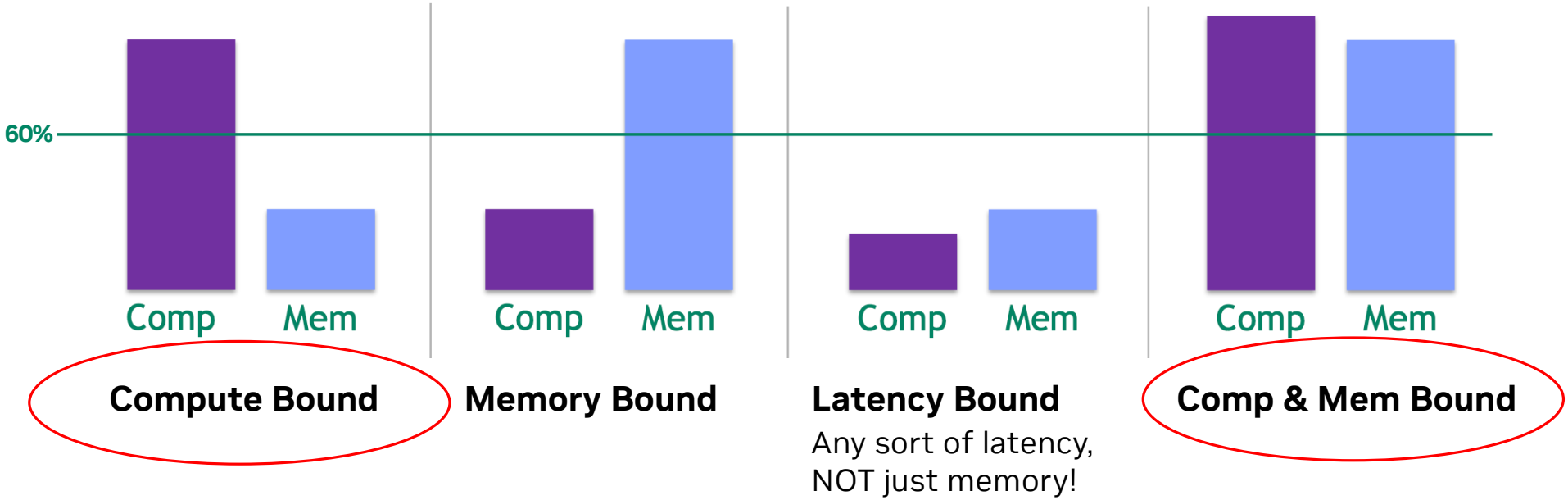
Fewer registers per thread.

Rely on thread parallelism to hide latencies!

Complex algorithms ⟷ Simple algorithms

Reducing instruction count and making throughput useful.

Reducing instruction count and Making throughput useful.



- Most time is spent issuing and executing instructions. The SM resources are busy.
- Goal: Now make sure the work performed is necessary / useful.
- Note: As you make progress optimizing kernel, it's very possible to bounce between various of the 4 states.

Ways to reduce instruction count

(Examples to come)

- Frequently zoom your focus in/out through all levels of the problem.
  - Source tweaks
  - Algorithm changes.
  - Everything in between.
- Focus as much on the problem being solved as the code being run.

- Perform "inexpensive prechecks" to see if you can avoid expensive operations.
- Algebraic optimizations
- Operate in a different numeric space.
- Use CCCL for high performance parallel primitives. Avoiding reinventing the wheel.
- Vectorized instructions (memory operations, DPX, f32x2 on Blackwell)

Cost hierarchy of math operations

Use the most lightweight tool possible

- Trig, sqrt, etc.
- Division, mod operators
  - No division or mod instructions in hardware. Depending on properties of divisor, might turn into 100s of instructions.
    - Division / mod by constexpr is usually fine.
- Multiply, add, subtract
- Fused multiply adds

Use only the precision your application needs

- Investigate if lower precisions are sufficient for your use case
- Beware of implicit cast to double! Use the .f suffix on your numeric literals to avoid it
- Make use of the fast math optimizations (--use_fast_math)
  - Incremental adoption is possible via Single Precision Intrinsics. __cosf(), __expf(), __frcp_*(), __fsqrt_*(), etc..
  - Single precision trigonometric math API functions may use some double precision instructions and local memory

expf()

Default math API code path

```
mov.f32 %f2, 0f3F000000;
mov.f32 %f3, 0f3BBB989D;
fma.rn.f32 %f4, %f1, %f3, %f2;
cvt.sat.f32.f32 %f5, %f4;
mov.f32 %f6, 0f4B400001;
mov.f32 %f7, 0f437C0000;
fma.rm.f32 %f8, %f5, %f7, %f6;
add.f32 %f9, %f8, 0fCB40007F;
neg.f32 %f10, %f9;
mov.f32 %f11, 0f3FB8AA3B;
fma.rn.f32 %f12, %f1, %f11, %f10;
mov.f32 %f13, 0f32A57060;
fma.rn.f32 %f14, %f1, %f13, %f12;
mov.b32 %r6, %f8;
shl.b32 %r7, %r6, 23;
mov.b32 %f15, %r7;
ex2.approx.ftz.f32 %f16, %f14;
mul.f32 %f17, %f16, %f15;
```

`__expf` or fast math

```
mul.f32 %f2, %f1, 0f3FB8AA3B;
ex2.approx.f32 %f3, %f2;
```

CUDA Programming Guide contains more information with error bounds on intrinsic functions

Function	Error bounds
`__fmaf_[rn,rz,ru,rd](x,y,z)`	IEEE-compliant.
`__expf(x)`	The maximum ulp error is `2 + floor(abs(1.173 * x))`.
`__sinf(x)`	For `x` in $[-\pi, \pi]$, the maximum absolute error is $2^{-21.41}$, and larger otherwise.
`__cosf(x)`	For `x` in $[-\pi, \pi]$, the maximum absolute error is $2^{-21.19}$, and larger otherwise.
`...`	

Algebraic Optimizations

Static Considerations

- Observe any code inside loops: Can multiplication by a constant be pulled out? Etc.
- Move divisors to the other side of comparison operators.
- If you have division by a runtime constant inside kernel: Instead, compute inverse on host, pass to kernel. Multiply by inverse in kernel.
- Use template parameters for any variable known at compile time or with a limited range of values. Runtime compilation can take this even further.

Runtime Considerations

- Do you know more than the compiler? About:
  - The possible range of values produced by an expression?
  - How an expression is conceptually used in subsequent expressions?
    - What are key thresholds?
    - When do conditional results change?
  - How can you leverage that?

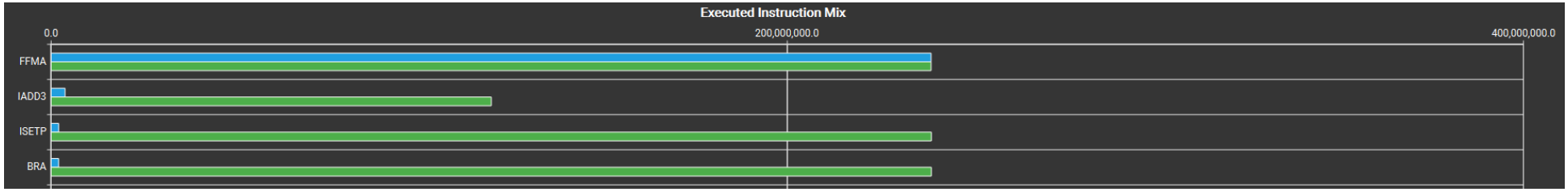
Small changes can have a large impact

Unsigned integer overflows are defined behavior and the compiling needs to account for this resulting in maybe additional instructions. Signed integer overflows are undefined behavior which gives the compiler more flexibility allowing for the generation of faster code

```
85   #pragma unroll 128
86   for (unsigned int i = 0; i < num_iters; i += 2) {
87     const unsigned int reg_index = (i % 8) / 2;
88     vals[reg_index] = vals[reg_index] * scale + offset;
89   }
90
91   #pragma unroll 128
92   for (int i = 0; i < num_iters; i += 2) {
93     const int reg_index = (i % 8) / 2;
94     vals[reg_index] = vals[reg_index] * scale + offset;
95   }
```

Optimization Tip

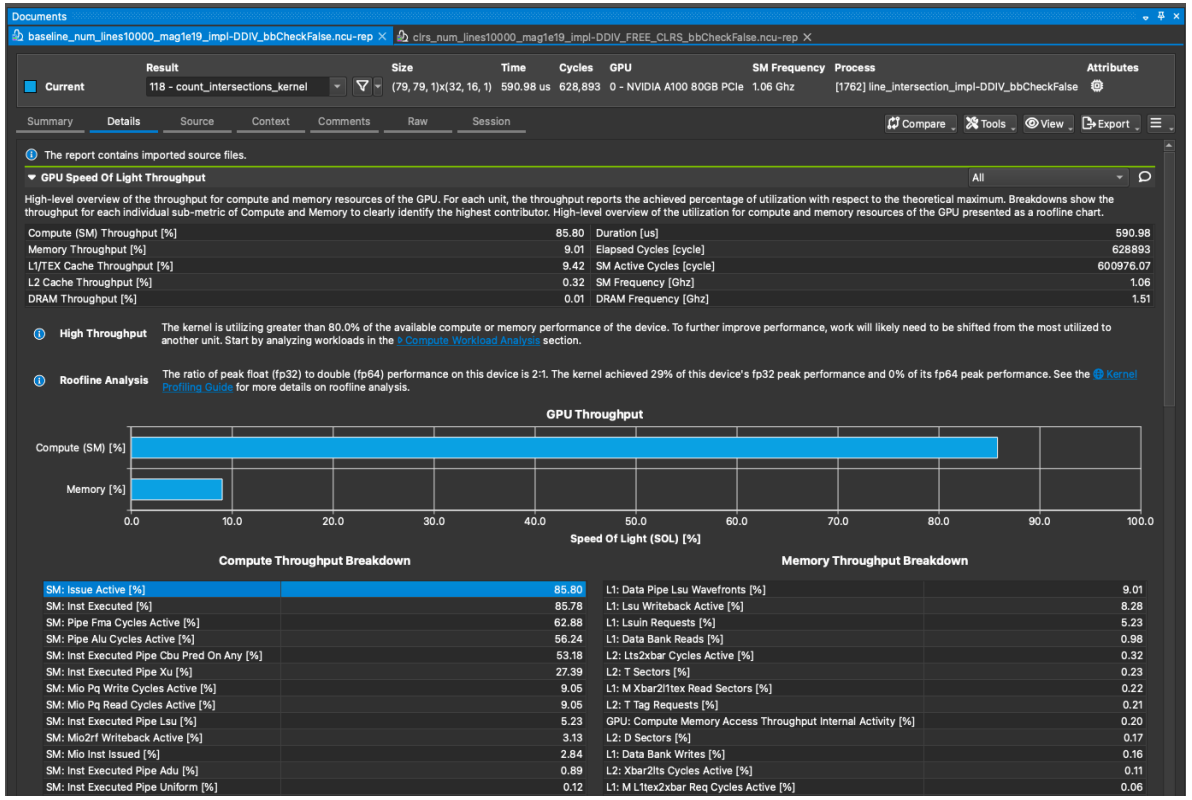
Use signed integers rather than unsigned integers as loop counters



Segment intersection example

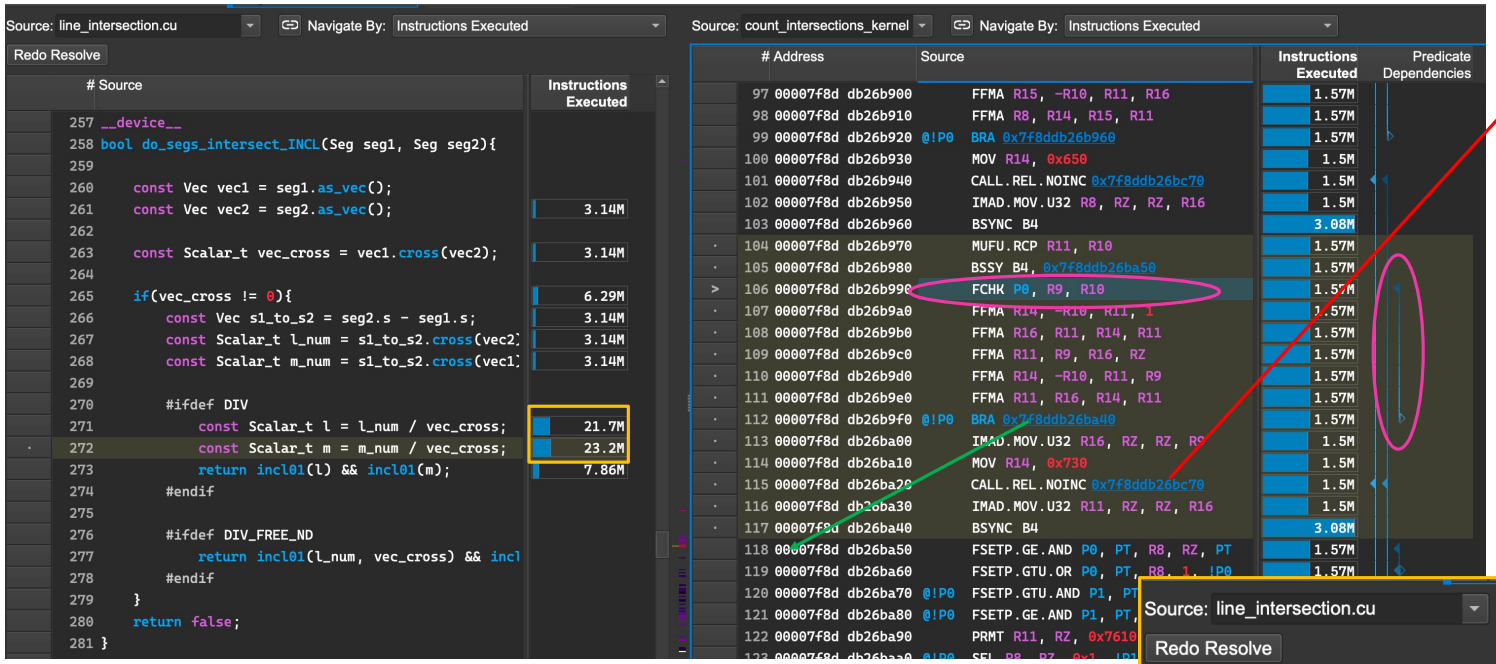
Given an array of line **segments**, count how many intersections occur.

```cpp
template<typename T>
__device__
bool incl01(T v) {
    return T(0) ≤ v && v ≤ T(1);
}

__device__
bool do_segs_intersect_INCL(Seg seg1, Seg seg2){

    const Vec vec1 = seg1.as_vec();
    const Vec vec2 = seg2.as_vec();

    const Scalar_t vec_cross = vec1.cross(vec2);

    if(vec_cross ≠ 0){
        const Vec s1_to_s2 = seg2.s - seg1.s;
        const Scalar_t l_num = s1_to_s2.cross(vec2);
        const Scalar_t m_num = s1_to_s2.cross(vec1);

        #ifdef DIV
            const Scalar_t l = l_num / vec_cross;
            const Scalar_t m = m_num / vec_cross;
            return incl01(l) && incl01(m);
        #endif



    }
    return false;
}
```



Full source code available on Accelerated Computing Hub

Compute bound on division slow path:
The SASS (assembly) correlated with C division operator calls the injected helper SASS (function).

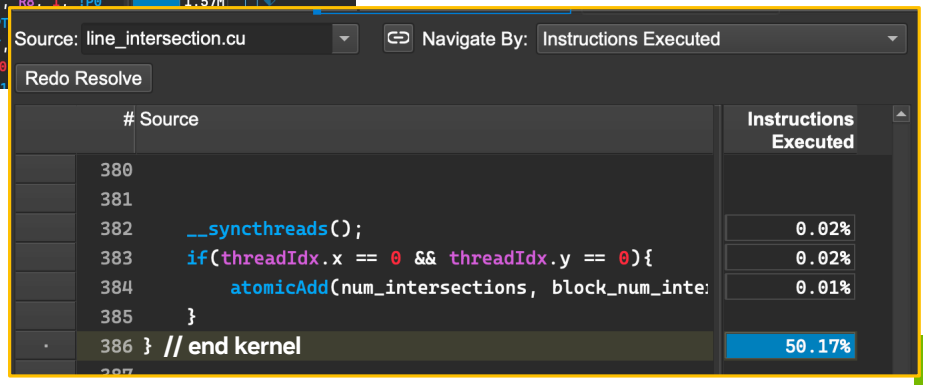


Injected SASS Slow path

Not associated with any C Source.

Shared by all div ops.

Often "correlated" with closing "}" of kernel.



Gory details: How to know how often you're taking the fast or slow path?

- Determined by float range at runtime.

Look for FCHK instruction. Follow the predicate dependency.

CALL.REL.NOINC instruction jumps to slow path.

BRANCH instruction jumps over that CALL instruction. (fast path)

Segment intersection example

Possible optimizations

1. Source tweak to avoid division
2. Algorithm change to avoid division.
3. Perform bounding box precheck
4. Operate on different range of floats to get off slow path.

Option 1: Source tweak to avoid division

Observe how the result of division is used.

- **We only care whether the result lies in the range [0, 1]**
- **Instead of performing division, compare the signs and magnitudes of numerator and denominator.**

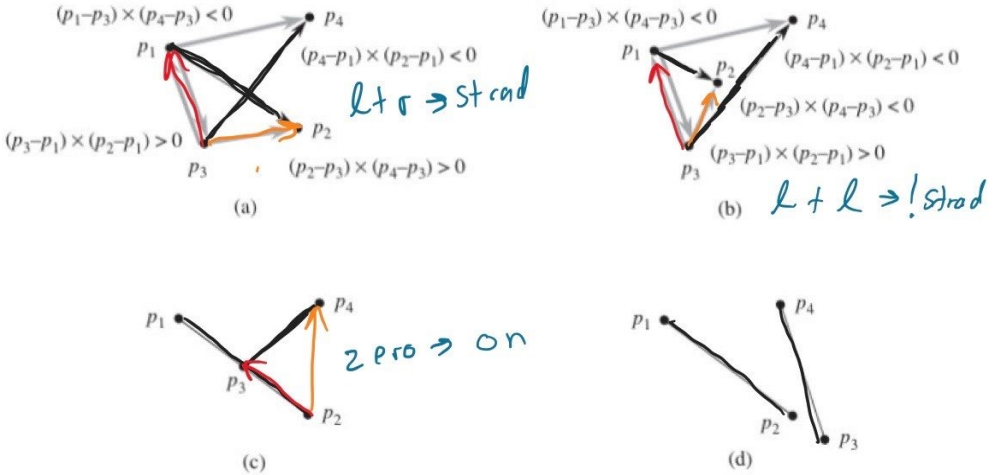
```
255  __device__
256  bool do_segs_intersect_INCL(Seg seg1, Seg seg2){
257
258      const Vec vec1 = seg1.as_vec();
259      const Vec vec2 = seg2.as_vec();
260
261      const Scalar_t vec_cross = vec1.cross(vec2);
262
263      if(vec_cross ≠ 0){
264          const Vec s1_to_s2 = seg2.s - seg1.s;
265          const Scalar_t l_num = s1_to_s2.cross(vec2);
266          const Scalar_t m_num = s1_to_s2.cross(vec1);
267
268          #ifdef DIV
269              const Scalar_t l = l_num / vec_cross;
270              const Scalar_t m = m_num / vec_cross;
271              return incl01(l) && incl01(m);
272          #endif
273
274          #ifdef DIV_FREE_ND
275              return incl01(l_num, vec_cross) && incl01(m_num, vec_cross);
276          #endif
277      }
278      return false;
279  }
280
```

```
230  __device__ __forceinline__
231  bool incl01(Scalar_t num, Scalar_t den){
232
233      if(abs(num) < 1.0e-7f){
234          return true;
235      }
236
237      const bool num_positive = num ≥ 0;
238      const bool den_positive = den ≥ 0;
239
240      const bool gt = num > den;
241      const bool lt = num < den;
242      return !(
243          (num_positive ≠ den_positive) |
244          (num_positive & gt) | // both have positive sign, numerator larger in magnitude
245          ((!num_positive) & lt) // both have negative sign, numerator larger in magnitude.
246      );
247  }
248
249  template<typename T>
250  __device__
251  bool incl01(T v) {
252      return T(0) ≤ v && v ≤ T(1);
253  }
254
```

Option 2: Different algorithm to avoid division: Segments divide the 2D plane.

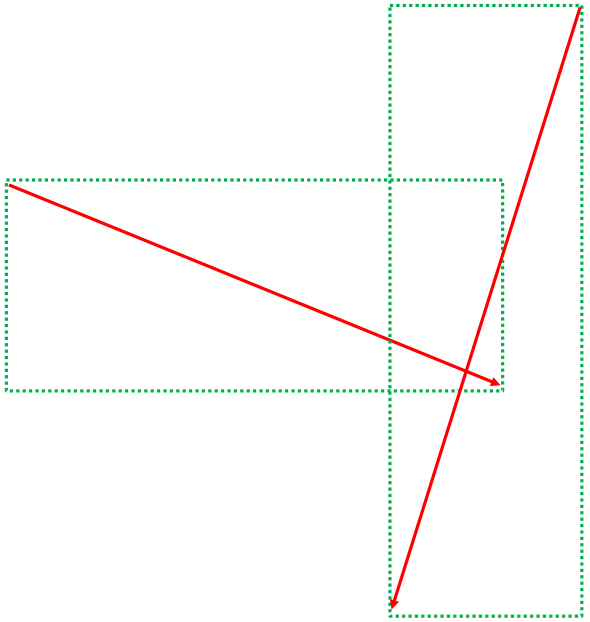
```
__device__
bool do_segs_intersect_divFreeCLRS(Seg seg1, Seg seg2){

    // if s1_reg and t1_reg are both non-zero and differ in sign,
        // then seg 1 straddles the LINE formed by seg 2
    const Scalar_t s1_reg = seg2.get_region_of_point(seg1.s);
    const Scalar_t t1_reg = seg2.get_region_of_point(seg1.t);
    // if s2_reg and t2_reg are both non-zero and differ in sign,
        // then seg 2 straddles the LINE formed by seg 1.
    const Scalar_t s2_reg = seg1.get_region_of_point(seg2.s);
    const Scalar_t t2_reg = seg1.get_region_of_point(seg2.t);

    const bool any_zero_region =
        (s1_reg == 0 || t1_reg == 0 || s2_reg == 0 || t2_reg == 0);

    // Check for intersections s.t. the intersection point IS NOT ANY END POINT.
    if(!any_zero_region
            && s1_reg < 0 != t1_reg < 0 // regions must differ
            && s2_reg < 0 != t2_reg < 0){
        return true;
    }

    // Check for intersections s.t. the intersection point IS ANY ENDPOINT.
    if(s1_reg == 0 && seg2.as_box().contains(seg1.s)){
        return true;  // seg1_s on seg2
    }
    if(t1_reg == 0 && seg2.as_box().contains(seg1.t)){
        return true;  // seg1_t on seg 2
    }
    if(s2_reg == 0 && seg1.as_box().contains(seg2.s)){
        return true;  // seg2_s on seg 1
    }
    if(t2_reg == 0 && seg1.as_box().contains(seg2.t)){
        return true;  // seg2_t on seg 1
    }

    // no intersection.
    return false;
}
```

// get_region_of_point() is just a few subtractions and a cross product.



Option 3: Add bounding box precheck

- Lightweight precheck: Before running any intersection test, check whether the bounding boxes overlap.
  - Adds instructions, may cause divergence. Is it worth it?



Option 4: Somehow change your input data

- Somehow change your input floating point numbers to be smaller.
  - Fast division path will be taken as FCHK won't flag for overflow risk.
- Application dependent. Might be impossible!

Line intersection results (A100)

Table for Float Magnitude (1e19):

Num Lines	BB_Check	Impl	Runtime Microseconds	
100000	False	DIV	41691 (1.00x)	Baseline (slow path)
100000	False	DIV_FREE_ND	16376 **(2.54x)**	Source tweak
100000	False	DIV_FREE_CLRS	15639 **(2.67x)**	New algorithm
100000	True	DIV	45795 (All BB worse!)	Bounding box doesn't help anyone for this data!
100000	True	DIV_FREE_ND	22655	
100000	True	DIV_FREE_CLRS	21076	

Table for Float Magnitude (1e18):

Num Lines	BB_Check	Impl	Runtime Microseconds	Work with smaller magnitudes.
100000	False	DIV	17154 (1.00x)	Baseline (fast path)
100000	False	DIV_FREE_ND	16379 **(1.05x)**	Source tweak
100000	False	DIV_FREE_CLRS	15640 **(1.10x)**	New algorithm
100000	True	DIV	22919 (All BB worse!)	
100000	True	DIV_FREE_ND	22653	
100000	True	DIV_FREE_CLRS	21076	

Table for Float Magnitude (1e2):

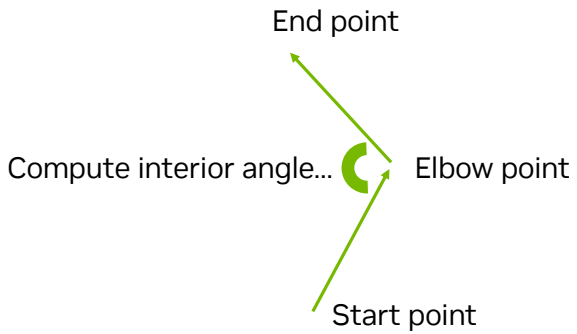
Num Lines	BB_Check	Impl	Runtime Microseconds	Work with less frequent intersections.
100000	False	DIV	16661 (1.00x)	
100000	False	DIV_FREE_ND	14514 (1.15x)	
100000	False	DIV_FREE_CLRS	14404 (1.16x)	
100000	True	DIV	11612 **(1.43x)**	BBox helps only on tiny data!
100000	True	DIV_FREE_ND	11425 **(1.46x)**	
100000	True	DIV_FREE_CLRS	11254 **(1.48x)**	

Several ways to achieve similar results:

Operate in a different numeric space

- Example: Using Log Probabilities for improved accuracy and performance.
- Example: Comparing squared distances rather than distances to avoid sqrt().
- Example: Angle Comparison
  - Say we want to determine which interior angles of a polygon are greater than some threshold.
  - Assume we are compute bound on the inverse trig performed in this function:

```
__device__
bool is_interior_angle_greater(Point start, Point elbow, Point end, float query_radians){
    // ...
    float candidate_interior_angle = acosf(... ... );
    return candidate_interior_angle > query_radians;
    // ....
}
```



- Optimization:
  - Currently: Every thread computes the candidate angle via inverse cosine. Then compares to query in the space of radians.
  - Instead: Compare angles in the space of "normalized dot products" rather than radians.
    - Have the host compute the cosine of the query angle once. Expressing query via the range [-1, 1]
    - Now have the threads compute a normalized dot product of the vectors and compare in that space.
  - Removed trig function in exchange for 2 divisions.

Optimizing Polynomial Evaluation

Repeatedly evaluating a high order polynomial with coefficients known at compile time.
(Degree 64, evaluated 32 times per thread.)

$$y = 1.0 + 1.01x + 1.02x^2 + \cdots$$

```
4   __device__ __forceinline__
5   float polynomial(float x) {
6       float val = 1.0;
7       val += pow(x, 1) * 1.01;
8       val += pow(x, 2) * 1.02;
9       ...
10      return val;
11  }
```

	Runtime (us)	Speedup
Baseline	40,862	1x

Optimizing Polynomial Evaluation

Repeatedly evaluating a high order polynomial with constants known at compile time

Always use “f” suffix to make sure literals are single precision!

```
27  __device__ __forceinline__
28  float polynomial(float x) {
29      float val = 1.0f;
30      val += pow(x, 1.0f) * 1.01f;
31      val += pow(x, 2.0f) * 1.02f;
32      ...
33      return val;
34  }
```

	Runtime (us)	Speedup
Baseline	40,862	1x
FP32 literals	6,198	6.6x

Optimization Tip

Ensure all literals are correct precision to avoid unnecessary conversions

`static_cast` can be used if code needs to support multiple types

```
11  template<typename T>
12  void __global__ kernel(const int N, T* data) {
13      const T myLiteral = static_cast<T>(3.14):
14      ...
15  }
```

Optimizing Polynomial Evaluation

Repeatedly evaluating a high order polynomial with constants known at compile time

Replace pow function calls with running calculation of exponentiation

```
37  __device__ __forceinline__
38  float polynomial(float x) {
39      float val = 1.0f;
40      const float x_1 = x;
41      val += x_1 * 1.01f;
42      const float x_2 = x * x_1;
43      val += x_2 * 1.02f;
44      const float x_3 = x * x_2;
45      val += x_3 * 1.03f;
46      ...
47    return val;
48  }
```

	Runtime (us)	Speedup
Baseline	40,862	1x
FP32 literals	6,198	6.6x
Avoiding general purpose functions	454.8	90x

Optimization Tip

Prefer faster, more specialized math functions over slower, more general ones when possible

Optimizing Polynomial Evaluation

Repeatedly evaluating a high order polynomial with constants known at compile time

Use **Horner's method** to decrease instruction count

```
51  __device__ __forceinline__
52  float polynomial(float x) {
53      ...
54      val = val * x + 1.02f;
55      val = val * x + 1.01f;
56      val = val * x + 1.00f;
57      return val;
58  }
```

	Runtime (us)	Speedup
Baseline	40,862	1x
FP32 literals	6,198	6.6x
Avoiding general purpose functions	454.8	90x
Horner's method	319.1	128x

Optimization Tip

Check if expressions can be reformulated to yield fewer instructions

$$p(x) = a_0 + x\left(a_1 + x\left(a_2 + x\left(a_3 + \cdots + x(a_{n-1} + x\,a_n)\cdots\right)\right)\right)$$

Optimizing Polynomial Evaluation

Repeatedly evaluating a high order polynomial with constants known at compile time

Make use of fused multiple add instructions either by compiling with fmad=true or intrinsics

```
61  __device__ __forceinline__
62  float polynomial(float x) {
63      ...
64      val = __fmaf_rn(val, x, 1.02f);
65      val = __fmaf_rn(val, x, 1.01f);
66      val = __fmaf_rn(val, x, 1.00f);
67      return val;
68  }
```

Optimization Tip

Use fused multiple add instructions when possible

	Runtime (us)	Speedup
Baseline	40,862	1x
FP32 literals	6,198	6.6x
Avoiding general purpose functions	454.8	90x
Horner's method	319.1	128x
Fused multiple adds	170.6	240x

Optimizing Polynomial Evaluation

Repeatedly evaluating a high order polynomial with constants known at compile time

Horner's method creates very little instruction level parallelism

Estrin's scheme can be used to add avoid serialization of instructions with minimal extra cost—as we don't need too much ILP, we can do a single recursive step to break the stream of instructions, then evaluate the two polynomials with Horner's method

Both cases have: Grid stride loop, fixed problem and block size.

Grid size varies: Sabotage block parallelism via smaller grid size.

	Thread blocks per SM	Runtime (us)
No ILP (Horner's Method)	8	170
	1	326
Increased ILP (Estrin's and Horner's method)	8	171
	1	192

Optimization Tip

Break chains of dependent instructions if ILP is needed

Take $P_n(x)$ to mean the nth order polynomial of the form: $P_n(x) = C_0 + C_1x + C_2x^2 + C_3x^3 + \cdots + C_nx^n$

Written with Estrin's scheme we have:

$P_3(x) = (C_0 + C_1x) + (C_2 + C_3x)\ x^2$

$P_4(x) = (C_0 + C_1x) + (C_2 + C_3x)\ x^2 + C_4x^4$

$P_5(x) = (C_0 + C_1x) + (C_2 + C_3x)\ x^2 + (C_4 + C_5x)\ x^4$

$P_6(x) = (C_0 + C_1x) + (C_2 + C_3x)\ x^2 + ((C_4 + C_5x) + C_6x^2)x^4$

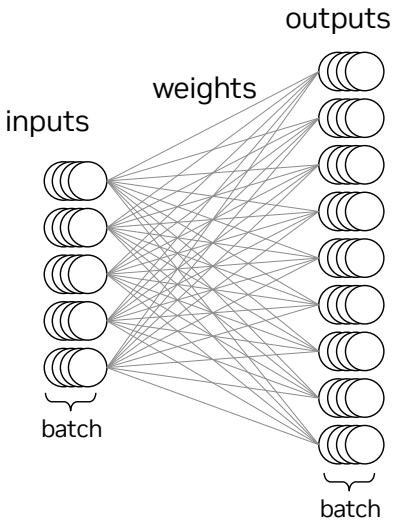
$P_7(x) = (C_0 + C_1x) + (C_2 + C_3x)\ x^2 + ((C_4 + C_5x) + (C_6 + C_7x)\ x^2)x^4$

$P_8(x) = (C_0 + C_1x) + (C_2 + C_3x)\ x^2 + ((C_4 + C_5x) + (C_6 + C_7x)\ x^2)x^4 + C_8x^8$

Tensor Core Summary

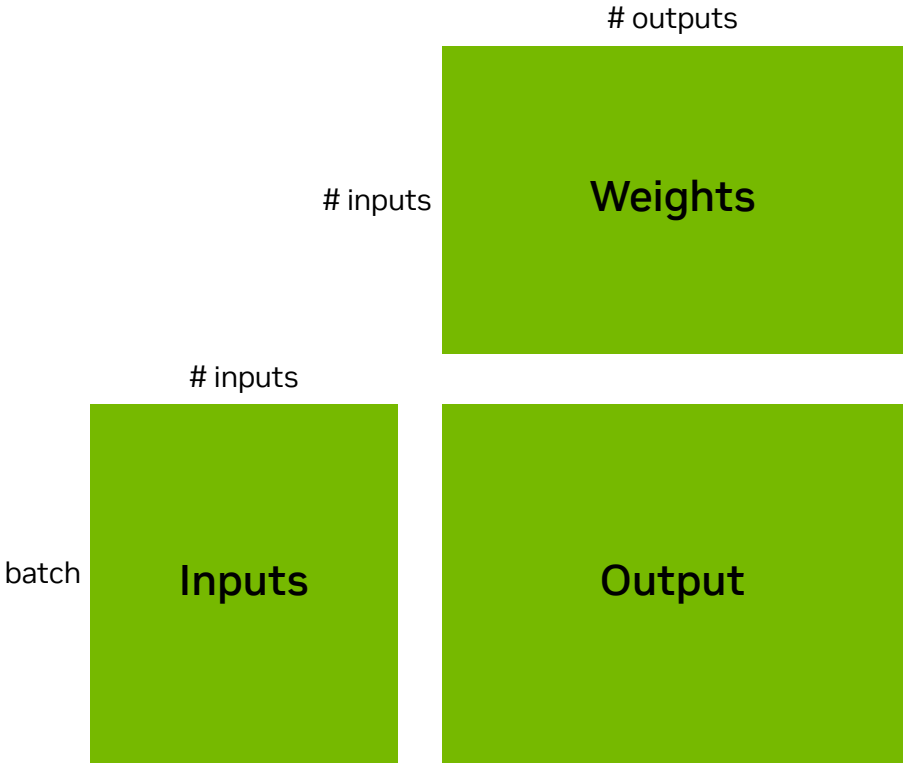
Matrix Multiplication

A key operation in Deep Learning



Fully connected layer

Batch dimension makes input a 2D matrix

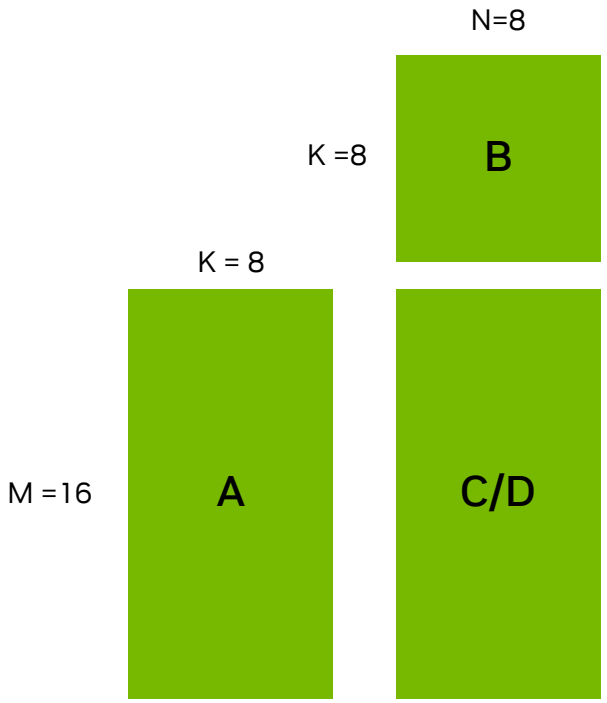


Tensor Core = Pipeline dedicated to matrix multiplications

Example : HMMA 1688

FP16 16 x 8 x 8 Tensor Core operation

A single HMMA1688 instruction computes 128 results (1024 FMAs)



D = A x B + C

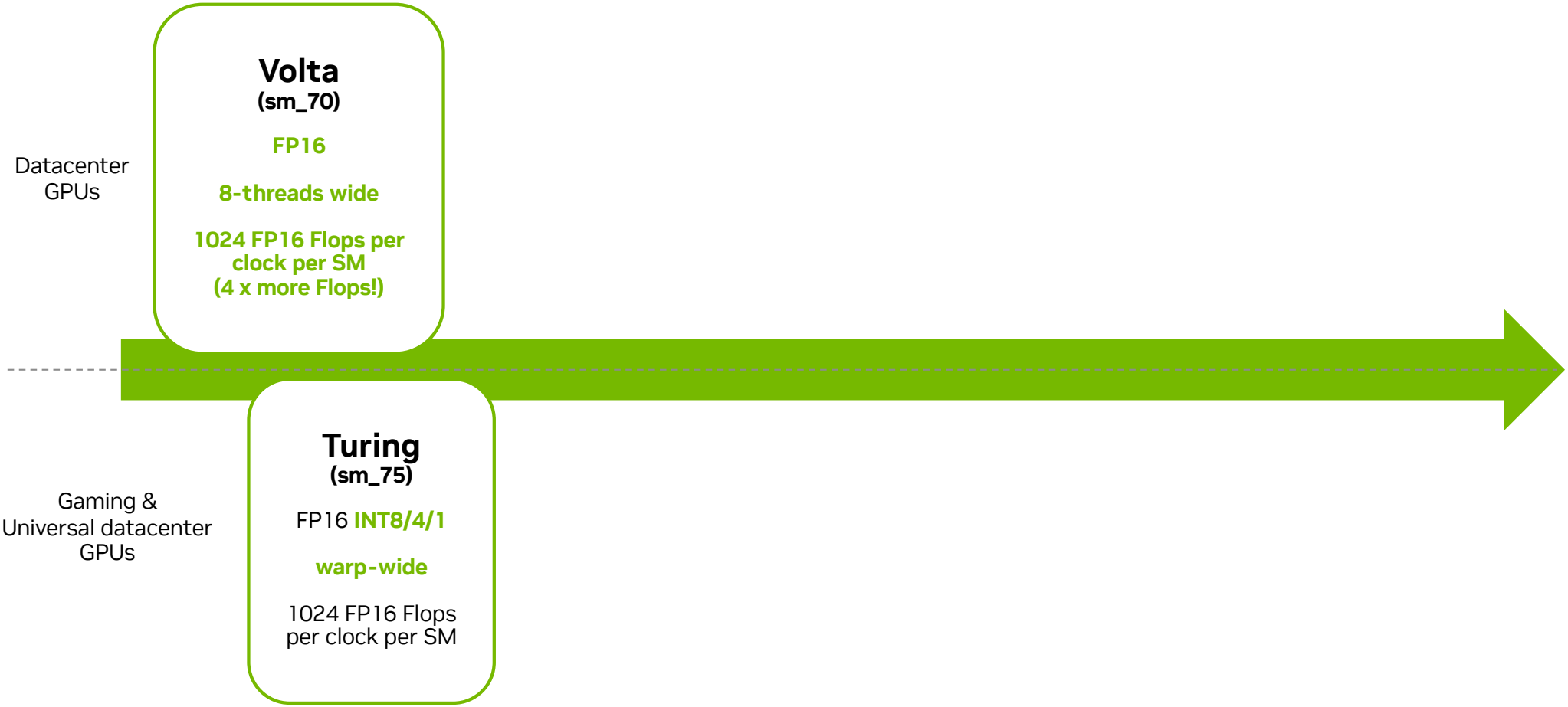
A = 16 x 8 FP16 input
B = 8 x 8 FP16 input
C/D = 16 x 8 FP16 or FP32 output

The whole warp participates

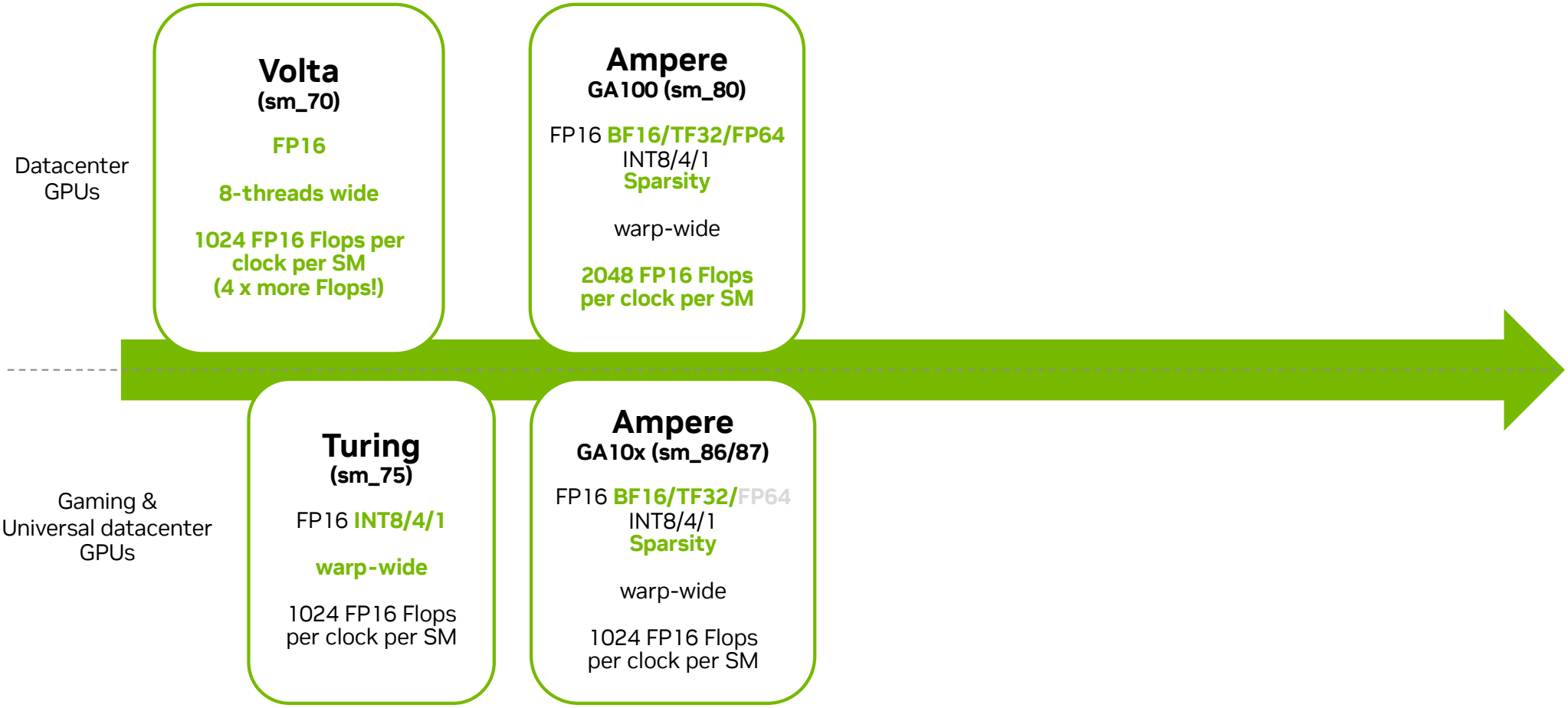
Values are stored in registers

Tensor Core History & Features

Tensor Core History & Features

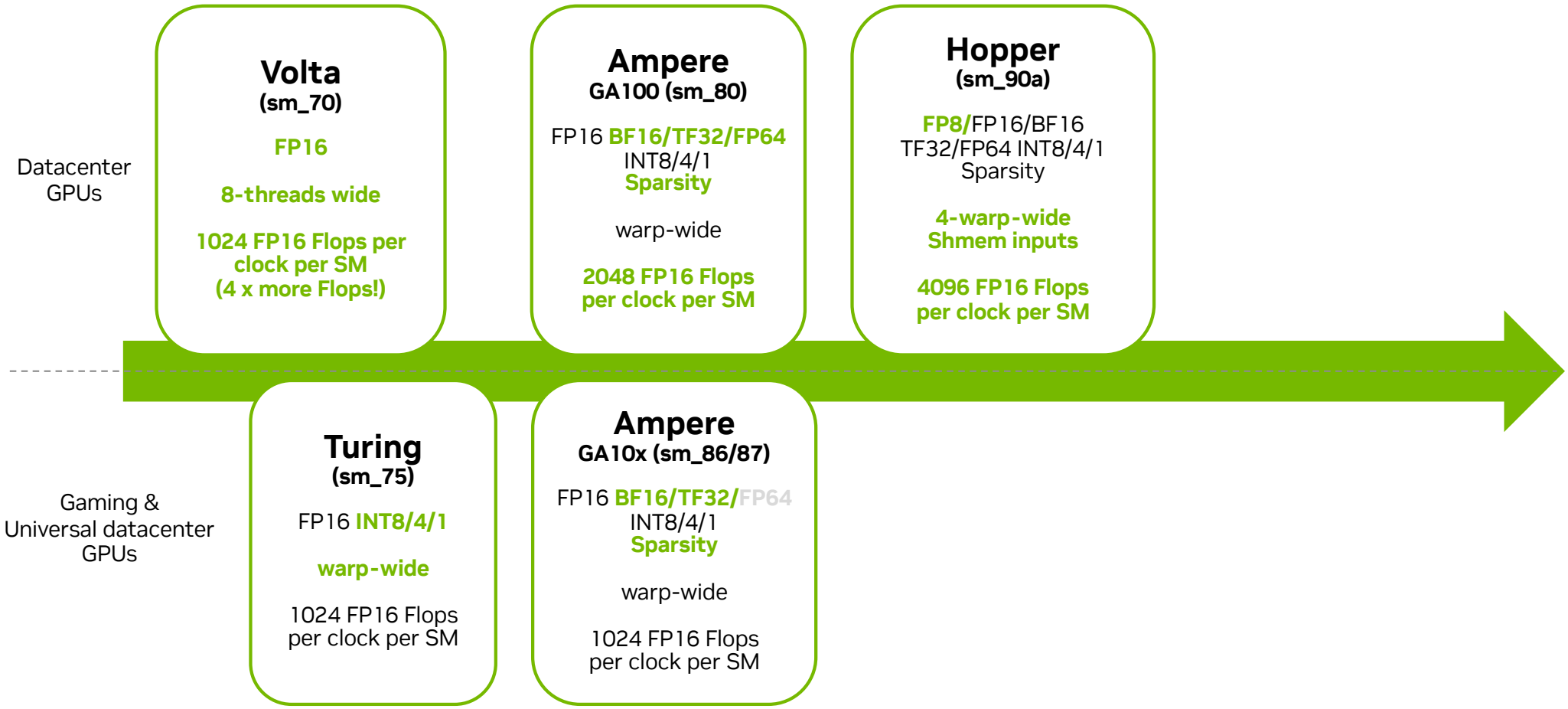


Tensor Core History & Features



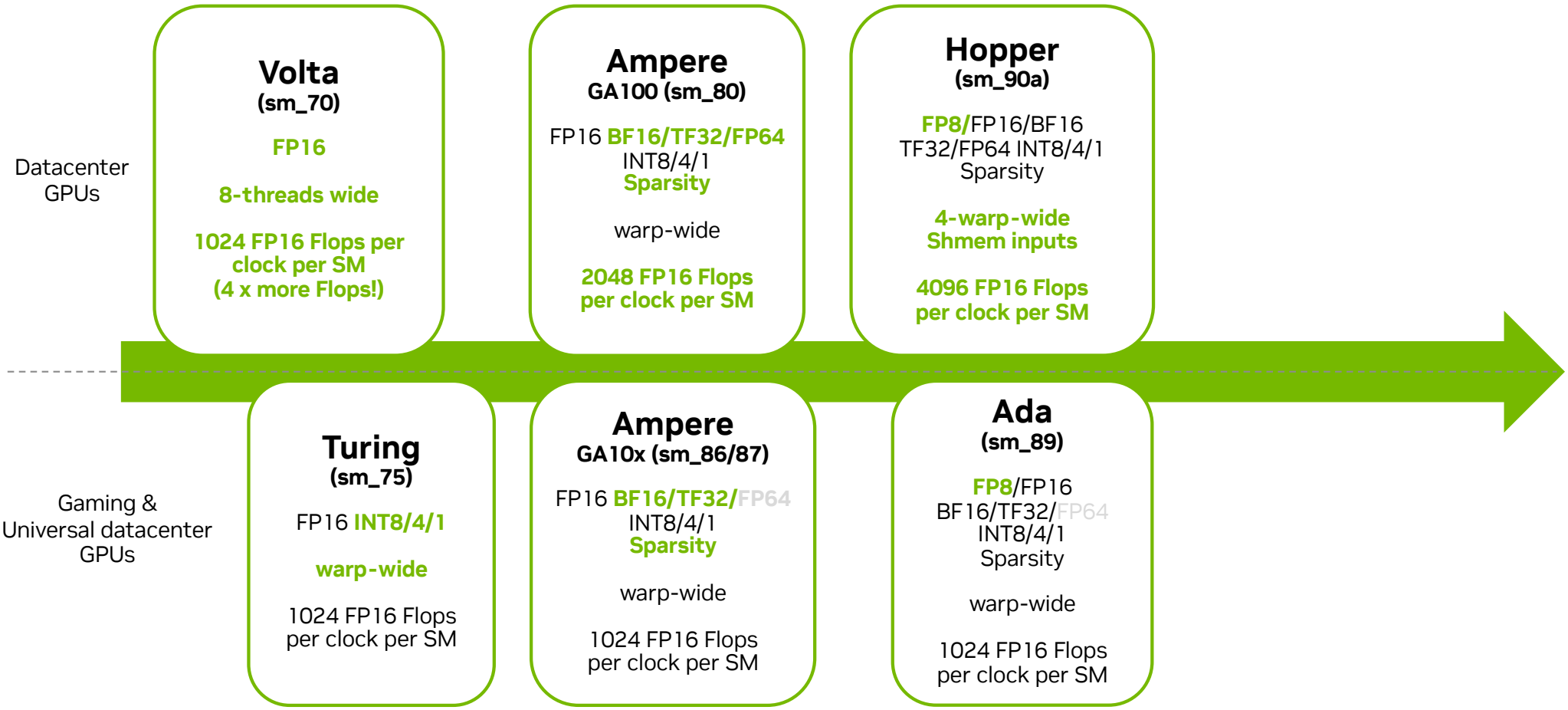
(Peak FP16 Flops per clock per SM, without sparsity)

Tensor Core History & Features



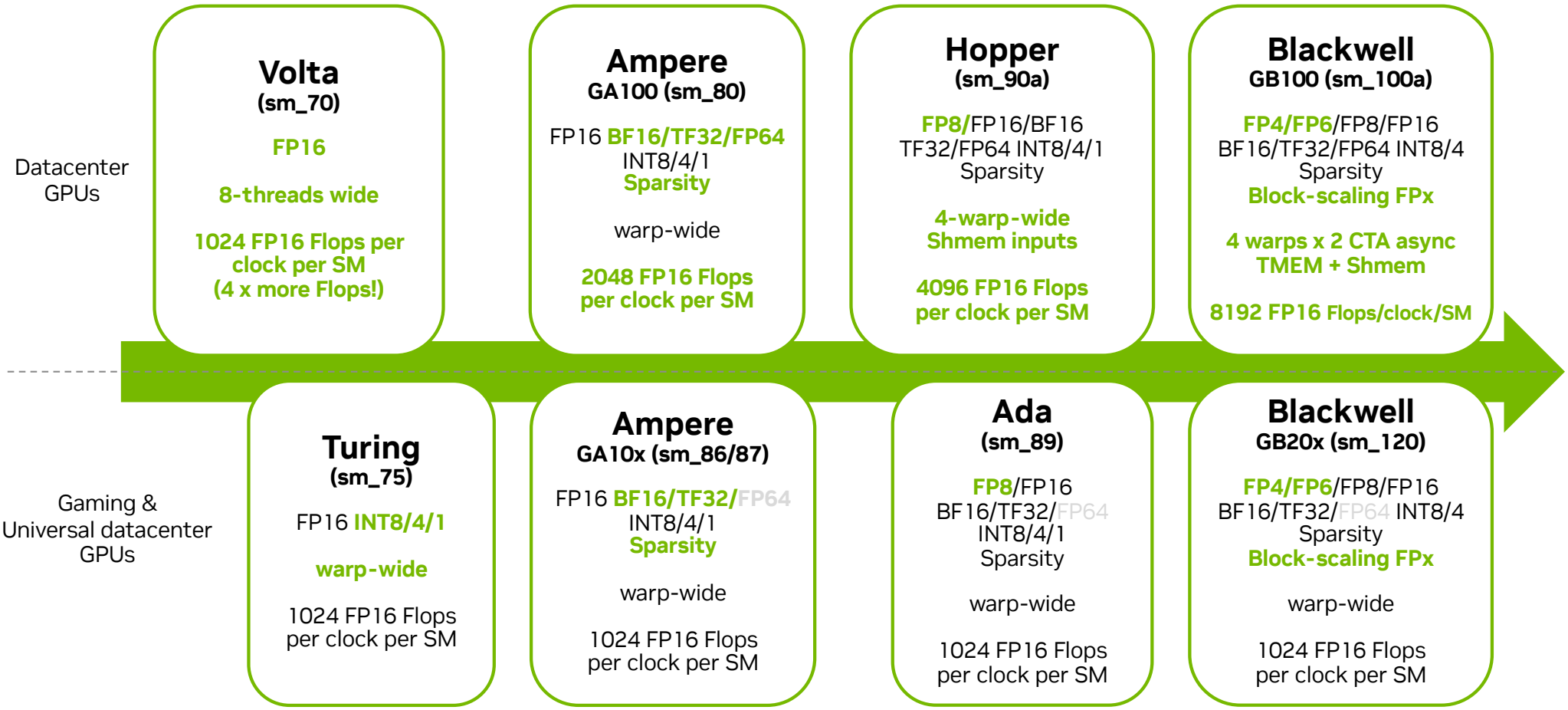
(Peak FP16 Flops per clock per SM, without sparsity)

Tensor Core History & Features



(Peak FP16 Flops per clock per SM, without sparsity)

Tensor Core History & Features

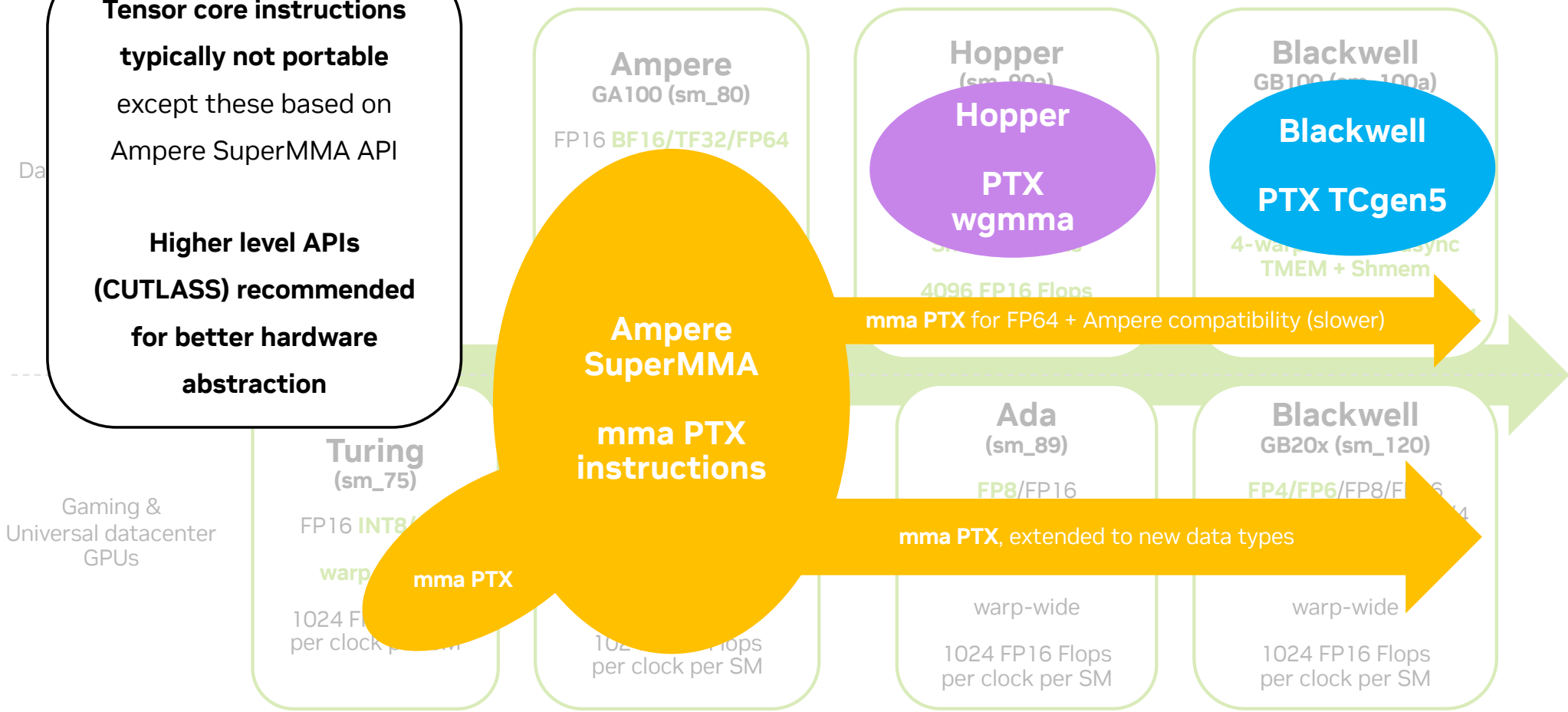


(Peak FP16 Flops per clock per SM, without sparsity)

GPU-specific instructions

Tensor core instructions typically not portable except these based on Ampere SuperMMA API

Higher level APIs (CUTLASS) recommended for better hardware abstraction



Tensor Core providers

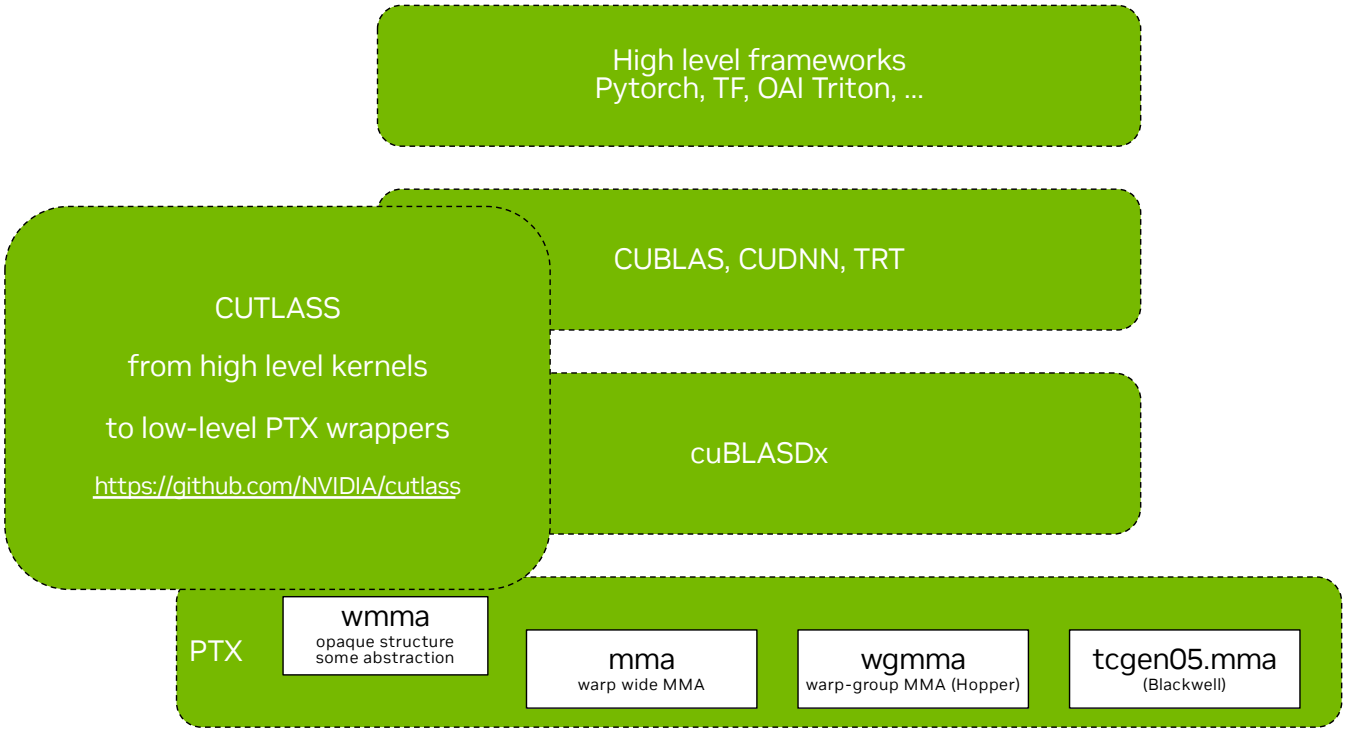
Many different levels to choose from

High level APIs

Easier to program
Better HW abstraction
Harder to get SOL performance

Low level APIs

Harder to program
Can be GPU-specific
Easier to get SOL performance



Tensor Core APIs

Writing your own Tensor Core kernels

- For most cases, CUTLASS should work and provide some HW abstraction and amazing performance

  https://github.com/NVIDIA/cutlass

  Check CUTLASS talks
  - Programming Blackwell Tensor Cores with CUTLASS [S72720]
  - Enable Tensor Core Programming in Python with CUTLASS 4.0 [S74639]

- If you want to use PTX directly, check PTX documentation or check CUTLASS code

  PTX mma for Turing, Ampere, and all “smaller” GPUs

  wgmma on Hopper

  tcgen05.mma on GB100 Blackwell

  - Blackwell TensorCore and Memory Subsystem Optimizations for Applications [CWE72552]

Questions?

+ More CUDA Developer Sessions at GTC 2025

- **General CUDA**

S72571 - What's CUDA All About Anyways?

S72897 - How To Write A CUDA Program: The Parallel Programming Edition

- **CUDA Python**
  - S72450 - Accelerated Python: Tour of the Community and Ecosystem
  - S72448 - The CUDA Python Developer's Toolbox
  - S72449 - 1001 Ways to Write CUDA Kernels in Python
  - S74639 - Enable Tensor Core Programming in Python with CUTLASS 4.0
- **CUDA C++**
  - S72574 - Building CUDA Software at the Speed-of-Light
  - S72572 - The CUDA C++ Developer's Toolbox
  - S72575 - How You Should Write a CUDA C++ Kernel
- **Developer Tools**
  - S72527 - It's Easier than You Think – Debugging and Optimizing CUDA with Intelligent Developer Tools
- **Connect with the Experts**
- CWE72433 - CUDA Developer Best Practices
- CWE73310 - Using NVIDIA CUDA Compiler Tool Chain for Productive GPGPU Programming
- CWE72393 - What's in Your Developer Toolbox? CUDA and Graphics Profiling, Optimization, and Debugging Tools
- CWE75384 – Connect with Dr. Wen-mei Hwu, Author of *Programming Massively Parallel Processors*

Multi-GPU Programming

S72576 - Getting Started with Multi-GPU Scaling: Distributed Libraries

S72579 - Going Deeper with Multi-GPU Scaling: Task-based Runtimes

S72578 - Advanced Multi-GPU Scaling: Communication Libraries

Performance Optimization

S72683 - CUDA Techniques to Maximize Memory Bandwidth and Hide Latency

S72685 - CUDA Techniques to Maximize Compute and Instruction Throughput

S72686 - CUDA Techniques to Maximize Concurrency and System Util.

S72687 - Get the Most Performance from Grace Hopper

nvidia.com/gtc/sessions/cuda-developer