Advanced Topics on High Performance Computing (Part 1)

Johnson Sun | Solution Architect | NVIDIA AI Technology Center (NVAITC)

Resources to Follow Along

Slides, Code, and Profile Results

- Make sure you :

  (1) Can access the slides

  (2) Install Nsight Systems & Nsight Compute

  (3) Download sample profile results

https://github.com/j3soon/advanced-topics-on-hpc-supp

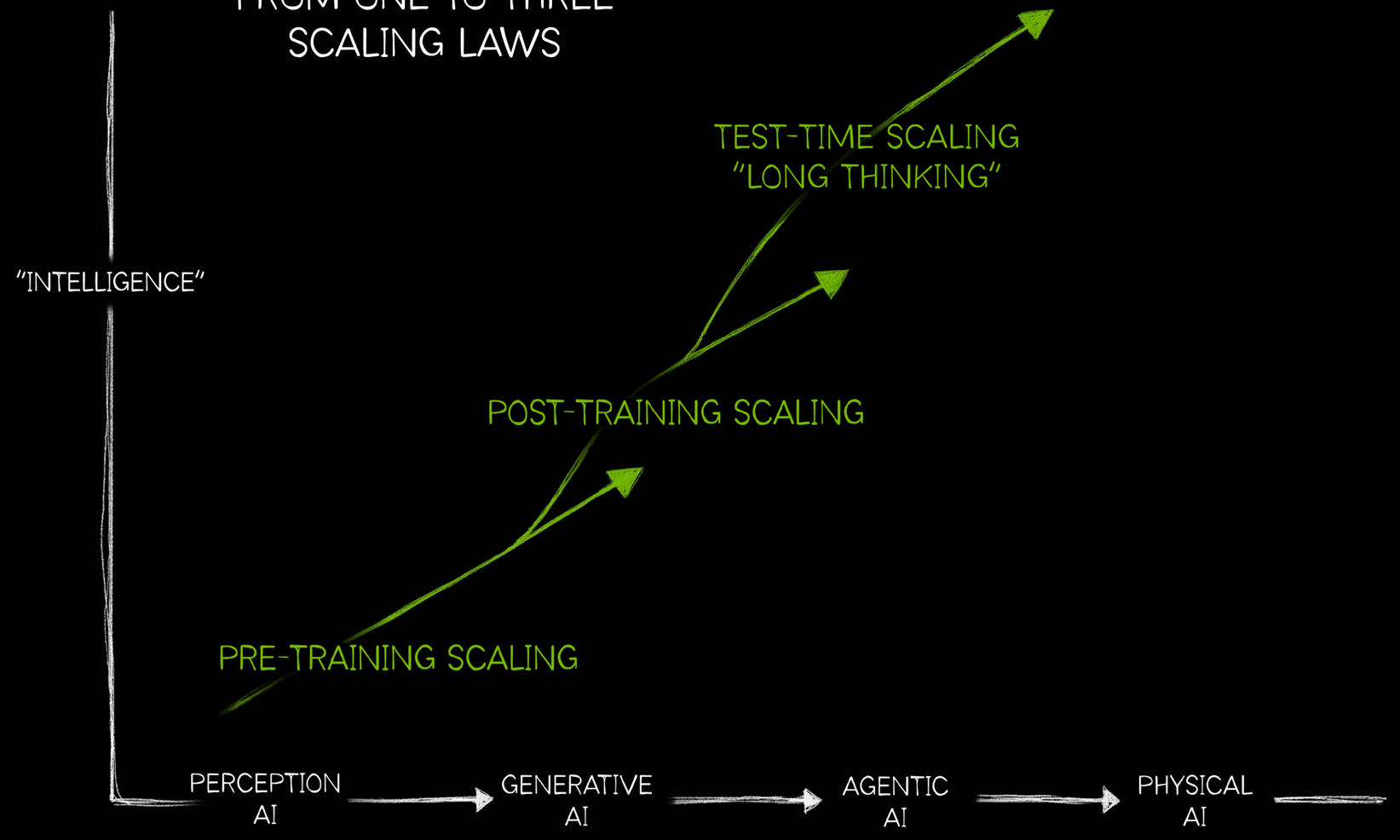
Outline

Topics that will be covered today

- Part 1
  - Current and Emerging GPU Trends
  - CUDA Binaries and Developer Tools
  - CUDA Optimization Basics
  - Hands-On: Analyzing Nsight Profiling Results
- Part 2
  - Advanced CUDA Optimization
  - NVIDIA HPC SDK
  - NVIDIA Communication Libraries (HPC-X)
- Supplementary Materials

Current and Emerging GPU Trends

FROM ONE TO THREE SCALING LAWS

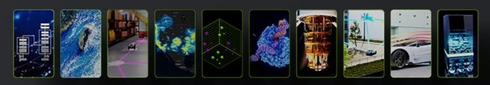


NVIDIA BUILDS AI INFRASTRUCTURE

One Year Rhythm | Supercluster Scale | Full-Stack | CUDA Everywhere

NIM
CUDA-Accelerated
Agentic AI Libraries

Omniverse
CUDA-Accelerated
Physical AI Libraries



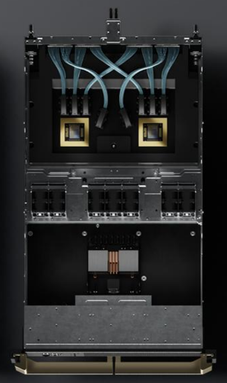
CUDA-X Libraries

CUDA • DOCA • NCCL
Cluster-Scale Software
System Software
Chip Software

Accelerated
Software Stack

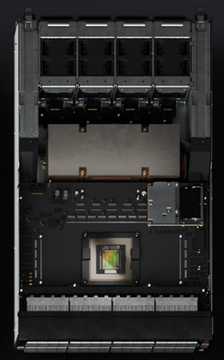
GB200 NVL72 SuperPOD

Grace Blackwell
MGX Node



NVLink Switch

Quantum Switch

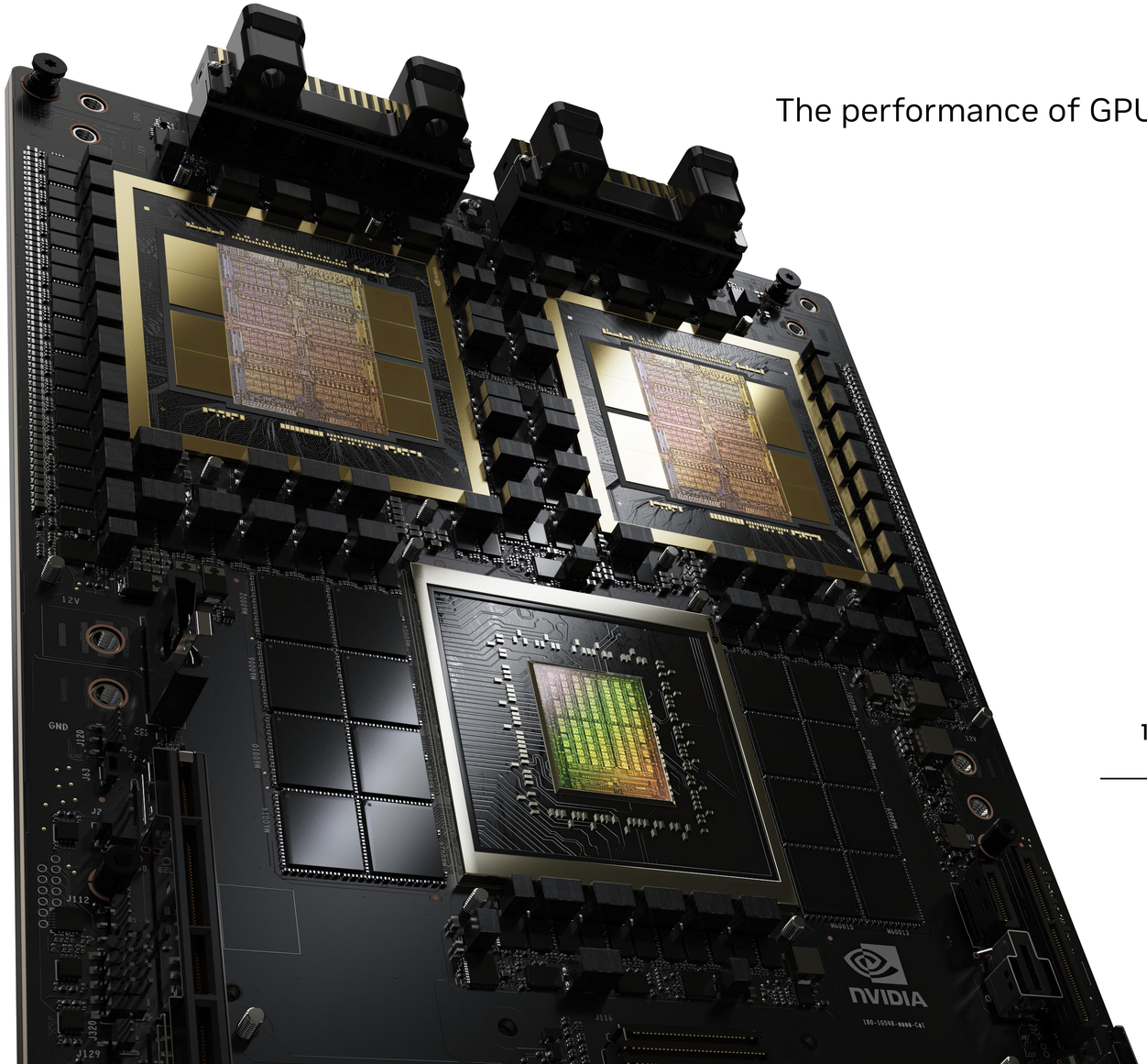


Spectrum-X Switch

Chips Purpose-Built for AI Supercomputing
GPU | CPU | DPU | NIC | NVLink Switch | IB Switch | ENET Switch

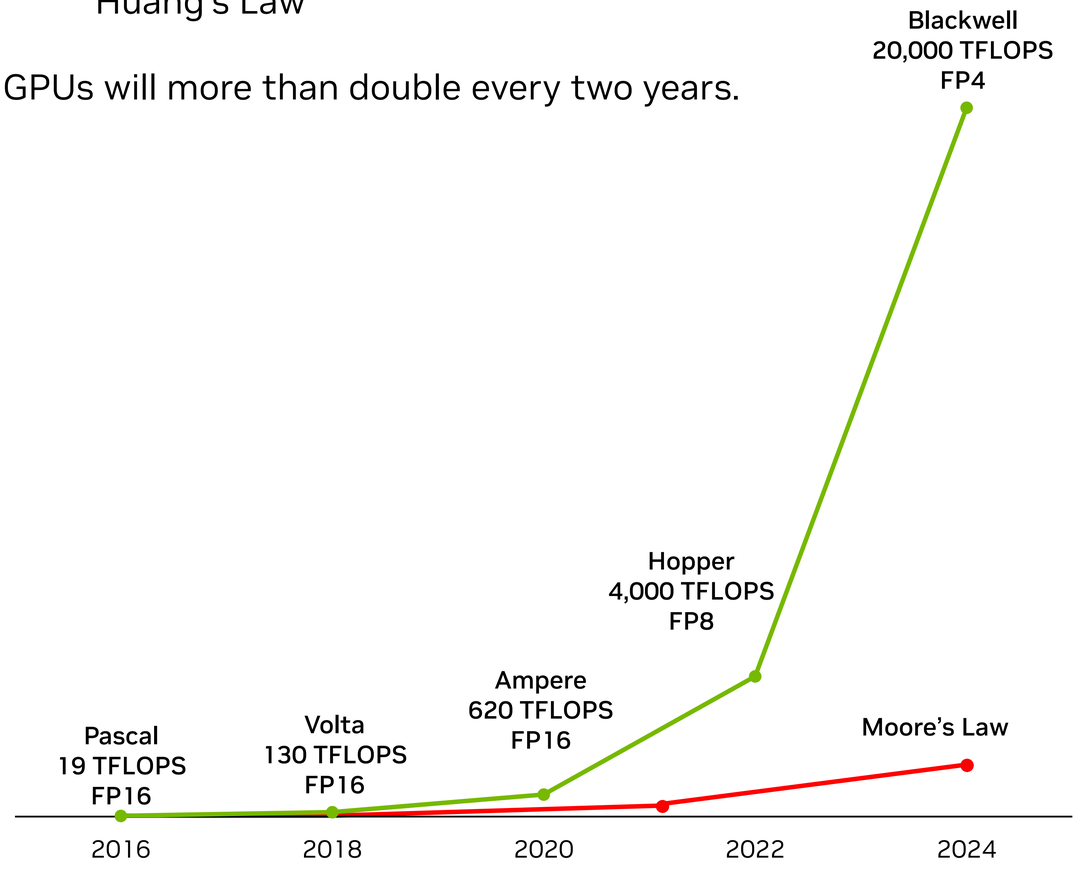
NVIDIA Blackwell Drives Exponential Compute

1,000X AI compute in eight years



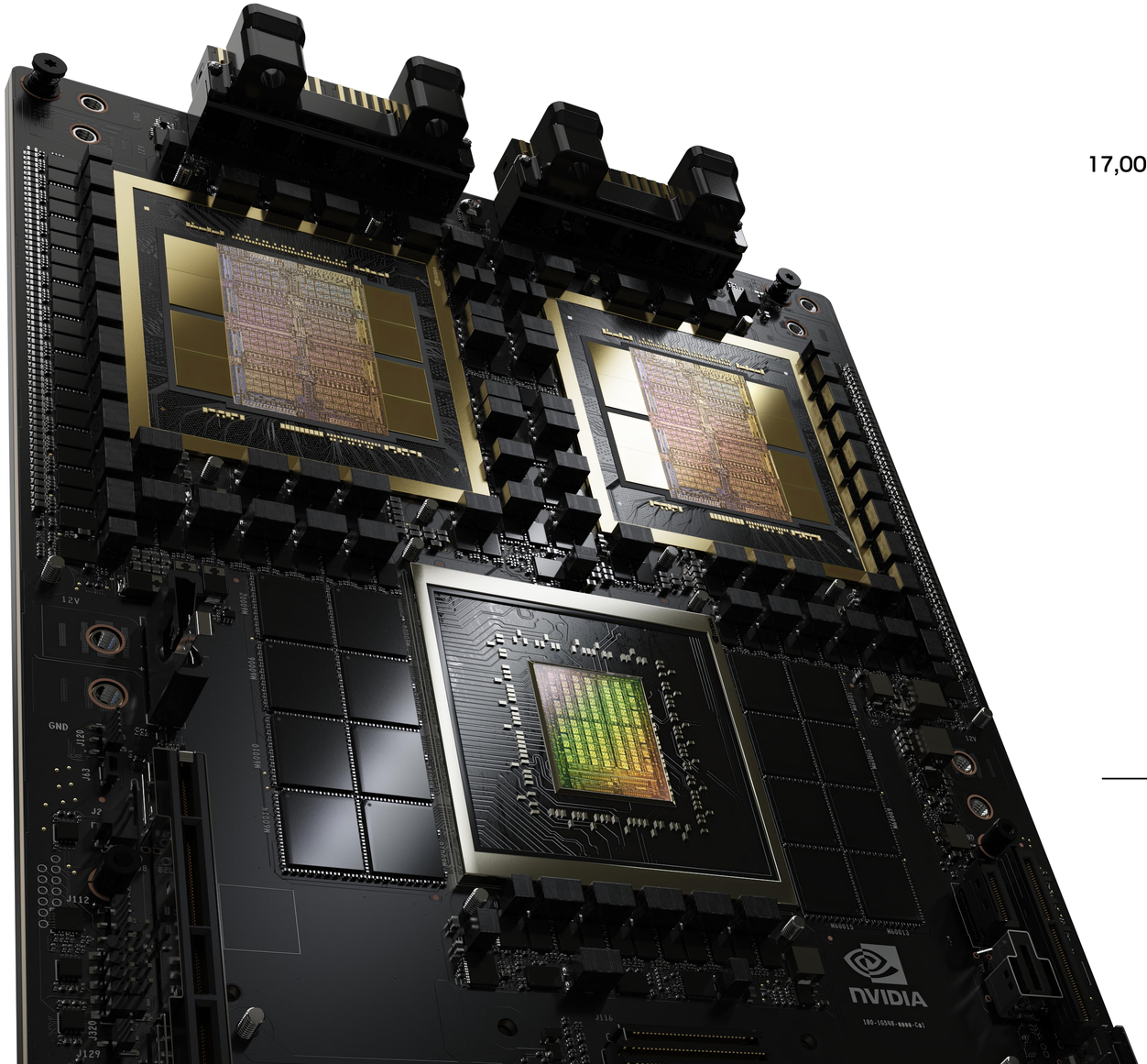
Huang's Law

The performance of GPUs will more than double every two years.



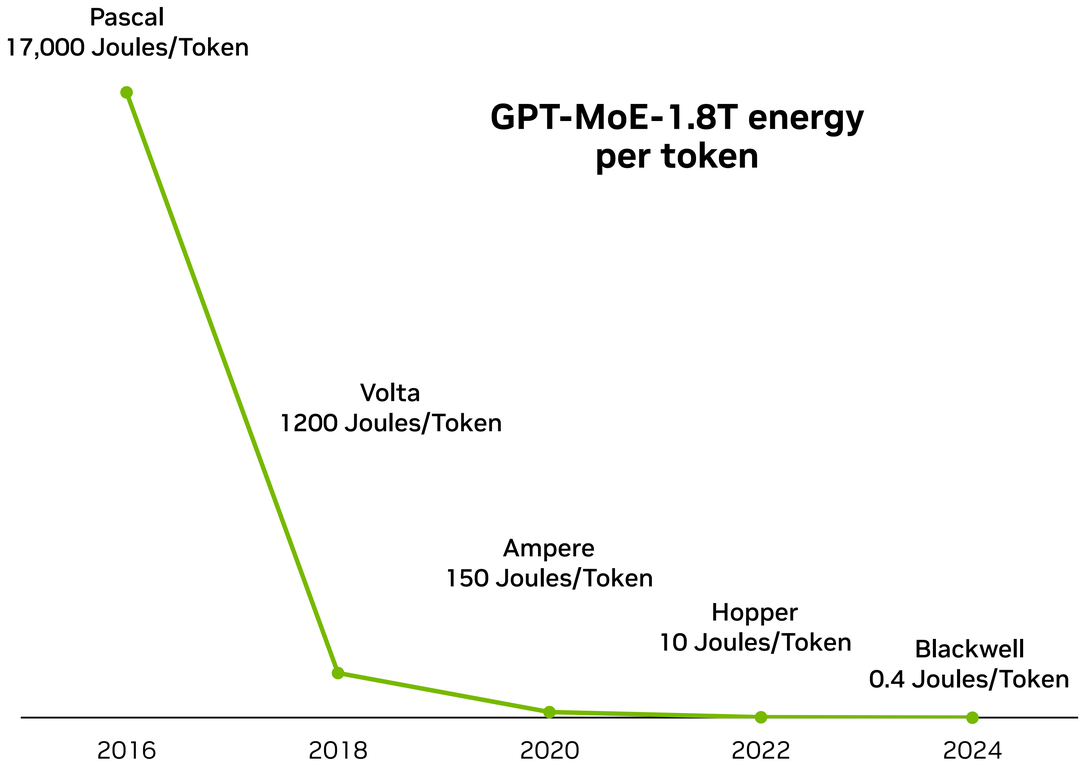
LLM Inference Continues to Get More Energy-Efficient

Energy required for tokens drops 45,000X in eight years



Accelerated Computing is not all about performance

It's about Performance per Watt

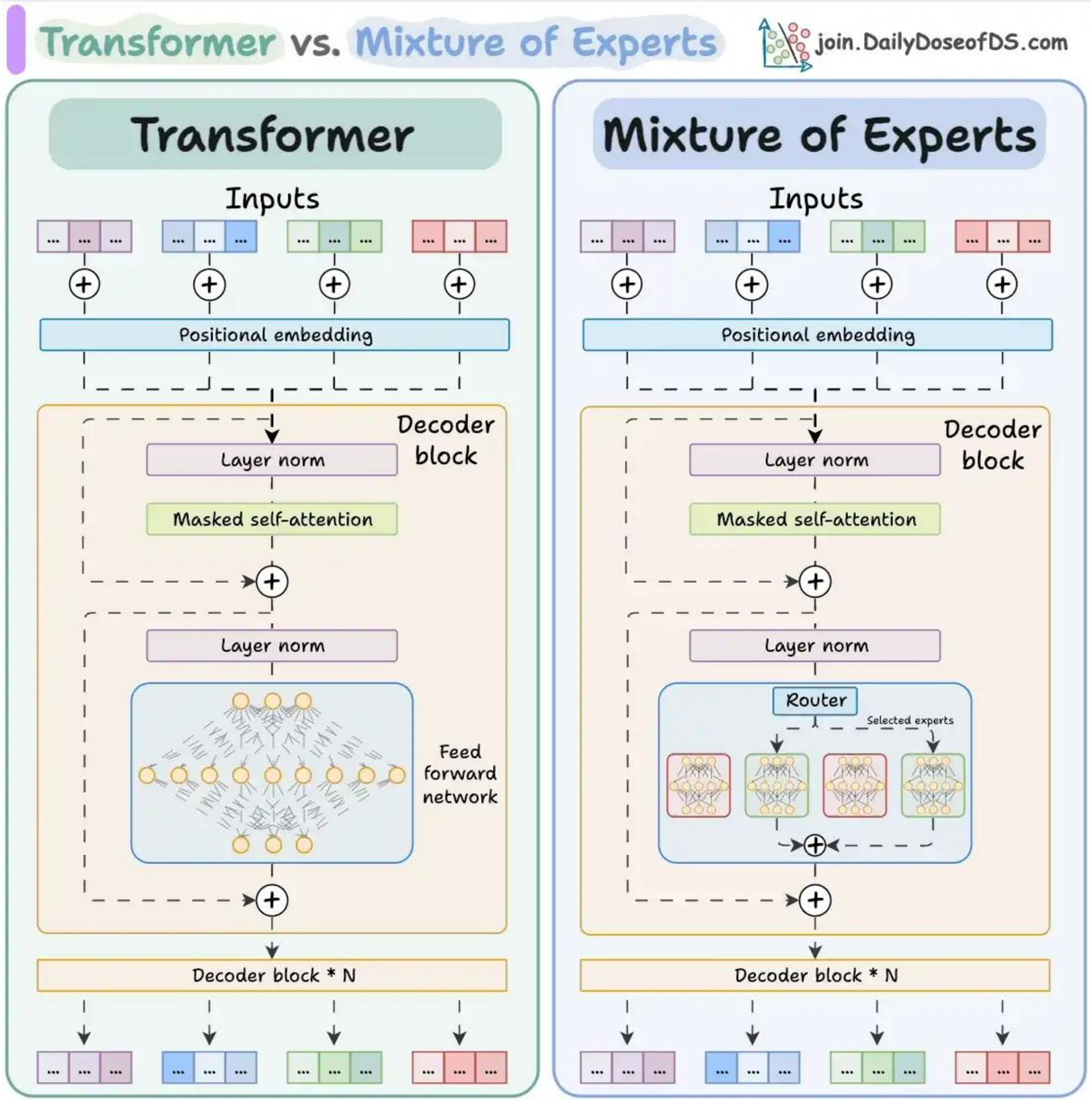


Next Generation MoE Models

Dense Models vs. Mixture of Experts (MoE)

Different parallelism strategies
- Data Parallelism (DP)
- Tensor Parallelism (TP)
- Pipeline Parallelism (PP)
- Expert Parallelism (EP)
- Sequence Parallelism (SP)
- Context Parallelism (CP)

Large models runs on multi-node

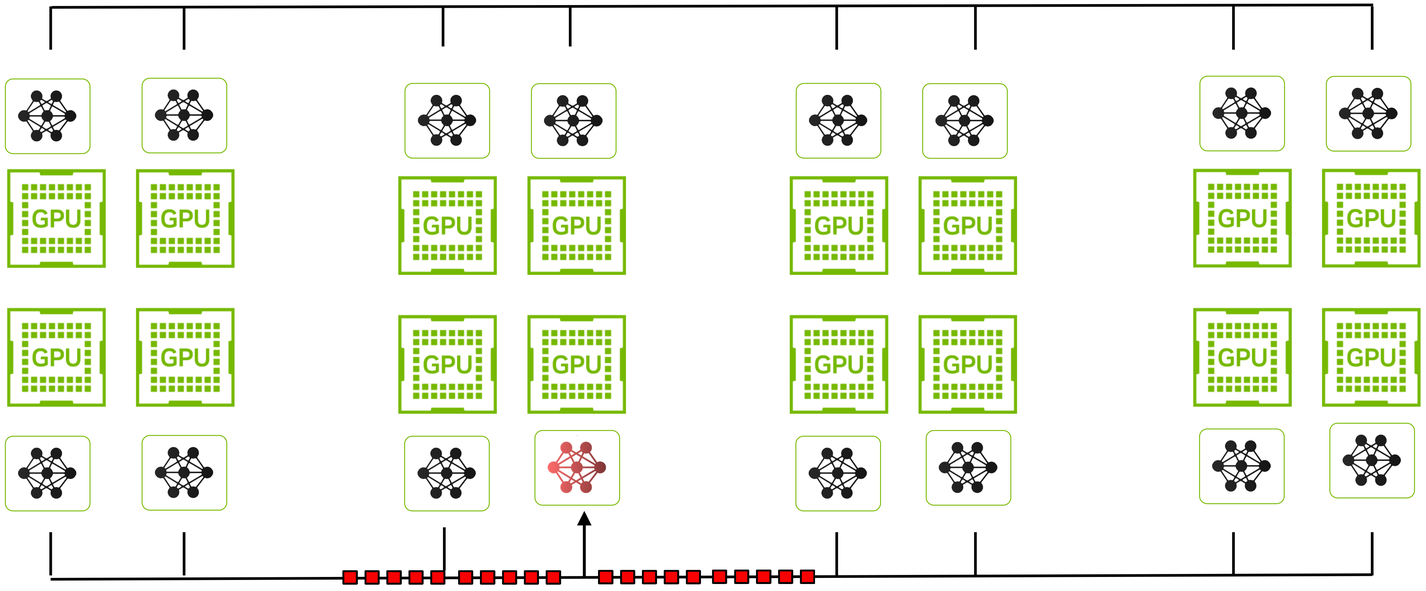


Transformer vs. Mixture of Experts in LLMs

Next Generation Models Communication Bottleneck

Mixture of Expert Models

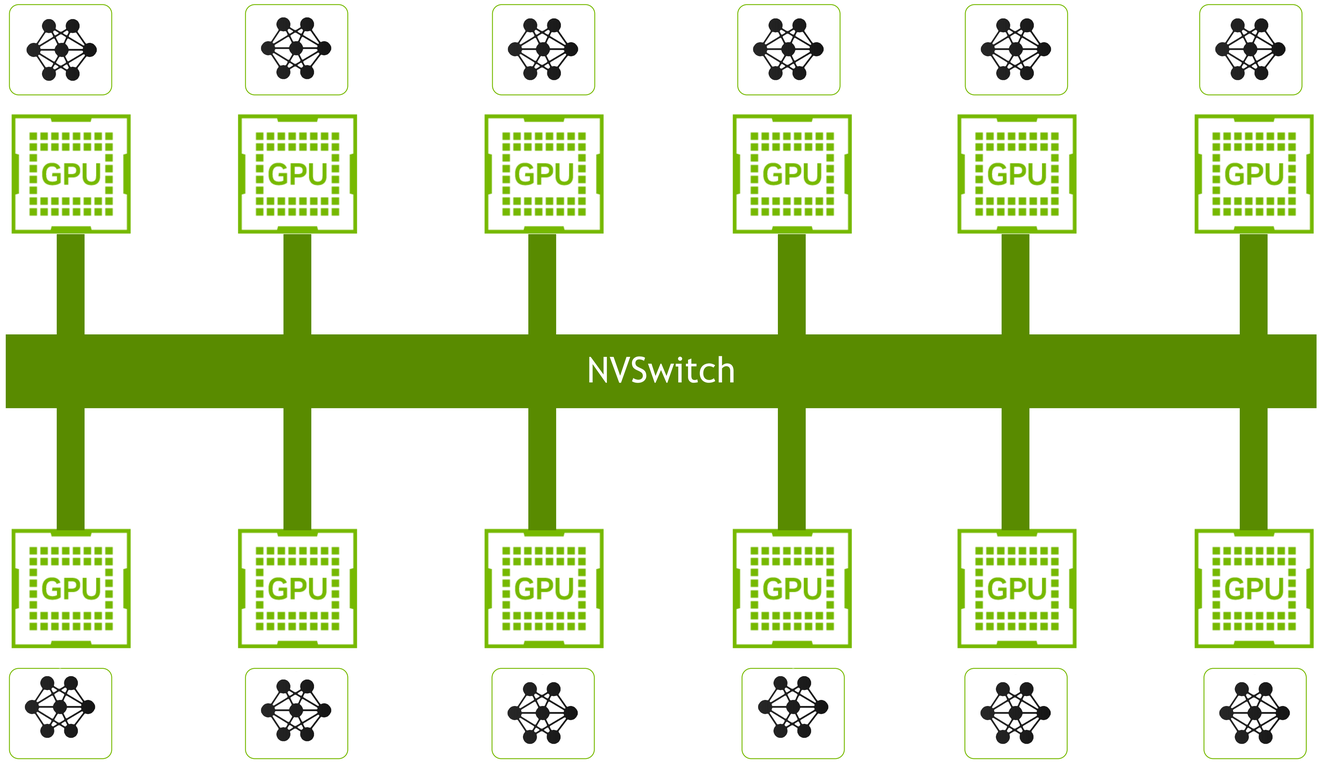
GPT MoE 1.8T Parameters



HDR InfiniBand
100 GB/s

15 GPUs Sending
to 1 GPU

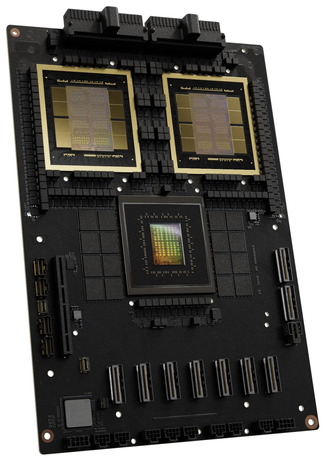
The World Needs a New AI Compute Fabric



NVLink and NVLink Switch
4-th Gen: 900 GB/s (max 8 GPUs)
5-th Gen: 1.8 TB/s (max 576 GPUs)

GB200 NVL72 Compute and Interconnect Nodes

Building Blocks for the GB200 NVL72 Rack



GB200 SUPERCHIP

40 PETAFLOPS FP4 AI INFERENCE
20 PETAFLOPS FP8 AI TRAINING
864GB FAST MEMORY

GB200 SUPERCHIP COMPUTE TRAY

2x GB200
80 PETAFLOPS FP4 AI INFERENCE
40 PETAFLOPS FP8 AI TRAINING
1728 GB FAST MEMORY
1U Liquid Cooled
18 Per Rack

NVLINK SWITCH TRAY

2x NVLINK SWITCH CHIP
14.4 TB/s Total Bandwidth
SHARPv4 FP64/32/16/8
1U Liquid Cooled
9 Per Rack

Announcing GB200 NVL72

Delivers New Unit of Compute

GB200 NVL72

- 36 GRACE CPUs
- 72 BLACKWELL GPUs
- Fully Connected NVLink Switch Rack

Training FP8	720 PFLOPs
Inference FP4	1,440 PFLOPs
NVL Model Size	27T params
Multi-Node All-to-All	130 TB/s
Multi-Node All-Reduce	260 TB/s

DGX SuperPOD

4 Racks of DGX GB200

2 x Management top of rack Ethernet switches

4 x Power shelves

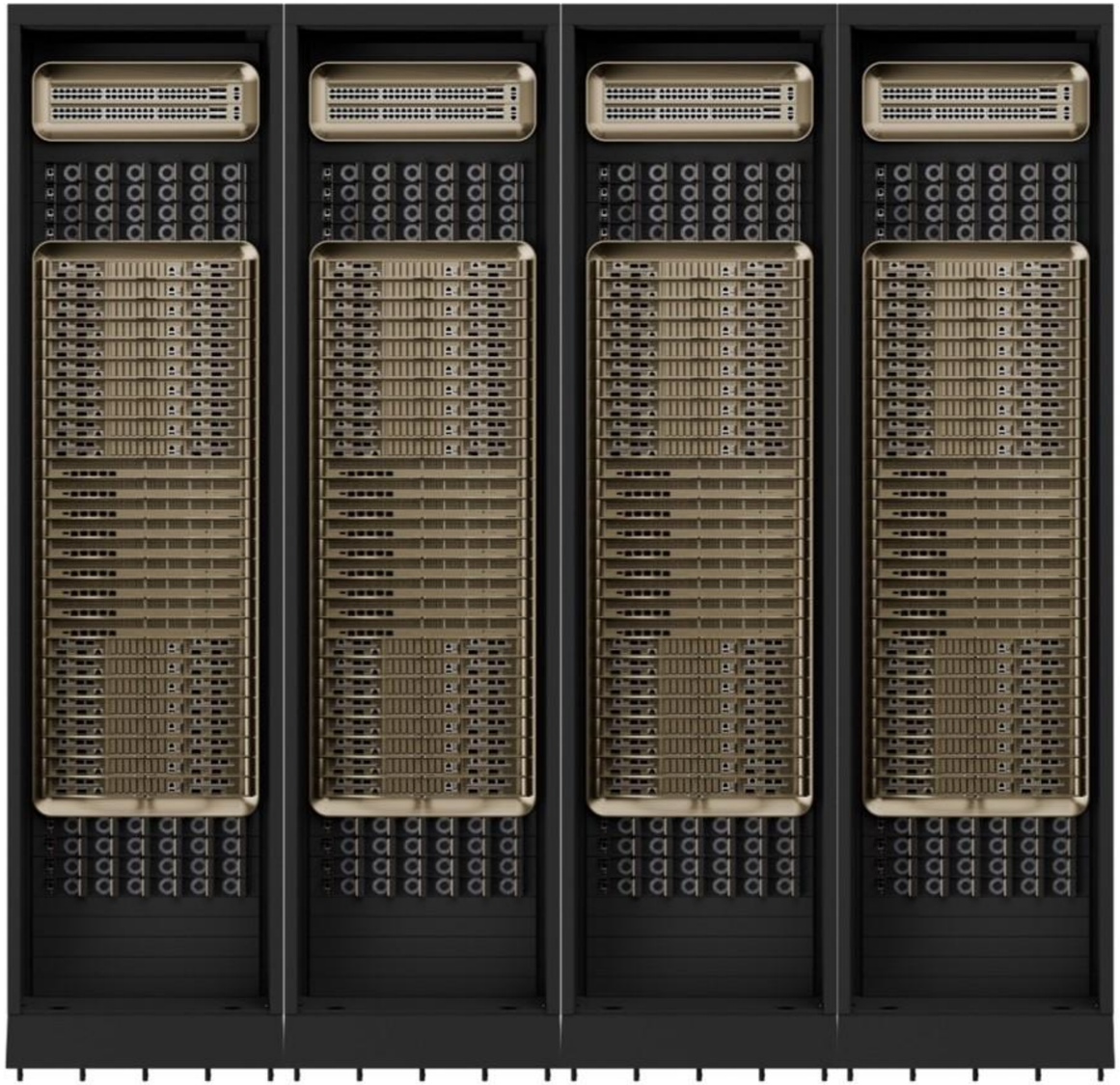
10 x Compute Nodes

9 x NVLink switches

8 x Compute Nodes

4 x Power shelves

Seismic bracing

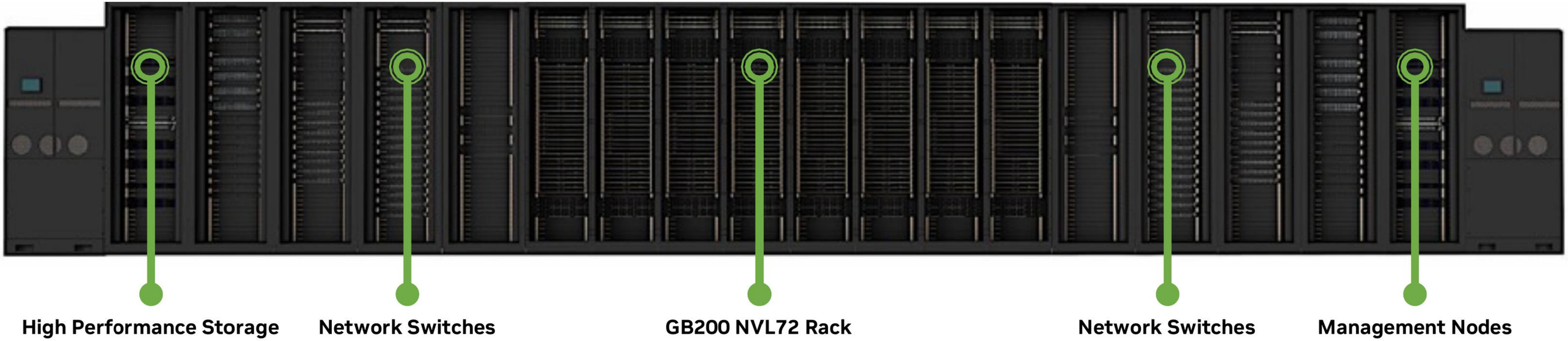


DGX SuperPOD

DGX GB200 Scalable Unit (SU)

- 1 Scalable Unit (SU) = 8 DGX GB200 NVL72 Rack = 144 nodes = 576 GPUs
- Scale up beyond 16 SUs (128 racks = 9216 GPUs)

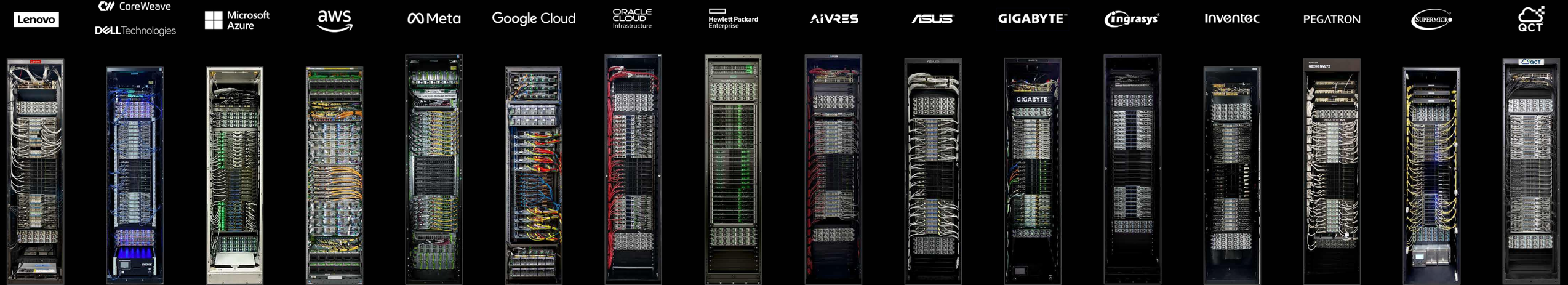
- Complete Rack Layout in One Row:



DGX SuperPod Reference Architecture

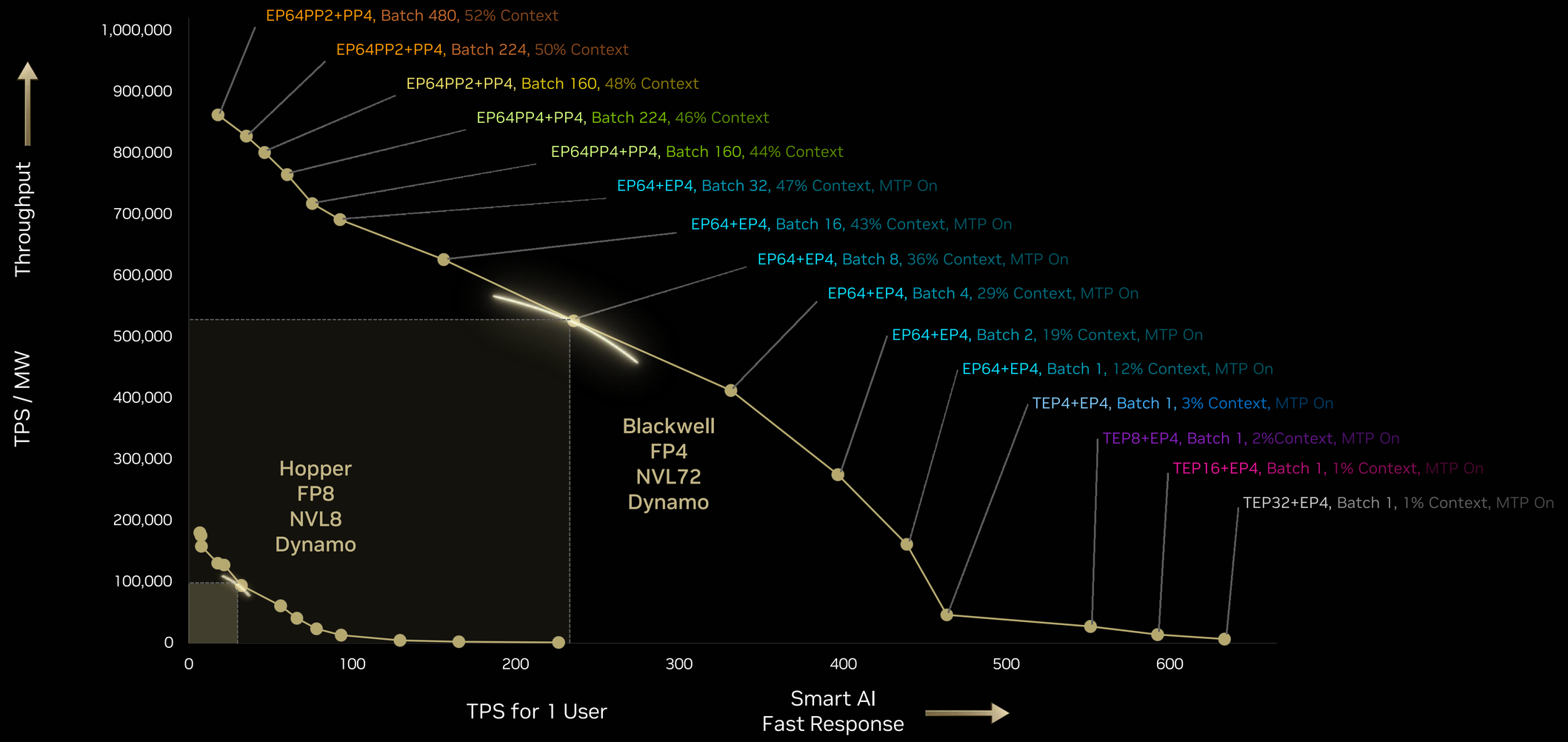
NVIDIA AI Infrastructure for Enterprise Computing

Grace Blackwell in Full Production



Blackwell 40X Hopper

FP4, NVL72, Dynamo, and TRT-LLM Continuous Optimization
32K ISL / 8K OSL



Full FP64 Precision on FP16 Tensor Cores

LU-factorization algorithm[1] using low-precision tensor cores to give an IEEE-754 FP64 accurate result

Input 64-bit matrix A^{64} for LU factorization

Convert precision $A^{64} \rightarrow A^{16}$

LU factorization of A^{16} gives solution x_0

Start by doing the compute-intensive factorization in FP16

Convert x_0 to 64-bit precision

Compute residual $r_i = b - Ax_i$ in FP64

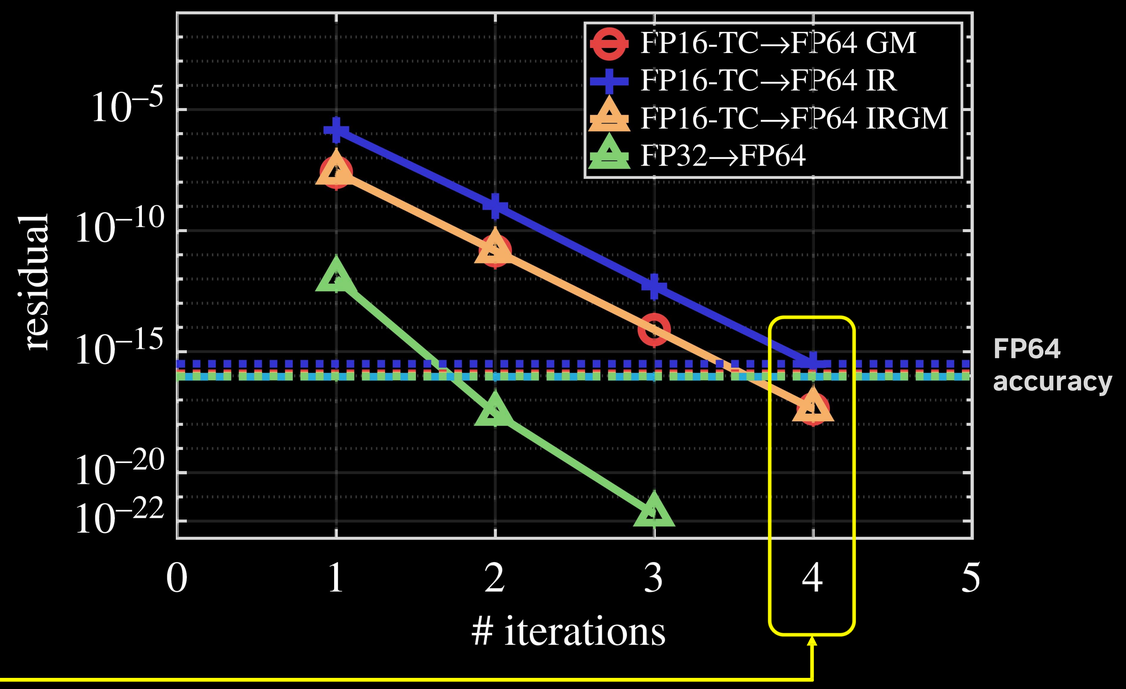
repeat until x_i is accurate enough

Solve $U^{-1}L^{-1}Ac_i = U^{-1}L^{-1}r_i$ by GMRES

Iterative step refines accuracy of result to desired precision

Correct current solution $x_{i+1} = x_i + c_i$ in FP64

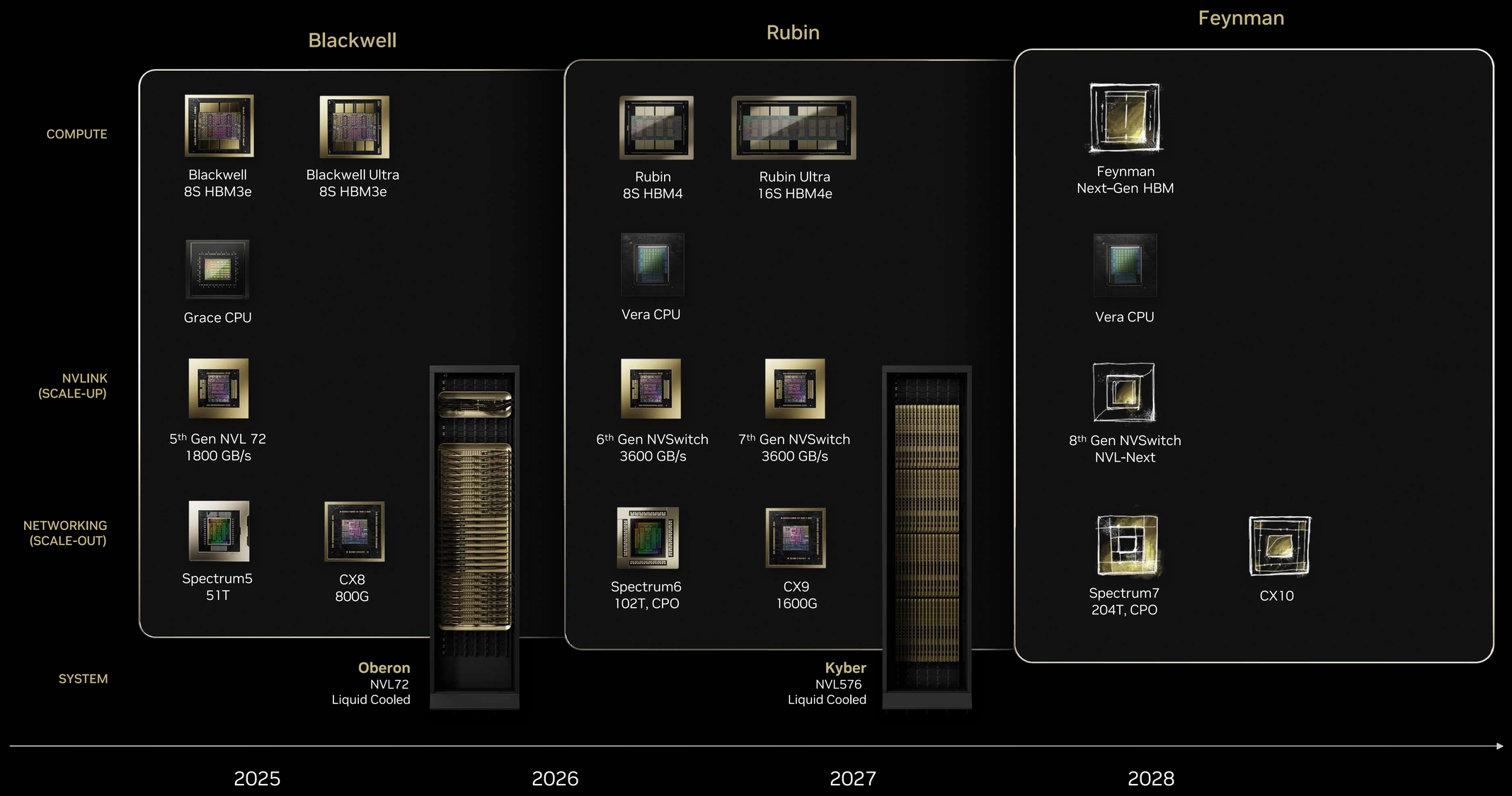
Output is IEEE-754 accurate FP64 result



[1] Haidar Azzam, Bayraktar Harun, Tomov Stanimire, Dongarra Jack and Higham Nicholas J., 2020, Mixed-precision iterative refinement using tensor cores on GPUs to accelerate solution of linear systems, Proc. R. Soc. A.476202001102020011 0 https://royalsocietypublishing.org/doi/10.1098/rspa.2020.0110

NVIDIA Paves Road to Gigawatt AI Factories

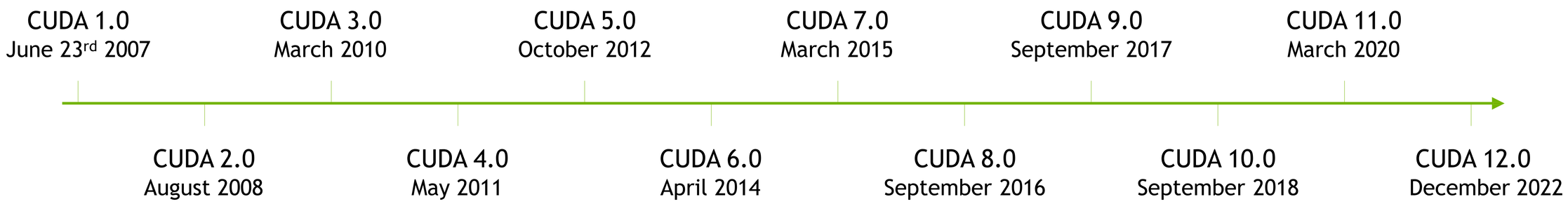
One-Year Rhythm | Full-Stack | One Architecture | CUDA Everywhere



Current and Emerging GPU Trends (Recap)

- Scaling Laws, NV Stack
- Huang’s Law, Energy Efficiency
- MoE, NV Switch, (MN-)NVLink
- GB200 NVL72, DGX SuperPod
- Pareto Frontier, Low-Precision

CUDA Binaries and Developer Tools


CUDA 1.0
June 23rd 2007
CUDA 2.0
August 2008
CUDA 3.0
March 2010
CUDA 4.0
May 2011
CUDA 5.0
October 2012
CUDA 6.0
April 2014
CUDA 7.0
March 2015
CUDA 8.0
September 2016
CUDA 9.0
September 2017
CUDA 10.0
September 2018
CUDA 11.0
March 2020
CUDA 12.0
December 2022

CUDA MAINTAINS BACKWARDS COMPATIBILITY

Backwards Compatibility

Every **new** version of CUDA can run any **older** CUDA program

Every **new** GPU can run any **older** GPU program

CUDA 1.0 June 23rd 2007	CUDA 3.0 March 2010	CUDA 5.0 October 2012	CUDA 7.0 March 2015	CUDA 9.0 September 2017	CUDA 11.0 March 2020
CUDA 2.0 August 2008	CUDA 4.0 May 2011	CUDA 6.0 April 2014	CUDA 8.0 September 2016	CUDA 10.0 September 2018	CUDA 12.0 December 2022

ANATOMY OF A CUDA BINARY

Hello world example

```
__global___void kernel() {
    printf("Hello, CUDA\n");
}
```

GPU code section

```
void main() {
    kernel<<< 1, 1 >>>();
    cudaDeviceSynchronize();
}
```

CPU code section

```
nvcc example.cu -o example
```

ANATOMY OF A CUDA BINARY

Hello world example

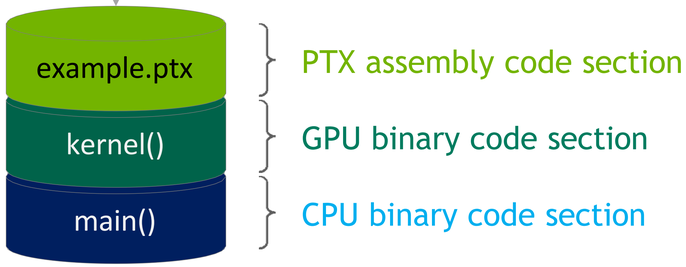
```
__global___void kernel() {
    printf("Hello, CUDA\n");
}
```

GPU code section

```
void main() {
    kernel<<< 1, 1 >>>();
    cudaDeviceSynchronize();
}
```

CPU code section

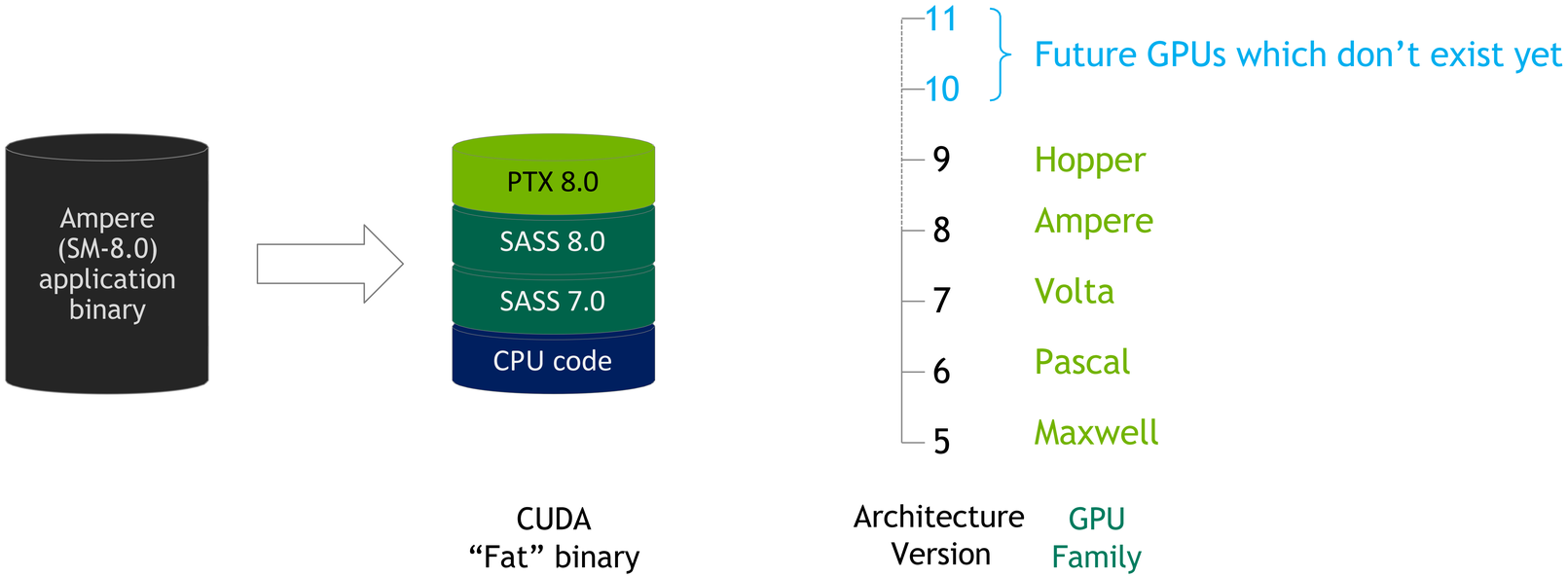
```
nvcc example.cu -o example
```



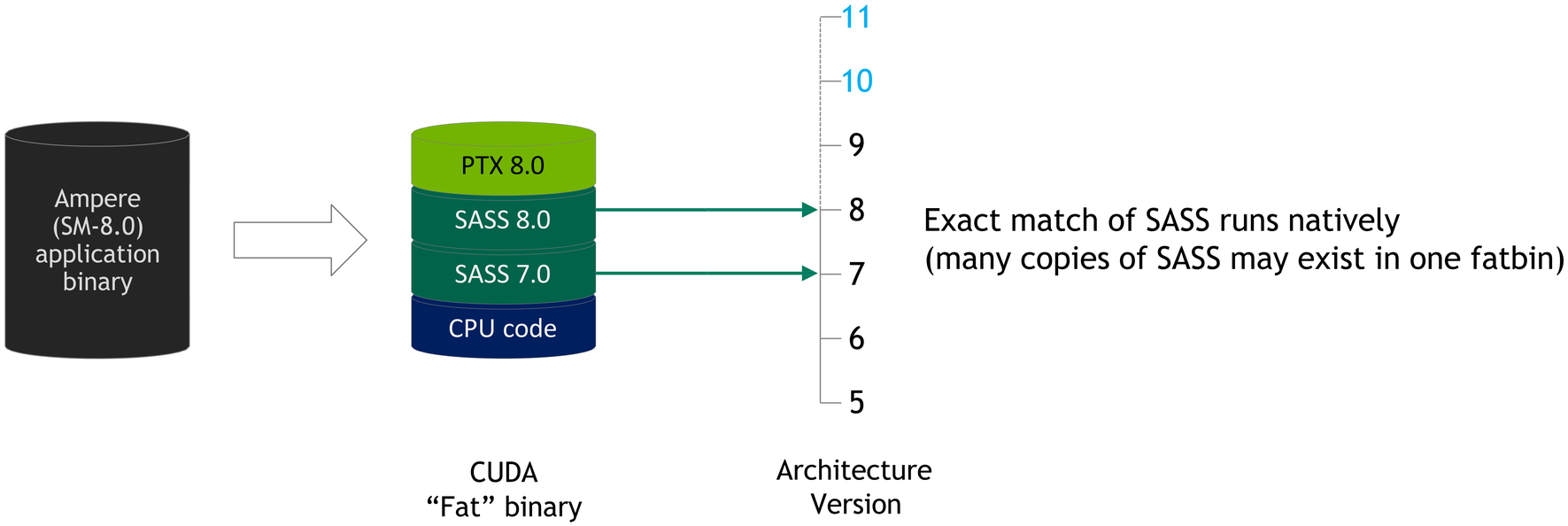
CUDA
"Fat" binary

Architecture Version	GPU Family
11	Future GPUs which don’t exist yet
10	
9	Hopper
8	Ampere
7	Volta
6	Pascal
5	Maxwell

Actually, we currently have compute capabilities 10–12 and are on CUDA 13, but let’s pretend Blackwell hasn't been released yet.
https://developer.nvidia.com/cuda-gpus

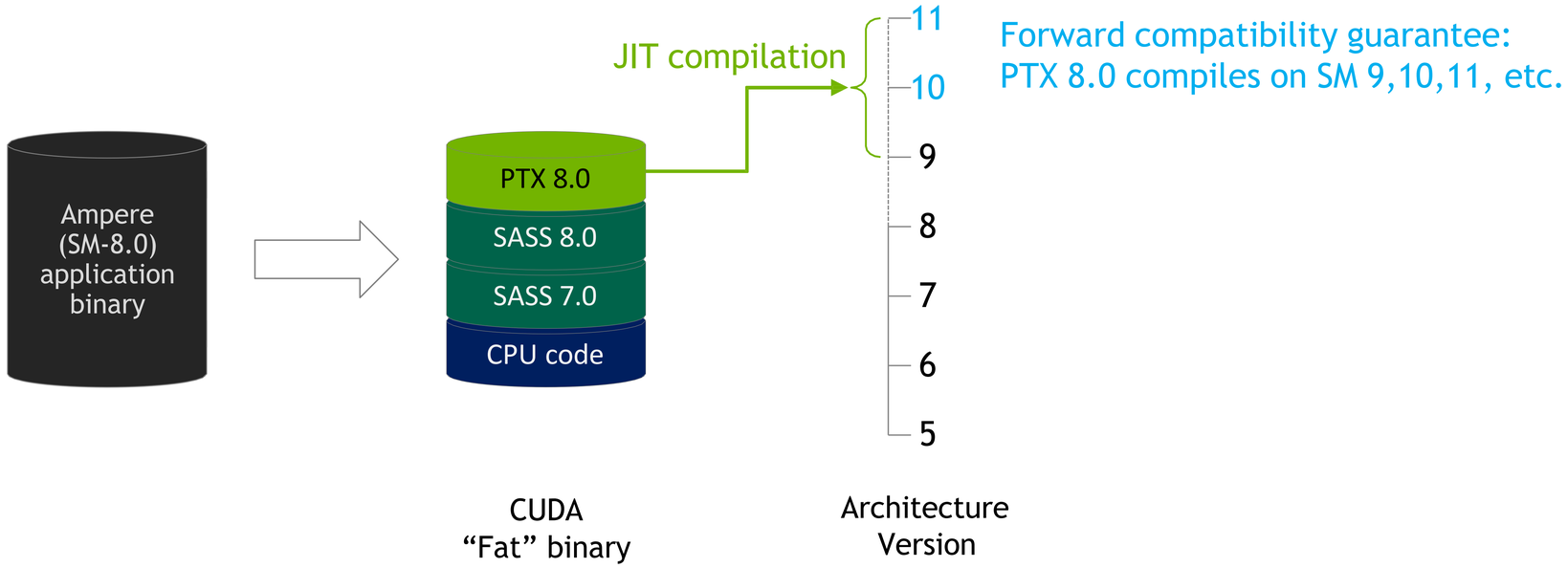


PTX (Parallel Thread Execution)
SASS (Streaming ASSembler)



Running this binary on an Ampere (SM-8.0) GPU
or a Volta (SM-7.0) GPU will just work

PORTABILITY VIA PTX JUST-IN-TIME COMPILATION



When running on a GPU for which SASS does not exist,
CUDA's built-in PTX compiler recompiles for the new GPU

Enabling Hopper-Specific Optimizations in CUDA Applications [A41147]

CUDA MAINTAINS BACKWARDS COMPATIBILITY

Backwards Compatibility

Older applications run on new hardware because of PTX JIT compilation

CUDA guarantees that all APIs still work, right back to CUDA 1.0

CUDA 1.0	June 23rd 2007
CUDA 2.0	August 2008
CUDA 3.0	March 2010
CUDA 4.0	May 2011
CUDA 5.0	October 2012
CUDA 6.0	April 2014
CUDA 7.0	March 2015
CUDA 8.0	September 2016
CUDA 9.0	September 2017
CUDA 10.0	September 2018
CUDA 11.0	March 2020
CUDA 12.0	December 2022

(cuda-memcheck has been deprecated in CUDA 12.0)

COMPUTE SANITIZER

AUTOMATICALLY SCAN FOR BUGS AND MEMORY ISSUES

- Compute Sanitizer checks correctness issues via sub-tools:
- *Memcheck* – The memory access error and leak detection tool.
- *Racecheck* – The shared memory data access hazard detection tool.
- *Initcheck* – The uninitialized device global memory access detection tool.
- *Synccheck* – The thread synchronization hazard detection tool.

```
~/W/m/c/build $ cmake ../ && cmake --build .
-- Configuring done
-- Generating done
-- Build files have been written to: /home/rmaynard/Work/misc/cuda_sanitizer_ctest/build
[2/2] Linking CUDA executable demo
~/W/m/c/build $ ctest -D MemoryCheck
   Site: RMAYNARD-DT
   Build name: Linux-unknown
Create new tag: 20210325-1346 - Experimental
Configure project
   Each . represents 1024 bytes of output
    . Size of output: 0K
Build project
   Each symbol represents 1024 bytes of output.
   '!' represents an error and '*' a warning.
    . Size of output: 0K
   0 Compiler errors
   0 Compiler warnings
Performing coverage
 Cannot find any coverage files. Ignoring Coverage request.
Memory check project /home/rmaynard/Work/misc/cuda_sanitizer_ctest/build
    Start 1: verify
1/1 MemCheck #1: verify ..........................   Passed    6.77 sec

100% tests passed, 0 tests failed out of 1

Total Test time (real) =   6.77 sec
-- Processing memory checking output:
1/1 MemCheck: #1: verify ..........................   Defects: 4
MemCheck log files can be found here: (<#> corresponds to test number)
/home/rmaynard/Work/misc/cuda_sanitizer_ctest/build/Testing/Temporary/MemoryChecker.<#>.log
Memory checking results:
Invalid __global__ read - 1
cudaErrorLaunchFailure - 3
Submit files
   SubmitURL: http://my.cdash.org/submit.php?project=CMakeTutorial
   Uploaded: /home/rmaynard/Work/misc/cuda_sanitizer_ctest/build/Testing/20210325-1346/Config
   Uploaded: /home/rmaynard/Work/misc/cuda_sanitizer_ctest/build/Testing/20210325-1346/Build.
   Uploaded: /home/rmaynard/Work/misc/cuda_sanitizer_ctest/build/Testing/20210325-1346/Dynami
   Uploaded: /home/rmaynard/Work/misc/cuda_sanitizer_ctest/build/Testing/20210325-1346/Done.x
   Submission successful
~/W/m/c/build $
```

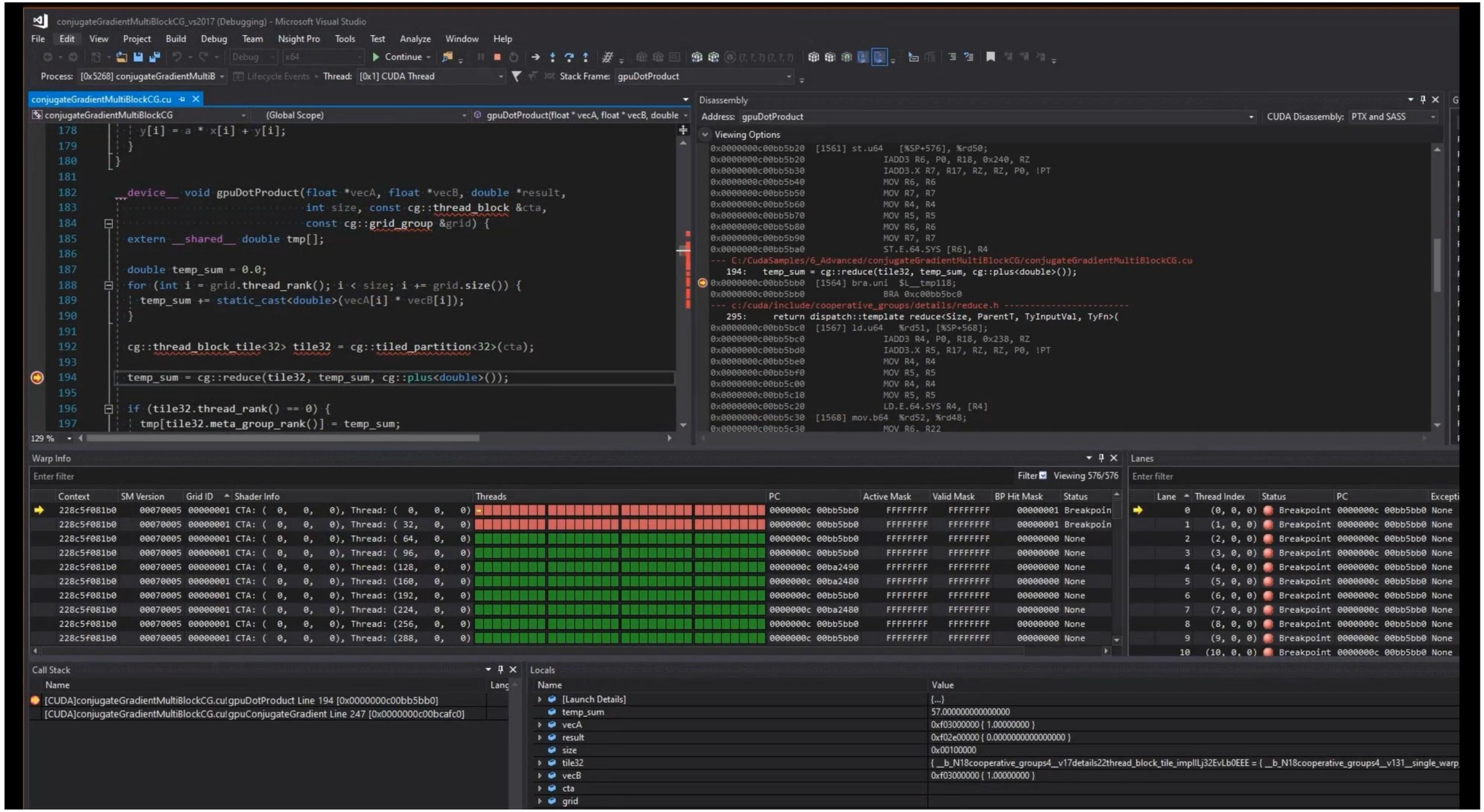
CUDA GDB

COMMAND LINE AND IDE BACKEND DEBUGGER

- Unified CPU and CUDA Debugging
- CUDA-C/PTX/SASS support
- Built on GDB and uses many of the same CLI commands

```
(cuda-gdb) info cuda threads breakpoint all
  BlockIdx ThreadIdx         Virtual PC Dev SM Wp Ln        Filename  Line
Kernel 0
   (1,0,0)   (0,0,0) 0x0000000000948e58   0 11  0  0 infoCommands.cu    12
   (1,0,0)   (1,0,0) 0x0000000000948e58   0 11  0  1 infoCommands.cu    12
   (1,0,0)   (2,0,0) 0x0000000000948e58   0 11  0  2 infoCommands.cu    12
   (1,0,0)   (3,0,0) 0x0000000000948e58   0 11  0  3 infoCommands.cu    12
   (1,0,0)   (4,0,0) 0x0000000000948e58   0 11  0  4 infoCommands.cu    12
   (1,0,0)   (5,0,0) 0x0000000000948e58   0 11  0  5 infoCommands.cu    12

(cuda-gdb) info cuda threads breakpoint 2 lane 1
  BlockIdx ThreadIdx         Virtual PC Dev SM Wp Ln        Filename  Line
Kernel 0
   (1,0,0)   (1,0,0) 0x0000000000948e58   0 11  0  1 infoCommands.cu    12
```

NSIGHT VISUAL STUDIO EDITION

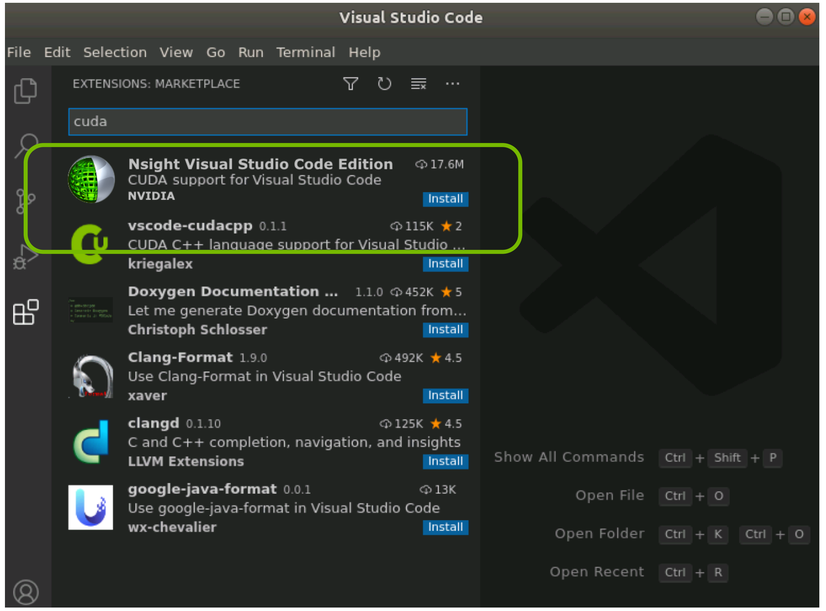


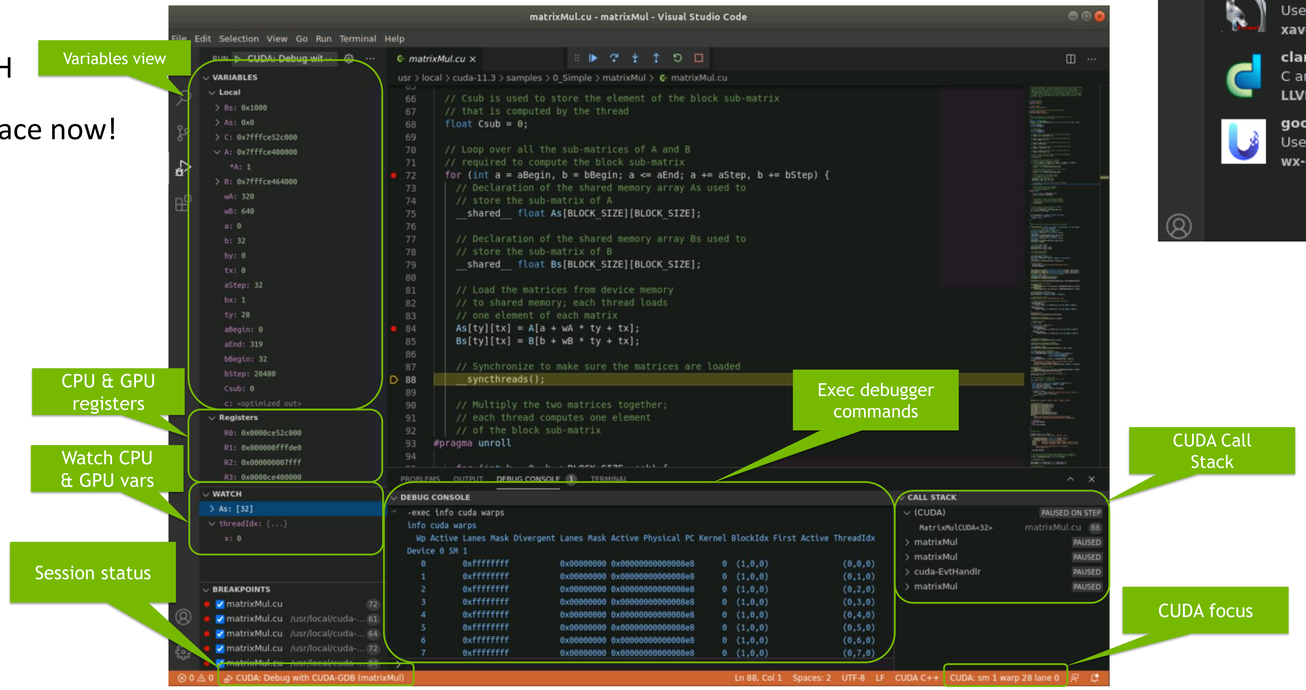
https://developer.nvidia.com/nsight-visual-studio-edition

NSIGHT VISUAL STUDIO CODE EDITION

Visual Studio Code extensions that provides:

- CUDA code syntax highlighting
- CUDA code completion
- Build warning/errors
- Debug CPU & GPU code
- Remote connection support via SSH
- Available on the VS Code Marketplace now!

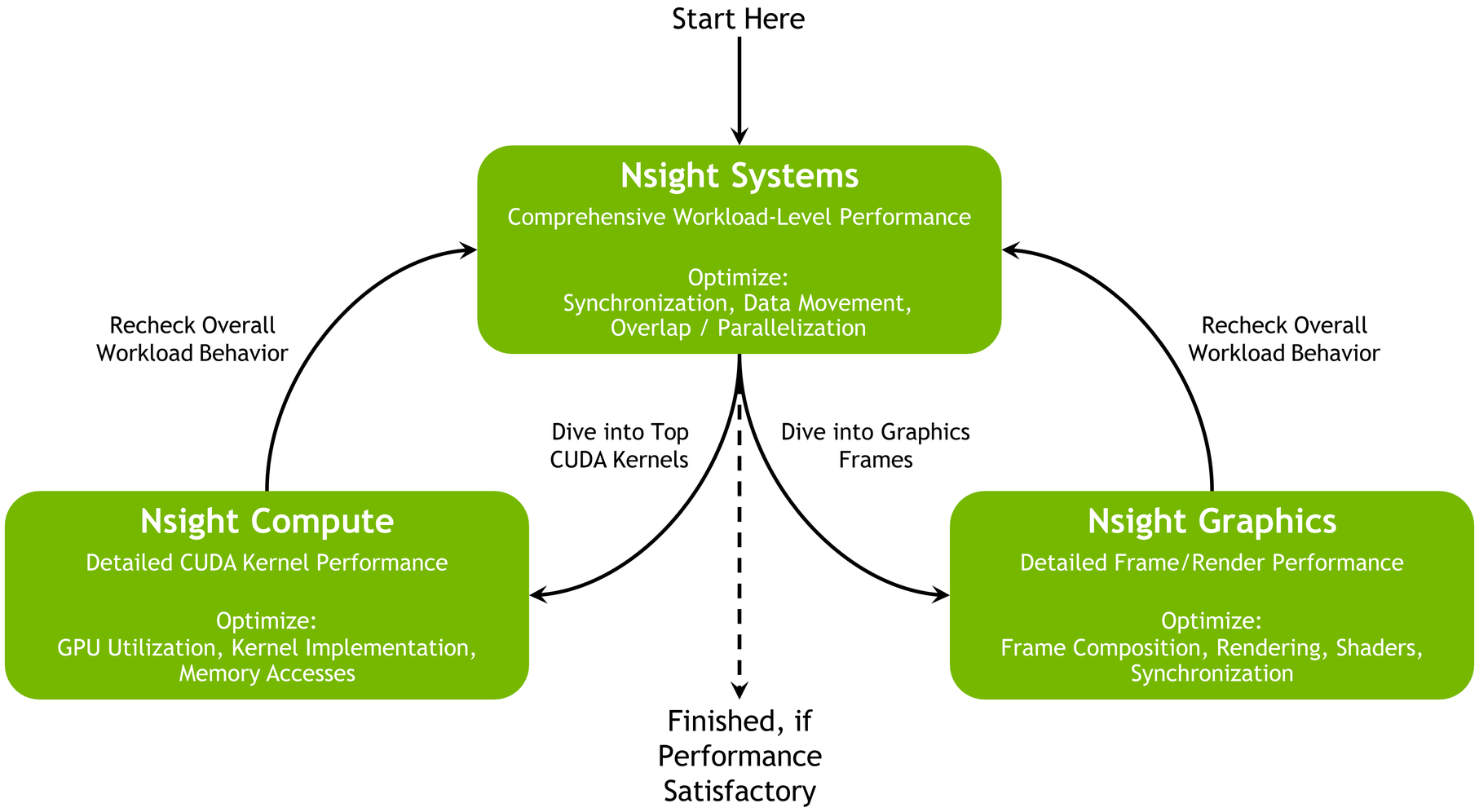




https://developer.nvidia.com/nsight-visual-studio-code-edition

NSIGHT TOOLS

Overview



NSIGHT SYSTEMS

SYSTEM PROFILER

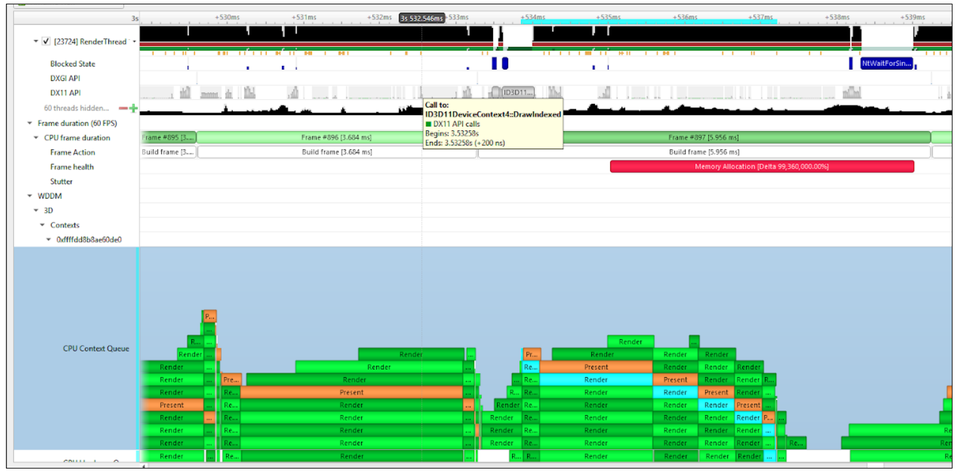
Key Features:

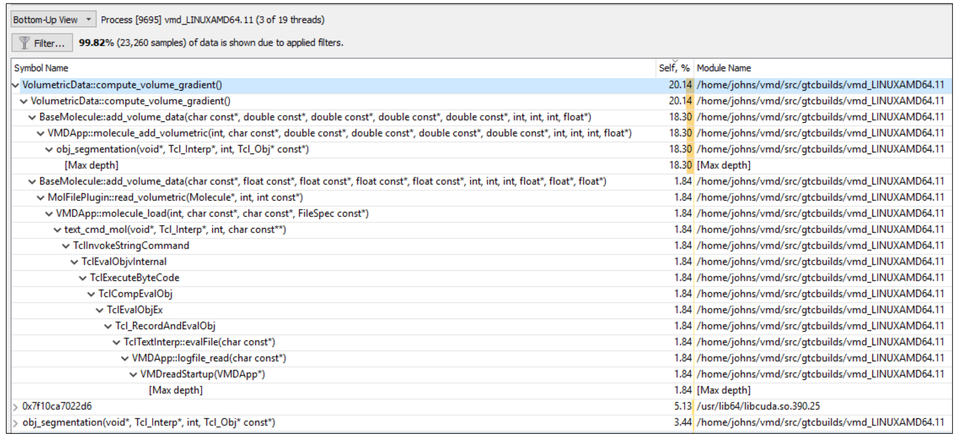
- System-wide application algorithm tuning
  - Multi-process tree support
- Locate optimization opportunities
  - Visualize millions of events on a very fast GUI timeline
  - Or gaps of unused CPU and GPU time
- Balance your workload across multiple CPUs and GPUs
  - CPU algorithms, utilization and thread state
    GPU streams, kernels, memory transfers, etc
- Command Line, Standalone, IDE Integration

OS: Linux (x86, Power, Arm SBSA, Tegra), Windows, MacOSX (host)
GPUs: Pascal+

Docs/product: https://developer.nvidia.com/nsight-systems

(More CUDA-level info compared to PyTorch Profiler)





Python Developer Tools

Detailed inspection of Python-based GPU & CPU execution

Python Call Stacks Samples and CUDA API Backtrace

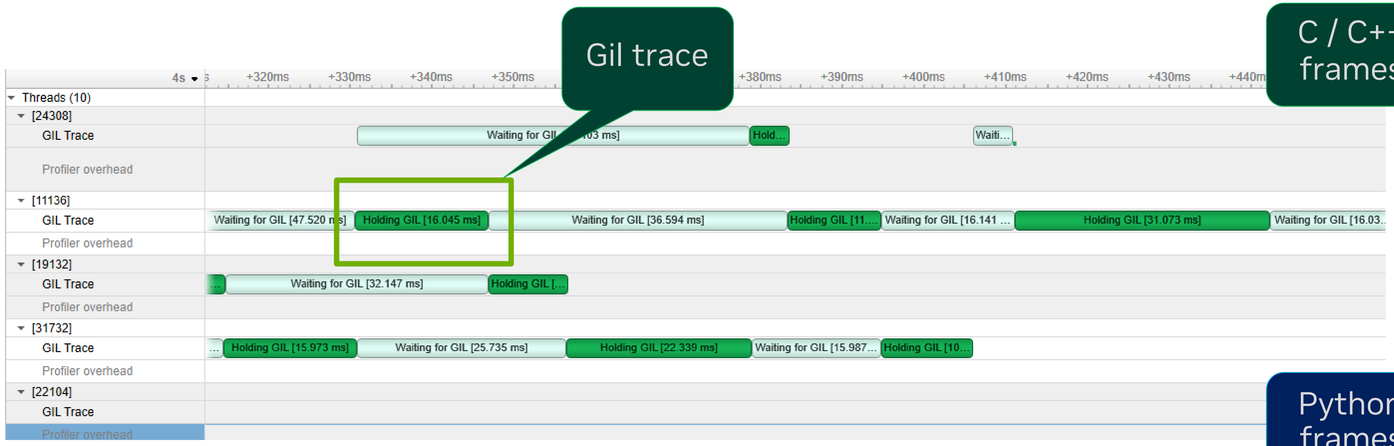
Identify where you are and how you got there

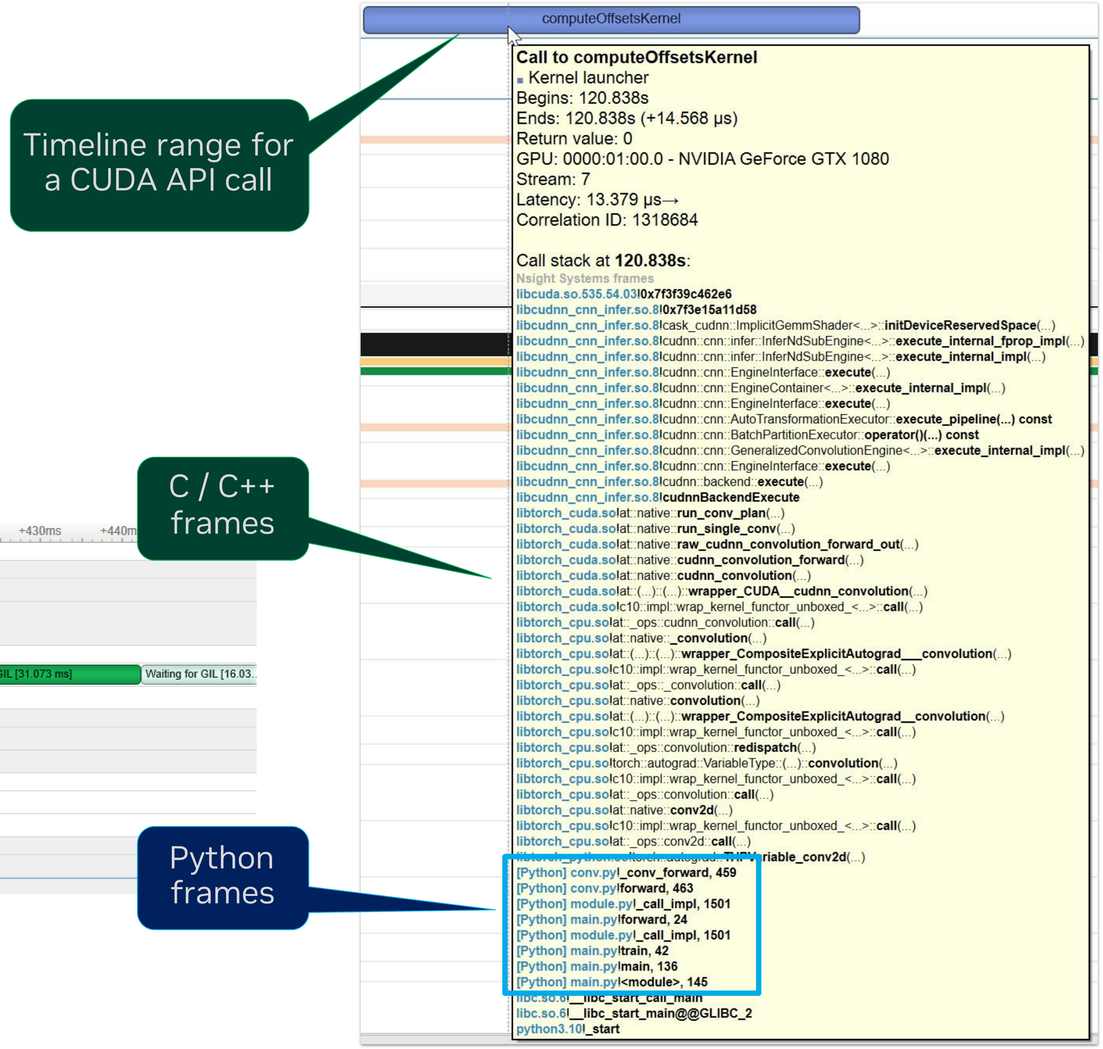
Global Interpreter Lock (GIL) trace

Common performance limiter in Python

Annotated dynamic code ranges

Built into in popular frameworks and libraries such as RAPIDS, Spark, CV-CUDA, and more…





NSIGHT COMPUTE

KERNEL PROFILING TOOL

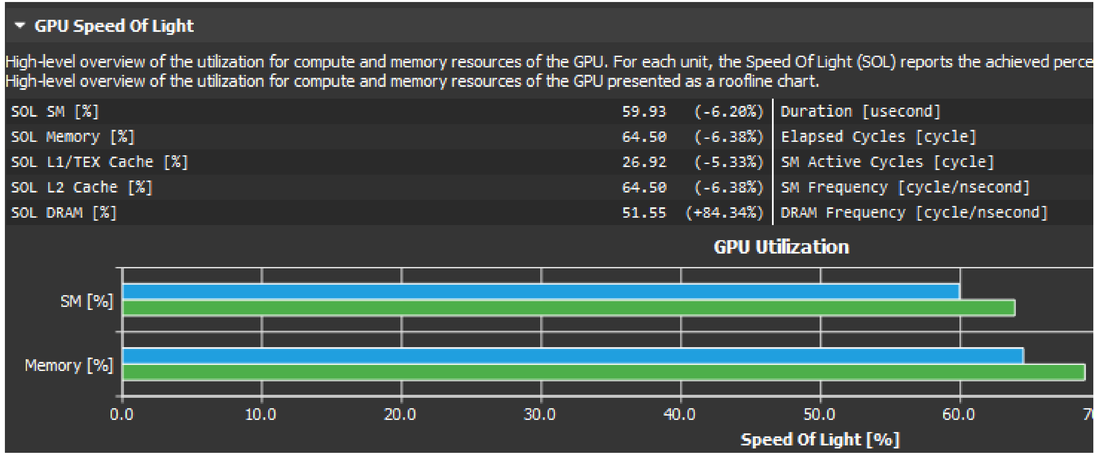
Key Features:

- Interactive CUDA API debugging and kernel profiling
- Built-in rules expertise
- Fully customizable data collection and display
- Command Line, Standalone, IDE Integration, Remote Targets

OS: Linux (x86, Power, Tegra, Arm SBSA), Windows, MacOSX (host only)

GPUs: Volta, Turing, Ampere GPUs

Docs/product: https://developer.nvidia.com/nsight-compute



inst_executed [inst]	63,021,056 (284 instances)
l1tex__data_bank_conflicts_pipe_lsu_mem_shared_op_ld.sum	0
l1tex__data_bank_conflicts_pipe_lsu_mem_shared_op_st.sum	0
l1tex__data_bank_reads.avg.pct_of_peak_sustained_elapsed [%]	9.66
l1tex__data_bank_writes.avg.pct_of_peak_sustained_elapsed [%]	3.23
l1tex__data_pipe_lsu_wavefronts.avg.pct_of_peak_sustained_elapsed [%]	46.16
l1tex__data_pipe_lsu_wavefronts_mem_shared_cmd_read.sum	25,165,824
l1tex__data_pipe_lsu_wavefronts_mem_shared_cmd_read.sum.pct_of_peak_sustained_active [%]	40.75
l1tex__data_pipe_lsu_wavefronts_mem_shared_cmd_write.sum	2,097,152
l1tex__data_pipe_lsu_wavefronts_mem_shared_cmd_write.sum.pct_of_peak_sustained_active [%]	3.40
l1tex__data_pipe_tex_wavefronts.avg.pct_of_peak_sustained_elapsed [%]	0
l1tex__f_wavefronts.avg.pct_of_peak_sustained_elapsed [%]	0.00
l1tex__lsu_writeback_active.avg.pct_of_peak_sustained_elapsed [%]	42.59
l1tex__lsu_writeback_active.sum [cycle]	27,803,648
l1tex__lsu_writeback_active.sum.pct_of_peak_sustained_active [%]	45.03
l1tex__lsuin_requests.avg.pct_of_peak_sustained_elapsed [%]	66.00
l1tex__m_l1tex2xbar_req_cycles_active.avg.pct_of_peak_sustained_elapsed [%]	3.40
l1tex__m_l1tex2xbar_write_bytes.sum [Mbyte]	4.19
l1tex__m_l1tex2xbar_write_bytes_mem_global_op_red.sum [byte]	0

@P0 EXIT	6	108	49	1,404,672
IADD3 R7, P2, R0, UR7, RZ	7	177	95	1,401,344
IADD3 R6, P1, R4, UR4, RZ	8	10	9	1,401,344
ISETP.GE.U32.AND P0, PT, R7, UR5, PT	8	12	5	1,401,344
IADD3.X R8, R2, UR8, RZ, P2, !PT	8	14	3	1,401,344
IMAD.X R7, RZ, RZ, R5, P1	9	9	0	1,401,344
ISETP.GE.U32.AND.EX P0, PT, R8, UR6, PT, P0	9	106	35	1,401,344
STG.E.U8 [R6.64], R3	8	116	75	1,401,344
@P0 EXIT	8	92	33	1,401,344
IADD3 R8, P2, R0, UR9, RZ	9	45	14	1,397,120
IADD3 R6, P1, R6, UR4, RZ	9	248	145	1,397,120
ISETP.GE.U32.AND P0, PT, R8, UR5, PT	9	57	17	1,397,120
IADD3.X R8, R2, UR12, RZ, P2, !PT	9	11	4	1,397,120
IMAD.X R7, RZ, RZ, R7, P1	9	7	0	1,397,120
ISETP.GE.U32.AND.EX P0, PT, R8, UR6, PT, P0	9	94	15	1,397,120
STG.E.U8 [R6.64], R3	8	104	61	1,397,120

CONTAINERIZATION AND NGC CATALOG

NVIDIA Docker, Apptainer/Singularity, Enroot/Pyxis

- NVIDIA Container Toolkit (i.e., NVIDIA Docker), optionally with Rootless-docker

```
docker pull nvcr.io/nvidia/cuda:13.0.2-devel-ubuntu24.04
docker run --rm -it --gpus all nvcr.io/nvidia/cuda:13.0.2-devel-ubuntu24.04
```

- Apptainer (previously Singularity, commands are interchangeable)

```
apptainer pull docker://nvcr.io/nvidia/cuda:13.0.2-devel-ubuntu24.04
apptainer run --nv cuda_13.0.2-devel-ubuntu24.04.sif

srun ... apptainer run --nv cuda_13.0.2-devel-ubuntu24.04.sif ...
```

- Enroot, optionally with Pyxis (i.e., Docker support for HPC)

```
enroot import docker://nvcr.io#nvidia/cuda:13.0.2-devel-ubuntu24.04
enroot create nvidia+cuda+13.0.1-devel-ubuntu24.04.sqsh
enroot start nvidia+cuda+13.0.1-devel-ubuntu24.04

srun --container-image=docker://nvcr.io#nvidia/cuda:13.0.2-devel-ubuntu24.04 ...
```

(For HPC-X: `nvcr.io/nvidia/nvhpc:25.9-devel-cuda13.0-ubuntu24.04`)

CUDA Binaries and Developer Tools (Recap)

- Backward Compatibility
- Portability, PTX, SASS
- Compute Sanitizer, CUDA GDB
- IDE Support (VS/VSCode/Eclipse)
- Nsight Systems/Compute
- Docker/Singularity/Apptainer/Enroot

CUDA Optimization Basics

GPU Overview

NVIDIA H200 SXM

PCIe Gen 5, 128 GB/s bidirectional

50 MB L2

141 GB HBM3e, 4.8 TB/s

132 SMs
4th Gen Tensor Cores

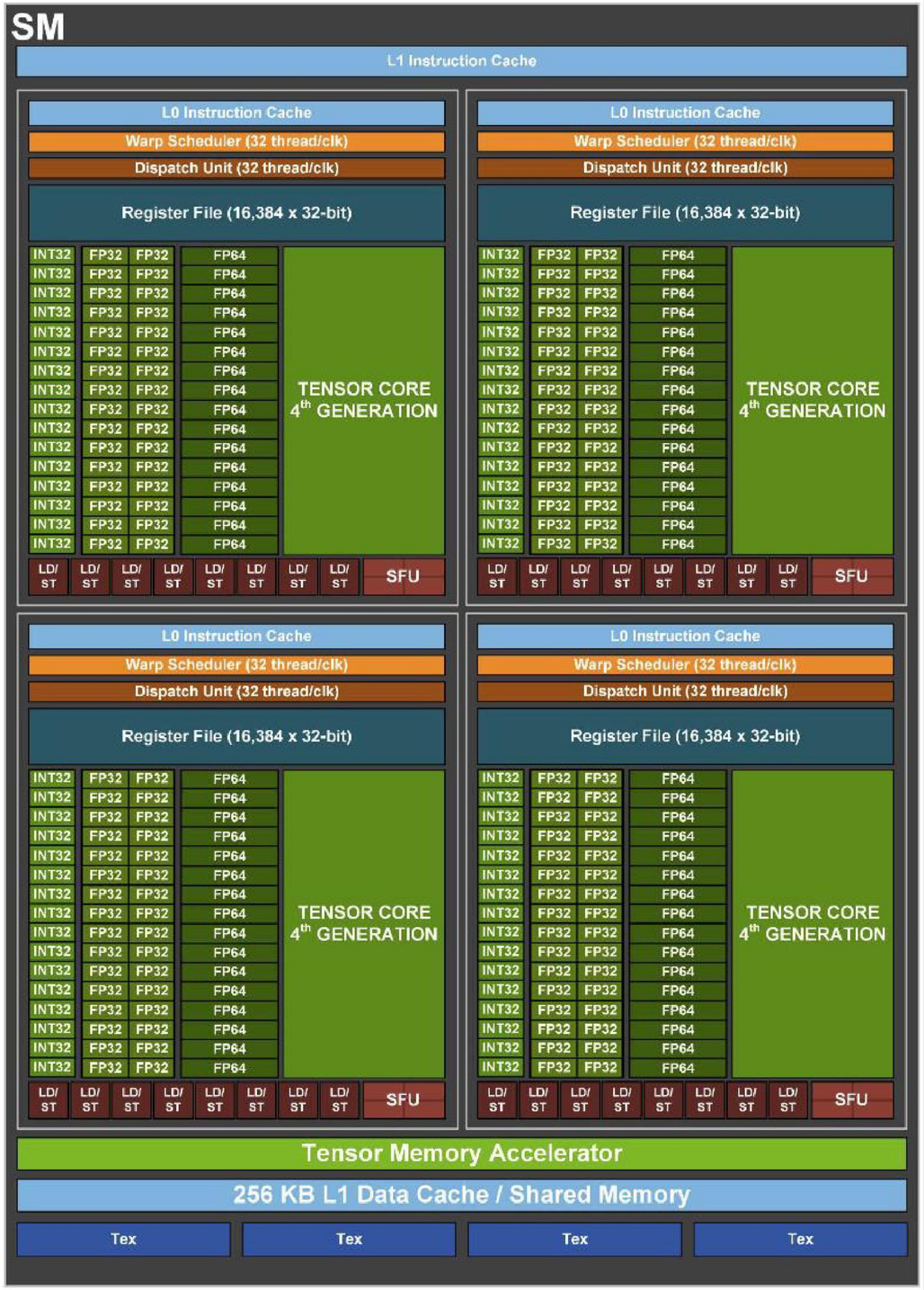
4th Gen NVLink, 900 GB/s bidirectional

Streaming Multiprocessor (SM)

Hopper architecture

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

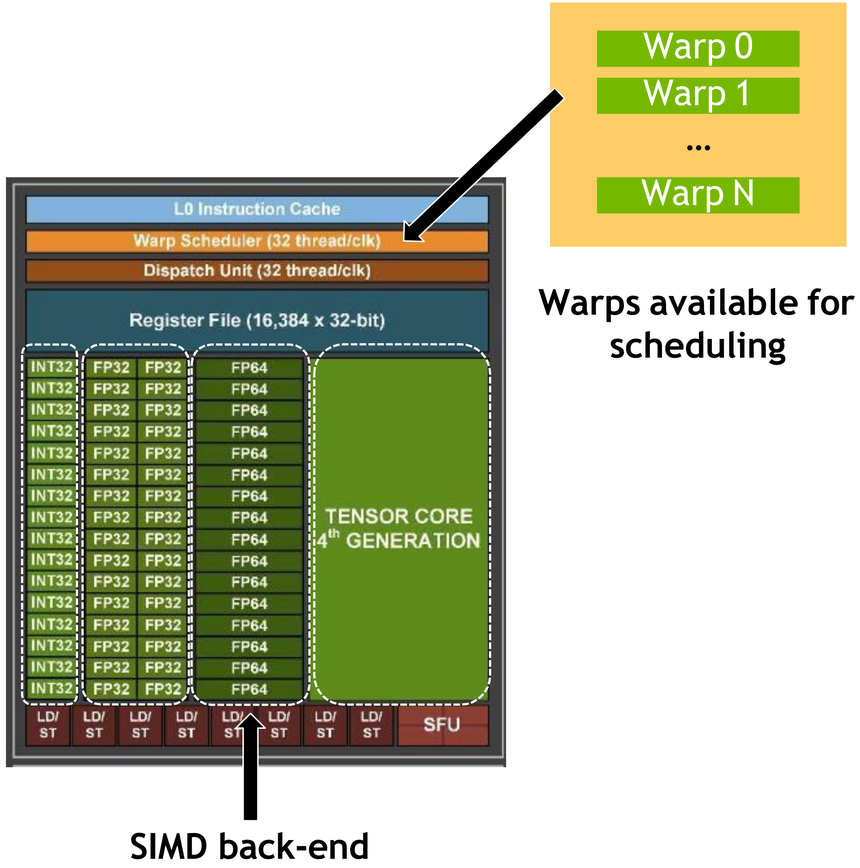
- 32 LD/ST units
- 64K 32-bit registers
- 256 KiB unified L1 data cache and shared memory
- Tensor Memory Accelerator (TMA)



SIMT Architecture

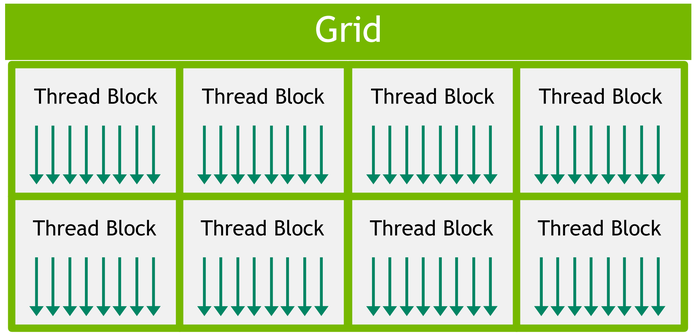
Single-Instruction, Multiple-Thread

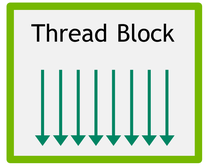
- Akin to a **single-instruction multiple-data (SIMD) array processor** per Flynn's taxonomy combined with **fine-grained multithreading**.
- SIMT architectures expose a large set of hardware threads, which is partitioned into groups called **warps.**
  - Interleave warp execution to hide latencies.
  - Execution context for each warp is kept on-chip for fast interleaving.
- When scheduled, each thread of a warp executes on a given lane of a SIMD functional unit.
- Each SM sub-partition can be thought of as a SIMT engine that creates, manages, schedules, and executes warps of 32 parallel threads.



Thread Hierarchy

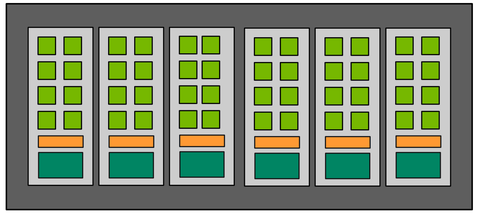
CUDA/Software



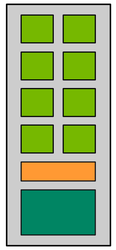


Thread

Hardware



Device



SM

Scalar CUDA core

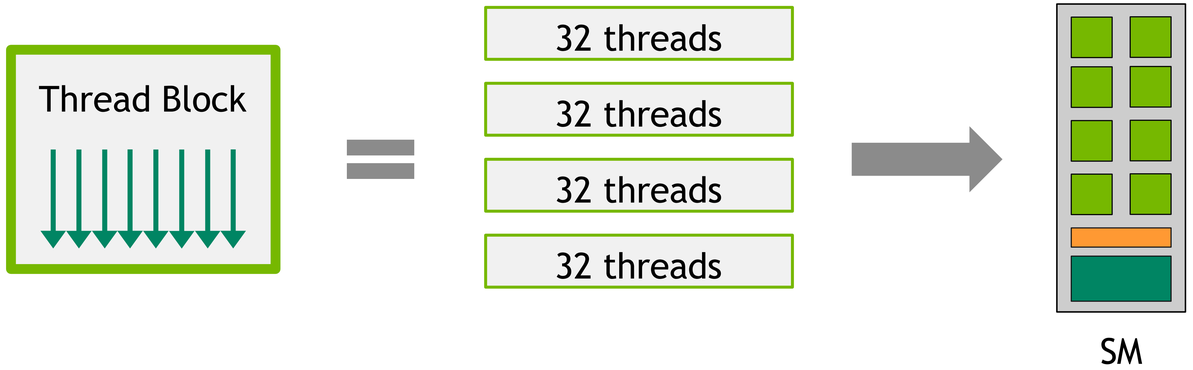
- A CUDA kernel is launched on a grid of thread blocks, which are completely independent.
- Thread blocks are executed on SMs.
  - Several concurrent thread blocks can reside on an SM.
  - Thread blocks do not migrate.
  - Each block can be scheduled on any of the available SMs, in any order, concurrently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

Thread Hierarchy

What about warps?

- At runtime, a block of threads is divided into warps for SIMT execution.
  - The way a block is partitioned into warps is always the same.
    - Each warp contains threads of consecutive, increasing thread IDs with the first warp containing thread 0.

- The total number of warps in a block is defined as:
  - $ceil\left(\frac{threads\ per\ block}{warp\ size}, 1\right)$



Warp Scheduling

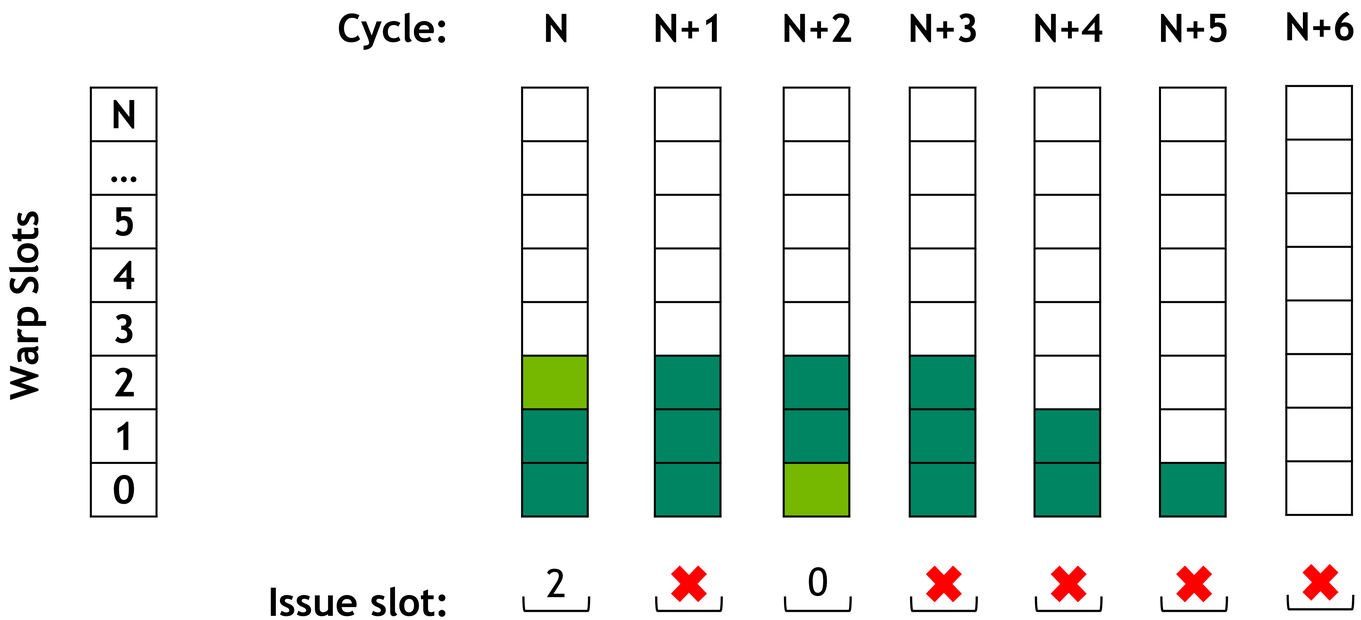
Mental model

Active Warp States:

Stalled
Waiting on:
an instruction fetch,
a memory dependency,
an execution dependency, or
a synchronization barrier.

Eligible
Active warp that is not stalled.

Selected
Eligible warp that is selected to issue an instruction.



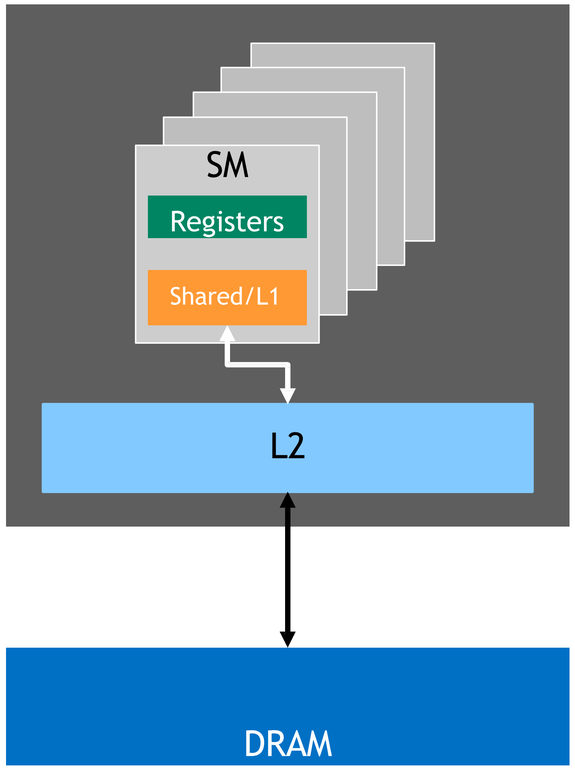
Having more **active warps** would help reduce idle issue slots and hide latencies of stalled warps.

Memory Hierarchy

(A100 GPU)

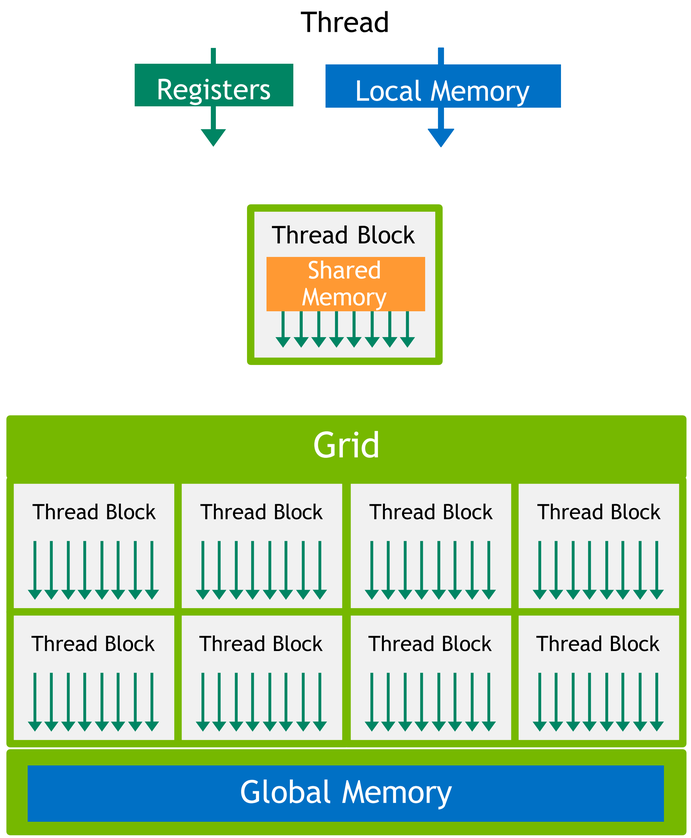
B/W	Latency
13x	1x
3x	5x
1x	15x
0.02x	25x

Hardware



PCIe

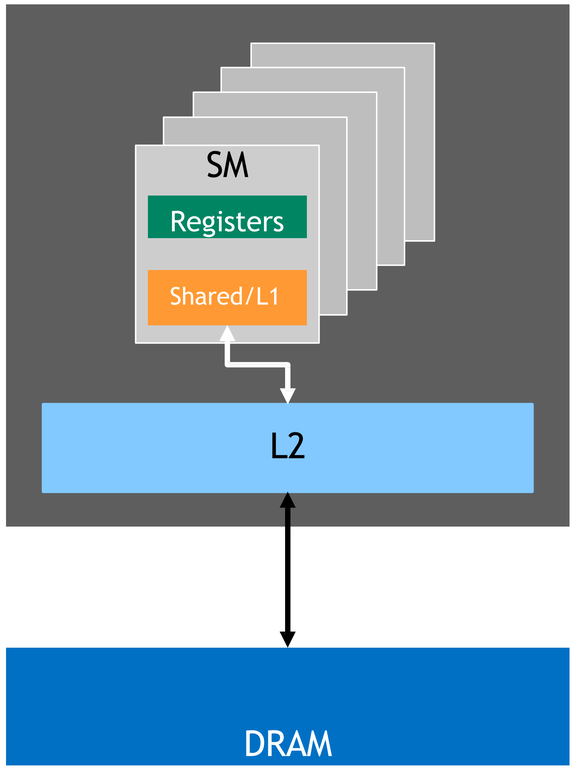
CUDA/Software



- Per-thread **registers**.
  - Lowest possible latency.
- Per-thread **local** memory.
  - Private storage.
  - Slowest access.
- Per-block **shared** memory.
  - Visible by all threads in a block.
  - Can be used to exchange data between threads in a thread block.
  - Very fast access.
- **Global** memory.
  - Visible by all threads in a grid.
  - Slowest access.

(local memory is in fact thread-local global memory)

Memory Reads & Writes



Reads

Check if data is in L1 (if not, check L2)

Check if data is in L2 (if not, get from DRAM)

Unit of data moved: **full sector**

Writes

L1 is write-through: update both L1 and L2

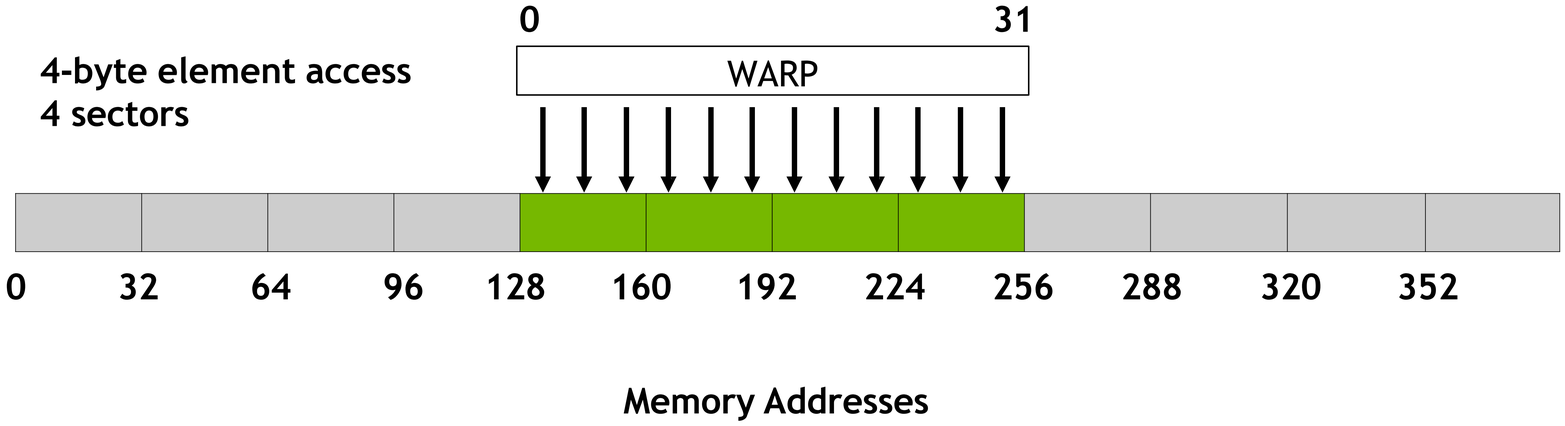
L2 is write back: flush data to DRAM only when needed

Unit of data moved: **partial sector***

* Depends on whether ECC is enable/disabled.

Global Memory Access Patterns

Aligned and sequential

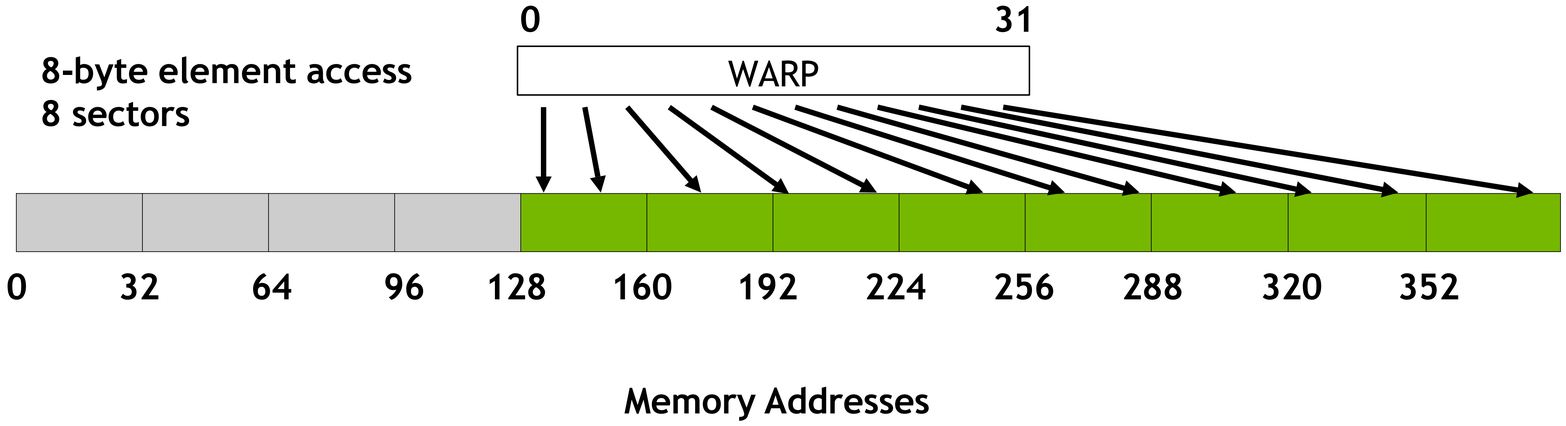


(1 sector = 32 bytes)

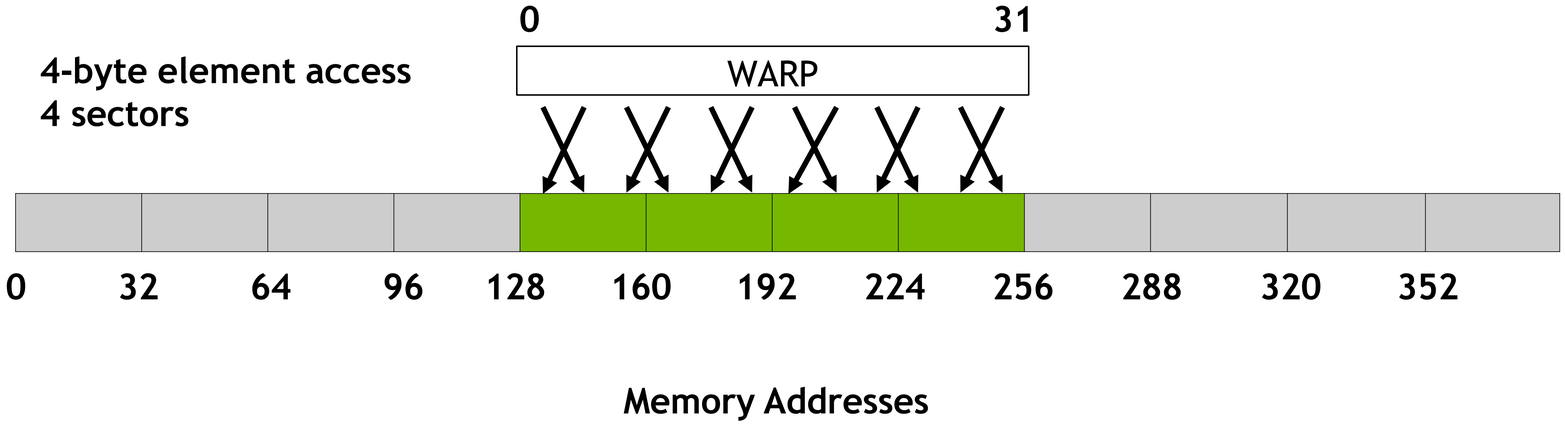
COALESCED!

Global Memory Access Patterns

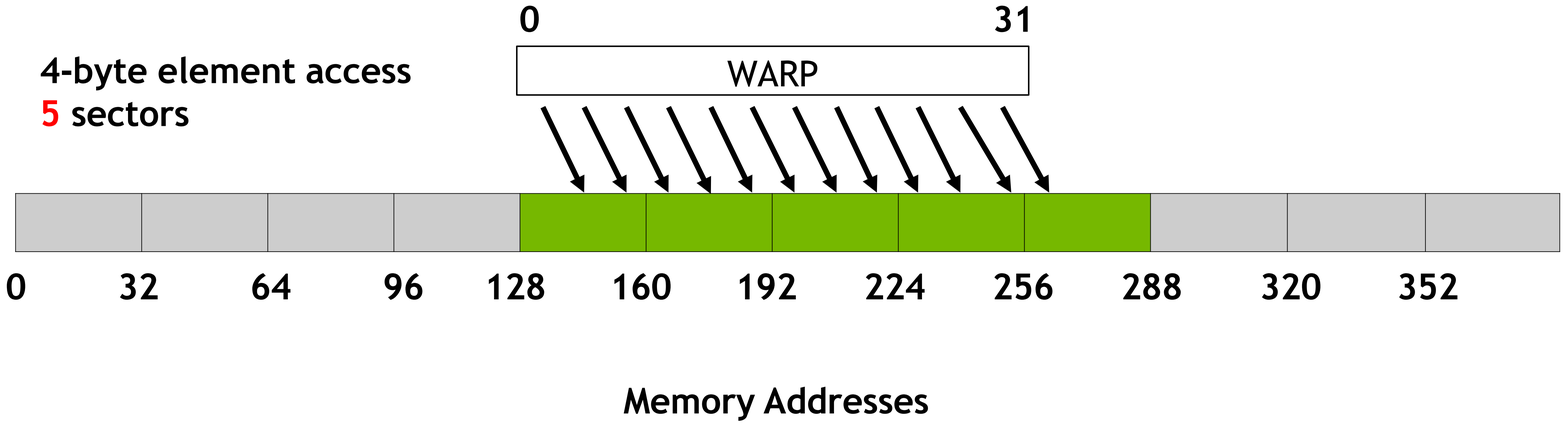
Aligned and sequential

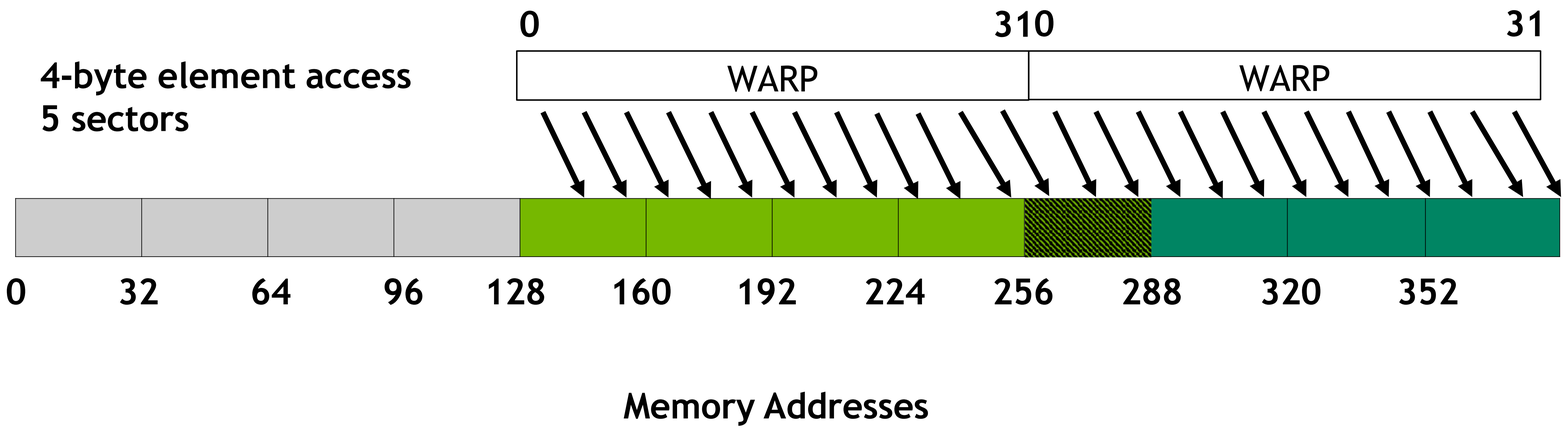


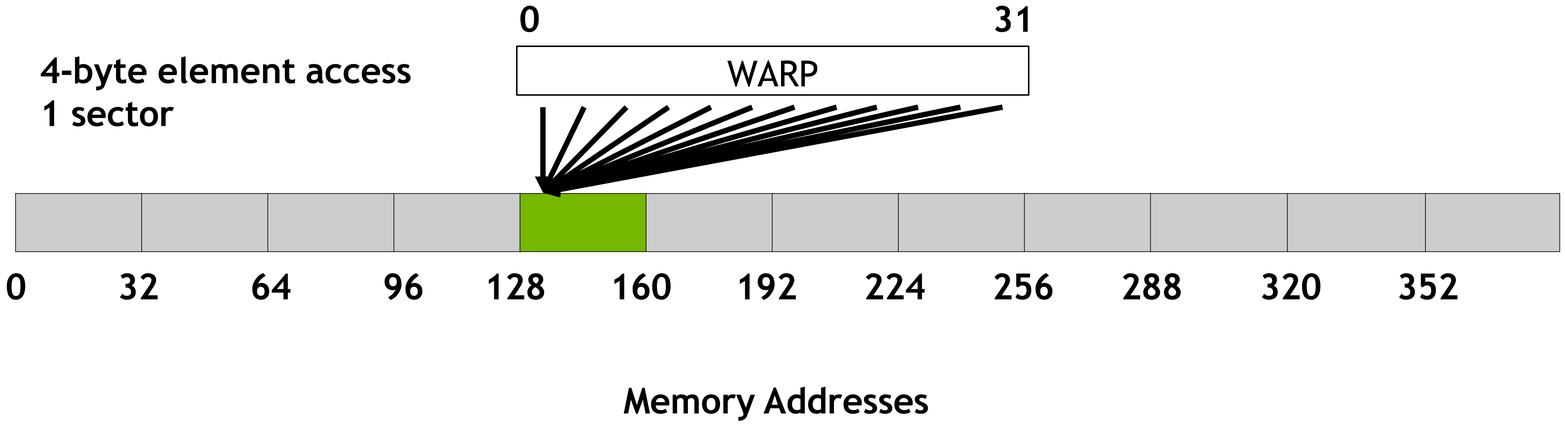
COALESCED!


4-byte element access
4 sectors
0
31
WARP
0
32
64
96
128
160
192
224
256
288
320
352
Memory Addresses

COALESCED!

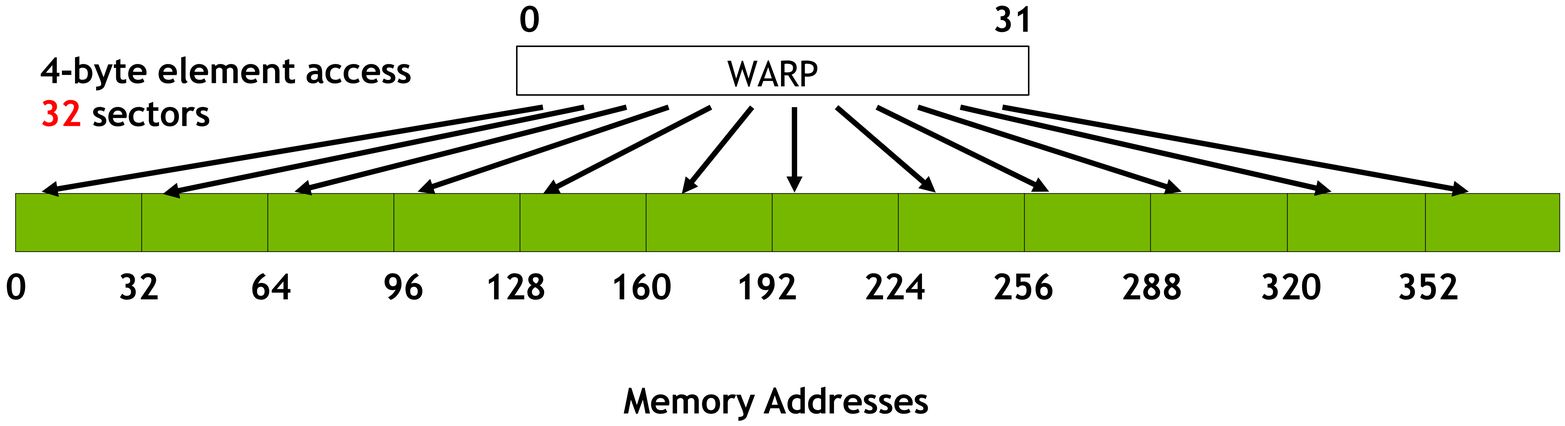

4-byte element access
5 sectors
0
31
WARP
0
32
64
96
128
160
192
224
256
288
320
352
Memory Addresses


4-byte element access
5 sectors
0
310
31
WARP
WARP
0
32
64
96
128
160
192
224
256
288
320
352
Memory Addresses


0
31
4-byte element access
1 sector
WARP
0
32
64
96
128
160
192
224
256
288
320
352
Memory Addresses

Global Memory Access Patterns

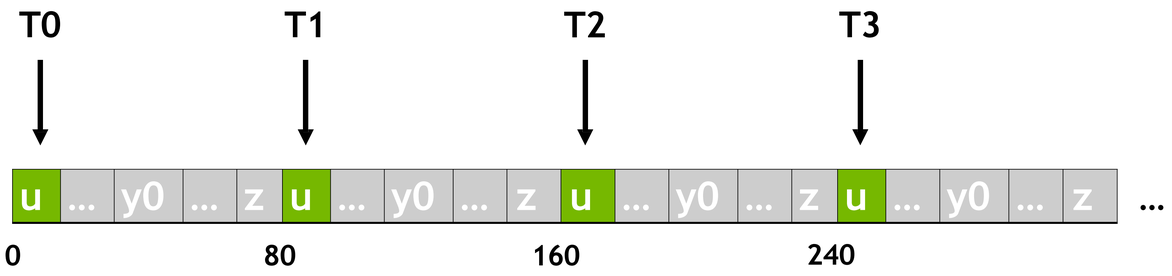
Aligned and sequential



Impact of Data Layout

Array-of-Structures (AoS) vs Structure-of-Arrays (SoA)

- When loading coefficients `u` and y[0]:
  - Successive threads in a warp read 4 bytes at 80-byte stride.

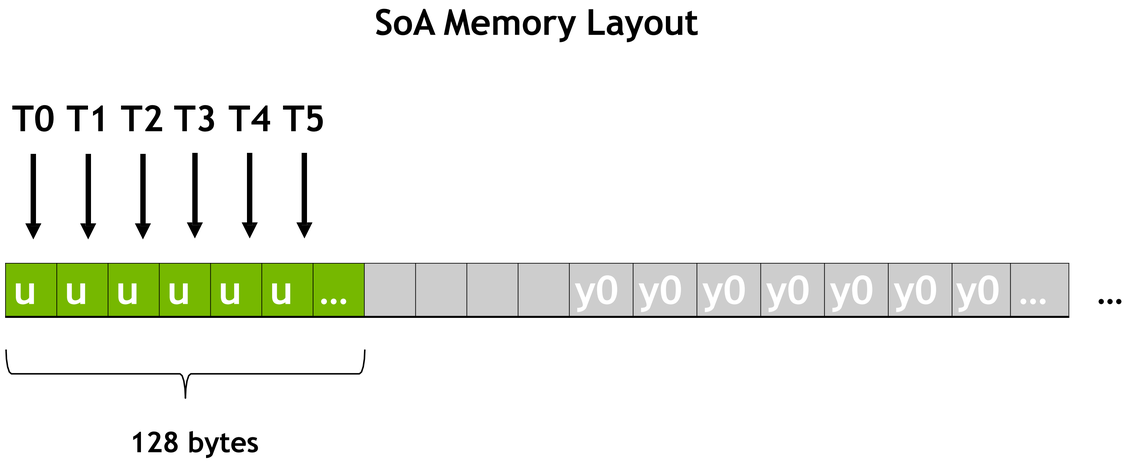



```
struct Coefficients
{
  float u, v, w;
  float x[8], y[8], z;
};

__global__void kernel(Coefficients *data)
{
  int i = cg::this_grid.thread_rank();

  data[i].u = data[i].u + 10.f;
  data[i].y[0] = data[i].y[0] + 10.f;
}
```

Impact of Data Layout

Array-of-Structures vs Structure-of-Arrays

- Refactoring from **AoS** to **SoA** leads to coalesced memory accesses for `u` and `y[0]`.




```
struct Coefficients
{
  float *u, *v, *w;
  float *x0, …, *x7, *y0, … *y7, *z;
};

__global__void kernel(Coefficients data)
{
  int i = cg::this_grid.thread_rank();

  data.u[i] = data.u[i] + 10.f;
  data.y0[i] = data.y0[i] + 10.f;
}
```

Shared Memory Access Patterns

Bank conflicts

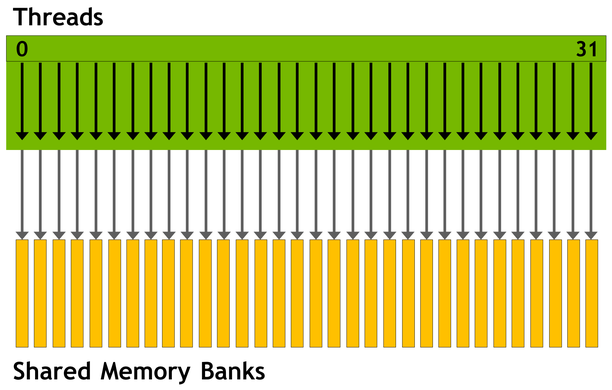
- Bank conflicts occur when threads in the **same phase** want to access the **same bank**.

Coalesced access

(No bank conflicts)

shmem[threadIdx.x] = data[tid]

4-byte data

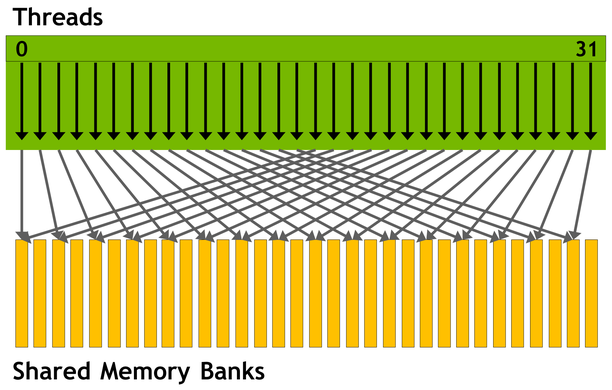


Conflict access

(2-way bank conflicts)

shmem[threadIdx.x * 2] = data[tid]

4-byte data

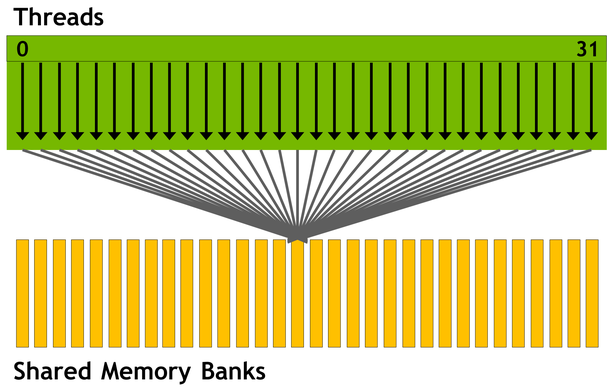


Broadcast access

(No bank conflicts)

data = shmem[0]

4-byte data

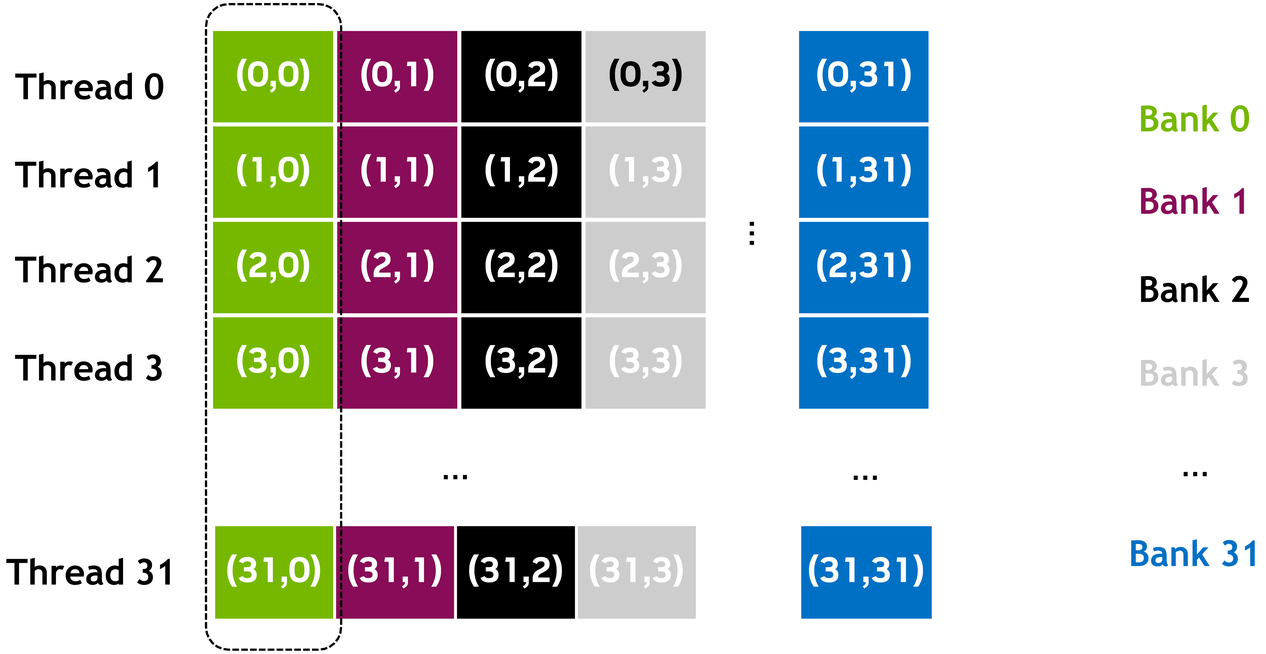


Bank Conflicts

Example

- 32 x 32 array of floats in shared memory
  - 4-byte data, 1 array element per bank
  - Row-major layout
  - 2D thread block

- **Access pattern:**
  - `idx := threadIdx.x*32 + threadIdx.y`
  - 32-way bank conflicts



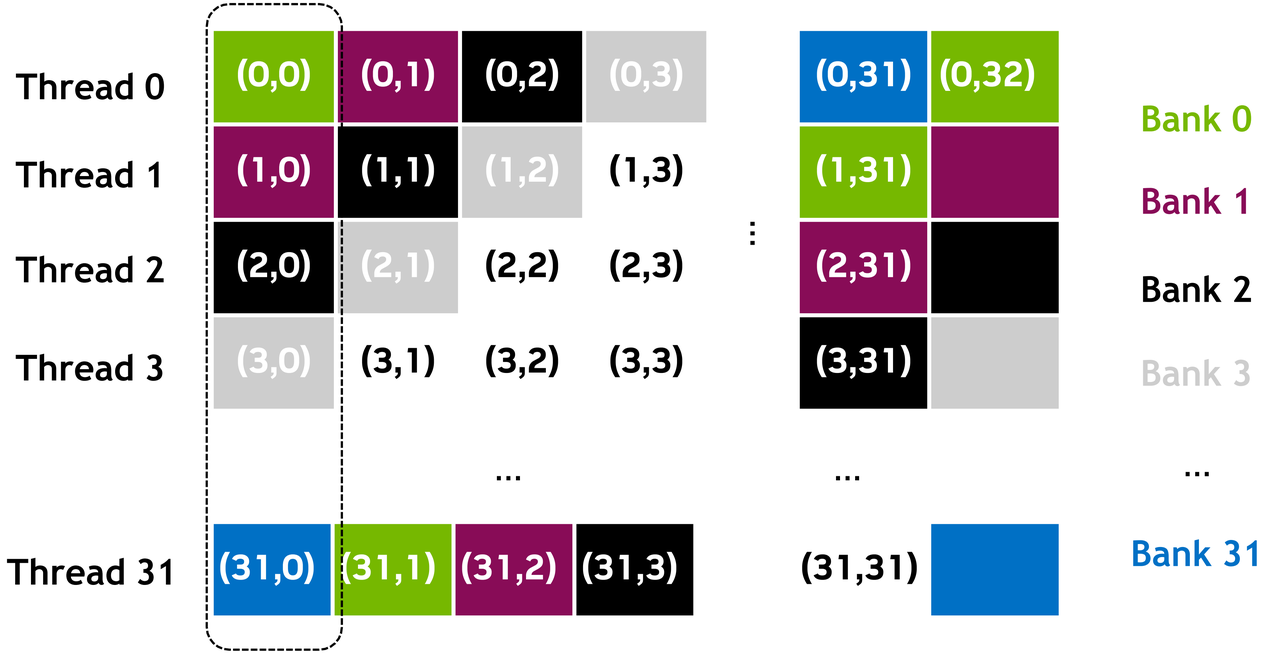
All threads in a warp access the same bank!

Resolving Bank Conflicts

Padding

- 32 x **33** array of floats in shared memory
  - 4-byte data, 1 array element per bank
  - Row-major layout
  - 2D thread block

- **Access pattern:**
  - `idx := threadIdx.x*33 + threadIdx.y`
  - No conflicts!



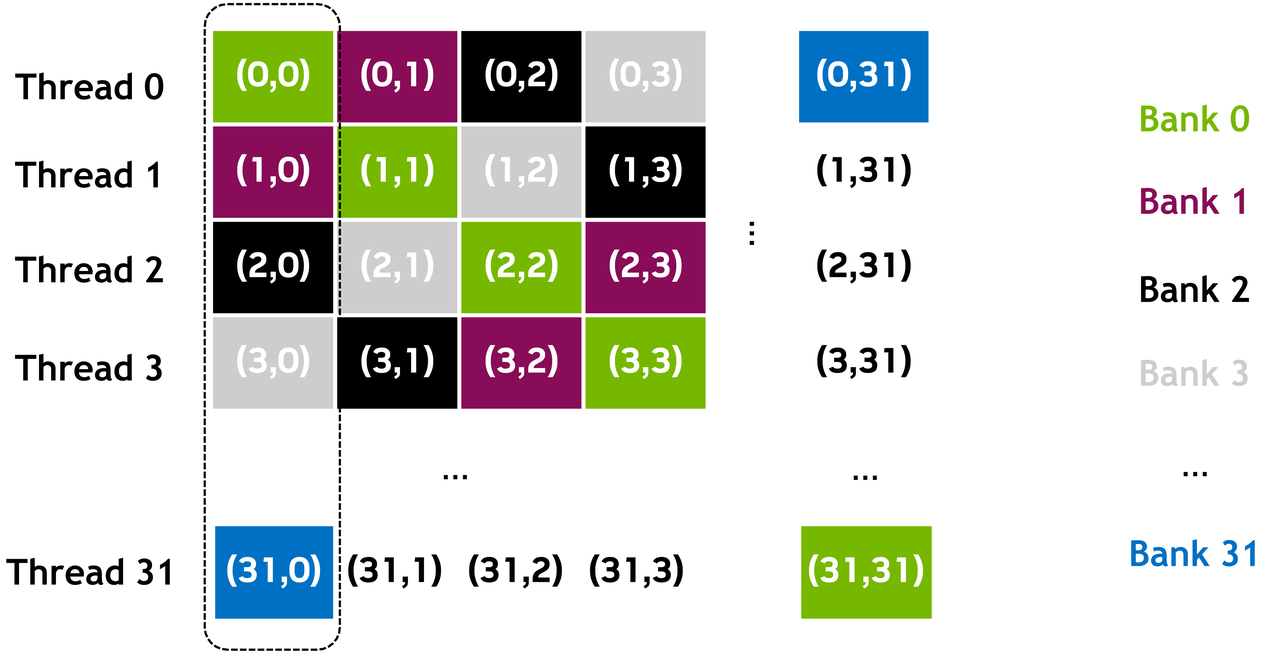
Each thread in a warp accesses a distinct bank!

Resolving Bank Conflicts

Swizzling

- 32 x 32 array of floats in shared memory
  - 4-byte data, 1 array element per bank
  - Row-major layout
  - 2D thread block

- **Access pattern:**
  - `idx = threadIdx.x*32 + threadIdx.y ^ threadIdx.x`
  - No conflicts!
  - No shared memory wasted!



Each thread in a warp accesses a distinct bank!

L1 / SHARED MEMORY

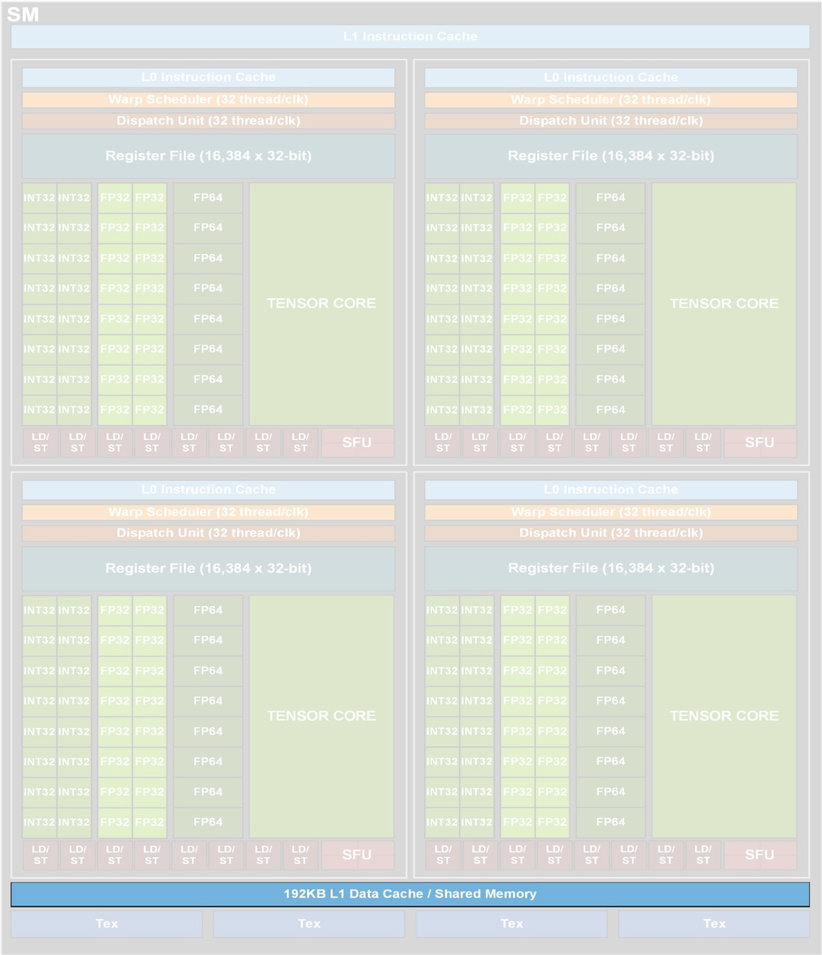
Overview

- Combined L1 Data Cache and Shared Memory
  - 1 L1TEX unit per SM
  - Same physical memory used for L1 cache and shared memory
  - L1 cache uses physical memory as data banks
  - Carveout reserves part of physical memory for shared memory
    - User configurable carveout sizes
  - Similar runtime expectations for shared memory and L1 cache

- Specification of Physical Memory (GA100):
  - Total Size per SM: 192KB
  - Shared Memory Configs: 0, 8, 16, 32, 64, 100, 132, or 164KB
  - Number of Banks: 32
  - Successive 4Byte words map to successive banks
  - 4Byte data access per bank per cycle

L1 / SHARED MEMORY

Overview

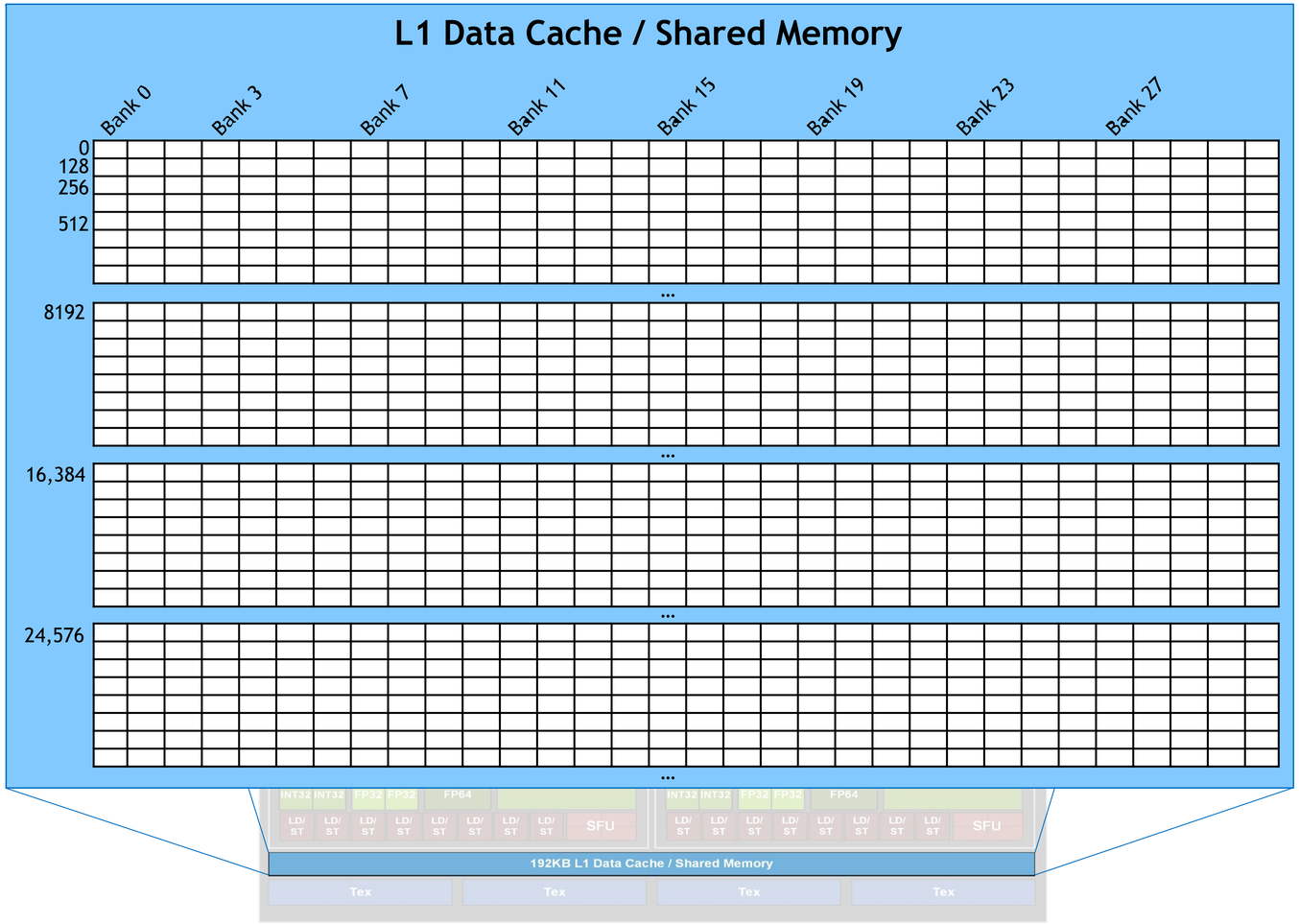


- Combined L1 Data Cache and Shared Memory
  - 1 L1TEX unit per SM
  - Same physical memory used for L1 cache and shared memory
  - L1 cache uses physical memory as data banks
  - Carveout reserves part of physical memory for shared memory
    - User configurable carveout sizes
  - Similar runtime expectations for shared memory and L1 cache

- Specification of Physical Memory (GA100):
  - Total Size per SM: 192KB
  - Shared Memory Configs: 0, 8, 16, 32, 64, 100, 132, or 164KB
  - Number of Banks: 32
  - Successive 4Byte words map to successive banks
  - 4Byte data access per bank per cycle

L1 / SHARED MEMORY

Physical Memory Layout

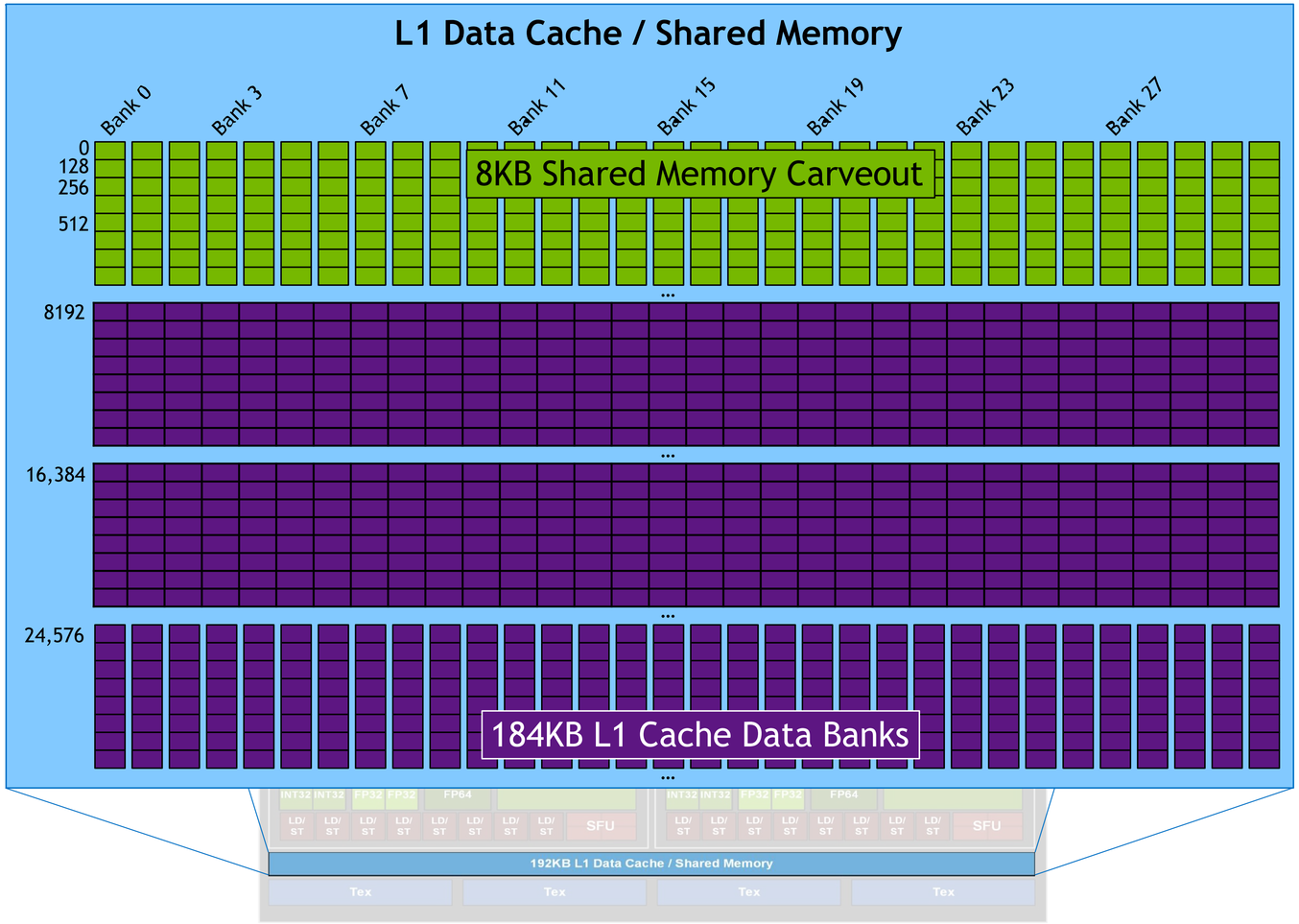


- Combined L1 Data Cache and Shared Memory
  - 1 L1TEX unit per SM
  - Same physical memory used for L1 cache and shared memory
  - L1 cache uses physical memory as data banks
  - Carveout reserves part of physical memory for shared memory
    - User configurable carveout sizes
  - Similar runtime expectations for shared memory and L1 cache

- Specification of Physical Memory (GA100):
  - Total Size per SM: 192KB
  - Shared Memory Configs: 0, 8, 16, 32, 64, 100, 132, or 164KB
  - Number of Banks: 32
  - Successive 4Byte words map to successive banks
  - 4Byte data access per bank per cycle

L1 / SHARED MEMORY

Carveout Example

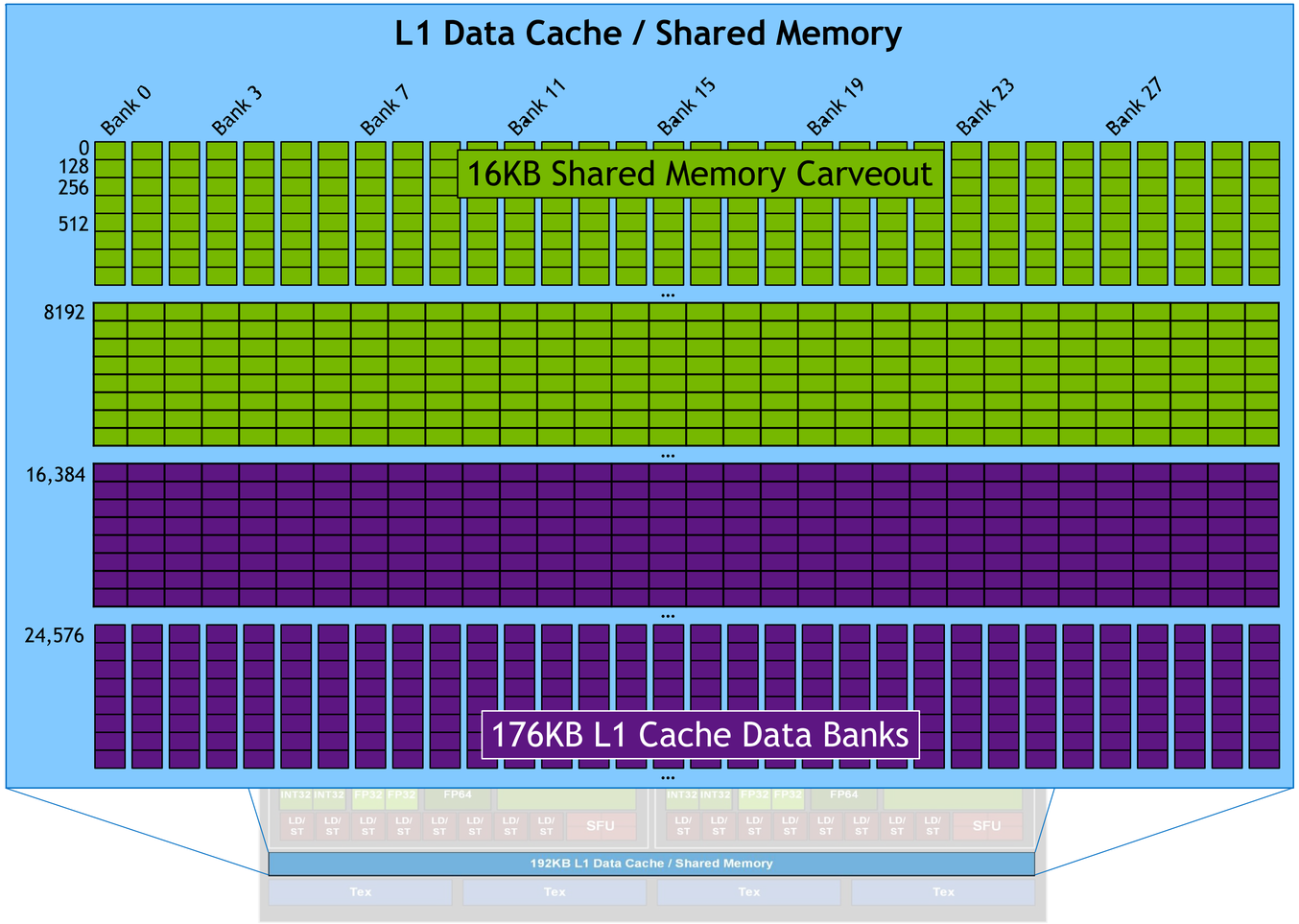


- Combined L1 Data Cache and Shared Memory
  - 1 L1TEX unit per SM
  - Same physical memory used for L1 cache and shared memory
  - L1 cache uses physical memory as data banks
  - Carveout reserves part of physical memory for shared memory
    - User configurable carveout sizes
  - Similar runtime expectations for shared memory and L1 cache

- Specification of Physical Memory (GA100):
  - Total Size per SM: 192KB
  - Shared Memory Configs: 0, 8, 16, 32, 64, 100, 132, or 164KB
  - Number of Banks: 32
  - Successive 4Byte words map to successive banks
  - 4Byte data access per bank per cycle

L1 / SHARED MEMORY

Carveout Example

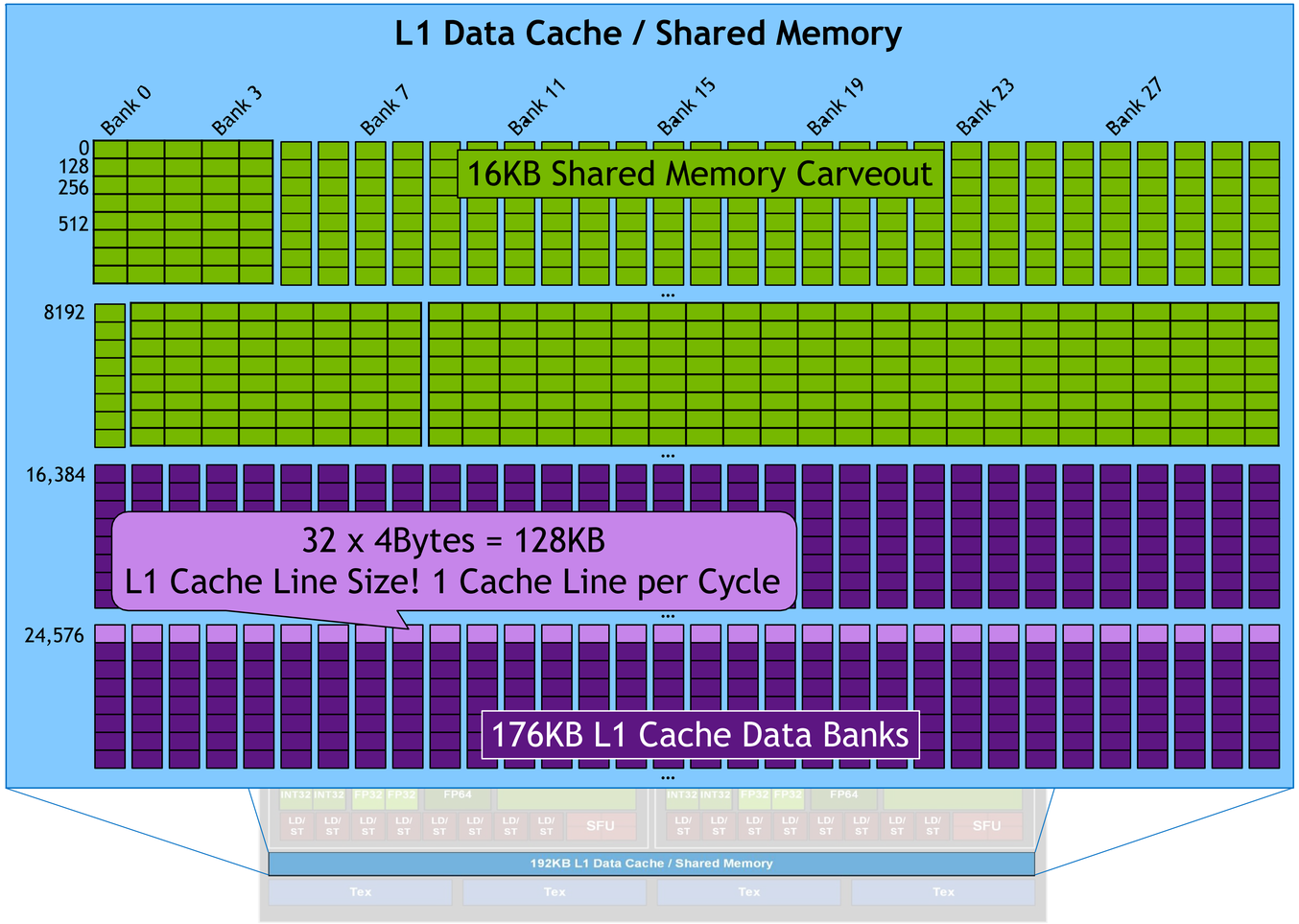


- Combined L1 Data Cache and Shared Memory
  - 1 L1TEX unit per SM
  - Same physical memory used for L1 cache and shared memory
  - L1 cache uses physical memory as data banks
  - Carveout reserves part of physical memory for shared memory
    - User configurable carveout sizes
  - Similar runtime expectations for shared memory and L1 cache

- Specification of Physical Memory (GA100):
  - Total Size per SM: 192KB
  - Shared Memory Configs: 0, 8, 16, 32, 64, 100, 132, or 164KB
  - Number of Banks: 32
  - Successive 4Byte words map to successive banks
  - 4Byte data access per bank per cycle

L1 / SHARED MEMORY

L1 Cache Perspective

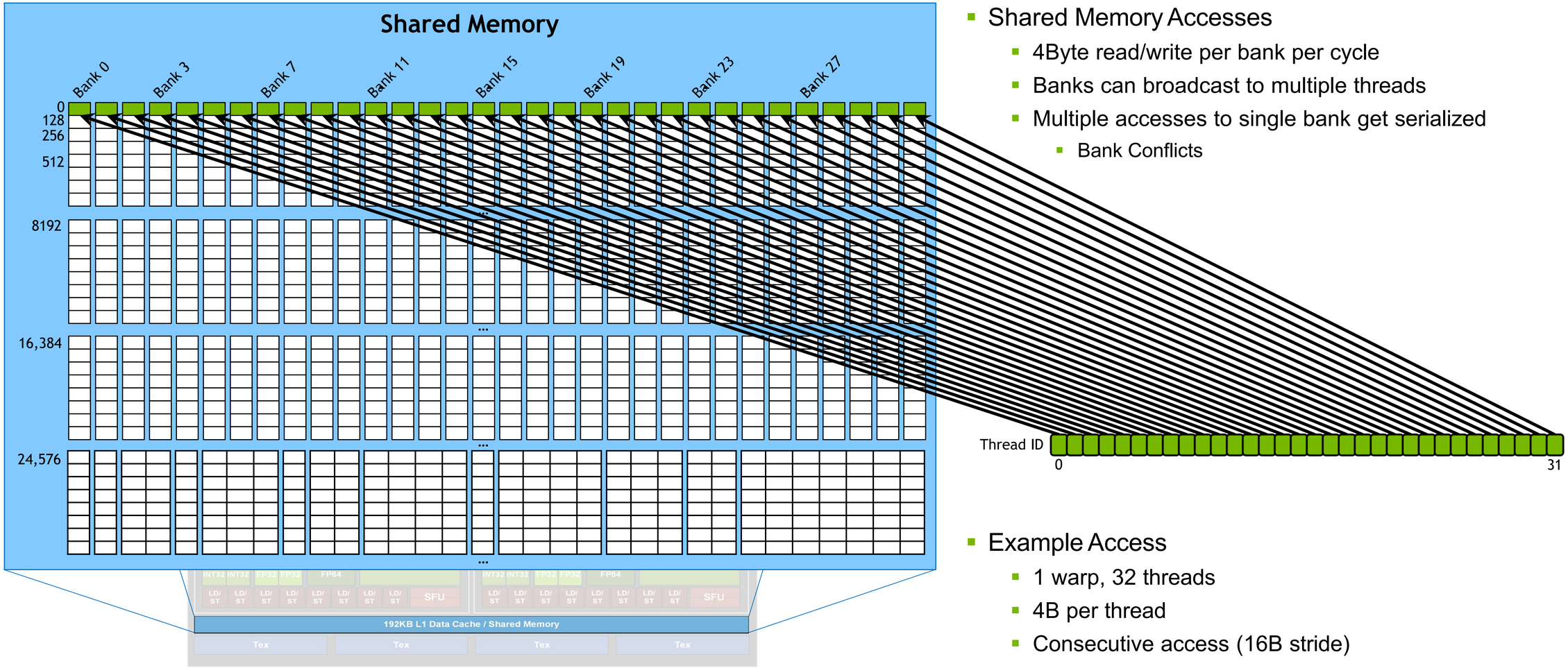


- Combined L1 Data Cache and Shared Memory
  - 1 L1TEX unit per SM
  - Same physical memory used for L1 cache and shared memory
  - L1 cache uses physical memory as data banks
  - Carveout reserves part of physical memory for shared memory
    - User configurable carveout sizes
  - Similar runtime expectations for shared memory and L1 cache

- Specification of Physical Memory (GA100):
  - Total Size per SM: 192KB
  - Shared Memory Configs: 0, 8, 16, 32, 64, 100, 132, or 164KB
  - Number of Banks: 32
  - Successive 4Byte words map to successive banks
  - 4Byte data access per bank per cycle

L1 / SHARED MEMORY

Shared Memory Perspective

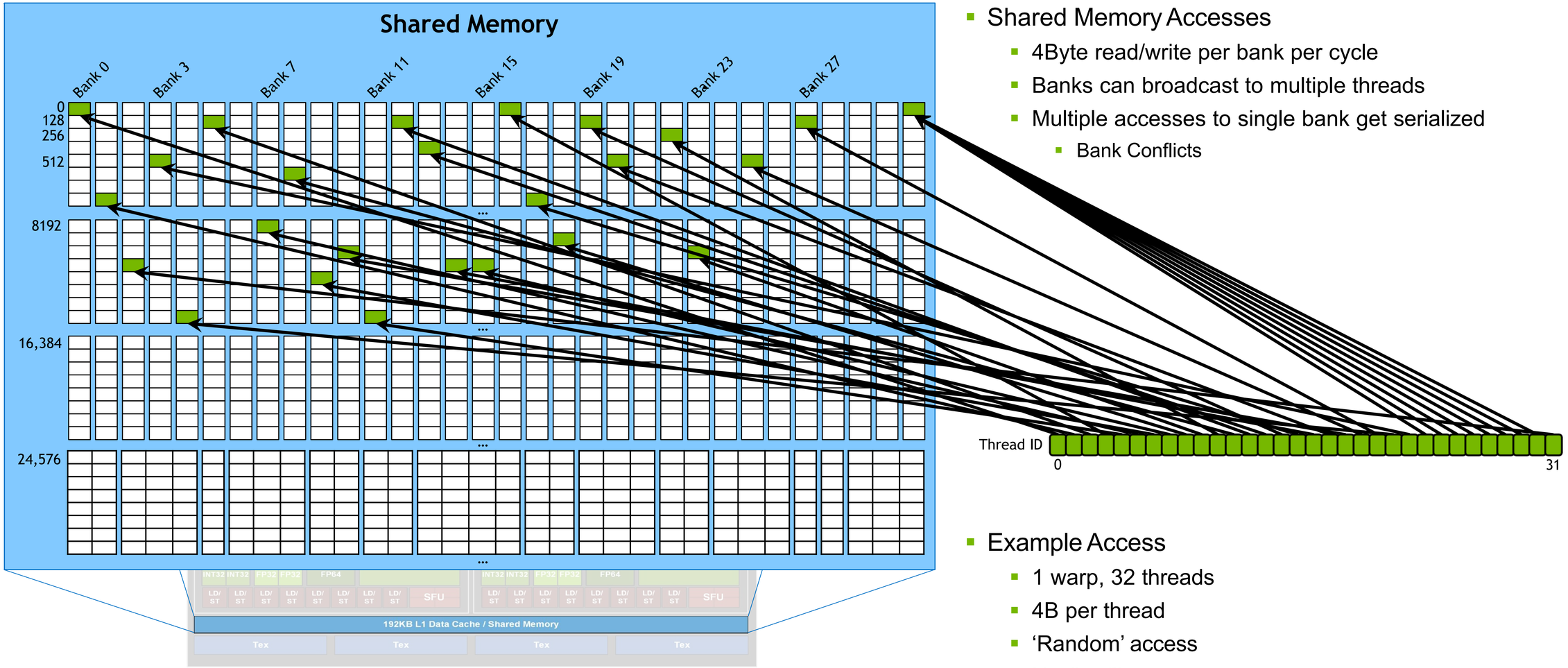


- Shared Memory Accesses
  - 4Byte read/write per bank per cycle
  - Banks can broadcast to multiple threads
  - Multiple accesses to single bank get serialized
    - Bank Conflicts

- Example Access
  - 1 warp, 32 threads
  - 4B per thread
  - Consecutive access (16B stride)
  - No Conflicts
  - 1 Cycle to access

L1 / SHARED MEMORY

Shared Memory Perspective

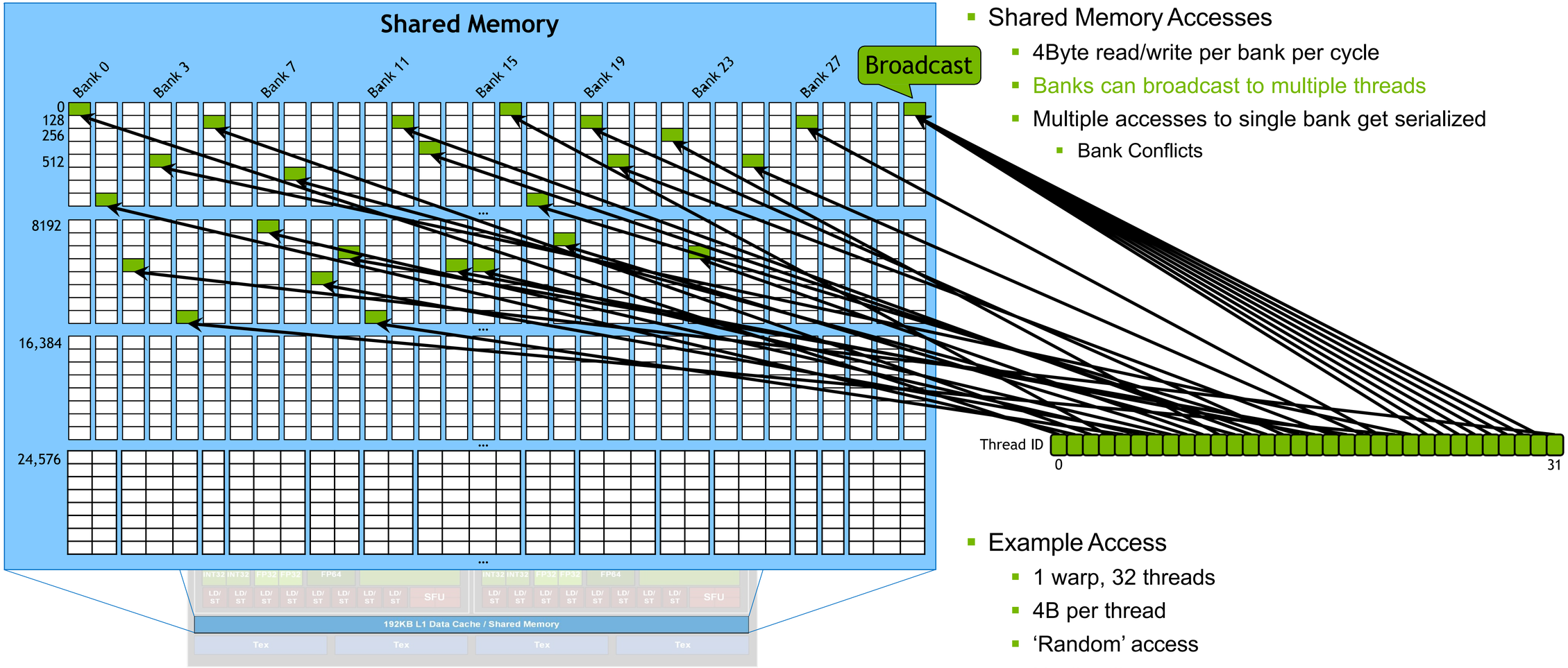


- Shared Memory Accesses
  - 4Byte read/write per bank per cycle
  - Banks can broadcast to multiple threads
  - Multiple accesses to single bank get serialized
    - Bank Conflicts

- Example Access
  - 1 warp, 32 threads
  - 4B per thread
  - ‘Random’ access
  - No Conflicts
  - 1 Cycle to access

L1 / SHARED MEMORY

Shared Memory Perspective

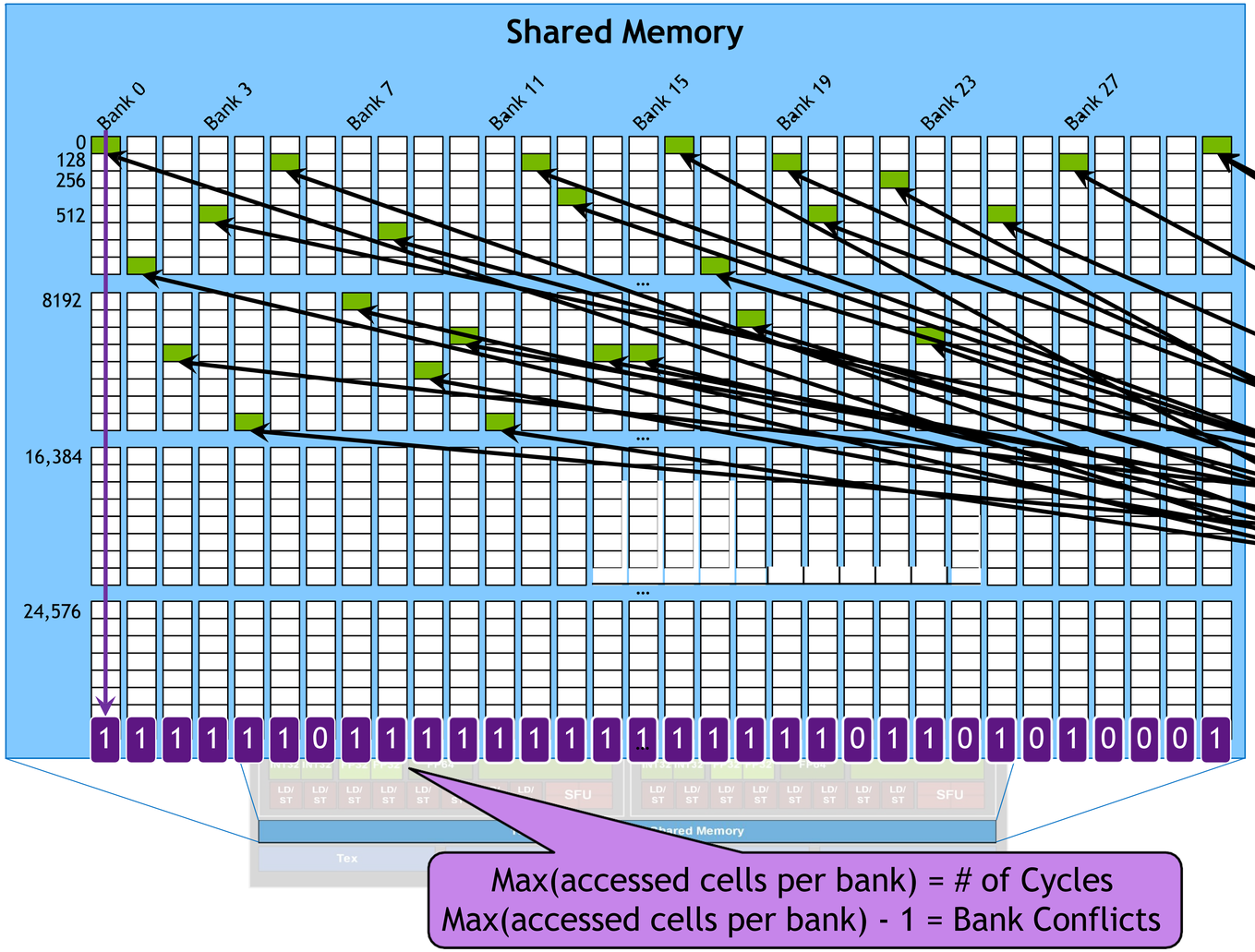


- Shared Memory Accesses
  - 4Byte read/write per bank per cycle
  - Banks can broadcast to multiple threads
  - Multiple accesses to single bank get serialized
    - Bank Conflicts

- Example Access
  - 1 warp, 32 threads
  - 4B per thread
  - ‘Random’ access
  - No Conflicts
  - 1 Cycle to access

L1 / SHARED MEMORY

Shared Memory Perspective

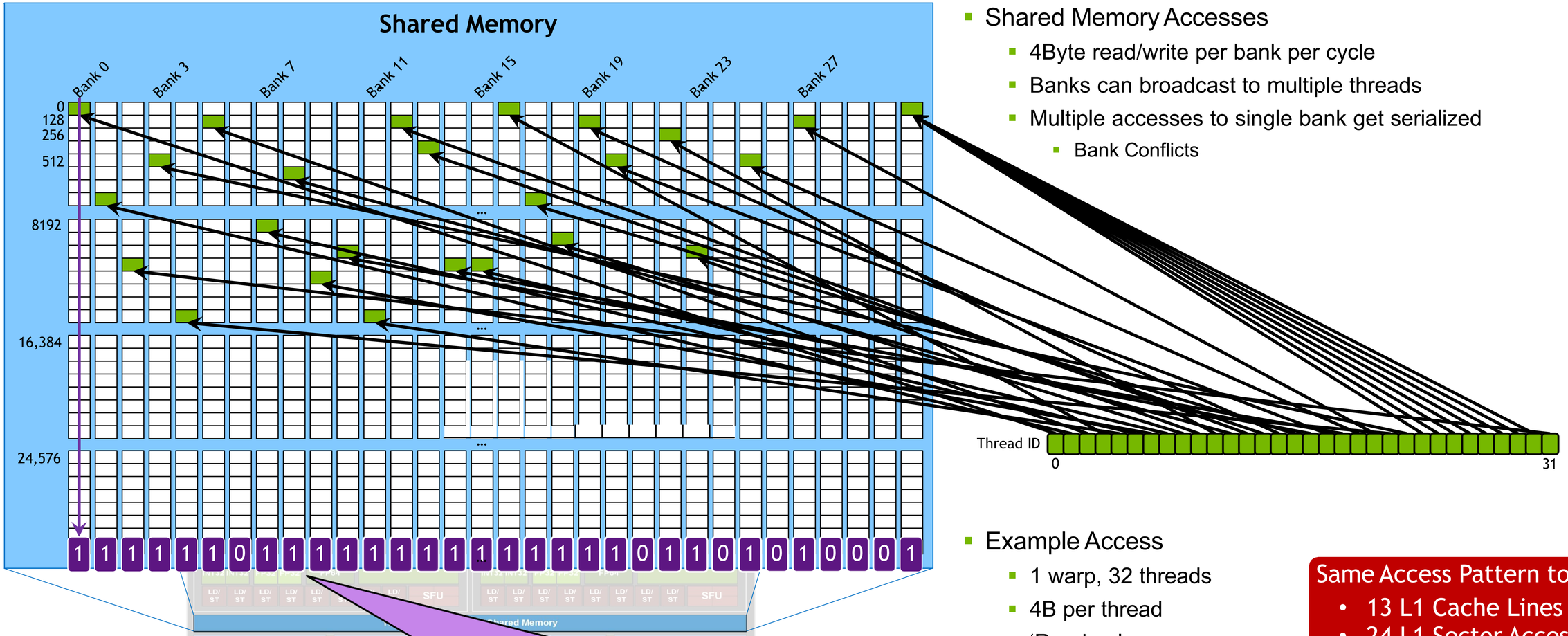


- Shared Memory Accesses
  - 4Byte read/write per bank per cycle
  - Banks can broadcast to multiple threads
  - Multiple accesses to single bank get serialized
    - Bank Conflicts

- Example Access
  - 1 warp, 32 threads
  - 4B per thread
  - ‘Random’ access
  - No Conflicts
  - 1 Cycle to access

L1 / SHARED MEMORY

Shared Memory Perspective



Max(accessed cells per bank) = # of Cycles
Max(accessed cells per bank) - 1 = Bank Conflicts

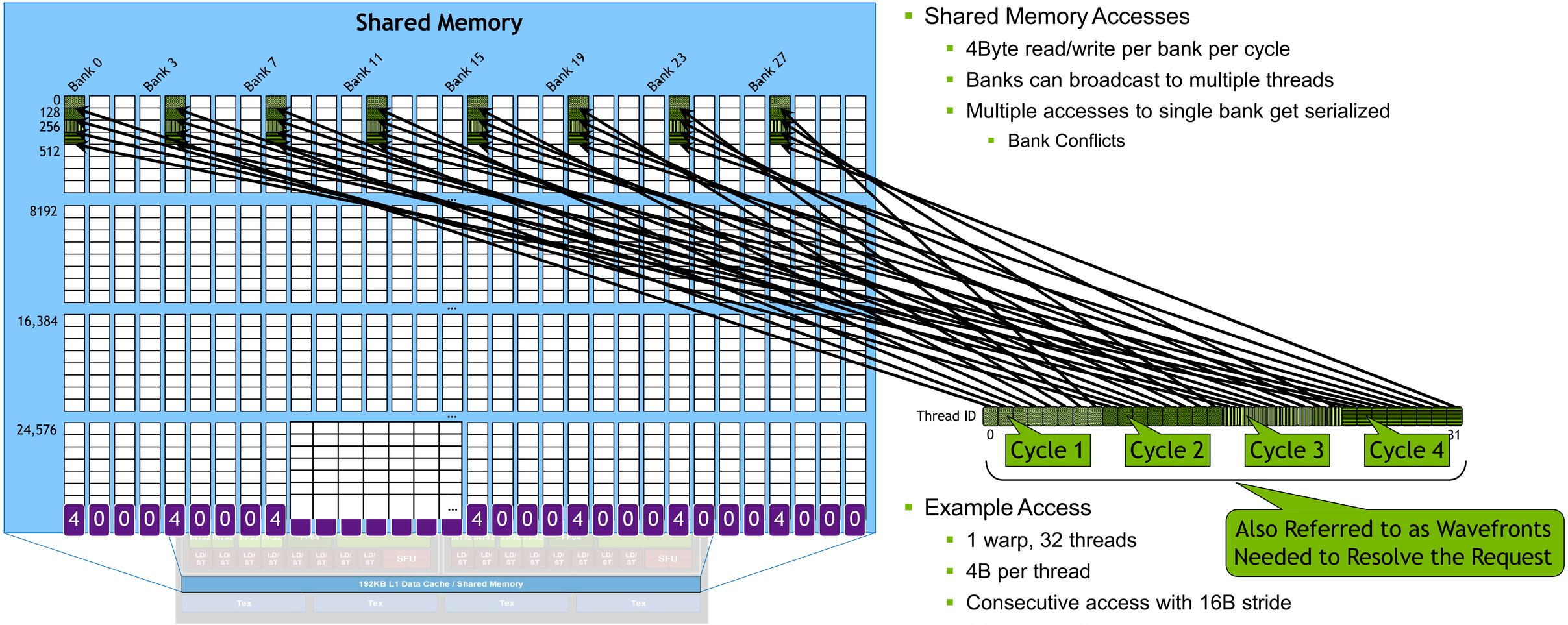
- Shared Memory Accesses
  - 4Byte read/write per bank per cycle
  - Banks can broadcast to multiple threads
  - Multiple accesses to single bank get serialized
    - Bank Conflicts

- Example Access
  - 1 warp, 32 threads
  - 4B per thread
  - 'Random' access
  - No Conflicts
  - 1 Cycle to access

Same Access Pattern to L1 Cache:
- 13 L1 Cache Lines
- 24 L1 Sector Accesses

L1 / SHARED MEMORY

Shared Memory Perspective



- Shared Memory Accesses
  - 4Byte read/write per bank per cycle
  - Banks can broadcast to multiple threads
  - Multiple accesses to single bank get serialized
    - Bank Conflicts

- Example Access
  - 1 warp, 32 threads
  - 4B per thread
  - Consecutive access with 16B stride
  - 3 Bank Conflicts
  - 4 Cycle to access

NSIGHT COMPUTE

Source Page

NVIDIA Nsight Compute

File Connection Debug Profile Tools Window Help

Connect Disconnect Terminate Profile Kernel Baselines

Untitled 2 *

Page: Source Result: 2 - 552 - read_global_store_shared_4 Add Baseline Apply Rules Occupancy Calculator Copy as Image

	Result	Time	Cycles	Regs	GPU	SM Frequency	CC	Process
Current	552 - read_global_store_shared_4bytes (65536, 1, 1)x(128, 1, 1)	239,616.00 nsecond	323,596	16	0 - NVIDIA GeForce RTX 2080 Ti	1,348,657,157.01 cycle/second	7.5	[2144] CuMemAccessPatterns.exe

View: SASS

Source: read_global_store_shared_4bytes Find... Navigation: Instructions Executed

#	Address	Source	Warp Stall Sampling (Not-issued Cycles)	Instructions Executed	Address Space	Access Operation	Access Size	L1 Wavefronts Shared	L1 Wavefronts Shared Excessive	L1 Wavefronts Shared Ideal	Register Dependencies	Predicate Dependencies
		read_global_store_shared_4bytes										
1	0000000b 00b9ea00	MOV R1, c[0x0][0x28]	77	262,144								
2	0000000b 00b9ea10	S2R R0, SR_CTAID.X	11	262,144								
3	0000000b 00b9ea20	S2R R7, SR_TID.X	37	262,144								
4	0000000b 00b9ea30	IMAD R0, R0, c[0x0][0x0], R7	176	262,144								
5	0000000b 00b9ea40	ISETP.GE.U32.AND P0, PT, R0, c[0x0][0x170], PT	49	262,144								
6	0000000b 00b9ea50	SHF.R.S32.HI R2, RZ, 0x1f, R0	14	262,144								
7	0000000b 00b9ea60	ISETP.GE.U32.AND.EX P0, PT, R2, c[0x0][0x174], PT, P0	36	262								
8	0000000b 00b9ea70	@P0 EXIT	93	262								
9	0000000b 00b9ea80	IMAD R3, R2, c[0x0][0x168], RZ	33	262								
10	0000000b 00b9ea90	IMAD.WIDE.U32 R4, R0, c[0x0][0x168], RZ	9	262,144								
11	0000000b 00b9eaa0	IMAD R3, R0, c[0x0][0x16c], R3										
12	0000000b 00b9eab0	LEA R2, P0, R4, c[0x0][0x160], 0x2										
13	0000000b 00b9eac0	IADD3 R3, R5, R3,										
14	0000000b 00b9ead0	LEA.HI.X R3, R4,	22	262,144								
15	0000000b 00b9eae0	LDG.E.SYS R0, [R2	61	262,144	Global	Load						
16	0000000b 00b9eaf0	IMAD R5, R7, c[0x0][0x168], RZ	0	262,144								
17	0000000b 00b9eb00	FSETP.GEU.AND P0, PT, R0, RZ, PT	8,432	262,144								
18	0000000b 00b9eb10	STS [R5.X4], R0	58	262,144	Shared	Store	32	1,048,576	786,432	262,144		
19	0000000b 00b9eb20	BAR.SYNC 0x0	33	262,144								
20	0000000b 00b9eb30	@P0 EXIT	717	262,144								
21	0000000b 00b9eb40	LDS.U R0, [R7.X4]	219	0				0	0	0		
22	0000000b 00b9eb50	FADD R5, RZ, R0	0	0								

Number of Ideal Wavefronts (1x)

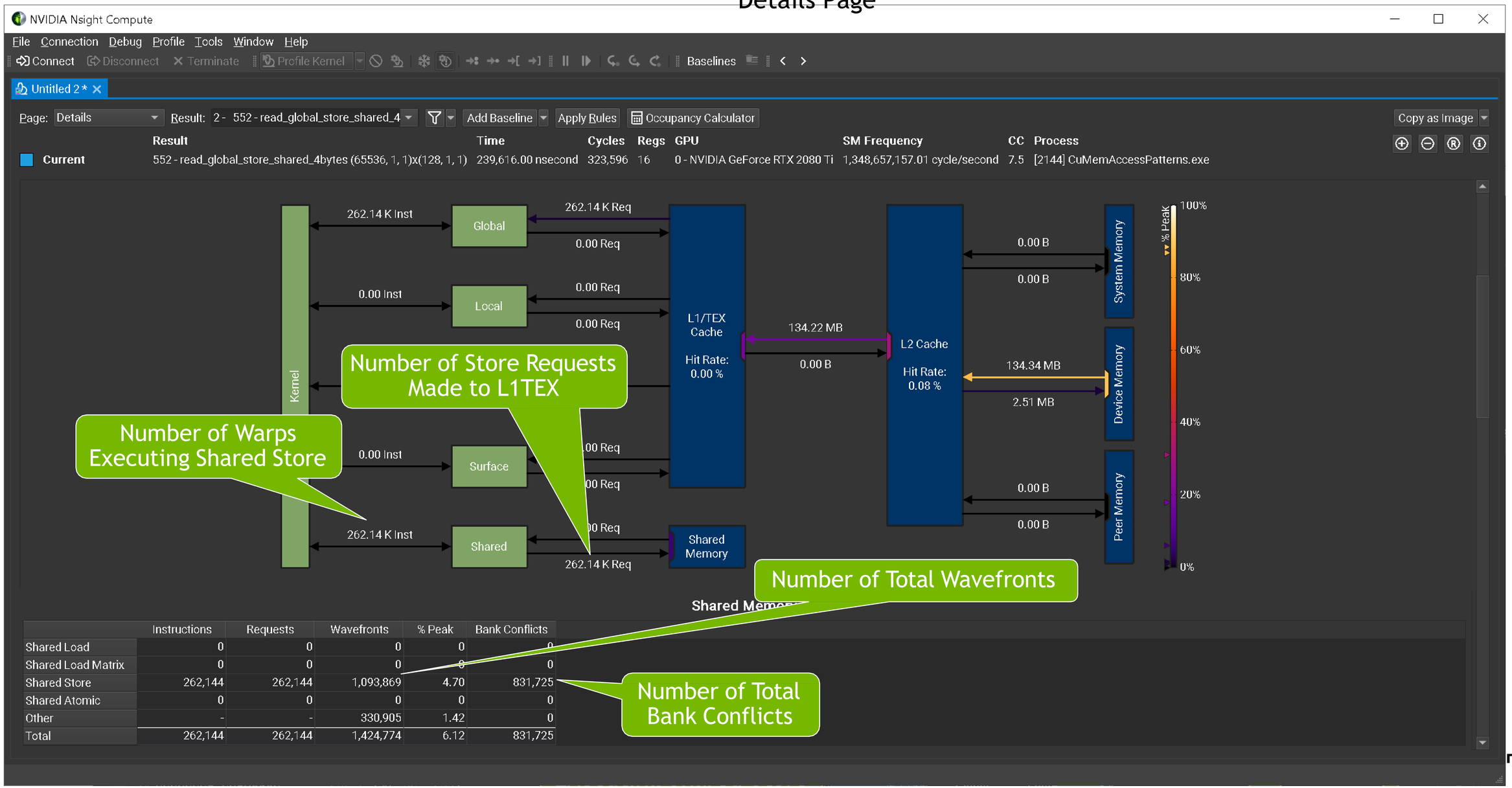
Number of Excessive Wavefronts for this Store = Bank Conflicts (3x)

Number of Shared Wavefronts for This Store (4x)

Number of Warps Executing This Shared Store

NSIGHT COMPUTE

Details Page



NSIGHT COMPUTE

Details Page

Source: read_global_store_shared_4bytes Find... Navigation: Instructions Executed

#	Address	Source	arp Stall Sampling Not-issued Cycles)	Instructions Executed	Address Space	Access Operation	Access Size	L1 Wavefronts Shared	L1 Wavefronts Shared Excessive	L1 Wavefronts Shared Ideal	Register Dependencies	Predica Dependenci
		read_global_store_shared_4bytes										
1	0000000b 00b9ea00	MOV R1, c[0x0][0x28]	77	262,144								
2	0000000b 00b9ea10	S2R R0, SR_CTAID.X	11	262,144								
3	0000000b 00b9ea20	S2R R7, SR_TID.X	37	262,144								
4	0000000b 00b9ea30	IMAD R0, R0, c[0x0][0x0], R7	176	262,144								
5	0000000b 00b9ea40	ISETP.GE.U32.AND P0, PT, R0, c[0x0][0x170], PT	49	262,144								
6	0000000b 00b9ea50	SHF.R.S32.HI R2, RZ, 0x1f, R0	14	262,144								
7	0000000b 00b9ea60	ISETP.GE.U32.AND.EX P0, PT, R2, c[0x0][0x174], PT, P0	36	262,144								
8	0000000b 00b9ea70	@P0 EXIT	93	262,144								
9	0000000b 00b9ea80	IMAD R3, R2, c[0x0][0x168], RZ	33	262,144								
10	0000000b 00b9ea90	IMAD.WIDE.U32 R4, R0, c[0x0][0x168], RZ	9	262,144								
11	0000000b 00b9eaa0	IMAD R3, R0, c[0x0][0x16c], R3	27	262,144								
12	0000000b 00b9eab0	LEA R2, P0, R4, c[0x0][0x160], 0x2	0	262,144								
13	0000000b 00b9eac0	IADD3 R3, R5, R3, RZ	20	262,144								
14	0000000b 00b9ead0	LEA.HI.X R3, R4, c[0x0][0x164], R3, 0x2, P0	22	262,144								
15	0000000b 00b9eae0	LDG.E.SYS R0, [R2]	61	262,144	Global	Load						
16	0000000b 00b9eaf0	IMAD R5, R7, c[0x0][0x168], RZ	0	262,144								
17	0000000b 00b9eb00	FSETP.GEU.AND P0, PT, R0, RZ, PT	8,482	262,144								
18	0000000b 00b9eb10	STS [R5.X4], R0	58	262,144	Shared	Store	32	1,048,576	786,432	262,144		
19	0000000b 00b9eb20	BAR.SYNC 0x0	33	262,144								
20	0000000b 00b9eb30	@P0 EXIT	717	262,144								
21	0000000b 00b9eb40	LDS.U R0, [R7.X4]	219	0				0	0	0		
22	0000000b 00b9eb50	FADD R5, RZ, R0	0	0								

Number of Excessive Wavefronts for this Store = Bank Conflicts (3x)

?

Shared Memory

	Instructions	Requests	Wavefronts	% Peak	Bank Conflicts
Shared Load	0	0	0	0	0
Shared Load Matrix	0	0	0	0	0
Shared Store	262,144	262,144	1,093,869	4.70	831,725
Shared Atomic	0	0	0	0	0
Other	-	-	330,905	1.42	0
Total	262,144	262,144	1,424,774	6.12	831,725

Number of Total Bank Conflicts

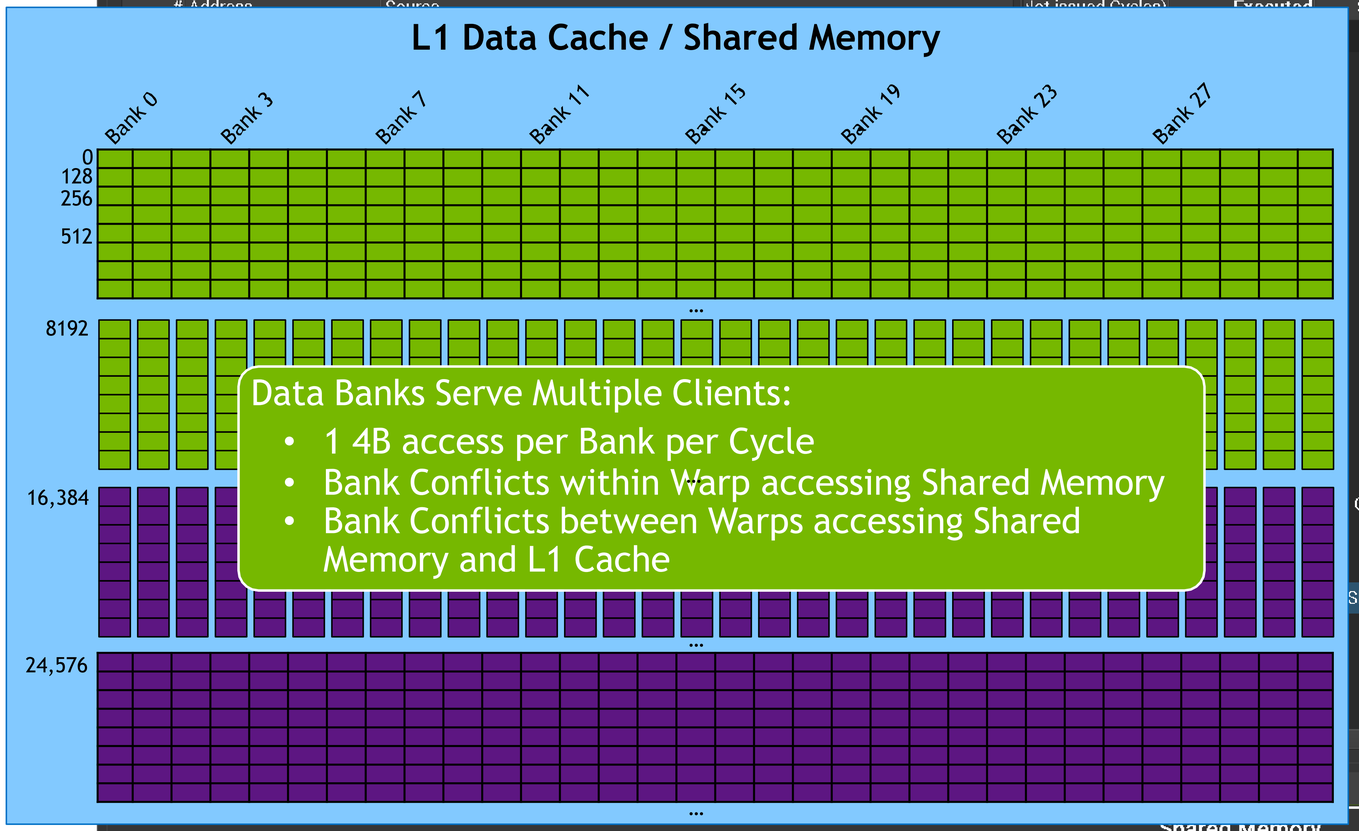
NVIDIA

NSIGHT COMPUTE

Details Page

Source: read_global_store_shared_4bytes | Find... | Navigation: Instructions Executed

#	Address	Source	arp Stall Sampling (Not Issued Cycles)	Instructions Executed	Address Space	Access Operation	Access Size	L1 Wavefronts Shared	L1 Wavefronts Shared Excessive	L1 Wavefronts Shared Ideal	Register Dependencies	Predica Dependenci
					Global	Load						
					Shared	Store	32	1,048,576	786,432	262,144		
								0	0	0		



Data Banks Serve Multiple Clients:

- 1 4B access per Bank per Cycle
- Bank Conflicts within Warp accessing Shared Memory
- Bank Conflicts between Warps accessing Shared Memory and L1 Cache

Number of Excessive Wavefronts for this Store = Bank Conflicts (3x)

Shared Memory

	Instructions	Requests	Wavefronts	% Peak	Bank Conflicts
Shared Load	0	0	0	0	0
Shared Load Matrix	0	0	0	0	0
Shared Store	262,144	262,144	1,093,869	4.70	831,725
Shared Atomic	0	0	0	0	0
Other	-	-	330,905	1.42	0
Total	262,144	262,144	1,424,774	6.12	831,725

Number of Total Bank Conflicts

NVIDIA

Vectorized Memory Accesses

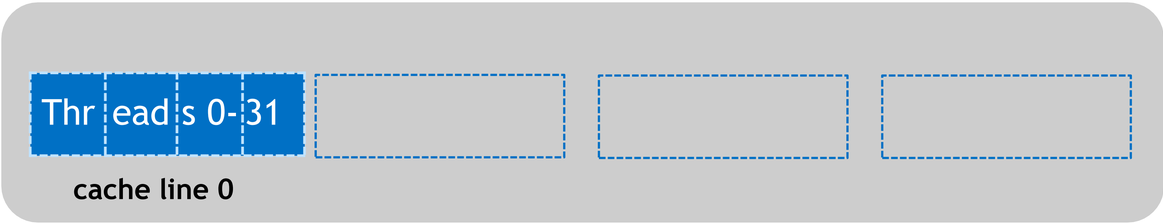
Multi-word as well as multi-thread

warp

contiguous, aligned memory access

int

Memory



Fills 1 cache line in a single fetch.

Vectorized Memory Accesses

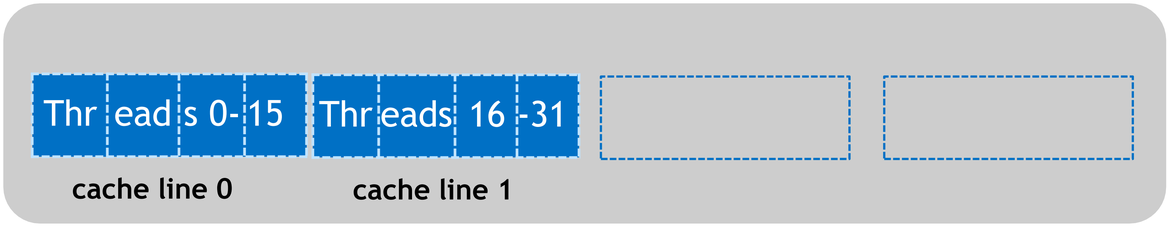
Multi-word as well as multi-thread

warp

contiguous, aligned memory access

int2

Memory



Fills 2 cache lines in a single fetch.

Vectorized Memory Accesses

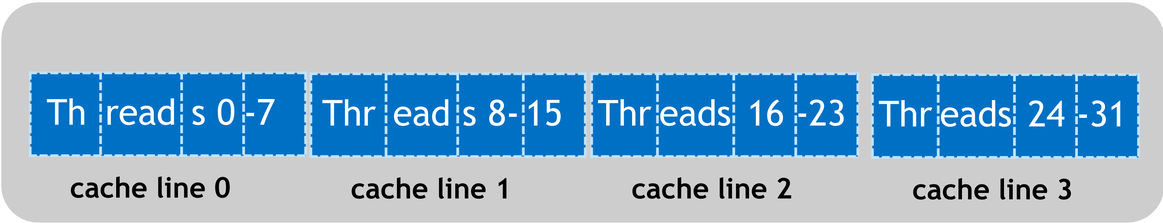
Multi-word as well as multi-thread

warp

contiguous, aligned memory access

int4

Memory



Fills 4 cache lines in a single fetch.

Vectorized Memory Accesses

Multi-thread, multi-word

- Vectorized **global** and **shared memory** accesses.
  - Require aligned data.
  - 64- or 128-bit width.

- Less executed instructions!
- More bytes in-flight!

- Approaches to enable vectorization:
  1) By using vector data types, e.g., `float2`, `float4`.
  2) Explicitly by casting to vector pointers.
     1) Proper alignment required.

```
// Using vectors data types
__global__
void copy(const float2 * __restrict__in,
          float2 * __restrict__out,
          int N)
{
  auto grid = cg::this_grid();
  int tid = grid.thread_rank();
  int stride = grid.size();

  for (int i = tid; i < N / 2; i += stride) {
    out[i] = in[i];
    // Same as:
    // out[i].x = in[i].x;
    // out[i].y = in[i].y;
  }
}
```

Vectorized Memory Accesses

Performance Analysis

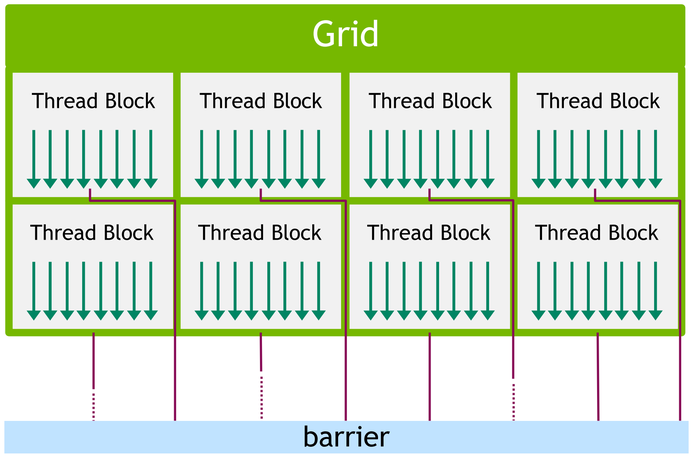
- **Experimental setup:**
  - NVIDIA H100 SXM, 1980 MHz
  - Problem size = 2^28
  - Thread block size = 256

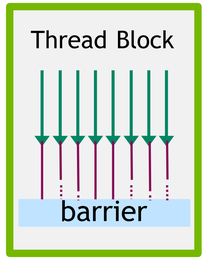
Implementation	Main Memory Bandwidth Utilization (%)	GPU Time (ms)
float	60.62	1.033
float2	84.34	0.737
float4	88.82	**0.706**

Synchronization

Barriers

CUDA/Software





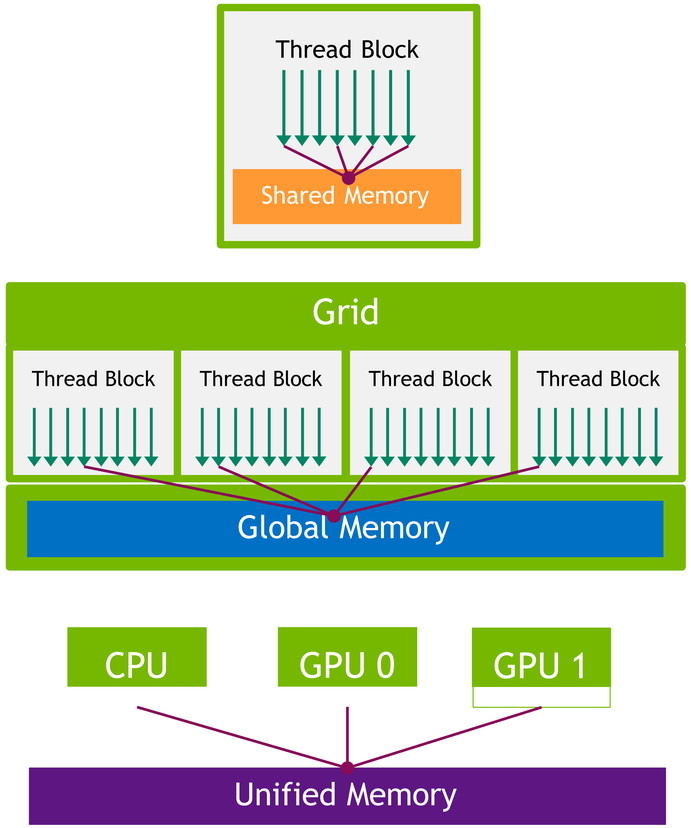
warp = 32 threads
barrier

- **Grid** boundary.
  - Kernel completion.
  - `grid_group::sync()` via Cooperative Groups API
    - Requires the kernel to be launched via the `cudaLaunchCooperativeKernel()` API
    - **Slow!** Avoid unless necessary.
- **Thread-block** boundary.
  - `__syncthreads()`
  - `thread_block::sync()` via Cooperative Groups API
  - **Fast!** The most common synchronization level.
- **Warp** or **sub-warp** boundary.
  - `__syncwarp()`
  - `coalesced_group::sync()` via Cooperative Groups API
  - **Very fast!**

Atomics

Memory spaces

CUDA/Software

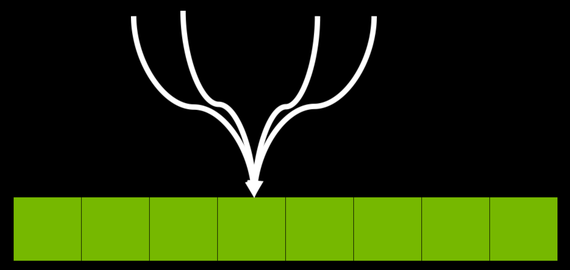


- Read-modify-write operations on 16–, 32-, 64- or 128-bit words.
  - Available as CUDA primitives or C++ atomics through `libcu++` extended API.
- **Shared memory** atomics.
- **Global memory** atomics.
  - Facilitated by special hardware in the L2 cache.
- **Unified memory** atomics.

Using Atomics Efficiently

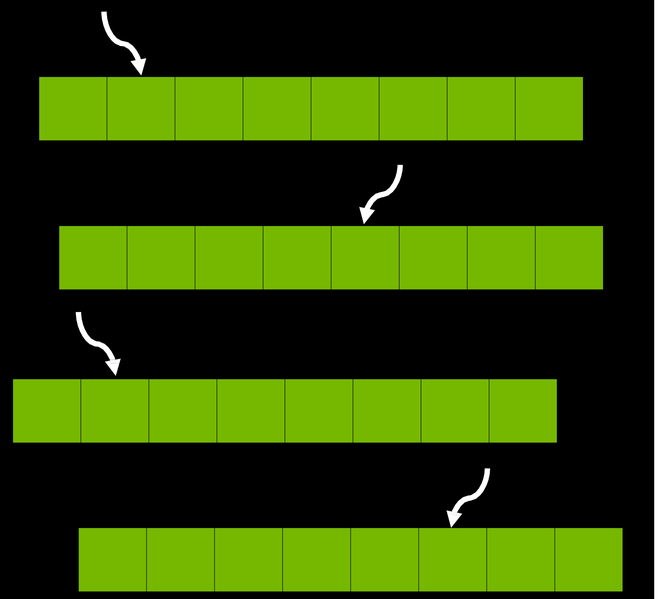
Access Patterns

Same address

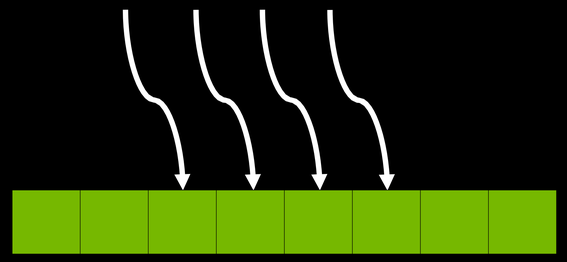


Serialized!
Least efficient access pattern.

Scattered



Coalesced



Most efficient access pattern.

Using Atomics Efficiently

Example #1: find the maximum value of an array

- **Problem description:** given an input array, find the maximum element in the array.
- **Naïve implementation:** every thread find its local maximum and then atomically updates the global maximum.
  - N / elements_per_thread same-address global atomics.

```
__global___
void find_max(const int * __restrict___in, int *max, int N)
{
  int grid_tid = cg::this_grid().thread_rank();
  int grid_stride = cg::this_grid().num_threads();

  // Find my local maximum
  int local_max = INT_MIN;
  for (int i = grid_tid; i < N; i += grid_stride) {
    if (in[i] > local_max)
      local_max = in[i];
  }

  // Atomically update the global max
  atomicMax(max, local_max);
}
```

Using Atomics Efficiently

Example #1: find the maximum value of an array

- **Optimization #1**: maintain a block-level max in shared memory.
  - Reduces the number of same-address global atomics by a factor equal to the thread block size.

```
__global__
void find_max(const int * __restrict__in, int *max, int N)
{
  int grid_tid = cg::this_grid().thread_rank();
  int grid_stride = cg::this_grid().num_threads();
  auto block = cg::this_thread_block();
  int block_tid = block.thread_rank();

  __shared__int block_max;

  // Find my local maximum int local_max = INT_MIN;
  for (int i = grid_tid; i < N; i += grid_stride) {
    if (in[i] > local_max)
      local_max = in[i];
  }

  // Atomically update the block-level max
  atomicMax(&block_max, local_max);
  block.sync();

  // Atomically update the global max
  if (block_tid == 0)
    atomicMax(max, block_max);
}
```

Using Atomics Efficiently

Example #1: find the maximum value of an array

- **Optimization #2**: use a parallel reduction to calculate the block-level max in shared memory.

```
//Assumes a block dimension of 256
__global___
void find_max(const int * __restrict___in, int *max, int N) {
  …
  auto tile = cg::tiled_partition<32>(block);
  extern __shared___int sdata[];

  // Find my local maximum as before
  // Each thread puts its local max into shared memory
  sdata[block_tid] = thread_max;

  // Block-level reduction
  if (block_tid < 128) {
    if (sdata[block_tid + 128] > thread_max)
      thread_max = sdata[block_tid + 128];
    sdata[block_tid] = thread_max;
  }
  block.sync();
  if (block_tid < 64) {
    if (sdata[block_tid + 64] > thread_max)
      thread_max = sdata[block_tid + 64];
    sdata[block_tid] = thread_max;
  }
  block.sync();

  // Warp-level reduction
  if (tile.meta_group_rank() == 0) {
    thread_max = cg::reduce(tile, thread_max,
                            cg::greater<int>());
  }

  if (block_tid == 0)
    atomicMax(max, thread_max);
}
```

Performance Analysis

- **Experimental setup:**
  - NVIDIA H100 SXM, 1980 MHz
  - Problem size = 2^28
  - Uniform distribution (-50, 50).

Implementation	Thread Block Size	GPU Time (ms)
global atomics	256	6.839
shared memory atomics	256	1.334
shared memory reduction	256	**1.066**

Using Atomics Efficiently

Example #2: vector update

- **Problem description:** a = a + b * c

```
__global___
void vector(const float * __restrict___b,
            const float * __restrict___c,
            float * __restrict___a,
            int N) {
  int  grid_tid  =  cg::this_grid().thread_rank();
  int grid_stride = cg::this_grid().num_threads();

  for (int i = grid_tid; i < N; i += grid_stride) {
    a[i] += b[i] * c[i];
  }
}
```

Memory operations = 3 reads + 1 write

Using Atomics Efficiently

Example #2: vector update

- **Optimization**: use atomics to update each vector element even though atomicity is not required.
  - Offload some of the computation to the L2 cache.
  - Saves reading the value of `a[i]` in registers.
  - This reduces the latency to compute each element of the vector.
  - Can result in **more bytes in-flight**!

```
__global___
void vector(const float * __restrict___b,
            const float * __restrict___c,
            float * __restrict___a,
            int N) {
  int  grid_tid  =  cg::this_grid().thread_rank();
  int grid_stride = cg::this_grid().num_threads();

  for (int i = grid_tid; i < N; i += grid_stride) {
    atomicAdd(&a[i], b[i] * c[i]);
  }
}
```

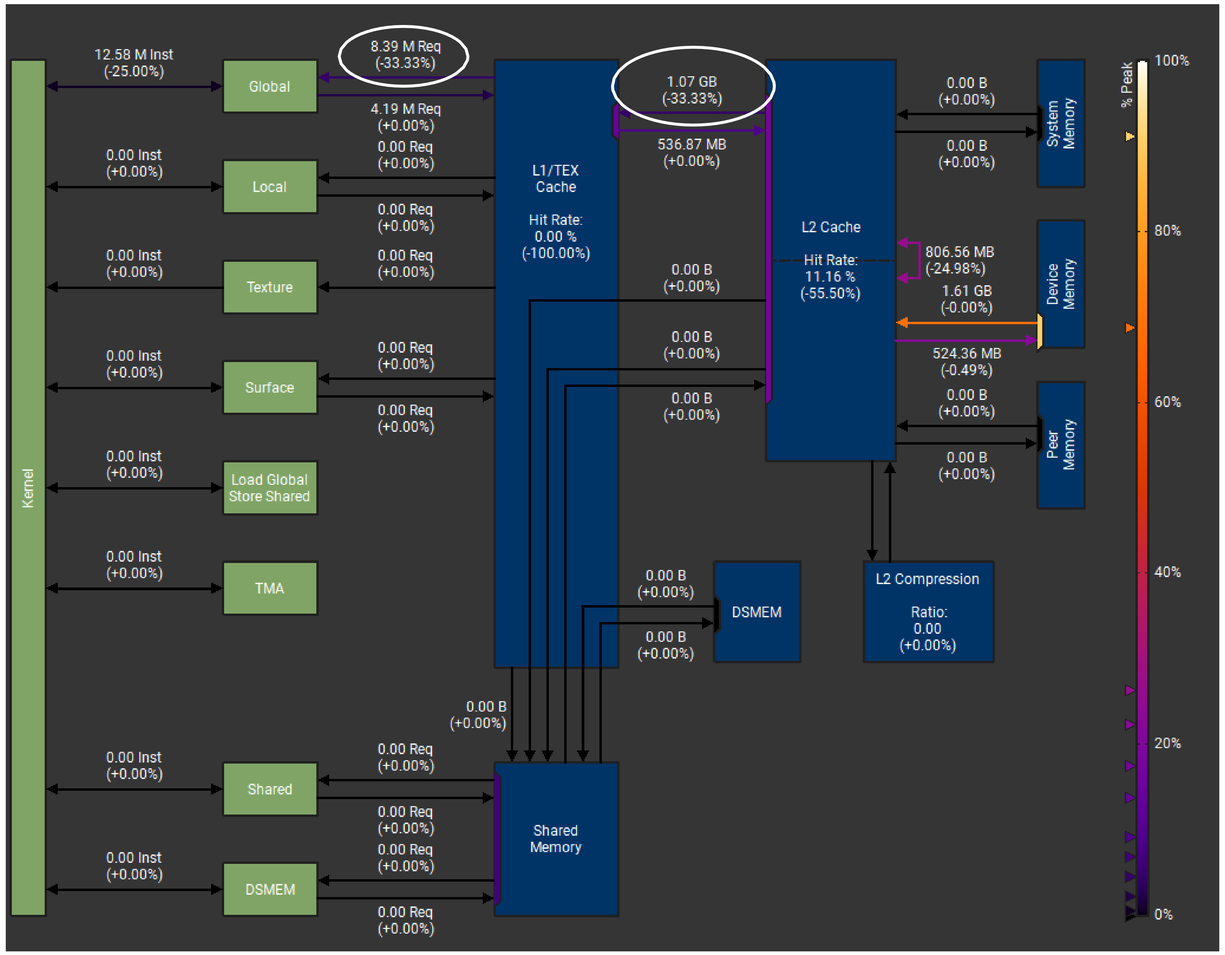
Memory operations = 2 reads + 1 write

Performance Analysis

NVIDIA H100 SXM

- **Experimental setup:**
  - NVIDIA H100 SXM, 1980 MHz
  - Problem size = 2^27

NCU Memory Chart (Transfer Size)

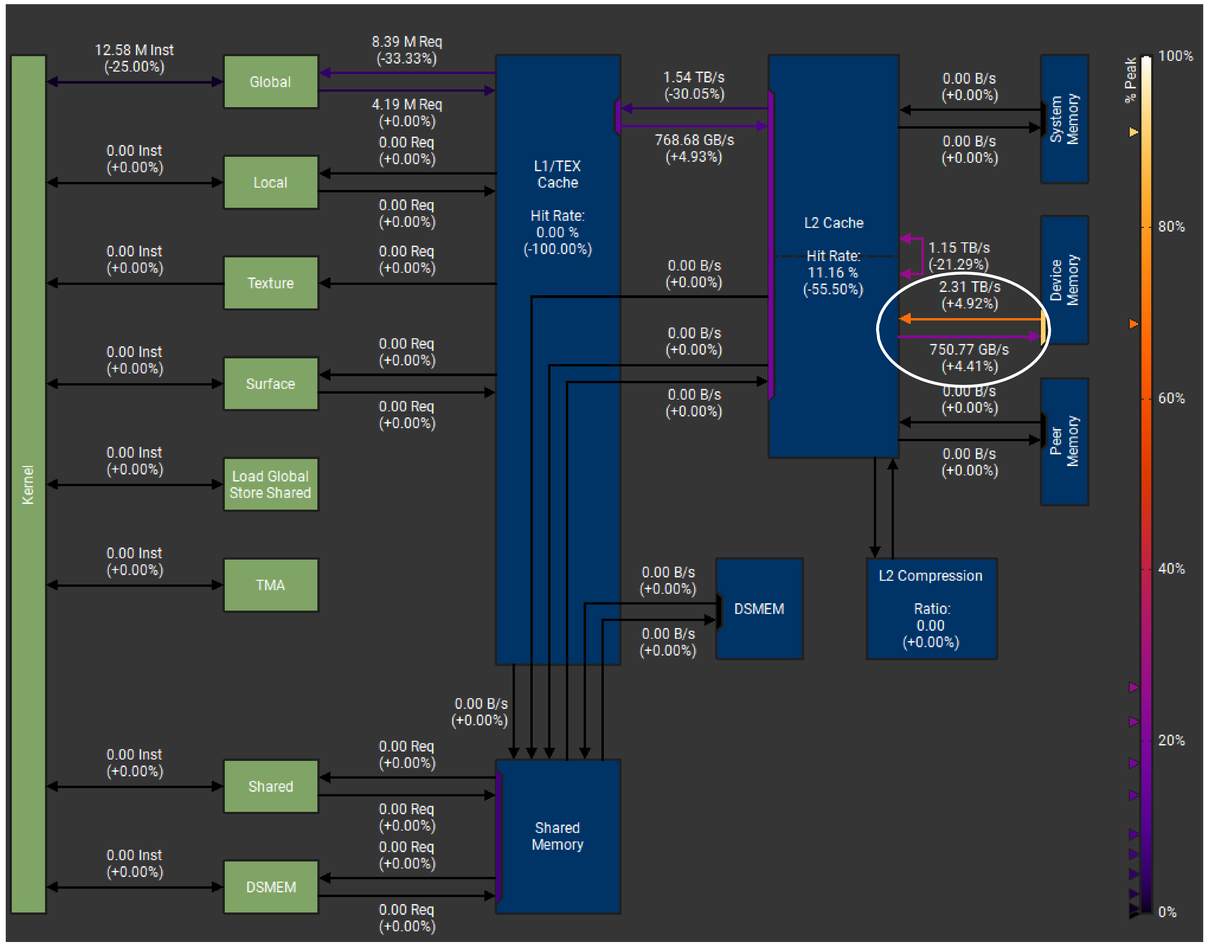


Performance Analysis

NVIDIA H100 SXM

- **Experimental setup:**
  - NVIDIA H100 SXM, 1980 MHz
  - Problem size = 2^27

NCU Memory Chart (Throughput)

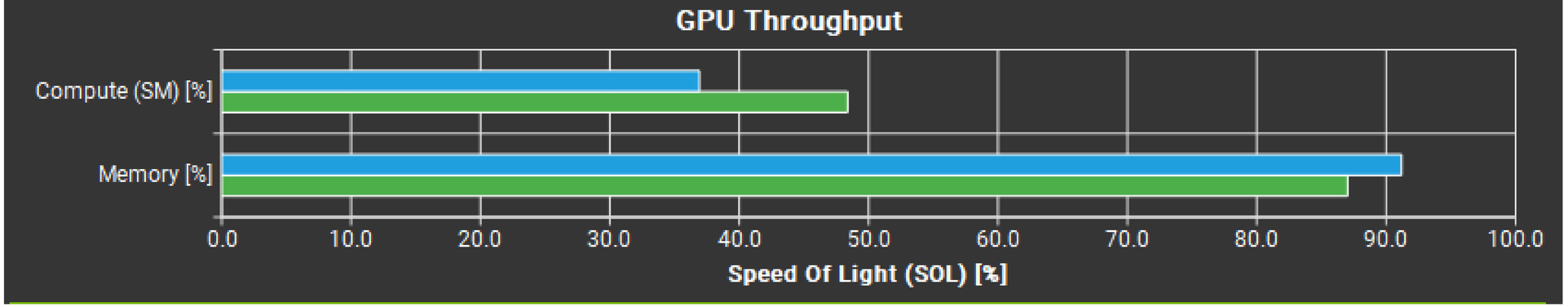


Performance Analysis

NVIDIA H100 SXM

	Report	Result	Time	Cycles	GPU	SM Frequency
Current	vector_atomics_grid1	1149 - vector_atomics (524288, 1, 1)x(256, 1, 1)	698.43 usecond	1,382,709	0 - NVIDIA H100 80GB HBM3	1.98 cycle/nsecond
Baseline 1	vector_baseline	1149 - vector (524288, 1, 1)x(256, 1, 1)	732.83 usecond	1,450,699	0 - NVIDIA H100 80GB HBM3	1.98 cycle/nsecond

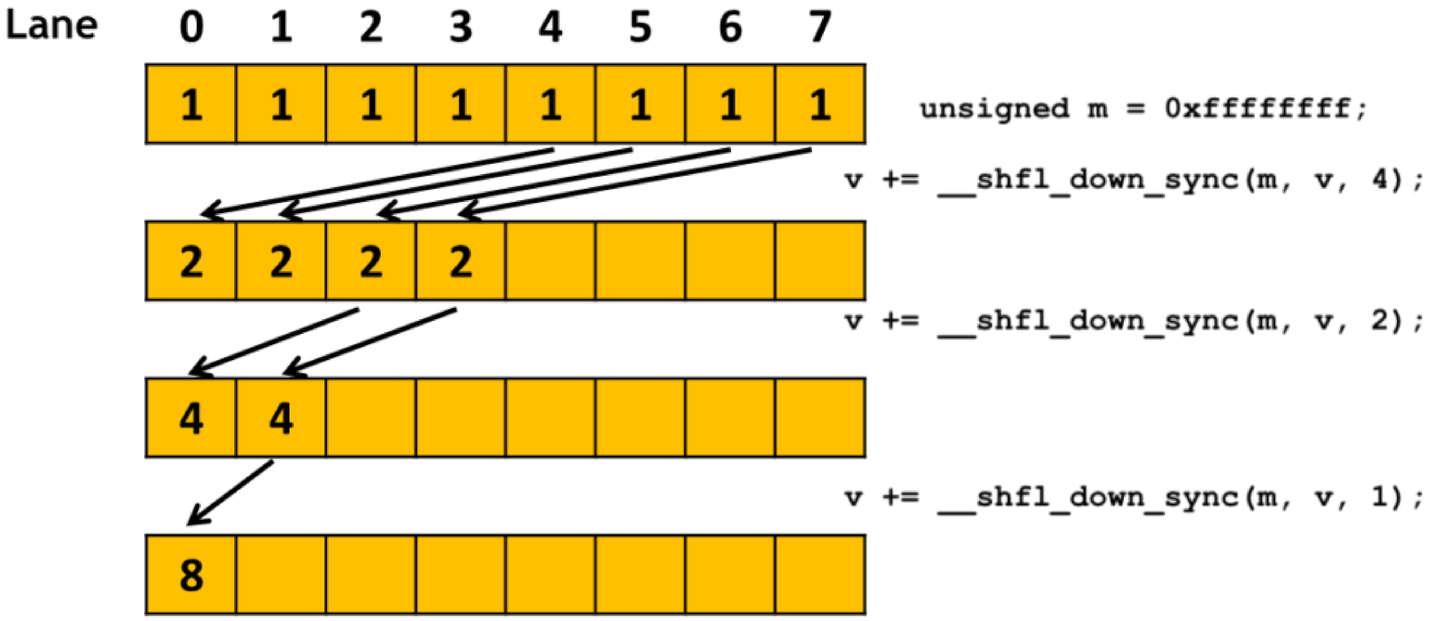
Compute (SM) Throughput [%]	36.93	(-23.68%)	Duration [usecond]	698.43	(-4.69%)
Memory Throughput [%]	91.19	(+4.81%)	Elapsed Cycles [cycle]	1,382,709	(-4.69%)
L1/TEX Cache Throughput [%]	16.36	(-10.92%)	SM Active Cycles [cycle]	1,375,961.72	(-4.45%)
L2 Cache Throughput [%]	68.43	(-15.59%)	SM Frequency [cycle/nsecond]	1.98	(-0.05%)
DRAM Throughput [%]	91.19	(+4.81%)	DRAM Frequency [cycle/nsecond]	2.62	(-0.01%)



- ~**5%** increase in memory throughput translates into a corresponding reduction in execution time. **Why?**
  - This kernel is DRAM bandwidth bound.

Warp-Level Primitives

Warp Shuffle/Vote/Match



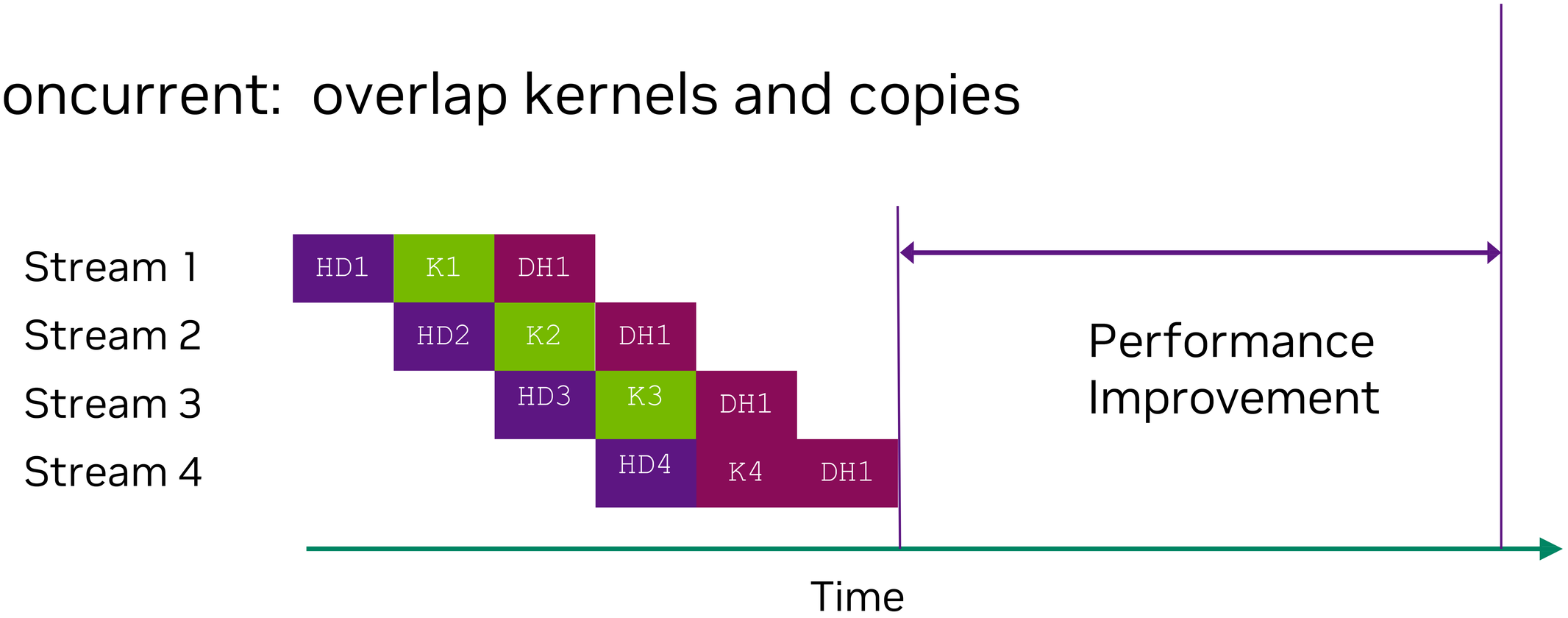

```
#define FULL_MASK 0xffffffff
for (int offset = 16; offset > 0; offset /= 2)
    val += __shfl_down_sync(FULL_MASK, val, offset);
```

Concurrency Through Pipelining

CUDA Streams and Concurrency

- Serial:

- Concurrent: overlap kernels and copies



CUDA Optimization Basics (Recap)

- SIMT, Thread/Memory Hierarchy
- Global Memory Coalesced Access
- L1 / Shared Memory Bank Conflict
- Vectorized Memory Access
- Synchronization (Barriers, Cooperative Groups)
- Atomics
- Efficient Atomics Example
- Warp-level Primitives
- Streams

Hands-On!

- Make sure you :

  (1) Can access the slides

  (2) Install Nsight Systems &
  Nsight Compute

  (3) Download sample profile resul

https://github.com/j3soon/advanced-topics-on-hpc-supp

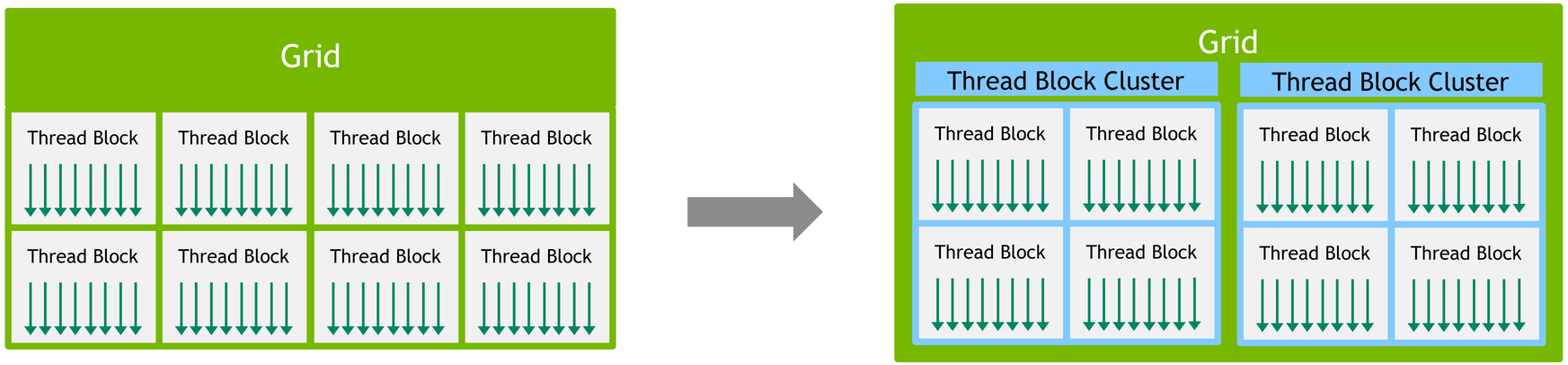
Advanced Topics on High Performance Computing (Part 2)

Johnson Sun | Solution Architect | NVIDIA AI Technology Center (NVAITC)

Advanced CUDA Optimization

Thread Block Clusters

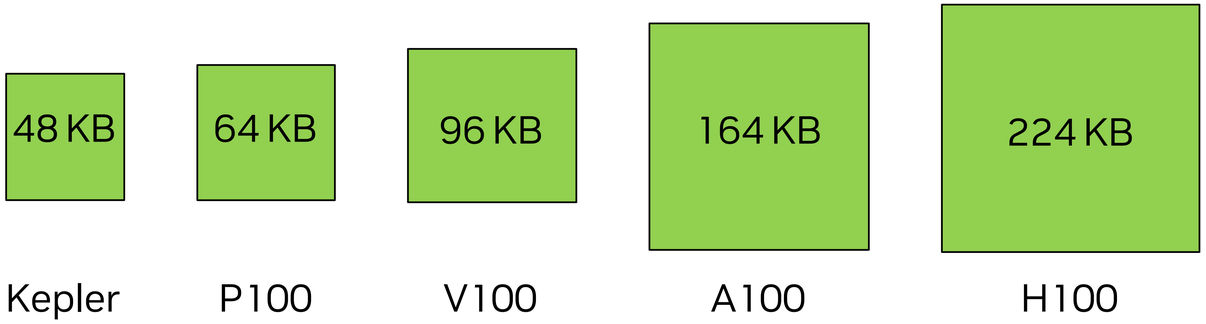
- For Hopper GPUs, CUDA introduced an optional level in the thread hierarchy called **Thread Block Clusters**.
- Thread blocks in a cluster are guaranteed to be concurrently scheduled and enable efficient cooperation and data sharing for threads across multiple SMs.
- For more information on this topic visit GTC session **[S62192]: "Advanced Performance Optimization in CUDA"**.

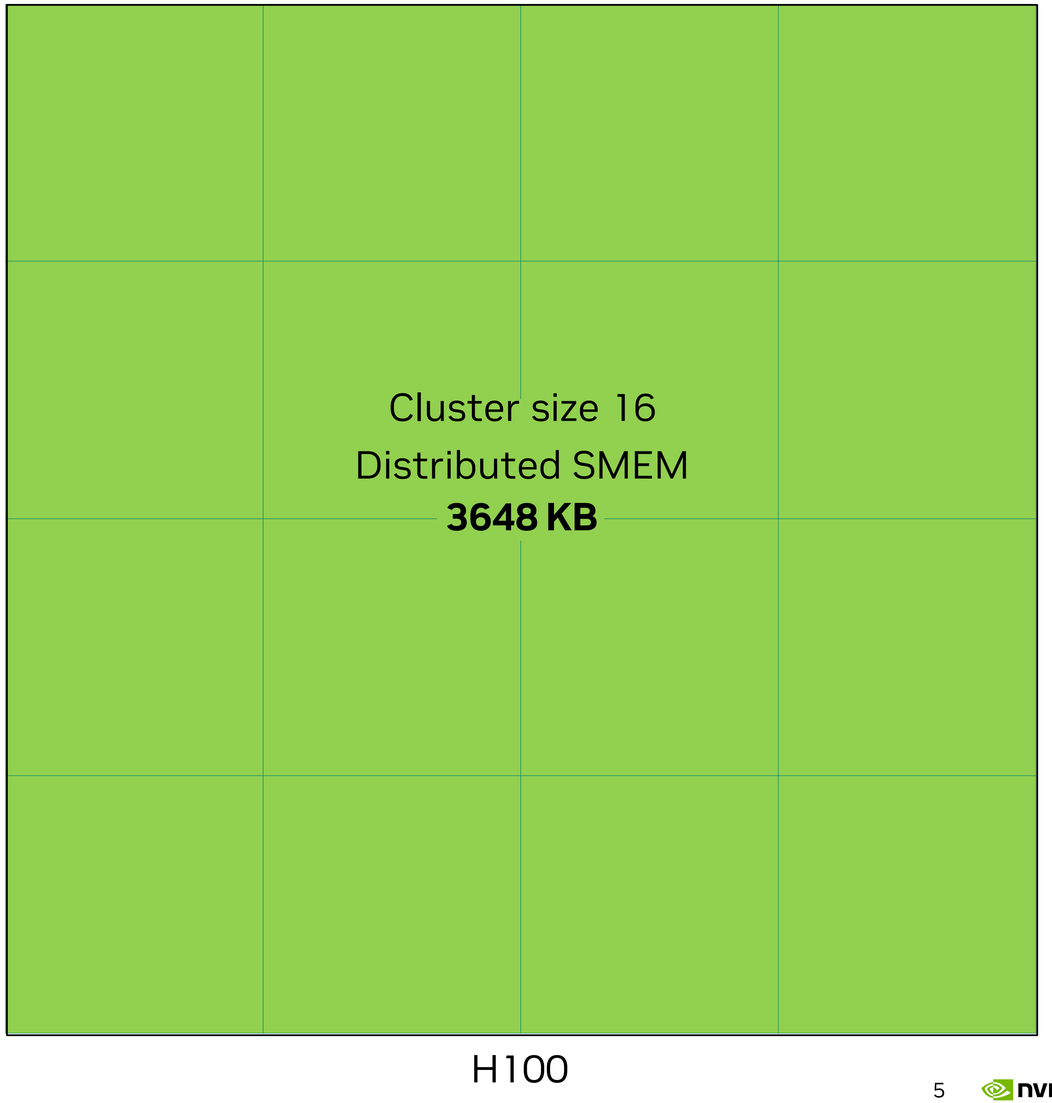


Cooperative Grid Arrays

Teaser

Increased SMEM saves GMEM trips in many algorithms…





Cooperative Thread Arrays (CTAs) = Thread Block
Cooperative Grid Array (CGA) = Thread Block Cluster

namespace cg = cooperative_groups;

Cooperative Grid Arrays

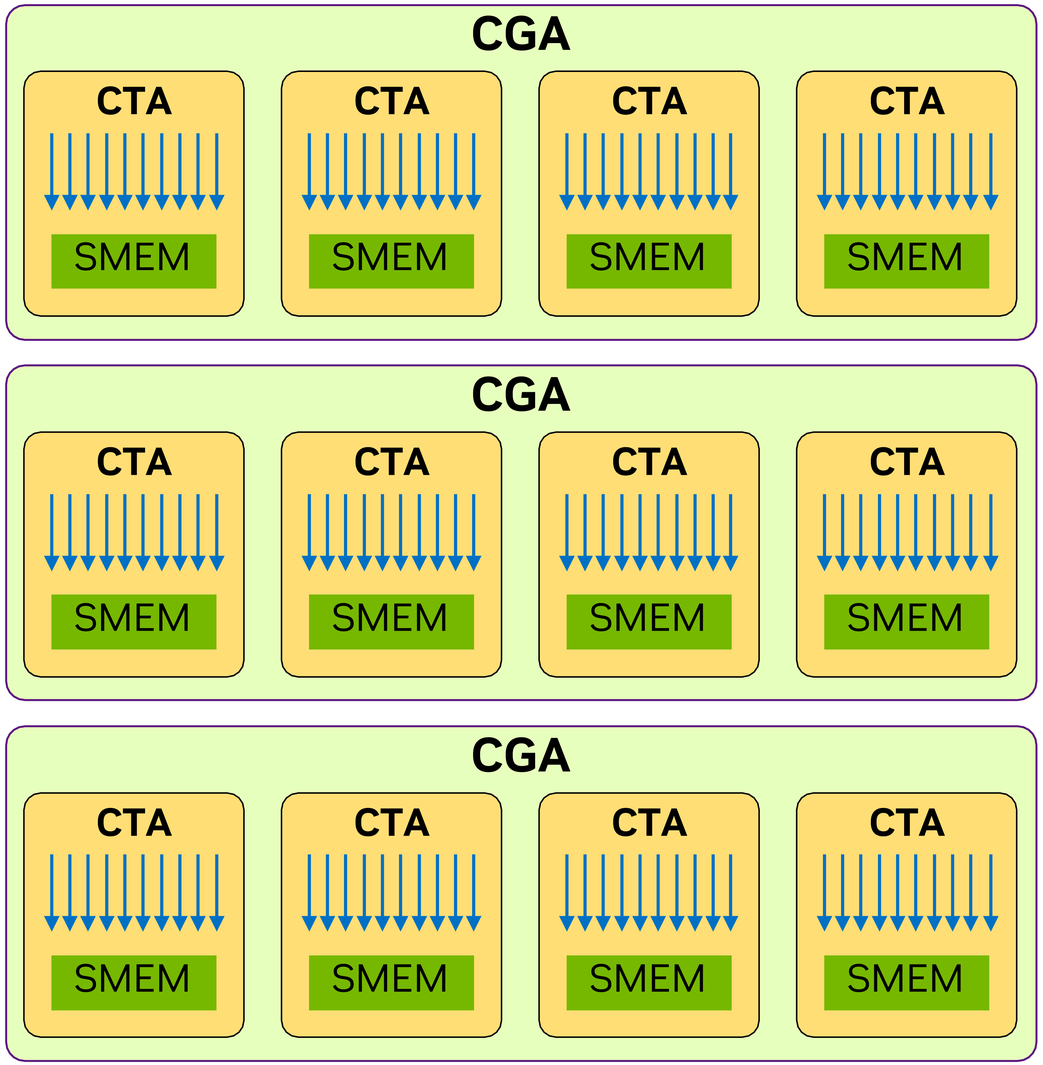
Properties

New level in the hierarchy:

T < CTA < **CGA** < GRID

Clusters:

- Support guaranteed for up to 8 CTAs / CGA
  need to opt-in for more than 8 (max 16 on Hopper)
- CTAs can be logically organized in 3D blocks:
  - linear rank: `cg.this_cluster().block_rank()`
  - 3D rank : `cg.this_cluster().block_index()`
- CTAs in cluster are co-scheduled
- Thread synchronization:
  - CTA ➔ `cg.this_block().sync()`
  - CGA ➔ `cg.this_cluster().sync()`
- Possible resource under-utilization:
  cluster size > 2 may lead to unused SMs



Cooperative Grid Arrays

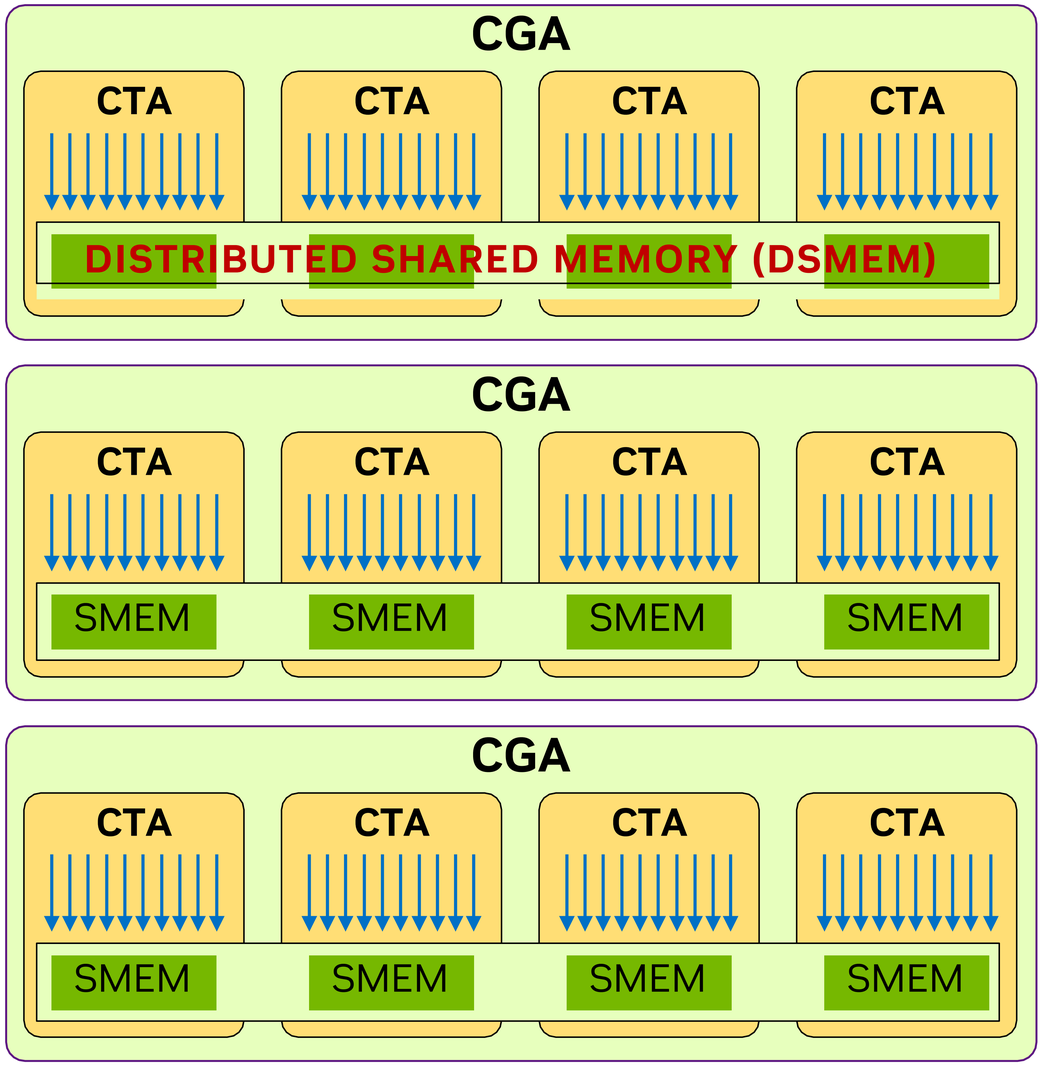
DSMEM properties

Distributed SMEM (DSMEM):

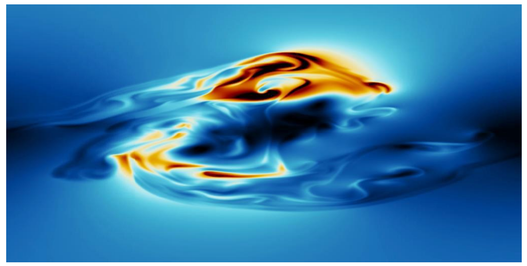
- SMEM within cluster is accessible by all CTAs ➔ DSMEM
  E.g., H100 total DSMEM size ↗ 3.6 MB
- DSMEM allows:
  - Access: LD / ST / ATOM
  - Async copies into remote DSMEM
  - Multicasting
- Note: remote DSMEM is slower than local:
  low latency, but relatively low bandwidth
- Remote DSMEM has to be mapped:

```
__shared___T smem;
auto dsmem_ptr =
    cg::this_cluster().map_shared_rank(&smem, rank);
```

- **Important:**
  make sure remote CTAs are alive while accessing DSMEM !
  (use `cluster.sync()` accordingly)

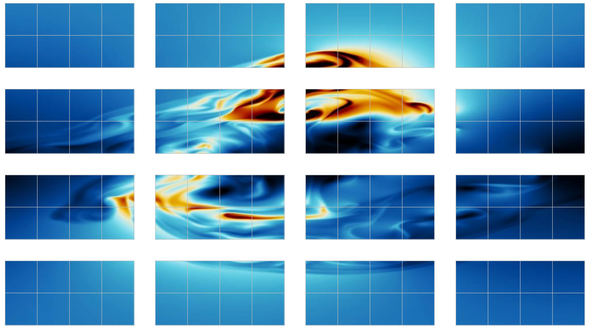


EXAMPLE: HIERARCHICAL EXECUTION, DATA EXCHANGE AND SYNCHRONIZATION



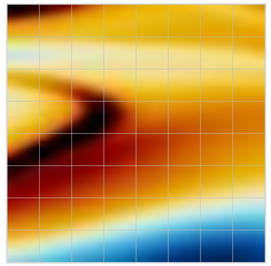
1

Launch cooperative grid of 4x4 clusters, each with 4x2 blocks

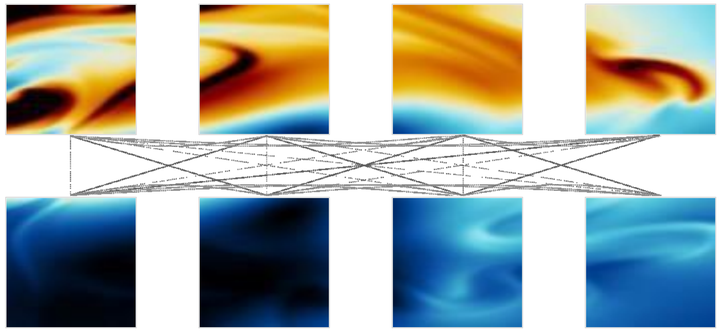


2

Use cooperative cluster indexing to pull data from neighbours



Solve locally within each block



4

Synchronize with cluster and solve at cluster level

Synchronize with grid and solve at grid level; go to 2

Use distributed shared memory for fast inter-block communication.

Memory Model

Weak memory model

Thread 0:

```
data = 42;
flag = 1;
```

Thread 1:

```
while (!flag) {}
assert(data == 42);   // not guaranteed !!
```

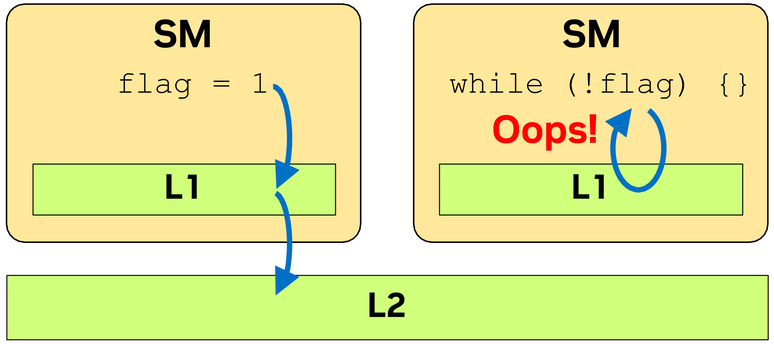
Moreover, Undefined Behavior (UB)!

CUDA uses **Weak Memory Model**:

- Writes by one thread may not be observed in the same order by other threads
- Concurrent access of same memory with at least one `'write'` is a **CONFLICT (UNDEFINED BEHAIVIOR)**
- Compiler assumes no-conflict code, **it's a programmer's responsibility to synchronize**

Synchronization between threads requires:

1. **Strong ordering: "happens before" relation**
2. **Reaching correct Point of Coherency**



Asynchronous Barriers

Barrier intro

CUDA barriers provide (within scope!):

- Synchronization points (including async mem transactions)
- Memory ordering

block till all threads arrived
writes **before** `__syncthreads`
visible **after** `__syncthreads`

write SMEM
`__syncthreads()`
read SMEM

write SMEM
`barrier.arrive()`
mark arrived (non-blocking)
writes **before** `arrive`
visible **after** `wait`
Independent ops (e.g., on other MEM)
`barrier.wait()`
block till all threads marked arrived
writes **before** `arrive`
visible **after** `wait`
read SMEM

If there is enough independent work for all threads, the barrier effectively becomes free because the `wait` instruction can retire immediately, as all threads have already arrived.

Asynchronous Data Copies

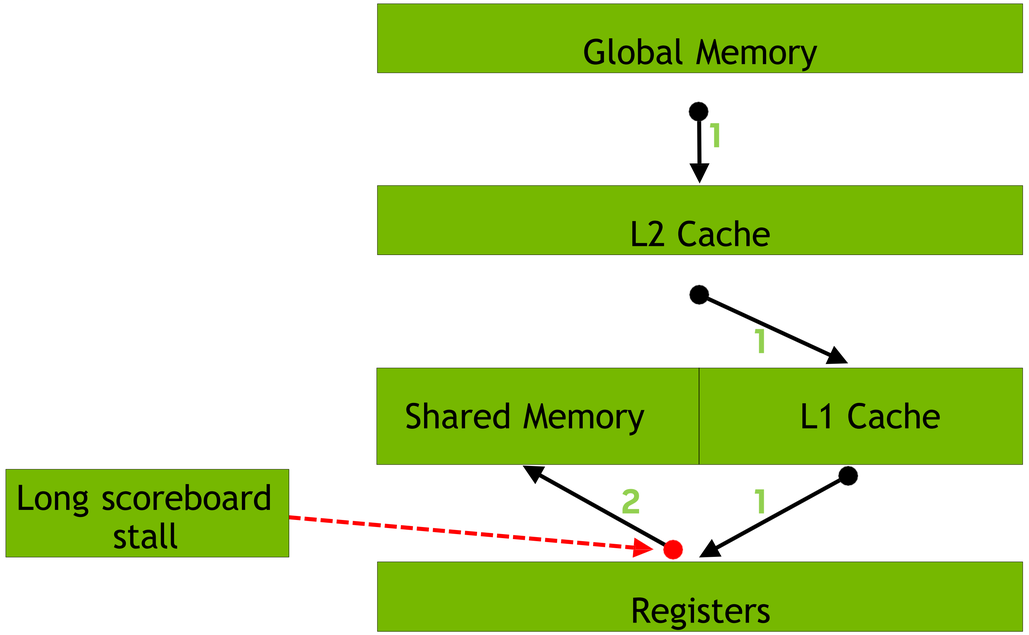
Synchronous copies dissected

```
T* gmem;
__shared__T smem[];
smem[sind] = gmem[gind];
```

Involves:

1. Copying data from global memory to register
2. Copying data from register to shared memory

- STALL [Scoreboard wait]
- Wasting registers and L1 bandwidth



Asynchronous Data Copies

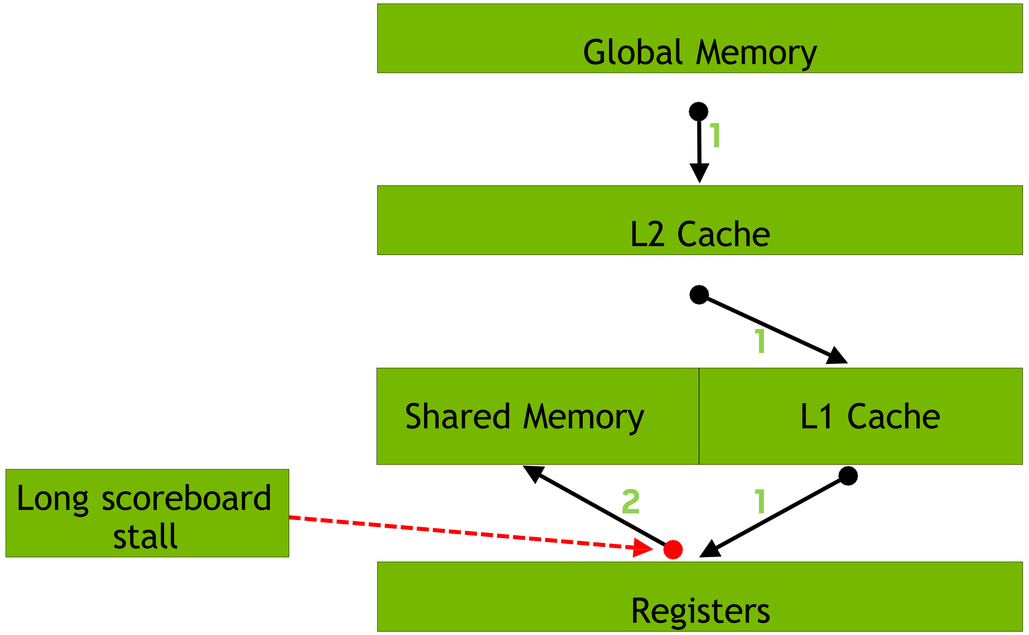
Synchronous copies dissected

```
T* gmem;
__shared__T smem[];
smem[sind] = gmem[gind];
```

Involves:

1. Copying data from global memory to register
2. Copying data from register to shared memory

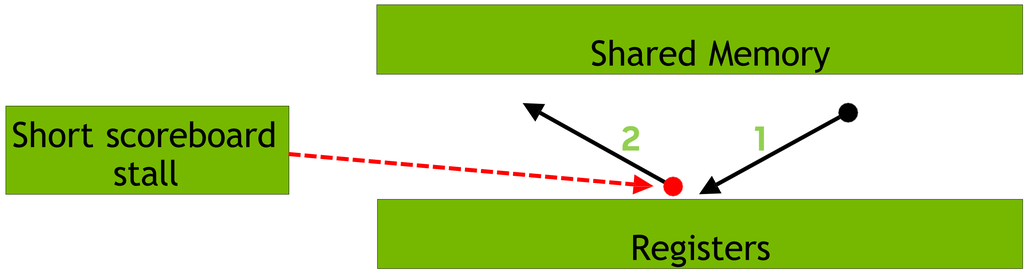
- STALL [Scoreboard wait]
- Wasting registers and L1 bandwidth




```
smem[sind1] = smem[sind2];
```

Involves:

1. Copying data from shared memory to register
2. Copying data from register to shared memory



Asynchronous Data Copies

Synchronous copies – less bytes in flight ☹

Representative code:

```
T* gmem;
__shared___T smem[..];
for (int i = 0; i < n; ++i) {
    smem[sind] = g_mem[gind];
    gind += stride;
    __syncthreads();
    // compute with smem[]
    __syncthreads();
}
```

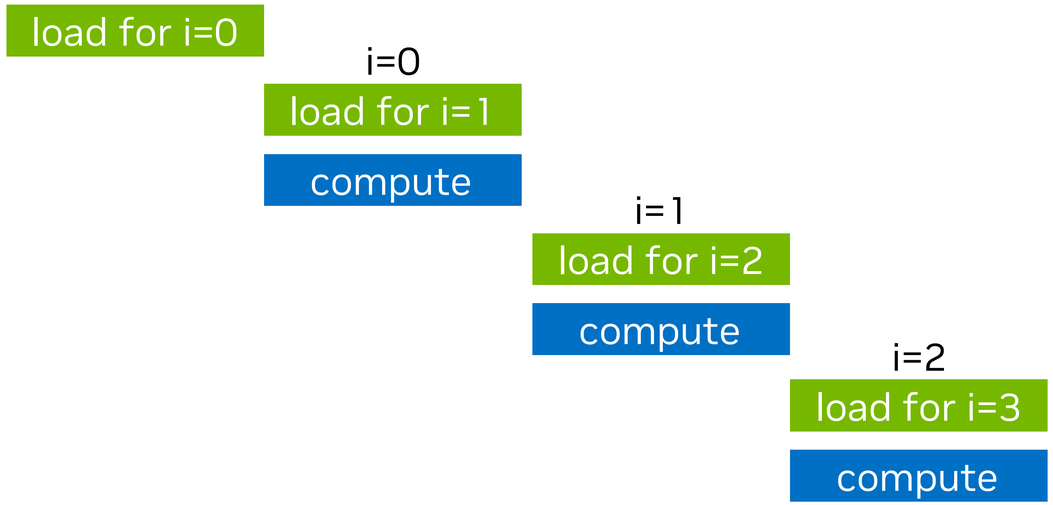
Copy and compute are sequenced.
Cannot fire SMEM prefetch during compute…

Asynchronous Data Copies

Asynchronous copies – more bytes in flight ☺

Asynchronous copy benefits:

- **Increases bytes in flight.**
- Other benefits:
  - Register bypass
  - Less L1 traffic
  - Less MIO pressure (fewer instructions)
  - Better dependency management



Asynchronous Data Copies

Asynchronous copies summary

HW	Direction	Operation	Synchronization	C++	SASS
Ampere	GMEM ➡ SMEM	Asynchronous version of `smem[sind] := gmem[gind]`	Thread-local SMEM barrier	`__pipeline_memcpy_async`	`LDGSTS`
Hopper	REG➡DSMEM	Asynchronous version of `dsmem[sind] := var`	DSMEM barrier	`cuda::ptx::st_async`	`STAS`

Hopper, TMA	GMEM➡(D)SMEM	1D – 5D block copy
		Uniform
			(D)SMEM barrier	`cuda::device::experimental::`
				`cp_async_bulk_tensor_Nd_global_to_shared`
				`cp_async_bulk_tensor_Nd_shared_to_global`
					`UTMALDG`

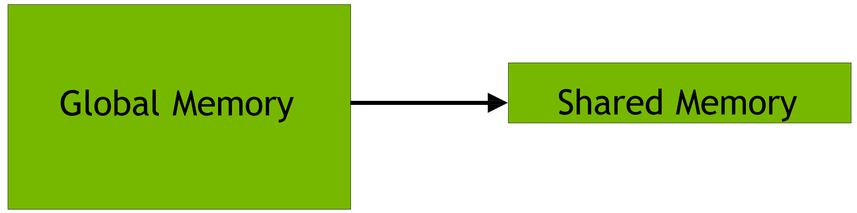
Hopper, TMA	SMEM➡GMEM		Thread-local		`UTMASTG`

Hopper, TMA	GMEM➡SMEM	1D block copy
		Uniform
			SMEM barrier	`cuda::device::experimental::`
				`cp_async_bulk_global_to_shared`
				`cp_async_bulk_shared_to_global`
					`UBLKCP`

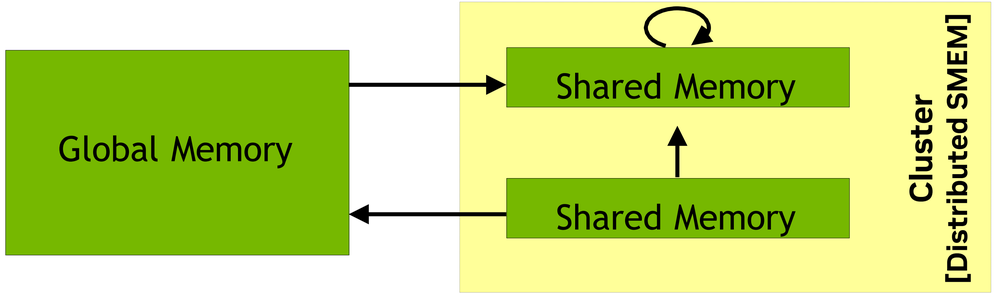
Hopper, TMA	SMEM➡GMEM		Thread-local		
Hopper, TMA	SMEM➡(D)SMEM		(D)SMEM barrier	`cuda::ptx::cp_async_bulk`*	

* Available in latest github CCCL

Ampere (LDGSTS)



Hopper (TMA)



Asynchronous Data Copies

Asynchronous copies summary

HW	Direction	Operation	Synchronization	C++	SASS
Ampere	GMEM ➡ SMEM	Asynchronous version of `smem[sind] := gmem[gind]`	Thread-local SMEM barrier	`__pipeline_memcpy_async`	`LDGSTS`
Hopper	REG➡DSMEM	Asynchronous version of `dsmem[sind] := var`	DSMEM barrier	`cuda::ptx::st_async`	`STAS`

Hopper, TMA	GMEM➡(D)SMEM	1D – 5D block copy
		Uniform
			(D)SMEM barrier	`cuda::device::experimental::`
				`cp_async_bulk_tensor_Nd_global_to_shared`
				`cp_async_bulk_tensor_Nd_shared_to_global`
					`UTMALDG`

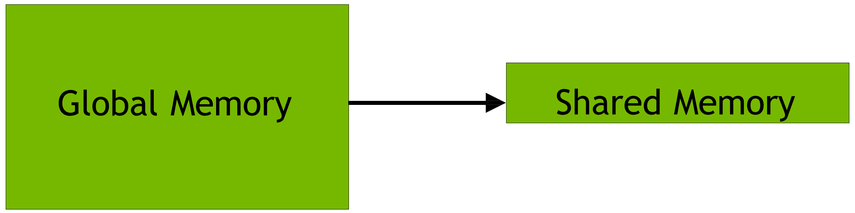
Hopper, TMA	SMEM➡GMEM		Thread-local		`UTMASTG`

Hopper, TMA	GMEM➡SMEM	1D block copy
		Uniform
			SMEM barrier	`cuda::device::experimental::`
				`cp_async_bulk_global_to_shared`
				`cp_async_bulk_shared_to_global`
					`UBLKCP`

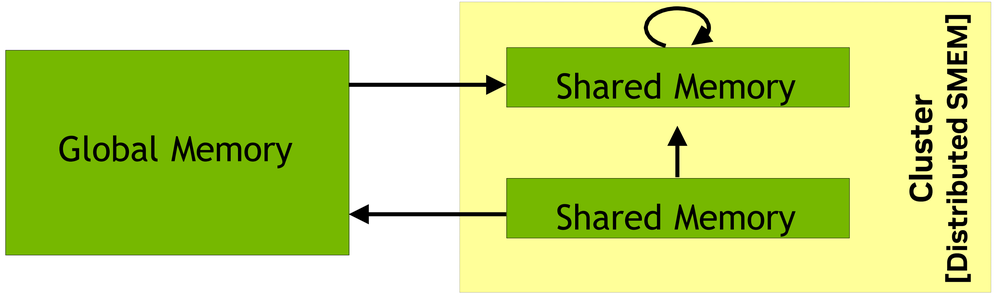
Hopper, TMA	SMEM➡GMEM		Thread-local		
Hopper, TMA	SMEM➡(D)SMEM		(D)SMEM barrier	`cuda::ptx::cp_async_bulk`*	

* Available in latest github CCCL

Ampere (LDGSTS)



Hopper (TMA)

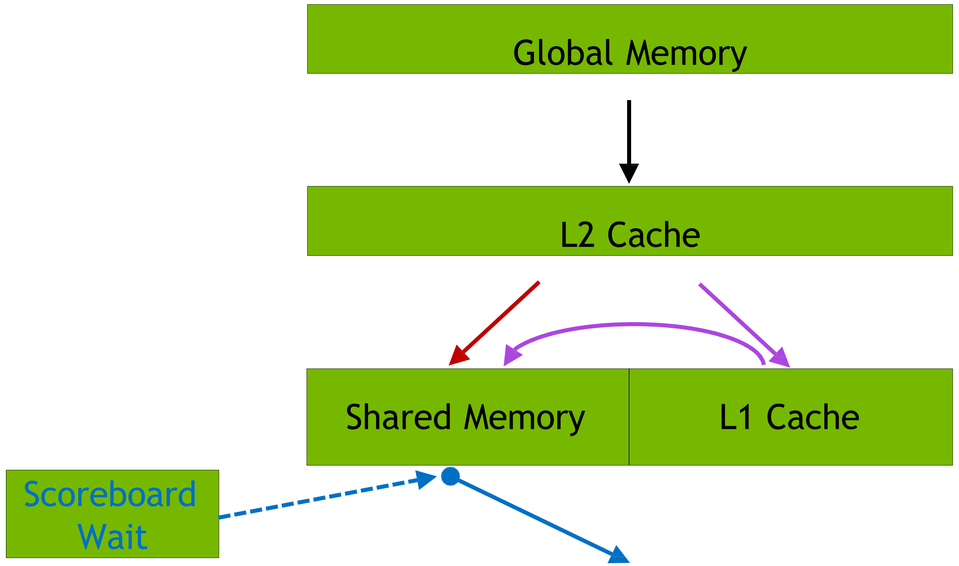


Asynchronous Data Copies

LDGSTS dissected

Asynchronous version of `smem[sind] = gmem[gind];`

```
__pipeline_memcpy_async(&smem[sind], &gmem[gind], sizeof(T));
```



Features:

1. Bypassing registers
   (less registers may lead to better occupancy / reduced spillage)
2. Two modes:
   - L1 bypass mode: kicks in when `sizeof(T)` and alignment are 16 B (benefit: no L1 pollution)
   - L1 access mode: when `sizeof(T)` and alignment are 4 or 8 B

Hopper note: shared memory must be local SMEM when CTA is part of CGA.

Use cases:

1. Batching loads in conditional code
2. Staging/Prefetching (producer-consumer)

Asynchronous Data Copies

LDGSTS copies and synchronization

```
T* gmem;

__shared___T smem[..];

__pipeline_commit();

__pipeline_memcpy_async(...);
...
__pipeline_memcpy_async(...);

__pipeline_commit();


__pipeline_wait_prior(N);
```

– issues async load instructions

– (non-blocking) batch-counting dependency barrier for previous loads released when data have written to SHMEM (for the executing thread)

– blocks until all previous commits except last `N` are “ready”

Asynchronous Data Copies

Ways to code LDGSTS

```
// primitives  <cuda_pipeline.h>:
__pipeline_memcpy_async(&smem[sind], &gmem[gind], sizeof(T)); // LDGSTS
__pipeline_commit();        // LDGDEPBAR
__pipeline_wait_prior(N);   // DEPBAR.LE SB,N

// libcu++  <cuda/pipeline.h>:
auto pipeline = cuda::make_pipeline();
pipeline.producer_acquire();
cuda::memcpy_async(&smem[sind], &gmem[gind], sizeof(T), pipeline); // LDGSTS
pipeline.producer_commit();                          // LDGDEPBAR
cuda::pipeline_consumer_wait_prior<N>(pipeline);     // DEPBAR.LE SB,N
pipeline.consumer_release();

// cooperative_groups  <cooperative_groups/memcpy_async.h>:
auto block = cg:: this_thread_block();
cg::memcpy_async(block, smem, gmem, sizeof(T) * block.size());   // LDGSTS + LDGDEPBAR
cg::wait_prior<N>(block);                                        // DEPBAR.LE SB,N
```

Asynchronous Data Copies

Asynchronous copies summary

HW	Direction	Operation	Synchronization	C++	SASS
Ampere	GMEM ➡ SMEM	Asynchronous version of `smem[sind] := gmem[gind]`	Thread-local SMEM barrier	`__pipeline_memcpy_async`	`LDGSTS`
Hopper	REG➡DSMEM	Asynchronous version of `dsmem[sind] := var`	DSMEM barrier	`cuda::ptx::st_async`	`STAS`

Hopper, TMA	GMEM➡(D)SMEM	1D – 5D block copy
		Uniform
			(D)SMEM barrier	`cuda::device::experimental::`
				`cp_async_bulk_tensor_Nd_global_to_shared`
				`cp_async_bulk_tensor_Nd_shared_to_global`
					`UTMALDG`

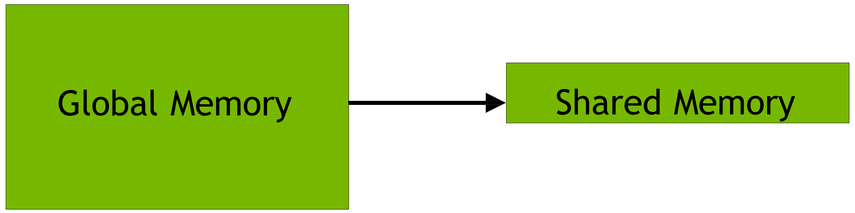
Hopper, TMA	SMEM➡GMEM		Thread-local		`UTMASTG`

Hopper, TMA	GMEM➡SMEM	1D block copy
		Uniform
			SMEM barrier	`cuda::device::experimental::`
				`cp_async_bulk_global_to_shared`
				`cp_async_bulk_shared_to_global`
					`UBLKCP`

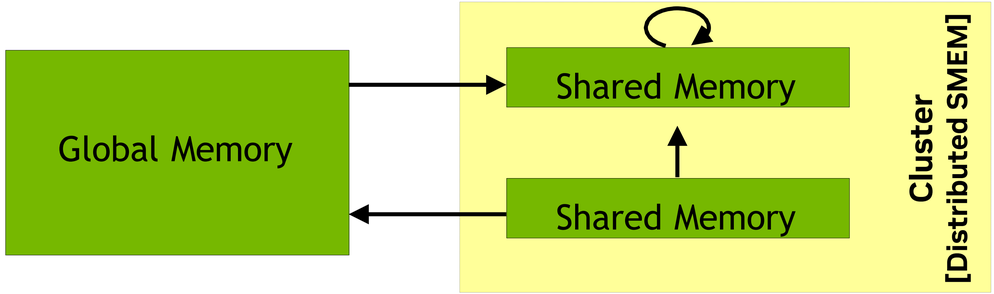
Hopper, TMA	SMEM➡GMEM		Thread-local		
Hopper, TMA	SMEM➡(D)SMEM		(D)SMEM barrier	`cuda::ptx::cp_async_bulk`*	

* Available in latest github CCCL

Ampere (LDGSTS)

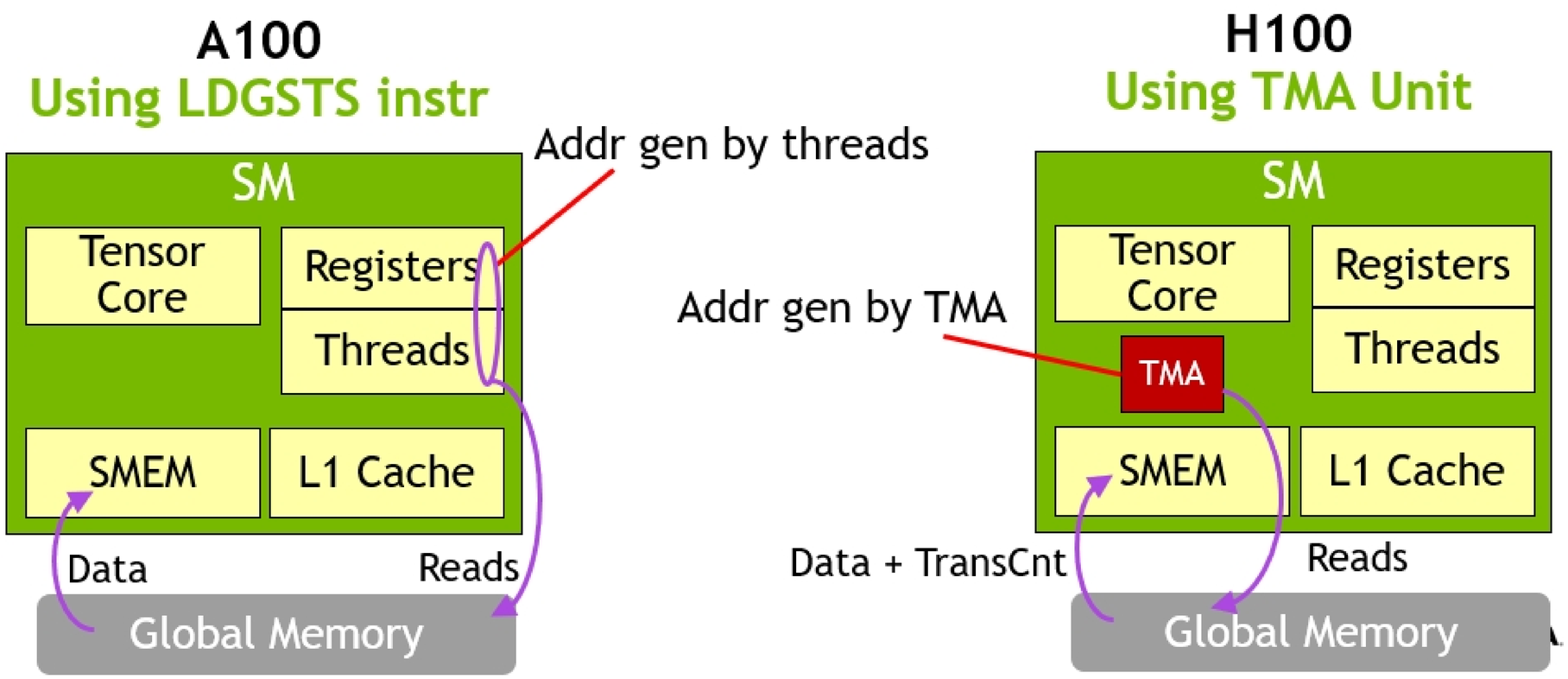


Hopper (TMA)



Tensor Memory Accelerator (TMA)

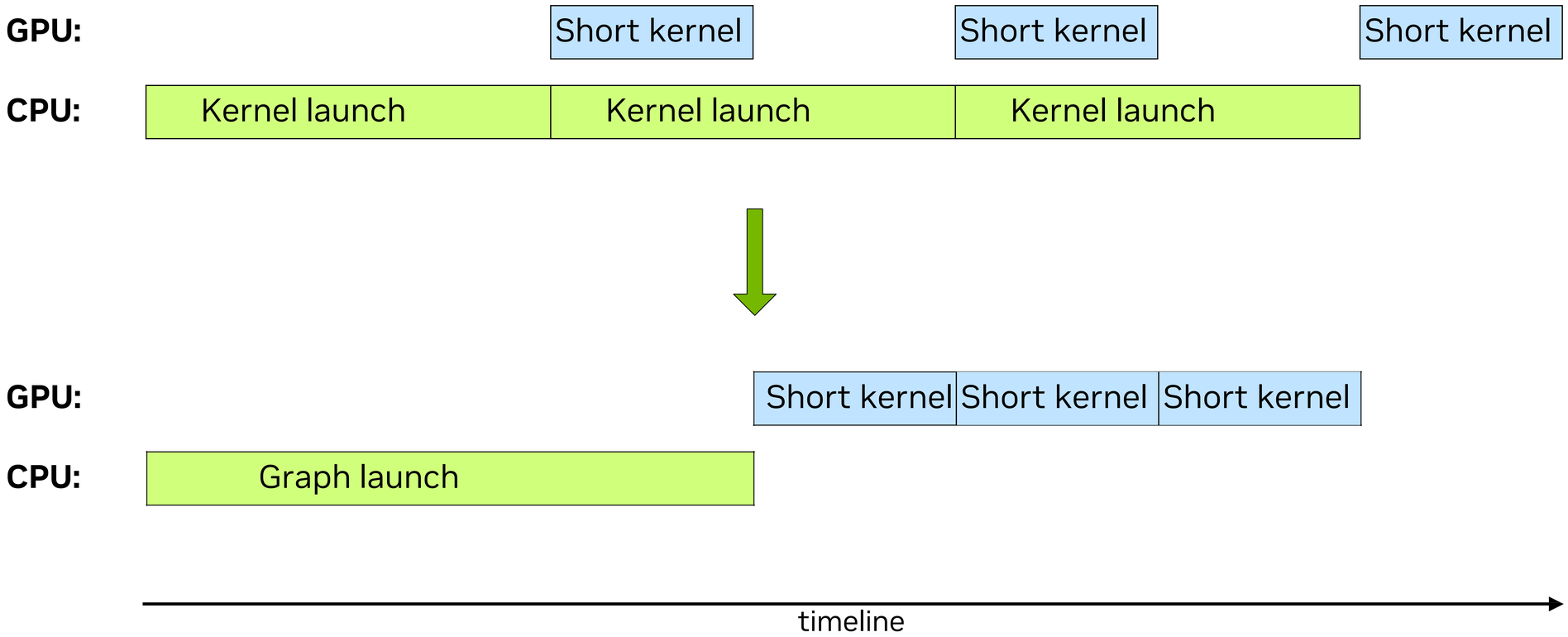
A single thread launches TMA via copy descriptor



CUDA Graphs

Reducing launch overhead

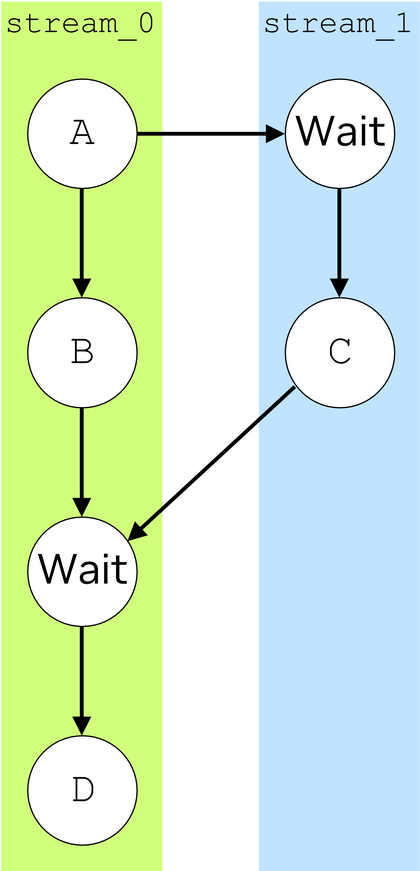
CUDA graphs reduce kernel launch overheads:



CUDA Graphs

Simple example – capturing

```
kernel_A<<<..., stream_0>>>();
cudaEventRecord(event_0, stream_0);
kernel_B<<<..., stream_0>>>();
cudaStreamWaitEvent(stream_1, event_0);
kernel_C<<<..., stream_1>>>();
cudaEventRecord(event_1, stream_1);
cudaStreamWaitEvent(stream_0, event_1);
kernel_D<<<..., stream_0>>>();
```

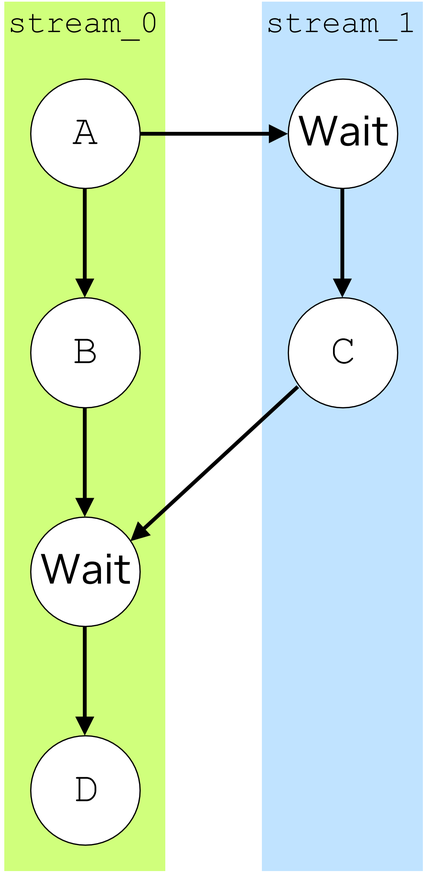


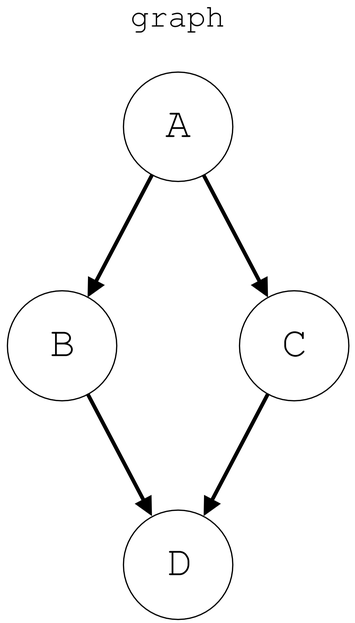
CUDA Graphs

Simple example – capturing

```
cudaStreamBeginCapture(stream_0, cudaStreamCaptureModeGlobal);
kernel_A<<<..., stream_0>>>();
cudaEventRecord(event_0, stream_0);
kernel_B<<<..., stream_0>>>();
cudaStreamWaitEvent(stream_1, event_0);
kernel_C<<<..., stream_1>>>();
cudaEventRecord(event_1, stream_1);
cudaStreamWaitEvent(stream_0, event_1);
kernel_D<<<..., stream_0>>>();
cudaGraph_t graph;
cudaStreamEndCapture(stream_0, &graph);
```

"dry-run"
during capture





CUDA Graphs

Simple example – steps

1. Capture graph (~ *write code*):

```
cudaStreamBeginCapture(stream_0, cudaStreamCaptureModeGlobal);
<CAPTURED ACTIONS>
cudaGraph_t graph;
cudaStreamEndCapture(stream_0, &graph);
```

2. Create executable graph (~ *compile code*):

```
cudaGraphExec_t graph_exec;
cudaGraphInstantiate(&graph_exec, graph);
```

3. Launch executable graph (~ *execute compiled code*):

```
cudaGraphLaunch(graph_exec, stream);
```

Graph encapsulates:

- Stream dependencies and concurrency
  - **Branches execute concurrently even though `cudaGraphLaunch` into a single stream**
  - Stream priorities can be inherited by nodes (requires `cudaGraphInstantiateFlagUseNodePriority` flag)
  - Streams used while capturing can be destroyed after capturing
- Kernel parameters
  - how do you change them?

CUDA Graphs

Simple example – graph modifications

Common time-stepping pattern:

```
for (int it = 0; it < nit; ++it) {
    kernel<<<...>>>(ptr_prev, ptr_next);
    // more kernels, etc.
    std::swap(ptr_prev, ptr_next);
}
```

Several approaches:

I. Capture even- and odd-iteration graphs and `cudaGraphLaunch(graph_exec[it & 1], stream);`
(see examples in https://github.com/NVIDIA/multi-gpu-programming-models/tree/master/nccl_graphs)

II. Always capture and `cudaGraphExecUpdate`

III. Explicit graph construction API:
`cudaStreamGetCaptureInfo`
`cudaGraphAddKernelNode` – returns a node handle used by `cudaGraphExecKernelNodeSetParams` during iterations
`cudaStreamUpdateCaptureDependencies`

Topology changes typically require graph re-instantiation.

Minor topology changes may not, e.g., turn node on/off (`cudaGraphNodeSetEnabled`)

CUDA Graphs

Other capture restrictions

Restrictions (not comprehensive list):

- Default legacy stream cannot be captured: `libraryCall(cudaStreamDefault)` ➔ `libraryCall(stream)`
- Synchronous calls cannot be captured: `cudaMemcpy` ➔ `cudaMemcpyAsync`
- Cannot synchronize: `cudaStreamSynchronize`, `cudaDeviceSynchronize`
- Host logic: `host_logic` ➔ `cudaLaunchHostFunc(stream, host_logic)`

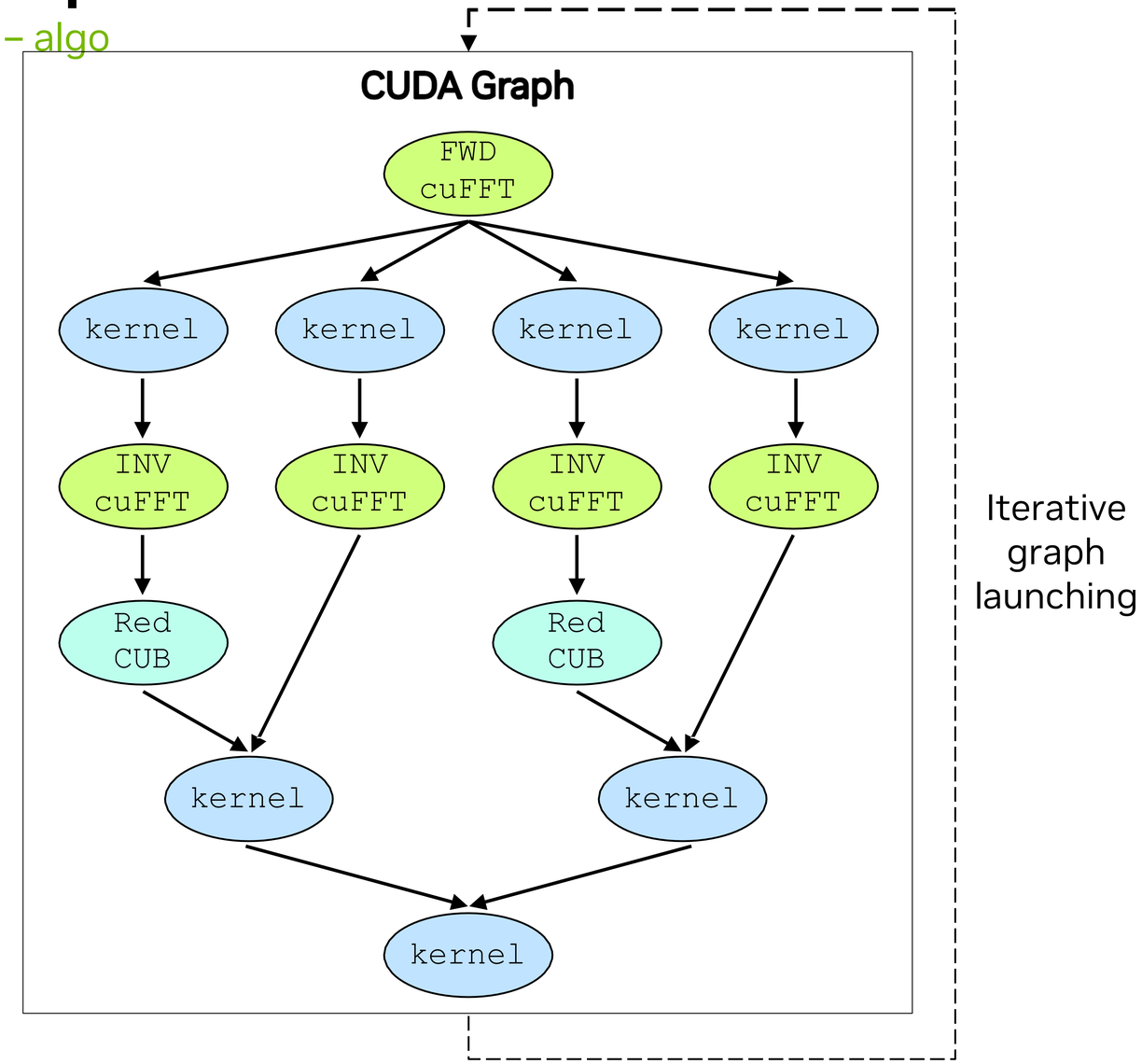
Can capture what can be put in a stream.

CUDA Graphs

Example – algo

Workflow based on seismic trace processing:

1. Forward `cuFFT`
2. Frequency continuations into multiple traces with different frequency bands
3. Inverse `cuFFT`s of multiple traces (cannot batch due to variable sizes)
4. Reductions (`CUB`) and combining even/odd
5. Final combining



CUDA Graphs

Example – code

```
cudaStreamBeginCapture(f_stream, cudaStreamCaptureModeGlobal);
cufftExecR2C(f_plan, f_data, of_data); // f_plan is associated with f_stream
cudaEventRecord(f_event, f_stream);
for (int i = 0; i < m; ++i) {
    cudaStreamWaitEvent(b_stream[i], f_event);
    kernel_0<<<1, BX, 0, b_stream[i]>>>(b_data[i], of_data, of_n, b_n[i]);
    cufftExecC2R(b_plan[i], b_data[i], ob_data[i]); // b_plan[i] is associated with b_stream[i]
}
for (int i = 0; i < m; i += 2) {
    cub::DeviceReduce::Sum(cub_storage[i].first, cub_storage[i].second, ob_data[i], ocub + i, ob_n[i], b_stream[i]);
    cudaEventRecord(b_event[i], b_stream[i]);
    cudaStreamWaitEvent(b_stream[i + 1], b_event[i]);
    kernel_1<<<1, BX, 0, b_stream[i + 1]>>>(ob_data[i + 1], ocub + i, ob_n[i + 1]);
    cudaEventRecord(b_event[i + 1], b_stream[i + 1]);
    cudaStreamWaitEvent(f_stream, b_event[i + 1]);
}
kernel_2<<<m, BX, 0, f_stream>>>(f_data, ob_data_gpu, ob_n_gpu, f_n);
cudaStreamEndCapture(f_stream, &graph);
```

No extra effort to capture library calls !! (into sub-graphs)

CUDA Graphs

Example – performance

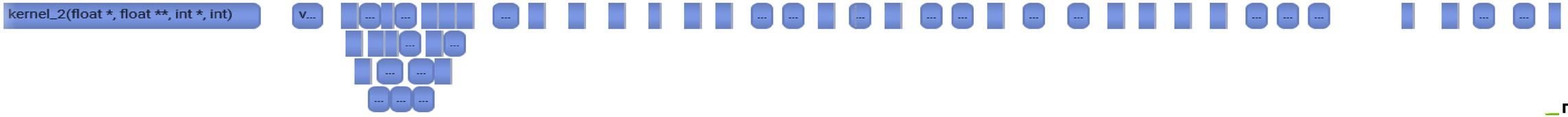
Performance:

- 1024 samples input trace.
- 100 output traces 128-8192 sample range.

Graphs provide **5X speedup** (H200 test).

Graphs: packed and compact

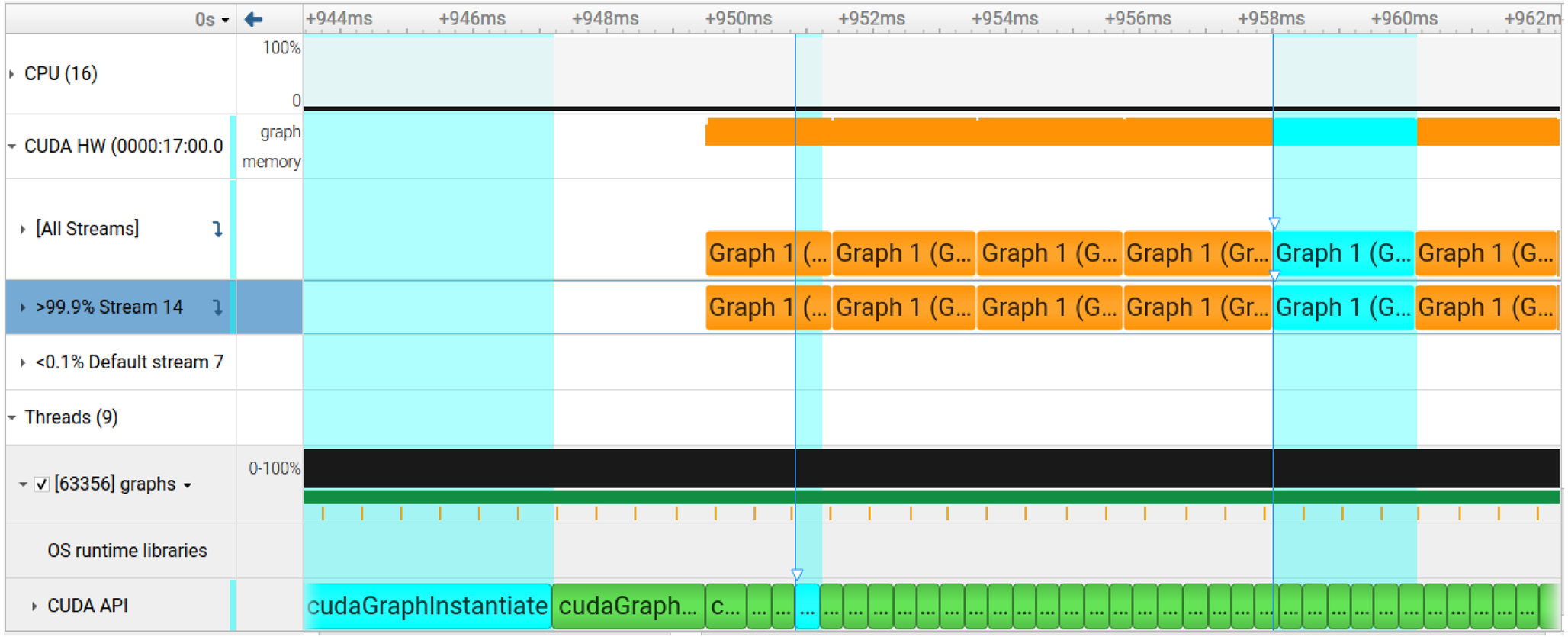
No-graphs: almost useless to employ multiple streams because launch overheads “serialize” kernels execution



CUDA Graphs

Nsight systems

- To trace nodes inside graphs, use `--cuda-graph-trace node` nsys option
  Note: has higher profiling overhead than default.
- Default is `--cuda-graph-trace graph` option:



Advanced CUDA Optimization

- Thread Block Cluster (CGA)
- Distributed Shared Memory (DSMEM)
- Asynchronous Barriers
- Asynchronous Data Copies
- Load Global Store Shared (LDGSTS)
- Tensor Memory Accelerator (TMA)
- CUDA Graphs

NVIDIA HPC SDK

PROGRAMMING THE NVIDIA PLATFORM

CPU, GPU, and Network

ACCELERATED STANDARD LANGUAGES

ISO C++, ISO Fortran

```
std::transform(par, x, x+n, y, y,
    [=](float x, float y){ return y +
a*x; }
);
```

```
do concurrent (i = 1:n)
   y(i) = y(i) + a*x(i)
enddo
```

```
import cupynumeric as np
…
def saxpy(a, x, y):
    y[:] += a*x
```

INCREMENTAL PORTABLE OPTIMIZATION

OpenACC, OpenMP

```
#pragma acc data copy(x,y) {
...
std::transform(par, x, x+n, y, y,
    [=](float x, float y){
        return y + a*x;
});
...
}

#pragma omp target data map(x,y) {
...
std::transform(par, x, x+n, y, y,
    [=](float x, float y){
        return y + a*x;
});
...
}
```

PLATFORM SPECIALIZATION

CUDA

```
__global__
void saxpy(int n, float a,
           float *x, float *y) {
  int i = blockIdx.x*blockDim.x +
          threadIdx.x;
  if (i < n) y[i] += a*x[i];
}

int main(void) {
  ...
  cudaMemcpy(d_x, x, ...);
  cudaMemcpy(d_y, y, ...);

  saxpy<<<(N+255)/256,256>>>(...);

  cudaMemcpy(y, d_y, ...);
```

ACCELERATION LIBRARIES

Core	Math	Communication	Data Analytics	AI	Quantum

Programming the NVIDIA Platform

Unmatched Developer Flexibility

Languages & Programming Models	Accelerated Standard Languages	Incremental Optimization	Platform Specialization
	C++, python, Fortran	OpenACC (More Science, Less Programming), OpenMP	NVIDIA CUDA, C++ | Fortran | Python
Libraries & Frameworks	Acceleration Libraries: Core, Math, Communication, Data Analytics, AI, DSLs		
Hardware	GPU	CPU	Interconnect

Wherever You Run

PCs | Workstations | On Prem | Cloud | At the Edge

NVIDIA HPC SDK

Available at developer.nvidia.com/hpc-sdk, on NGC, via Spack, and in the Cloud

DEVELOPMENT

Programming Models
- Standard C++ & Fortran
- OpenACC & OpenMP
- CUDA (C++ & Fortran)

Compilers
- nvcc
- nvc
- nvc++
- nvfortran

Core Libraries
- libcu++
- Thrust
- CUB

Math Libraries
- cuBLAS
- cuTENSOR
- cuSPARSE
- cuSOLVER
- cuFFT
- ...

Communication Libraries
- HPC-X
  - MPI
  - UCX
  - SHMEM
  - SHARP
  - HCOLL
- NVSHMEM
- NCCL

ANALYSIS

Profilers
- Nsight
  - Systems
  - Compute

Debugger
- cuda-gdb
  - Host
  - Device

Develop for the NVIDIA Platform: GPU, CPU and Interconnect
Libraries | Accelerated C++ and Fortran | Directives | CUDA
x86_64 | Arm
6 Releases Per Year | Freely Available

ISO is the place for portable concurrency and parallelism

Preview support coming to NVC++

C++17

Parallel Algorithms

- In NVC++
- Parallel and vector concurrency

Forward Progress Guarantees

- Extend the C++ execution model for accelerators

Memory Model Clarifications

- Extend the C++ memory model for accelerators

C++20

Scalable Synchronization Library

- Express thread synchronization that is portable and scalable across CPUs and accelerators
- In libcu++:
  - `std::atomic<T>`
  - `std::barrier`
  - `std::counting_semaphore`
  - `std::atomic<T>::wait/notify_*`
  - `std::atomic_ref<T>`

C++23 and Beyond

Executors / Senders-Recievers

- Simplify launching and managing parallel work across CPUs and accelerators

`std::mdspan/mdarray`

- HPC-oriented multi-dimensional array abstractions.

Range-Based Parallel Algorithms

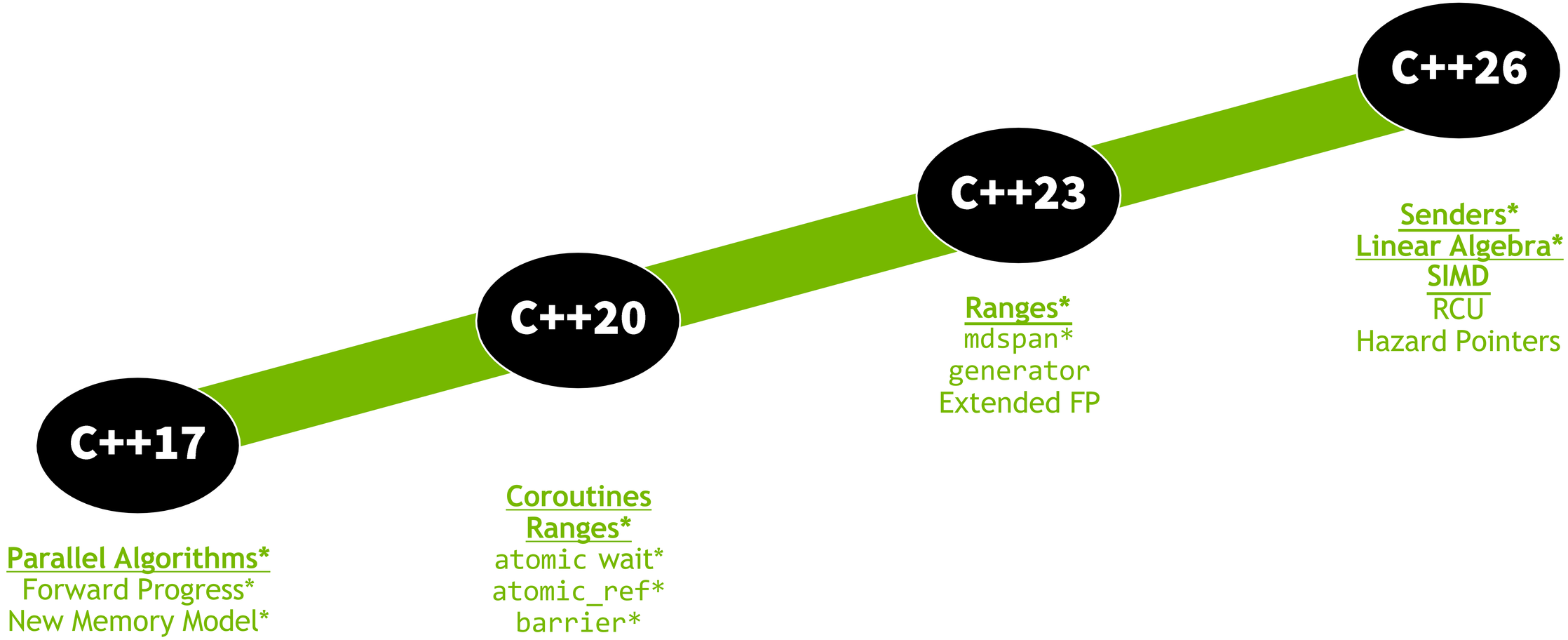
- Improved multi-dimensional loops

Linear Algebra

- C++ standard algorithms API to linear algebra
- Maps to vendor optimized BLAS libraries

Extended Floating Point Types

- First-class support for formats new and old: `std::float16_t/float64_t`



* = Available now in HPC SDK

Copyright (C) 2023 NVIDIA

ACCELERATED STANDARD LANGUAGES

Parallel performance for wherever your code runs

ISO C++

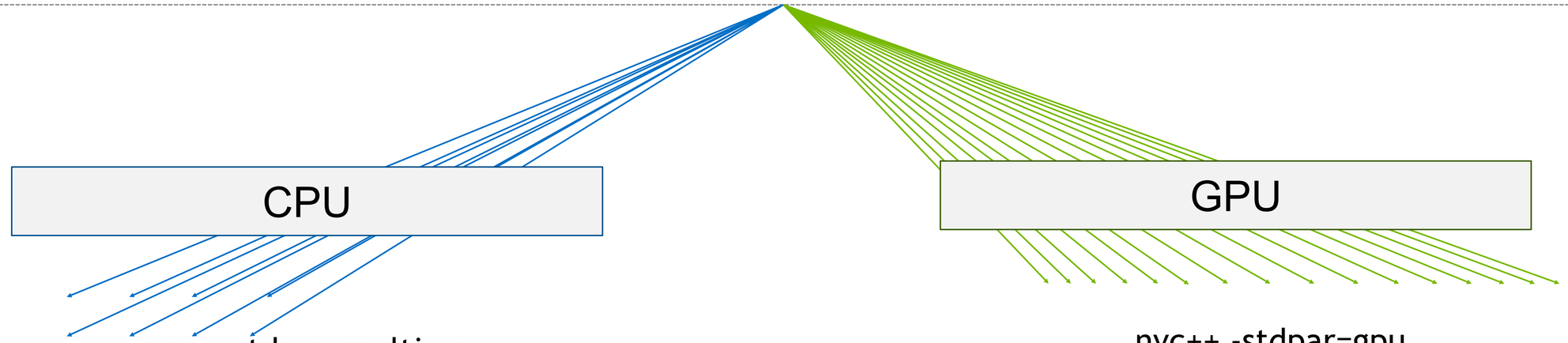
```
std::transform(par, x, x+n, y,
   y,[=](float x, float y){
      return y + a*x;
    }
);
```

ISO Fortran

```
do concurrent (i = 1:n)
   y(i) = y(i) + a*x(i)
enddo
```

Python

```
import cupynumeric as np
…
def saxpy(a, x, y):
    y[:] += a*x
```



nvc++ -stdpar=multicore
nvfortran -stdpar=multicore
legate -cpus 16 saxpy.py

nvc++ -stdpar=gpu
nvfortran -stdpar=gpu
legate -gpus 1 saxpy.py

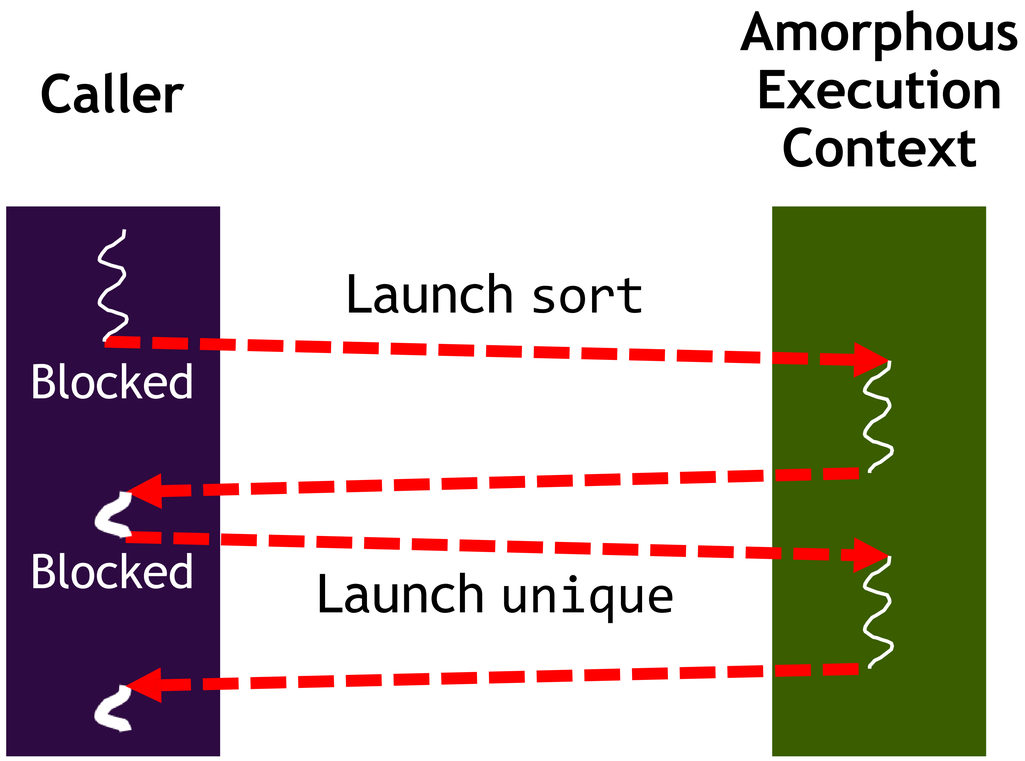
Standard Parallel Algorithms & Ranges

```
std::mdspan A{input,  N, M};
std::mdspan B{output, M, N};

auto v = stdv::cartesian_product(
  stdv::iota(0, A.extent(0)),
  stdv::iota(0, A.extent(1)));

std::for_each(ex::par_unseq,
  begin(v), end(v),
  [=] (auto idx) {
    auto [i, j] = idx;
    B[j, i] = A[i, j];
  });
```

In C++23 and NVC++ 22.5!

Copyright (C) 2023 NVIDIA

```
std::vector<std::string_view> s{…};

std::sort(ex::par_unseq, begin(s), end(s));
std::unique(ex::par_unseq, begin(s), end(s));
```



Standard Senders

Today C++ has:

No standard model for asynchrony.
No standard way to express where things should execute.

Solution is coming soon:

```
ex::scheduler auto sch = thread_pool.scheduler();

ex::sender auto begin  = ex::schedule(sch);
ex::sender auto hi     = ex::then(begin, [] { return 13; });
ex::sender auto add    = ex::then(hi, [] (int a) { return a + 42; });


auto [i] = this_thread::sync_wait(add).value();
```

Standard Senders

```
std::mdspan A{input, N, N}, B{output, N, N};

auto v = stdv::cartesian_product(
  stdv::iota(0, A.extent(0)), stdv::iota(0, A.extent(1)));
```

Synchronous

```
std::for_each(stdex::par_unseq,
  begin(v), end(v),
  [=] (auto idx) {
    auto [i, j] = idx;
    B[j, i] = A[i, j];
  });

std::matrix_product(stdex::par_unseq,
  A, B, B);
```

Asynchronous

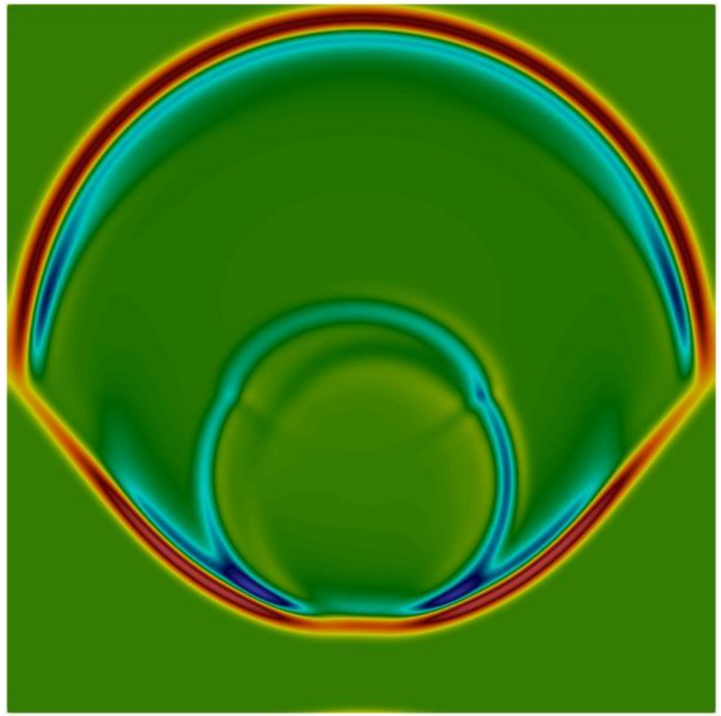
```
auto s = stdex::just_on(sch, v)
       | stdex::bulk(N,
           [=] (auto idx) {
             auto [i, j] = idx;
             B[j, i] = A[i, j];
           })
       | std::matrix_product_async(B, B);
```

Planned for C++26 and available at github.com/nvidia/stdexec!

Copyright (C) 2022 NVIDIA

Standard Senders

```
sender auto maxwell_eqs(scheduler auto &compute,
                        grid_accessor A, …) {
  return repeat_n(n_outer_iterations,
            repeat_n(n_inner_iterations,
                schedule(compute)
              | bulk(G.cells, update_h(G))
              | halo_exchange(G, hx, hy)
              | bulk(G.cells, update_e(time, dt, G))
              | halo_exchange(G, hx, hy))
            | transfer(cpu_serial_scheduler)
            | then(output_results))
          );
}
```



Planned for C++26 and available at github.com/nvidia/stdexec!

Copyright (C) 2022 NVIDIA

Standard Linear Algebra

```
std::mdspan A{…, N, M};
std::mdspan x{…, M};
std::mdspan y{…, N};

// y = 3.0 A x + 2.0 y
std::matrix_vector_product(
  ex::par_unseq,
  std::scaled(3.0, A), x,
  std::scaled(2.0, y), y);
```

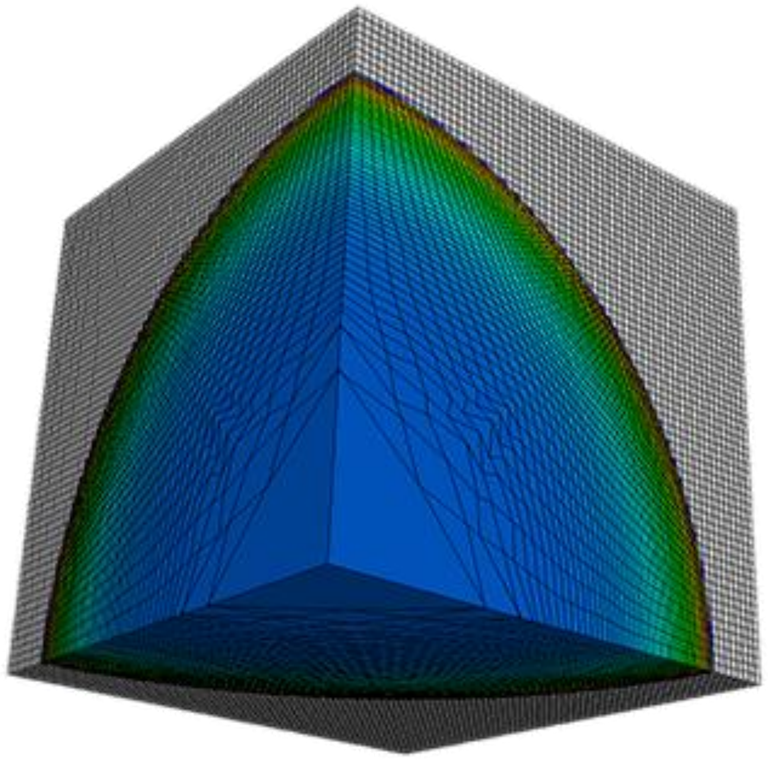
On the C++26 roadmap and in NVC++ 22.7!

Copyright (C) 2022 NVIDIA

C++17 PARALLEL ALGORITHMS

Lulesh Hydrodynamics Mini-app

- ~9000 lines of C++
- Parallel versions in MPI, OpenMP, OpenACC, CUDA, RAJA, Kokkos, ISO C++…
- Designed to stress compiler vectorization, parallel overheads, on-node parallelism



codesign.llnl.gov/lulesh

STANDARD C++

- Composable, compact and elegant
- Easy to read and maintain
- ISO Standard
- Portable – nvc++, g++, icpc, MSVC, …

```cpp
static inline
void CalcHydroConstraintForElems(Domain &domain, Index_t length,
    Index_t *regElemlist, Real_t dvovmax, Real_t& dthydro)
{
#if _OPENMP
  const Index_t threads = omp_get_max_threads();
  Index_t hydro_elem_per_thread[threads];
  Real_t dthydro_per_thread[threads];
#else
  Index_t threads = 1;
  Index_t hydro_elem_per_thread[1];
  Real_t dthydro_per_thread[1];
#endif
#pragma omp parallel firstprivate(length, dvovmax)
  {
    Real_t dthydro_tmp = dthydro ;
    Index_t hydro_elem = -1 ;
#if _OPENMP
    Index_t thread_num = omp_get_thread_num();
#else
    Index_t thread_num = 0;
#endif
#pragma omp for
    for (Index_t i = 0 ; i < length ; ++i) {
      Index_t indx = regElemlist[i] ;

      if (domain.vdov(indx) != Real_t(0.)) {
        Real_t dtdvov = dvovmax / (FABS(domain.vdov(indx))+Real_t(1.e-20)) ;

        if ( dthydro_tmp > dtdvov ) {
          dthydro_tmp = dtdvov ;
          hydro_elem = indx ;
        }
      }
    }
    dthydro_per_thread[thread_num] = dthydro_tmp ;
    hydro_elem_per_thread[thread_num] = hydro_elem ;
  }
  for (Index_t i = 1; i < threads; ++i) {
    if(dthydro_per_thread[i] < dthydro_per_thread[0]) {
      dthydro_per_thread[0] = dthydro_per_thread[i];
      hydro_elem_per_thread[0] = hydro_elem_per_thread[i];
    }
  }
  if (hydro_elem_per_thread[0] != -1) {
    dthydro = dthydro_per_thread[0] ;
  }
  return ;
}
```

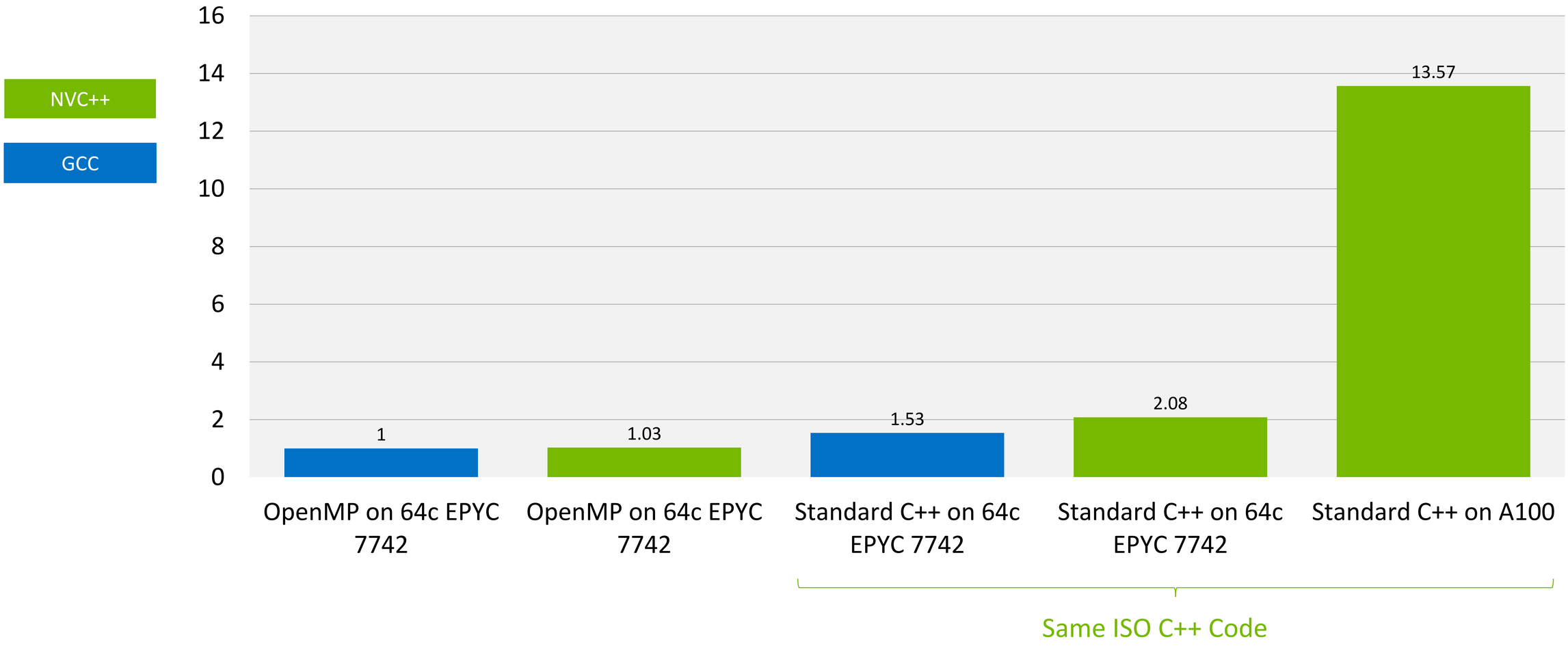
C++ with OpenMP

```cpp
static inline
void CalcHydroConstraintForElems(Domain &domain, Index_t length,
    Index_t *regElemlist, Real_t dvovmax, Real_t &dthydro)
{
  dthydro = std::transform_reduce(
    std::execution::par, counting_iterator(0), counting_iterator(length),
    dthydro, [](Real_t a, Real_t b) { return a < b ? a : b; },
    [=, &domain](Index_t i)
  {
      Index_t indx = regElemlist[i];
      if (domain.vdov(indx) == Real_t(0.0)) {
        return std::numeric_limits<Real_t>::max();
      } else {
        return dvovmax / (std::abs(domain.vdov(indx)) + Real_t(1.e-20));
      }
  });
}
```

Standard C++

C++ STANDARD PARALLELISM

Lulesh Performance



CUDA PYTHON

Native Access to Platform Specialization

EA Release – Out in August

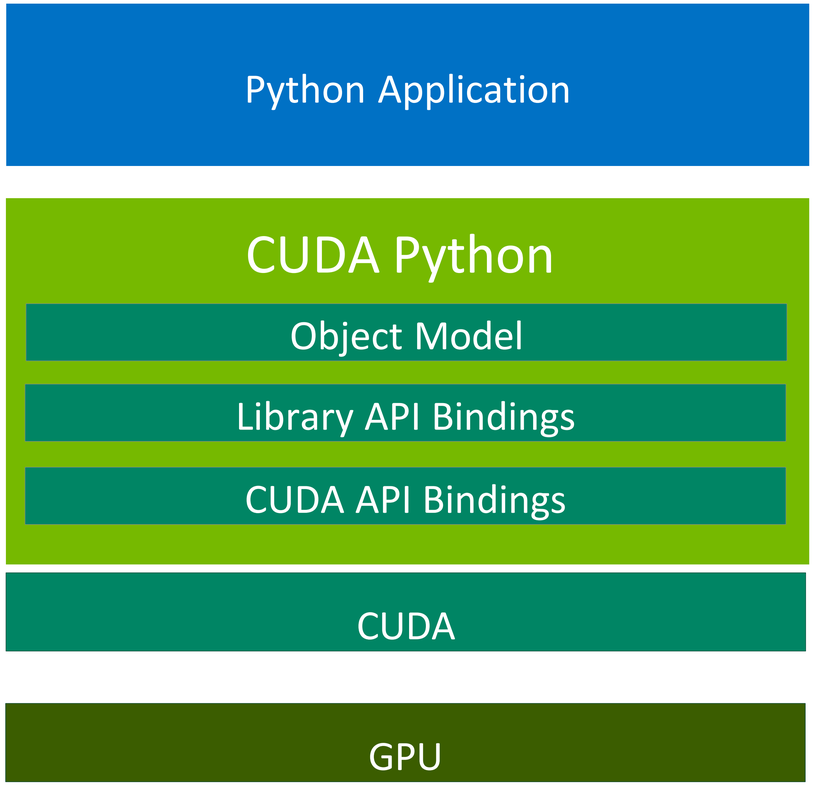
- Python Bindings on CUDA 11.4 Driver/Runtime API
- GitHub and Documentation Site

GA Release – Out now with CUDA 11.5

- GitHub source code
- Packaging: PIP and Conda
- Additional CUDA Functions

Future Releases

- Python bindings for Libraries
- Introduce Python Object Model



GitHub: https://github.com/NVIDIA/cuda-python
Docs: https://nvidia.github.io/cuda-python/

Email: cuda-python-dev@nvidia.com

HISTOGRAM: SERIAL PYTHON CODE

Generate input data for histogram and time serial Python code

```
def cpu_histogram(x, xmin, xmax, histogram_out):
    nbins = histogram_out.shape[0]
    bin_width = (xmax - xmin) / nbins
    for element in x:
        bin_number = np.int32(
            (element - xmin)/bin_width)
        if bin_number >= 0 and \
           bin_number < histogram_out.shape[0]:

           histogram_out[bin_number] += 1
```

```
def main():
    loops = int(sys.argv[1])

    data = np.random.random(2 ** 20)
    xmin = np.float32(-1.0)
    xmax = np.float32(1.0)
    histogram_out = np.zeros(shape=1000, dtype=np.int32)

    with prof.time_range("scipy", 0):
        cpu_histogram(data, xmin, xmax, histogram_out)

    for _ in range(loops):
        with prof.time_range("scipy_loop", 0):
            cpu_histogram(data, xmin, xmax, histogram_out)
```

HISTOGRAM: SERIAL PYTHON VS. NUMBA CODE

```python
def cpu_histogram(x, xmin, xmax, histogram_out):
    nbins = histogram_out.shape[0]
    bin_width = (xmax - xmin) / nbins

    for element in x:
        bin_number = np.int32((element - xmin)/bin_width)
        if bin_number >= 0 and bin_number < histogram_out.shape[0]:
            histogram_out[bin_number] += 1
```

```python
@cuda.jit
def numba_histogram(x, xmin, xmax, histogram_out):
    nbins = histogram_out.shape[0]
    bin_width = (xmax - xmin) / nbins

    start = cuda.grid(1)
    stride = cuda.gridsize(1)

    for i in range(start, x.shape[0], stride):
        bin_number = np.int32((x[i] - xmin)/bin_width)
        if bin_number >= 0 and bin_number < histogram_out.shape[0]:
            cuda.atomic.add(histogram_out, bin_number, 1)
```

HISTOGRAM: NUMPY VS. CUPY CODE

```
def main():
    loops = int(sys.argv[1])
    data = np.random.random(2 ** 20)
    d_data = array(data)

    histogram_out = cp.zeros(shape=1000, dtype=cp.int32)
    histogram_out_numpy = np.zeros(shape=1000, dtype=np.int32)
    nbins = histogram_out.shape[0]

    with prof.time_range("NumPy built in", 0):
        histogram_out, bins = np.histogram(data, nbins, range=(-1.0, 1.0))

    with prof.time_range("CuPy built in", 0):
        d_histogram_out, bins = cp.histogram(d_data, nbins, range=(-1.0, 1.0))

    # Compare results
        np.testing.assert_allclose( histogram_out, cp.asnumpy(d_histogram_out), 1e-3 )
```

General Purpose and Domain-Specific Libraries

RAPIDS

Data Preparation/ETL → Analytics/ML/Graph → Visualization

cuDF

- GPU-accelerated ETL functions
- Tracks Pandas and other common PyData APIs
- Dask + UCX integration for scaling

RAPIDS for Apache Spark

- RAPIDS accelerator for Apache Spark

RAPIDS ML

- GPU-native cuML library, plus XGBoost, RAFT, FIL, HPO, and more

cuGraph

- GPU graph analytics, including Louvain, PageRank, and more
- Multi-Node Multi-GPU features

cuxfilter

- GPU-accelerated cross-filtering

Viz integration

- pyViz: Plotly Dash, Bokeh, Datashader, HoloViews, hvPlot
- Node-RAPIDS bindings for node.js

Application and Domain-Specific Frameworks

Morpheus

cuSignal

cuSpatial
Spatial analytics

Merlin

cuCIM

NVTabular

cuDF

A GPU DataFrame library in Python with a pandas-like API built into the PyData ecosystem

Pandas-like API on the GPU

CPU

pandas

```
>>> import pandas as pd
>>> df = pd.read_csv("filepath")
>>> df.groupby("col").mean()
>>> df.rolling(window=3).sum()
```

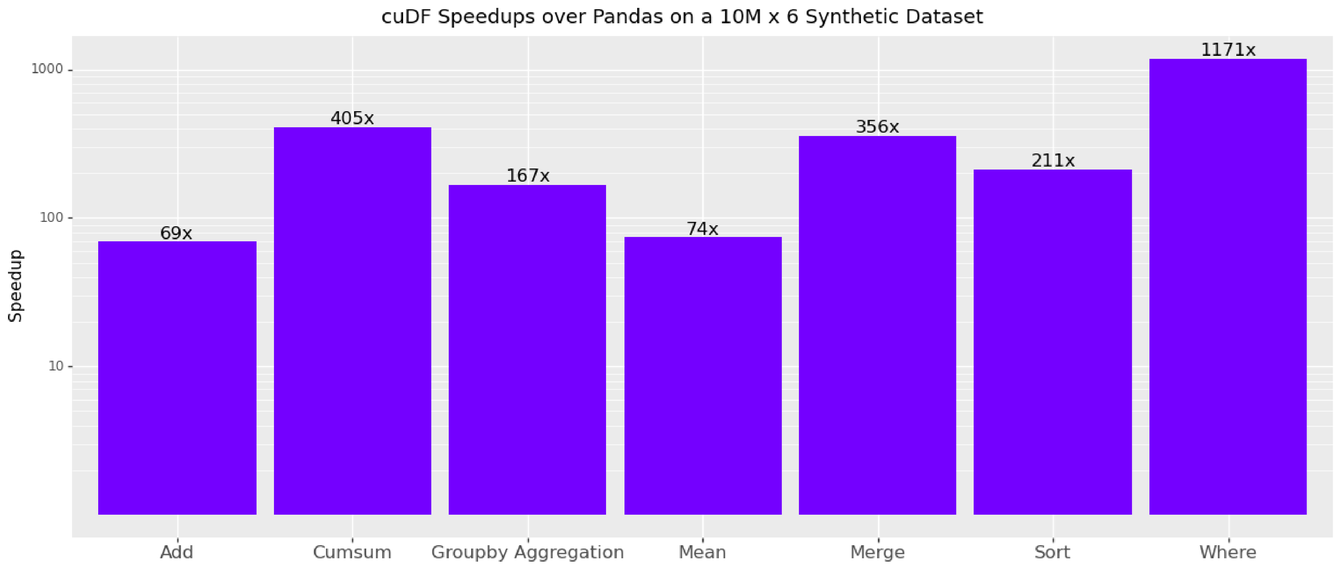
GPU

cuDF

```
>>> import cudf
>>> df = cudf.read_csv("filepath")
>>> df.groupby("col").mean()
>>> df.rolling(window=3).sum()
```

Average Speed-Ups: *10-100x*

Best-in-Class Performance (Benchmark)



Groupby	Strings and Regex	UDFs	Nested Types	Time Series
Indexing	Missing Data	CuPy Interoperability	Rolling Windows	

10 Minutes to cuDF

NVIDIA A100 vs. AMD EPYC 7642 48-Core Processor
cuDF Python vs. Pandas

cuML

Accelerated Machine Learning with a scikit-learn API

Scikit-learn

CPU

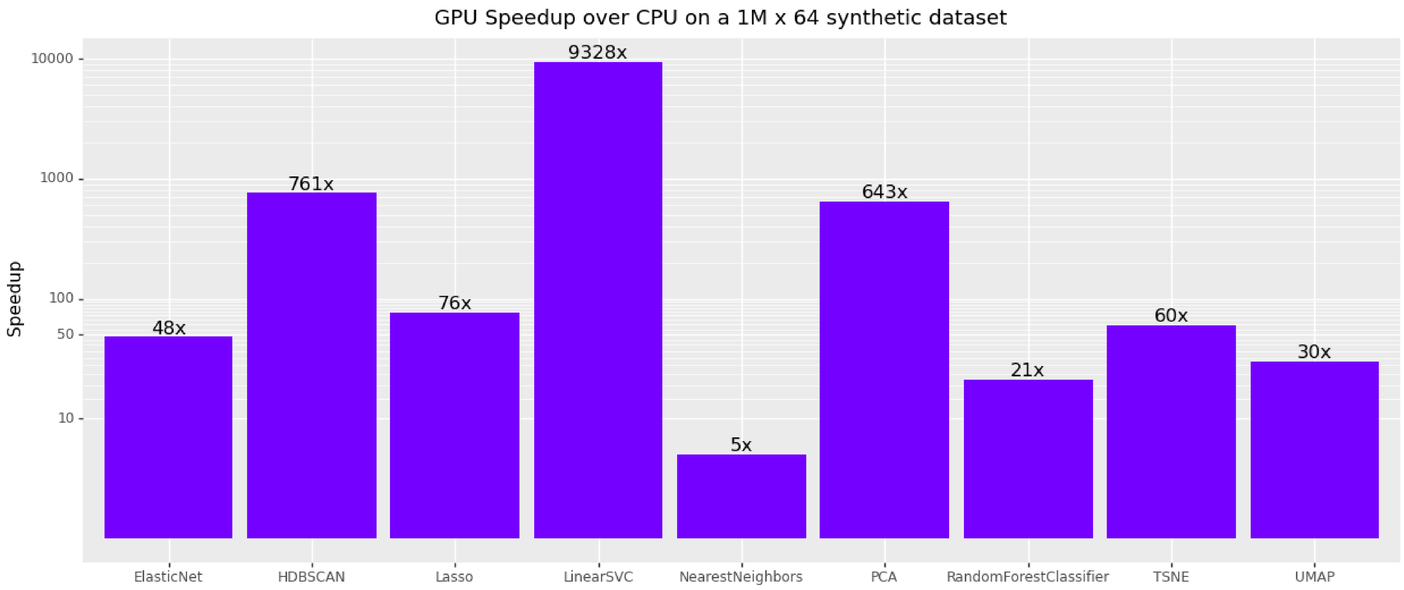
```
>>> from sklearn.ensemble import
RandomForestClassifier
>>> clf = RandomForestClassifier()
>>> clf.fit(x, y)
```

cuML

GPU

```
>>> from cuml.ensemble import
RandomForestClassifier
>>> clf = RandomForestClassifier()
>>> clf.fit(x, y)
```

50+ GPU-Accelerated Algorithms & Growing



Time Series	Classification	Regression	Clustering	Preprocessing
Cross Validation	Tree Models	Dimensionality Reduction	Explainability	

A100 GPU vs. AMD EPYC 7642 (96 logical cores)
cuML 23.04, scikit-learn 1.2.2, umap-learn 0.5.3

Announcing NVIDIA cuPyNumeric

Accelerating the Leading Foundational Library of the Python Ecosystem

DEVELOPER PRODUCTIVITY
Code Once, Scale to Many

Python code

Change NumPy to cuPyNumeric

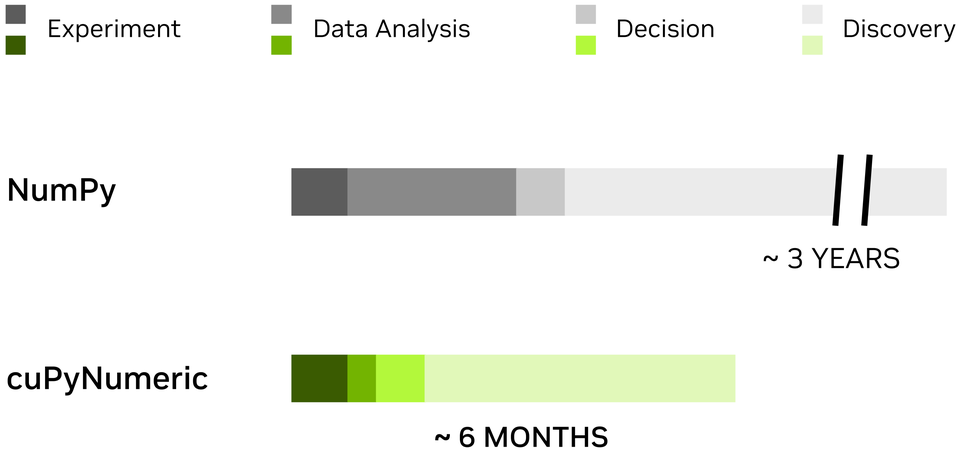
```
#import numpy as np
import cupynumeric as np

A = np.random.rand(M, K)
B = np.random.rand(K, N)
C = A @ B
```

SCALING WITH ZERO-CODE CHANGE

CPU | GPU | Multi-GPU + Multi-Node

LINAC COHERENT LIGHT SOURCE
Accelerating Discovery From Three Years to Six Months

LCLS Experiment Workflow



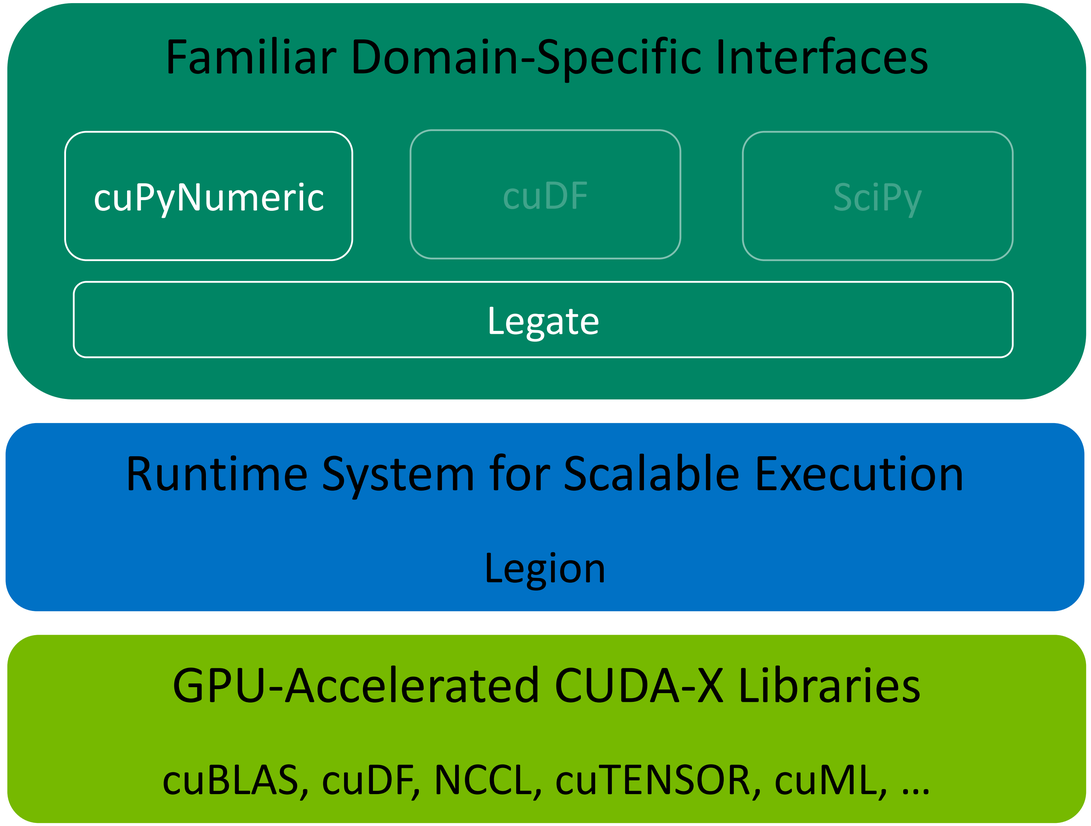
6X

Faster Analysis

Llama 3.1 405B inference 128 ISL, 2048 OSL maximum throughput | HGX H200 8-GPU | July 2024 TensorRT-LLM v0.13.0.dev2024082000 with official Llama FP8 Recipe | Oct 2024 TensorRT-LLM v.0.14 with TRT Model Optimizer FP8

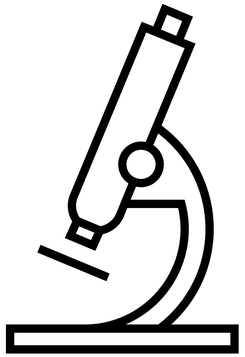
LEGATE ECOSYSTEM ARCHITECTURE

Scalable implementations of popular domain-specific APIs

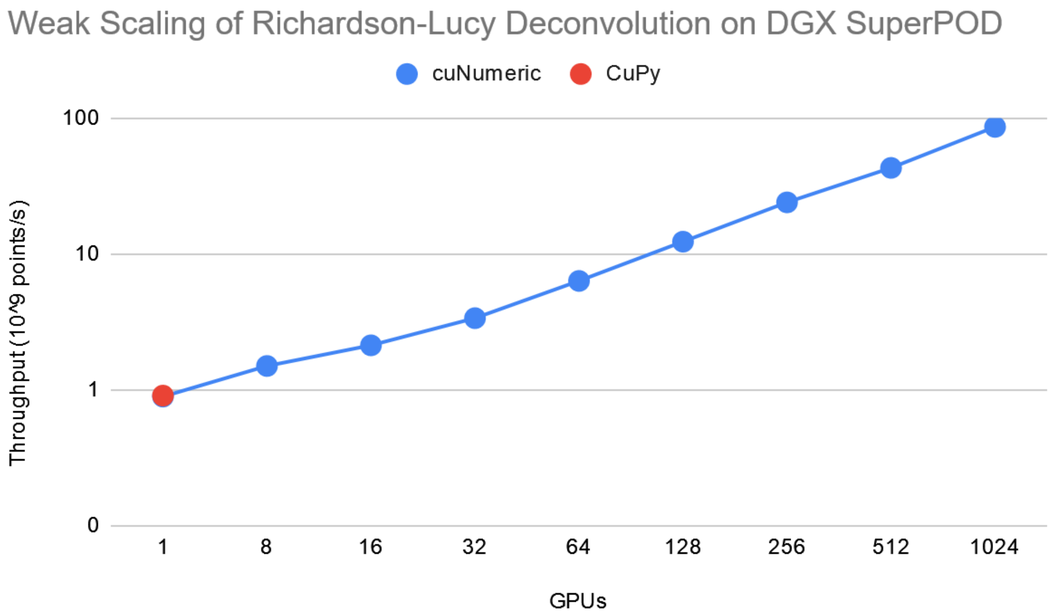


PERFORMANCE RESULTS

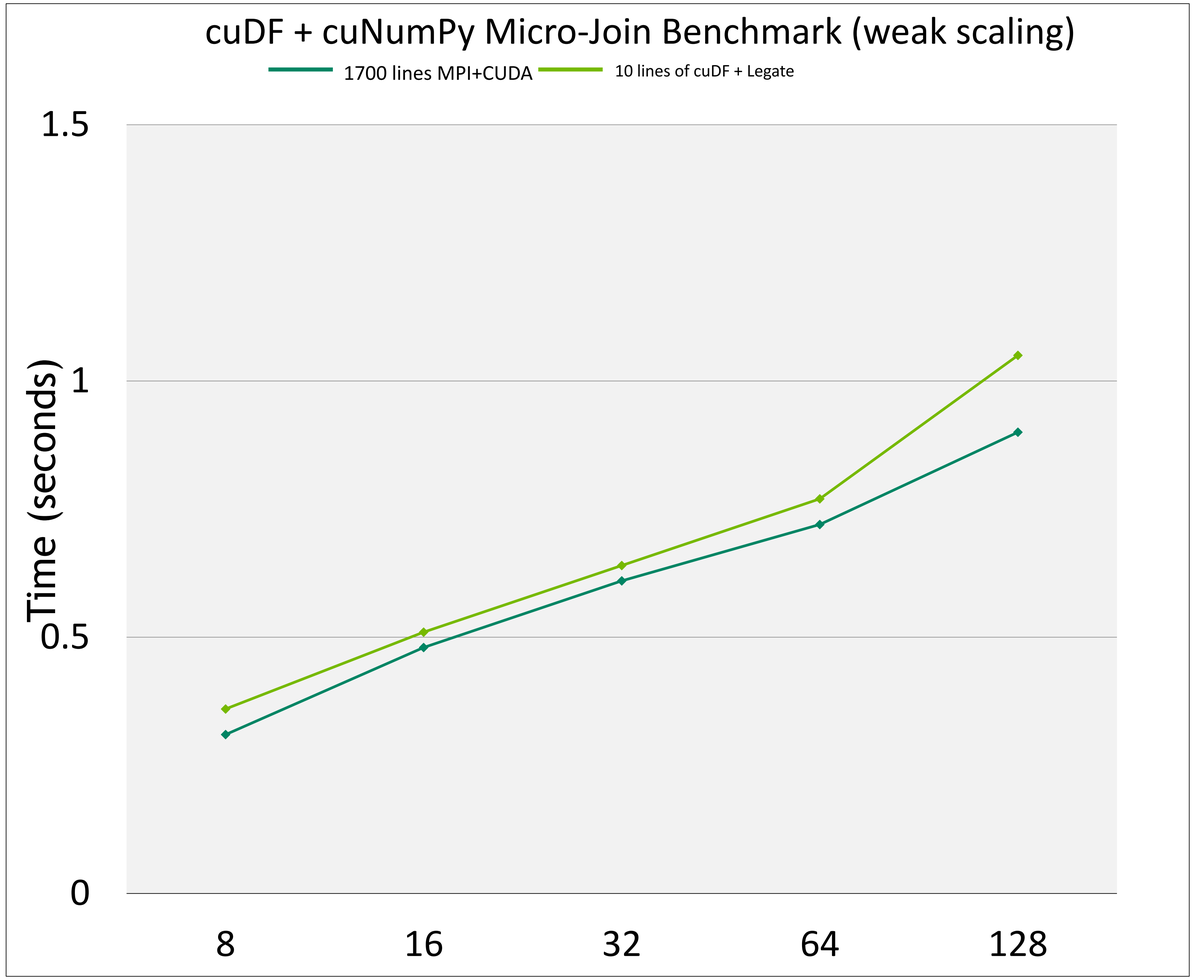
Microscopy Demo with Richardson-Lucy Deconvolution



```
def richardson_lucy(image, psf, num_iter=50,
                    clip=True, filter_epsilon=None):
  float_type = _supported_float_type(image.dtype)
  image = image.astype(float_type, copy=False)
  psf = psf.astype(float_type, copy=False)
  im_deconv = np.full(image.shape, 0.5, dtype=float_type)
  psf_mirror = np.flip(psf)

  for _ in range(num_iter):
    conv = convolve(im_deconv, psf, mode='same’)
      if filter_epsilon:
         with np.errstate(invalid='ignore’):
            relative_blur = np.where(conv < filter_epsilon, 0,
                                     image / conv)
      else:
         relative_blur = image / conv
      im_deconv *= convolve(relative_blur, psf_mirror,
                            mode='same’)

   if clip:
      im_deconv[im_deconv > 1] = 1
      im_deconv[im_deconv < -1] = -1

  return im_deconv
```



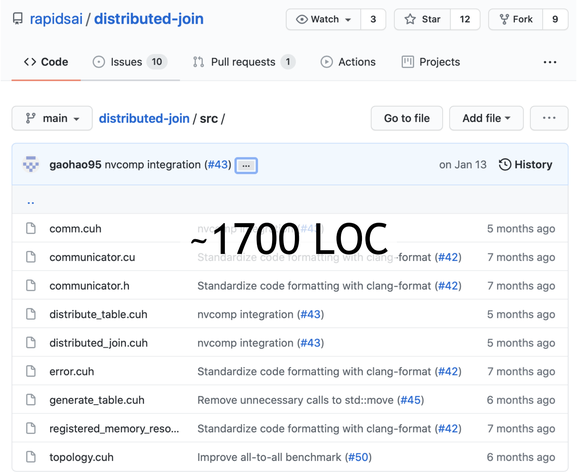
NEAR-MPI PERFORMANCE IN ~10 LINES OF CODE



Machine: DGX SuperPOD with A100-80GB GPUs

```
size = num_rows_per_gpu * num_gpus

key_l = np.arange(size)
val_l = np.random.randn(size)
lhs = pd.DataFrame({ "key": key_l, ”val": val_l })

key_r = key_l // 3 * 3    # selectivity: 0.33
payload_r = np.random.randn(size)
rhs = pd.DataFrame({ "key": key_r, ”val": val_r })

out = lhs.merge(rhs, on="key")
```

vs.



NVIDIA Warp

Differentiable Spatial Computing for Python

GPU Kernels in Python

Write CUDA kernels in 100% Python syntax
Runtime JIT compilation
Fast developer iteration

Spatial Computing

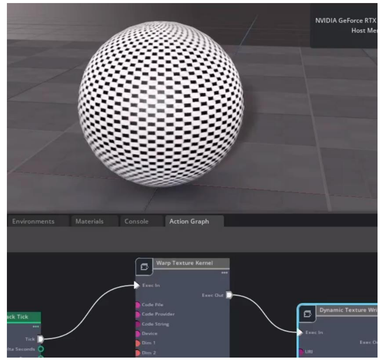
Rich vector math library
Mesh processing and queries
Sparse volumes (OpenVDB)
Hash grids

Differentiable programming

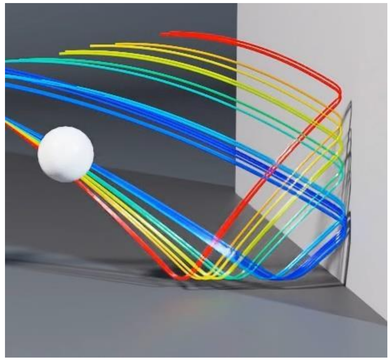
Auto-differentiation
Forward + backward kernel generation
Interop with DL frameworks (e.g.: Torch, JAX)

Omniverse integration

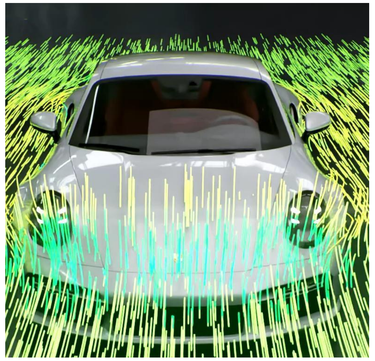
USD import/export
Runtime extensions for Isaac/USD Composer



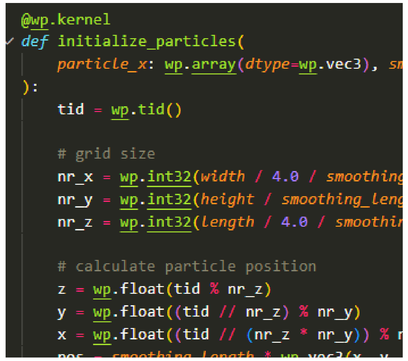
Data Processing



System Training



Simulation



```
@wp.kernel
def initialize_particles(
    particle_x: wp.array(dtype=wp.vec3), sm
):
    tid = wp.tid()

    # grid size
    nr_x = wp.int32(width / 4.0 / smoothing
    nr_y = wp.int32(height / smoothing_leng
    nr_z = wp.int32(length / 4.0 / smoothin

    # calculate particle position
    z = wp.float(tid % nr_z)
    y = wp.float((tid // nr_z) % nr_y)
    x = wp.float((tid // (nr_z * nr_y)) % n
```

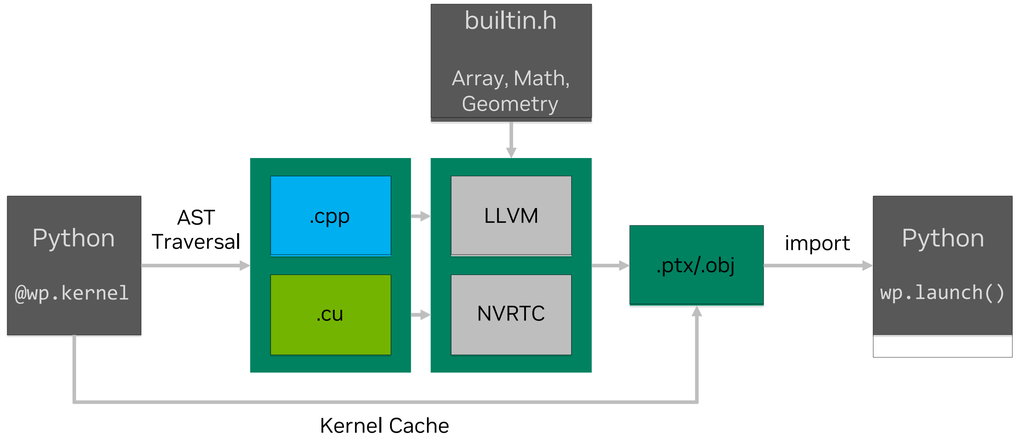
Scripting

Fast JIT Compilation of Differentiable Python Kernels

Just-in-time reverse-mode automatic differentiation

```python
import warp as wp

@wp.kernel
def integrate(p: wp.array(dtype=wp.vec3),
              v:  wp.array(dtype=wp.vec3),
              f:  wp.array(dtype=wp.vec3),
              m: wp.array(dtype=float)):

    # thread id
    tid = wp.tid()

    # Semi-implicit Euler step
    v[tid] = v[tid] + (f[tid] * m[tid] + wp.vec3(0.0, -9.8, 0.0)) * dt
    x[tid] = x[tid] + v[tid] * dt

# kernel launch
wp.launch(integrate, dim=1024, inputs=[x, v, f, …], device="cuda:0")
```

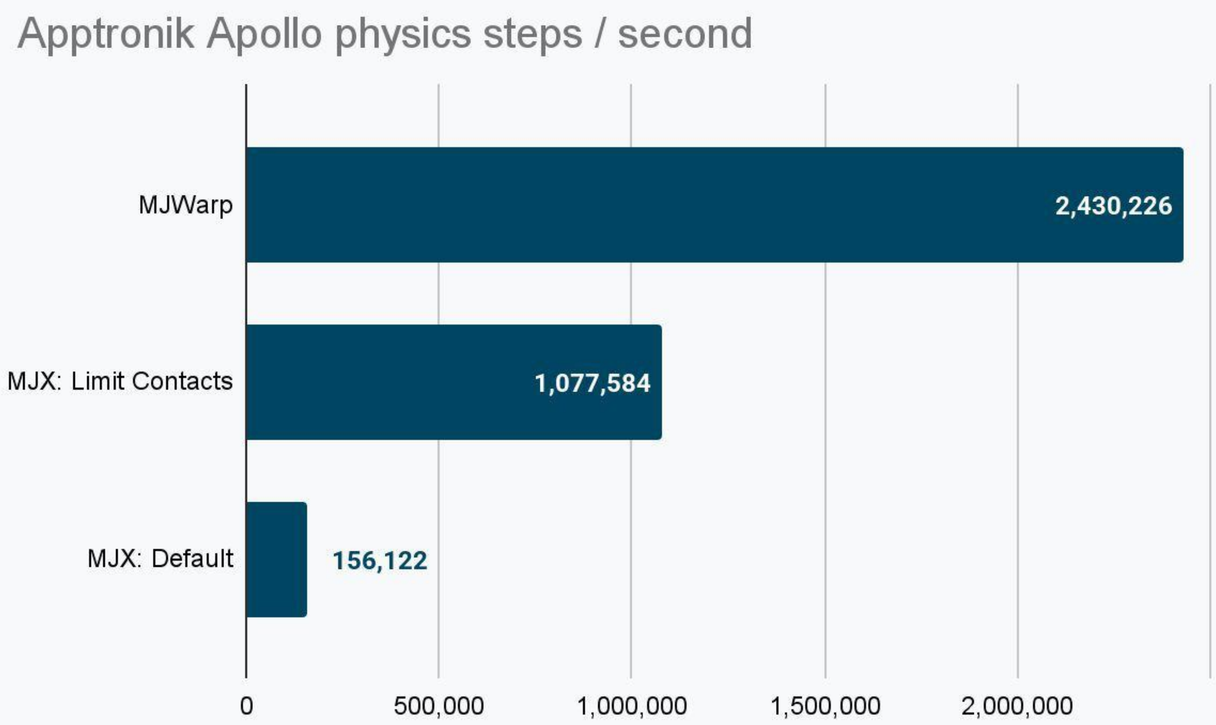
Warp Compilation Pipeline

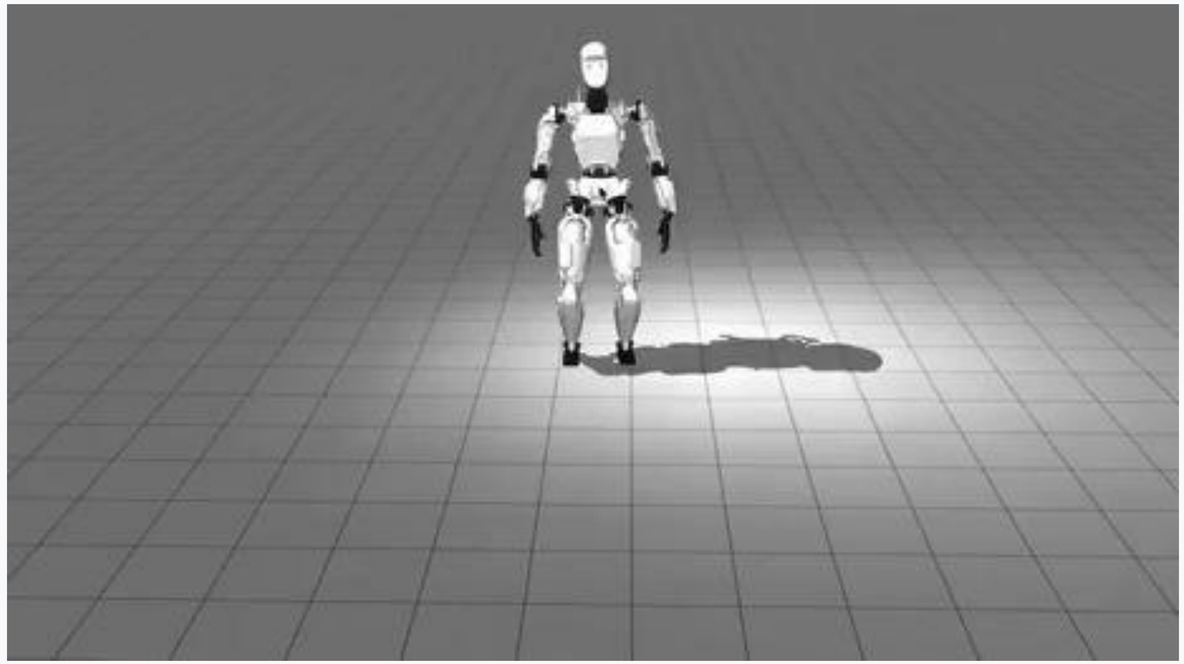


Get Warp at: https://github.com/nvidia/warp

or: `pip install warp-lang`

Example: Humanoid Robot Locomotion

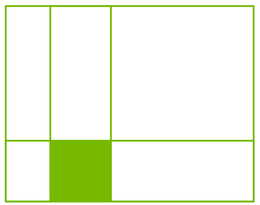




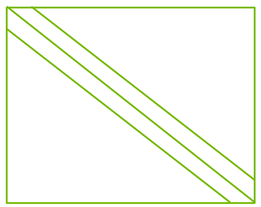
Apptronik Apollo
Joystick Policy

NVIDIA MATH LIBRARIES

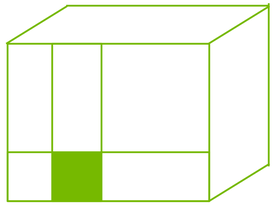
Linear Algebra, FFT, RNG and Basic Math



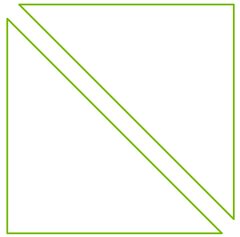
cuBLAS



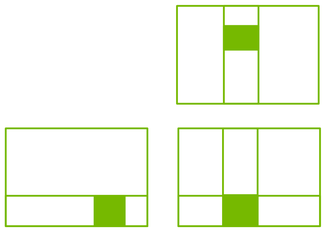
cuSPARSE



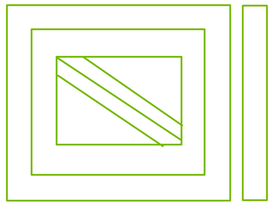
cuTENSOR



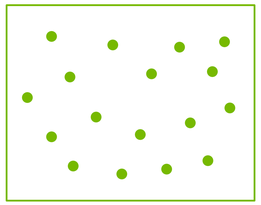
cuSOLVER



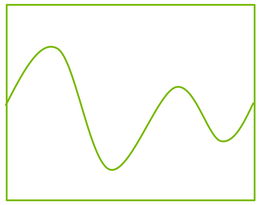
CUTLASS



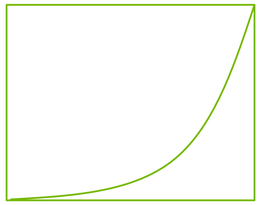
AMGX



cuRAND



cuFFT



CUDA Math API

cuBLAS API Extensions

cuBLAS Host API

cuBLAS Host APIs for CUDA-accelerated **BLAS** for **Level 1** (vector-vector), **Level 2** (matrix-vector), and **Level 3** (matrix-matrix) operations. cuBLAS also includes custom GEMM extension APIs that are simple to use for drop-in hardware acceleration.

cuBLAS APIs are available in the cuBLAS library.

cuBLASLt Host API

cuBLASLt Host APIs are **multi-stage GEMM APIs** that are highly expressive, allowing applications to leverage the latest NVIDIA architecture features for the best performance with support for **fusions** and performance tuning options.

cuBLASLt APIs are available in the cuBLAS library.

cuBLASXt Single-Process Multi-GPU Host API

cuBLASXt Host API exposes a multi-GPU capable interface for efficiently dispatching Level 3 workloads across one or multiple GPUs in a **single node**.

cuBLASXt APIs are available in the cuBLAS library.

cuBLASMp Multi-Node Multi-GPU Host API (Preview)

cuBLASMp (Preview) is a high-performance, **multi-process**, GPU-accelerated library for **distributed** basic dense linear algebra. cuBLASMp is available for standalone download and as part of the HPC SDK.

cuBLASDx Device API (Preview)

cuBLASDx (Preview) is a **device side** API extension to cuBLAS for performing BLAS calculations inside your CUDA kernel. **Fusing** numerical operations decreases the latency and improves the performance of your application.

C++ Standard Parallelism

C++ Core Compute Libraries (CCCL)

stdexec

The Standard C++ framework for asynchrony
https://github.com/NVIDIA/stdexec

The Standard Library for your entire system
https://github.com/NVIDIA/libcudacxx

Thrust

The C++ parallel algorithms library
https://github.com/NVIDIA/thrust

Cooperative primitives for CUDA C++
https://github.com/NVIDIA/cub

The C++ parallel algorithms library
https://github.com/NVIDIA/thrust

- High-Level Container Classes
  - Host_Vector
  - Device_Vector
- High-Level Algorithms
  - Transform
  - Fill
  - Copy
  - …
- Iterator Classes
  - Counting Iterator
  - Constant Iterator
  - …

Cooperative primitives for CUDA C++
https://github.com/NVIDIA/cub

- Parallel Communication Primitives
  - Warp-wide Collectives
  - Blow-wide Collectives
  - Device-wide Primatives
- Utility Operators
  - Iterators
  - Thread/Block I/O
  - PTX Intrinsics

The C++ parallel algorithms library
https://github.com/NVIDIA/thrust

```
thrust::universal_vector<float> h;
thrust::universal_vector<float> d;

thrust::event e = thrust::async::copy(
  par, h.begin(), h.end(), d.begin());
thrust::future<float> f = thrust::async::reduce(
  par.after(e), y.begin(), y.end());

…

float r = f.get();
```

Cooperative primitives for CUDA C++
https://github.com/NVIDIA/cub

```
template <typename R>
__global__ void f(R, …) {
  __shared__ typename R::TempStorage tmp;
  int local_s(…);
  int s = R(tmp).Sum(local_s);
}

f<<<32, 1>>>(cub::WarpReduce<int>{}, …);
f<<<128, 1>>>(cub::BlockReduce<int, 128>{}, …);
```

LIBCU++: A GPU-ENABLED STL

Host Compiler's Standard Library	
`#include <…>`	ISO C++, `__host__` only.
`std::`	Complete, strictly conforming to Standard C++.
`#include <cuda/std/…>`	CUDA C++, `__host__` `__device__`.
`cuda::std::`	Subset, strictly conforming to Standard C++.
`#include <cuda/…>`	CUDA C++, `__host__` `__device__`.
`cuda::`	Conforming extensions to Standard C++.
libcu++	

libcu++ does not interfere with or replace your host Standard Library.

The NVIDIA C++ Standard Library

https://github.com/NVIDIA/libcudacxx

1.0.0 (CUDA 10.2)

`atomic<T>` (SM60+)
Type Traits

1.1.0 (CUDA 11.0)

`atomic<T>::wait/notify` (SM70+)
`barrier` (SM70+)
`latch` (SM70+)
`*_semaphore` (SM70+)
`cuda::memcpy_async` (SM70+)
`chrono::` Clocks & Durations
`ratio<Num, Denom>`

1.2.0 (CUDA 11.1)

`cuda::pipeline` (SM80+)

1.3.0 (CUDA 11.2)

`tuple<T0, T1, ...>`

1.4.1 (CUDA 11.3)

`complex`
`byte`
`chrono::` Dates & Calendars

2.0.0

`atomic_ref<T>` (SM60+)
Memory Resources & Allocators
`cuda::stream_view`

Future

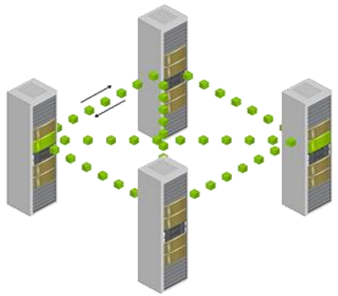
Executors
Range Factories & Adaptors
Parallel Range Algorithms
Parallel Linear Algebra Algorithms
`mdspan<T, ...>`
…

NVIDIA HPC SDK (Recap)

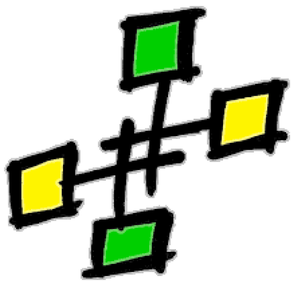
- ISO C++, OpenACC/OpenMP, CUDA C++
- CUDA Python, RAPIDS, cuPyNumeric
- NVIDIA Warp
- NVIDIA Math Libraries
- C++ Core Compute Libraries (CCCL)

NVIDIA Communication Libraries (HPC-X)

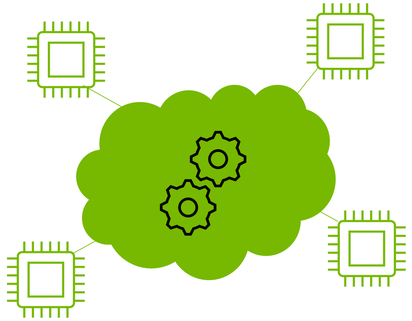
SOFTWARE AND ACCELERATION PACKAGES



HPC-X MPI



HPC-X OpenSHMEM



In-Network Computing

Unified Communication X

Unified Collective Communication

Unified Collective Communication

SHARP Scalable Hierarchical Aggregation and Reduction Protocol

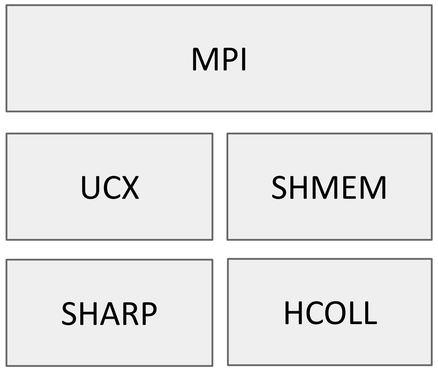
NCCL-SHARP and UCX Support

HPC-X

Accelerated Communications APIs and Operations for HPC

HPC-X is a comprehensive software package that includes Accelerated Communications API and Operations for HPC

- CUDA-aware MPI library based on Open MPI with support for GPUDirect™
- Fully compatible with CUDA C++, CUDA Fortran and the NVIDIA HPC Compilers
- Optimized for NVIDIA InfiniBand interconnect solutions
- Integrated into the NVIDIA HPC SDK



Components	Description
MPI/SHMEM	•Open MPI and OpenSHMEM •MPI profiler (IPM - open source tool from *http://ipm-hpc.org* /) •MPI tests (OSU, IMB, random ring, etc.)
HPC Acceleration Package	•HCOLL •UCX •Scalable Hierarchical Aggregation and Reduction Protocol (SHARP) •nccl-rdma-sharp-plugin

CUDA-AWARE MPI

Message Passing Interface

- Non-CUDA-aware MPI

```
//MPI rank 0
cudaMemcpy(s_buf_h, s_buf_d, size, cudaMemcpyDeviceToHost);
MPI_Send(s_buf_h, size, MPI_CHAR, 1, 100, MPI_COMM_WORLD);

//MPI rank 1
MPI_Recv(r_buf_h, size,MPI_CHAR, 0, 100, MPI_COMM_WORLD, &status);
cudaMemcpy(r_buf_d, r_buf_h, size, cudaMemcpyHostToDevice);
```

- CUDA-aware MPI (w/ UCX)

```
//MPI rank 0
MPI_Send(s_buf_d, size, MPI_CHAR, 1, 100, MPI_COMM_WORLD);

//MPI rank n-1
MPI_Recv(r_buf_d, size, MPI_CHAR, 0, 100, MPI_COMM_WORLD, &status);
```

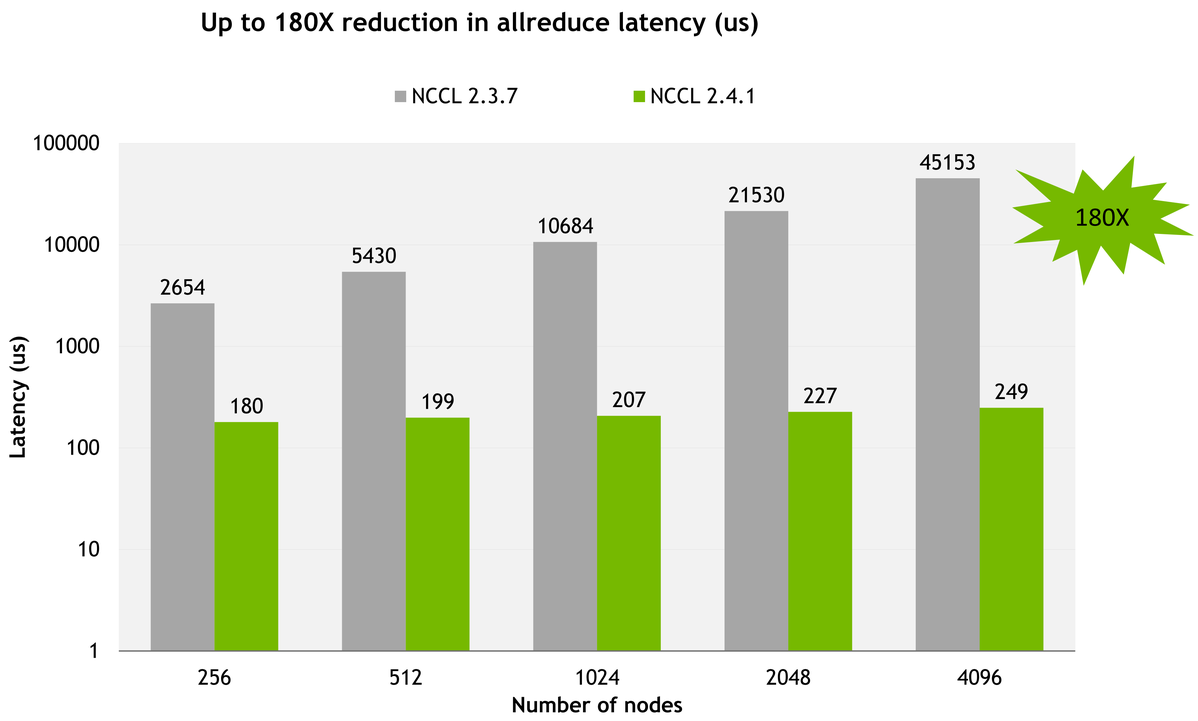
NCCL

NVIDIA Collective Communication Library

- Multi-GPU and Multi-Node Collectives Optimized for NVIDIA GPUs
- Automatic Topology Detection
- Easy to integrate | MPI Compatible
- Minimize latency | Maximize bandwidth

Impact

- Accelerates leading deep learning frameworks
- Adoption in HPC accelerating



AllReduce 8-byte (float) latency, Summit Supercomputer at Oak Ridge National Lab
4096 nodes, 6xV100, 2x IB EDR

NCCL

Initialization and Teardown with MPI

```
MPI_Init(&argc,&argv);
MPI_Comm_size(MPI_COMM_WORLD,&size);
MPI_Comm_rank(MPI_COMM_WORLD,&rank);

ncclUniqueId nccl_uid;
if (rank == 0) ncclGetUniqueId(&nccl_uid);
MPI_Bcast(&nccl_uid, sizeof(ncclUniqueId), MPI_BYTE, 0, MPI_COMM_WORLD);
MPI_Barrier(MPI_COMM_WORLD);

ncclComm_t nccl_comm;
ncclCommInitRank(&nccl_comm, size, nccl_uid, rank);
…
ncclCommDestroy(nccl_comm);
MPI_Finalize();
```

NCCL

Communication

Message ordering is important if messages are **not** unique, e.g. here if `top == bottom` (#GPU <= 2)

```
#ifdef USE_DOUBLE
#define NCCL_REAL_TYPE ncclDouble
#else
#define NCCL_REAL_TYPE ncclFloat
#endif
...
launch_jacobi_kernel(a_new, a, l2_norm_d, iy_start, iy_end, nx, compute_stream);
ncclGroupStart();
ncclRecv(a_new,                  nx, NCCL_REAL_TYPE, top, nccl_comm, compute_stream);
ncclSend(a_new +(iy_end - 1)*nx, nx, NCCL_REAL_TYPE, btm, nccl_comm, compute_stream);
ncclRecv(a_new +(iy_end*nx),     nx, NCCL_REAL_TYPE, btm, nccl_comm, compute_stream);
ncclSend(a_new + iy_start * nx,  nx, NCCL_REAL_TYPE, top, nccl_comm, compute_stream);
ncclGroupEnd();
```

NCCL

Overlapping Communication and Computation

Need to use CUDA high priority streams to avoid NCCL comms getting stuck behind compute.

```
int leastPriority = 0;
int greatestPriority = leastPriority;
cudaDeviceGetStreamPriorityRange(&leastPriority, &greatestPriority);
cudaStream_t compute_stream,push_stream;
cudaStreamCreateWithPriority(&compute_stream, cudaStreamDefault, leastPriority);
cudaStreamCreateWithPriority(&push_stream, cudaStreamDefault, greatestPriority);

...
launch_jacobi_kernel(a_new, a, l2_norm_d, iy_start,       (iy_start + 1), nx, push_stream);
launch_jacobi_kernel(a_new, a, l2_norm_d, (iy_end - 1),   iy_end,         nx, push_stream);
launch_jacobi_kernel(a_new, a, l2_norm_d, (iy_start + 1), (iy_end - 1),   nx, compute_stream);
ncclGroupStart();
ncclRecv(a_new,                 nx, NCCL_REAL_TYPE,     top, nccl_comm, push_stream)
ncclSend(a_new + (iy_end - 1) * nx, nx, NCCL_REAL_TYPE, btm, nccl_comm, push_stream);
ncclRecv(a_new + (iy_end * nx),     nx, NCCL_REAL_TYPE, btm, nccl_comm, push_stream);
ncclSend(a_new + iy_start * nx,     nx, NCCL_REAL_TYPE, top, nccl_comm, push_stream);
ncclGroupEnd();
```

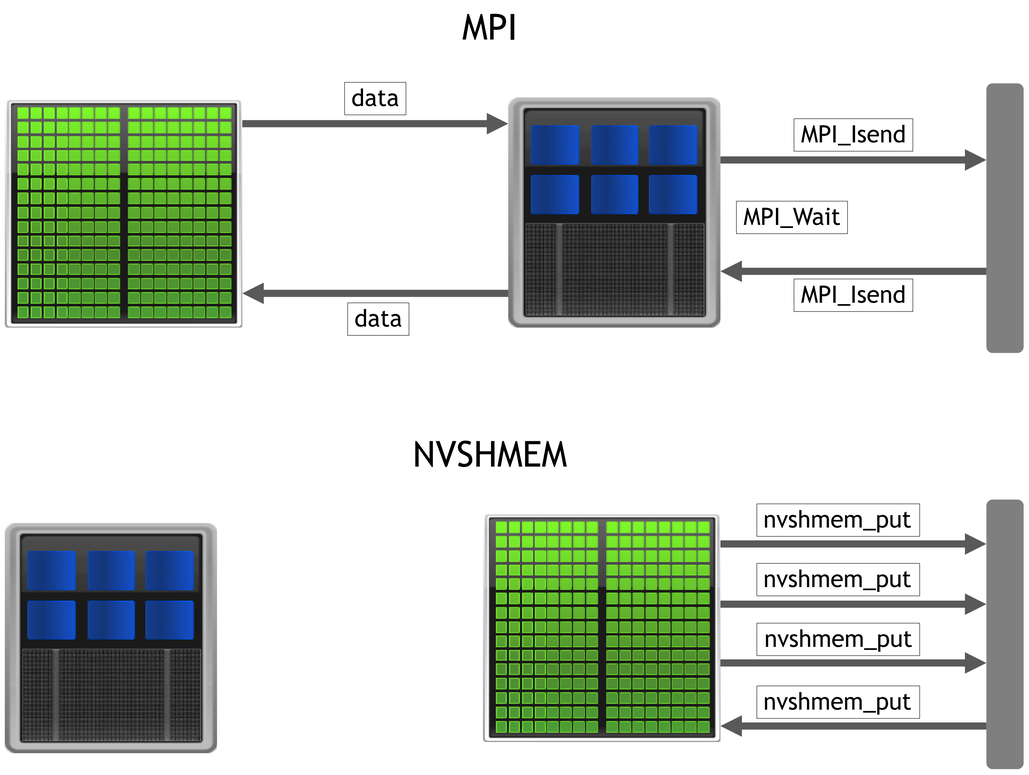
NVSHMEM

GPU Optimized SHMEM

- Initiate from CPU or GPU
- Initiate from within CUDA kernel
- Issue onto a CUDA stream
- Interoperable with MPI & OpenSHMEM

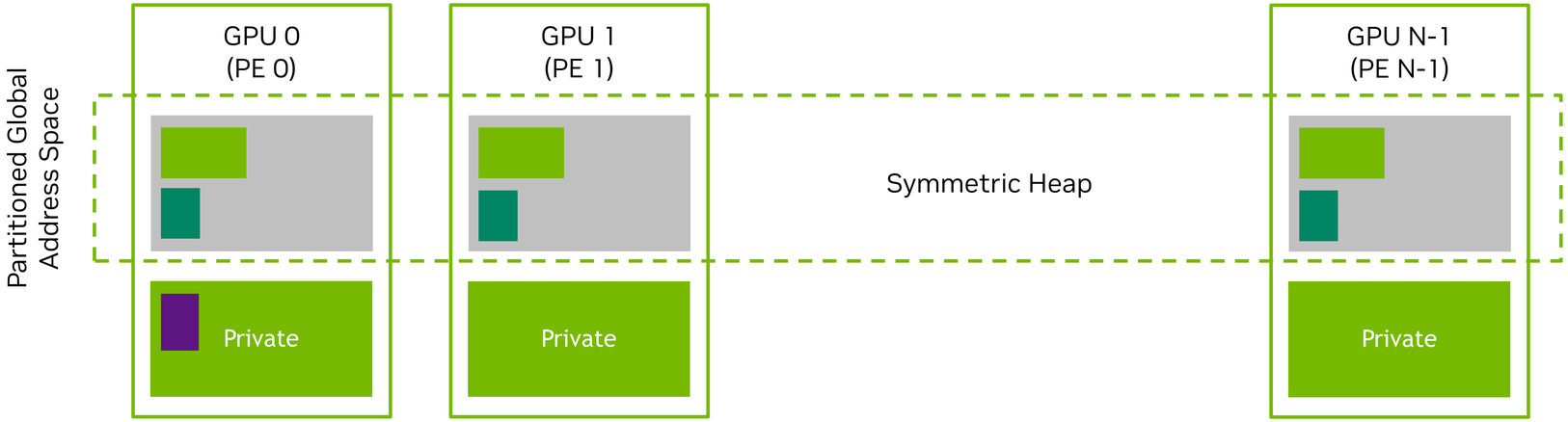
Application Impact

- LBANN, Kokkos/CGSolve, QUDA



NVSHMEM

Implementation of OpenSHMEM, a Partitioned Global Address Space (PGAS) library



Symmetric objects are allocated collectively with the same size on every PE

Symmetric memory: `nvshmem_malloc(…);` Private memory: `cudaMalloc(…);`

CPU (blocking and stream-ordered) and CUDA Kernel interfaces

Read: `nvshmem_get(…);` Write: `nvshmem_put(…);` Atomic: `nvshmem_atomic_add(…);`
Flush writes: `nvshmem_quiet();` Order writes: `nvshmem_fence();`

Synchronize: `nvshmem_barrier();` Poll: `nvshmem_wait_until(…);`

Interoperable with MPI

NVSHMEM - Skeleton

Interoperability with MPI and OpenSHMEM

```
MPI_Init(&argc, &argv);
MPI_Comm mpi_comm = MPI_COMM_WORLD;



nvshmemx_init_attr_t attr;
attr.mpi_comm = &mpi_comm;
nvshmemx_init_attr(NVSHMEMX_INIT_WITH_MPI_COMM, &attr);
assert( size == nvshmem_n_pes() );
assert( rank == nvshmem_my_pe() );
…
nvshmem_finalize();
MPI_Finalize();
```

MPI/SHMEM independent UID based init with NVSHMEM 2.11+

```
nvshmemx_uniqueid_t uid;
if (rank == 0) nvshmemx_get_uniqueid(&uid);
// broadcast uid to all ranks
...
nvshmemx_init_attr_t attr;
nvshmemx_set_attr_uniqueid_args(rank,size,&uid,&attr);
nvshmemx_init_attr(NVSHMEMX_INIT_WITH_UNIQUEID,&attr);
```

```
shmem_init();
nvshmemx_init_attr_t attr;
nvshmemx_init_attr(NVSHMEMX_INIT_WITH_SHMEM, &attr);
mype_node = nvshmem_team_my_pe(NVSHMEMX_TEAM_NODE);
```

NVSHMEM

Host Code

```
a = (float*)nvshmem_malloc( nx * (chunk_high + 2) * sizeof(float));
a_new = (float*)nvshmem_malloc(nx * (chunk_high + 2) * sizeof(float));
...
while (l2_norm > tol && iter < iter_max) {
    ...
    jacobi_kernel<<<dim_grid, dim_block, 0, compute_stream>>>(
        a_new, a, l2_norm_d, iy_start, iy_end, nx,
        top_pe, iy_end_top, bottom_pe,iy_start_bottom);
    CUDA_RT_CALL(cudaGetLastError());
    nvshmemx_barrier_all_on_stream(compute_stream);
    ...
}
nvshmem_free(a);
nvshmem_free(a_new);
```

NVSHMEM

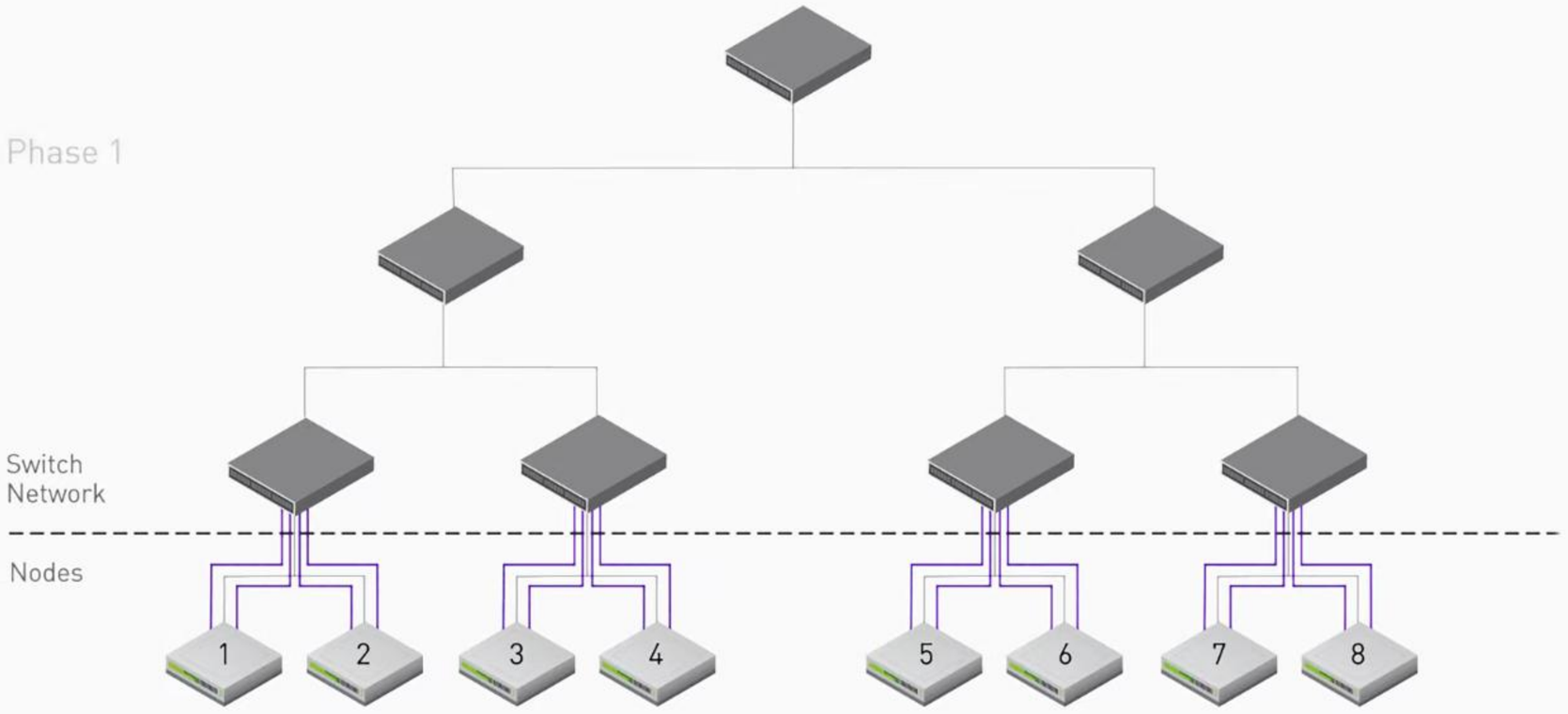
Device Code

Optimized for DGX H100/NVLink, other approach might be better for portable performance

```
__global__void jacobi_kernel(...) {
    int ix = blockIdx.x * blockDim.x + threadIdx.x + 1;
    int iy = blockIdx.y * blockDim.y + threadIdx.y + iy_start;
    real local_l2_norm = 0.0;
    if (iy < iy_end && ix < (nx - 1)) {
        const real new_val = 0.25 * (a[iy * nx + ix + 1] + a[iy * nx + ix - 1] +
                                     a[(iy + 1) * nx + ix] + a[(iy - 1) * nx + ix]);
        a_new[iy * nx + ix] = new_val;
        local_l2_norm += (new_val - a[iy * nx + ix])*(new_val - a[iy * nx + ix]);
        if (iy_start == iy)     nvshmem_float_p(a_new + t_iy * nx + ix, new_val, top_pe);
        if ((iy_end - 1) == iy) nvshmem_float_p(a_new + b_iy * nx + ix, new_val, bottom_pe);
    }
    atomicAdd(l2_norm, local_l2_norm);
}
```

SHARP

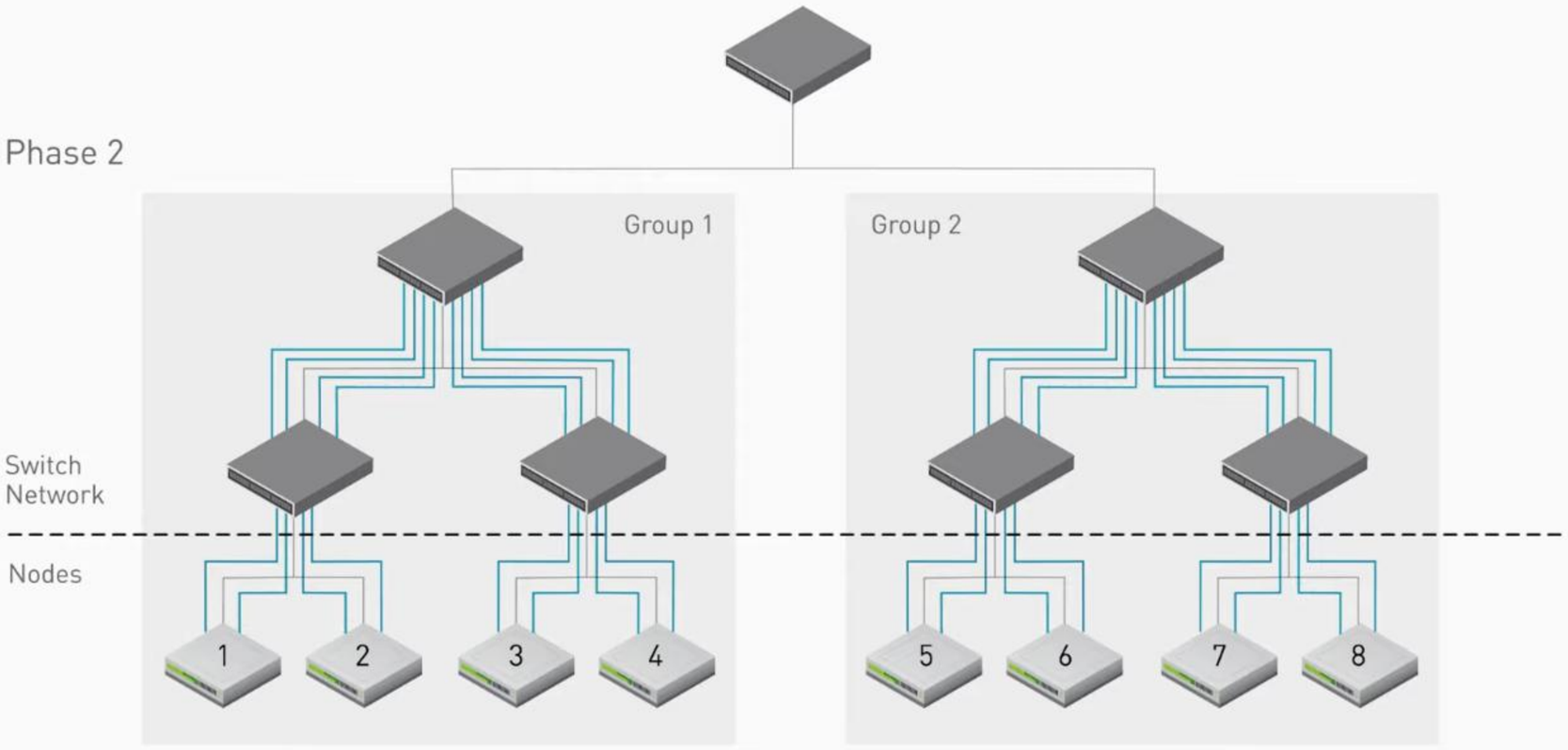
~~Scalable Hierarchical Aggregation and Reduction Protocol~~



(w/ HCOLL, UCC)

SHARP

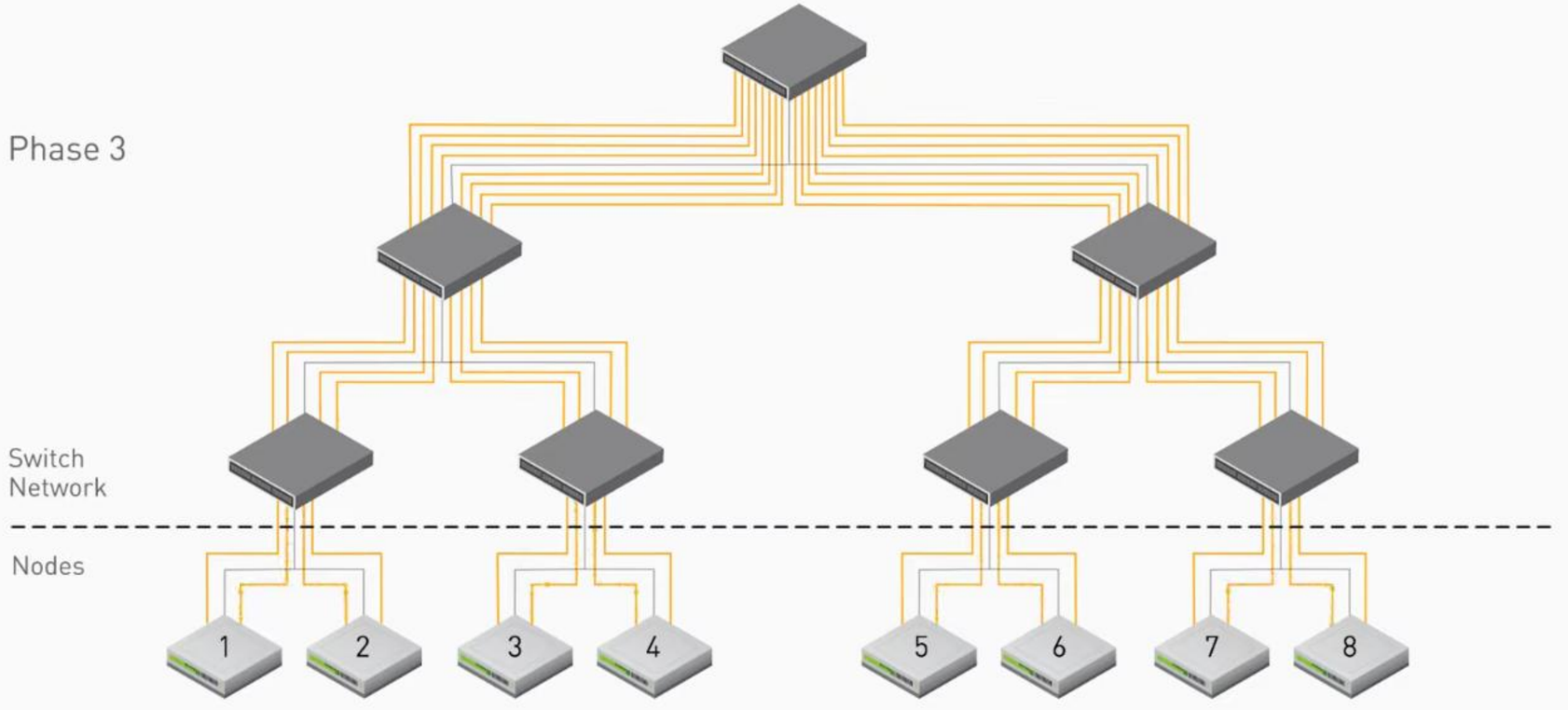
~~Scalable Hierarchical Aggregation and Reduction Protocol~~



(w/ HCOLL, UCC)

SHARP

~~Scalable Hierarchical Aggregation and Reduction Protocol~~

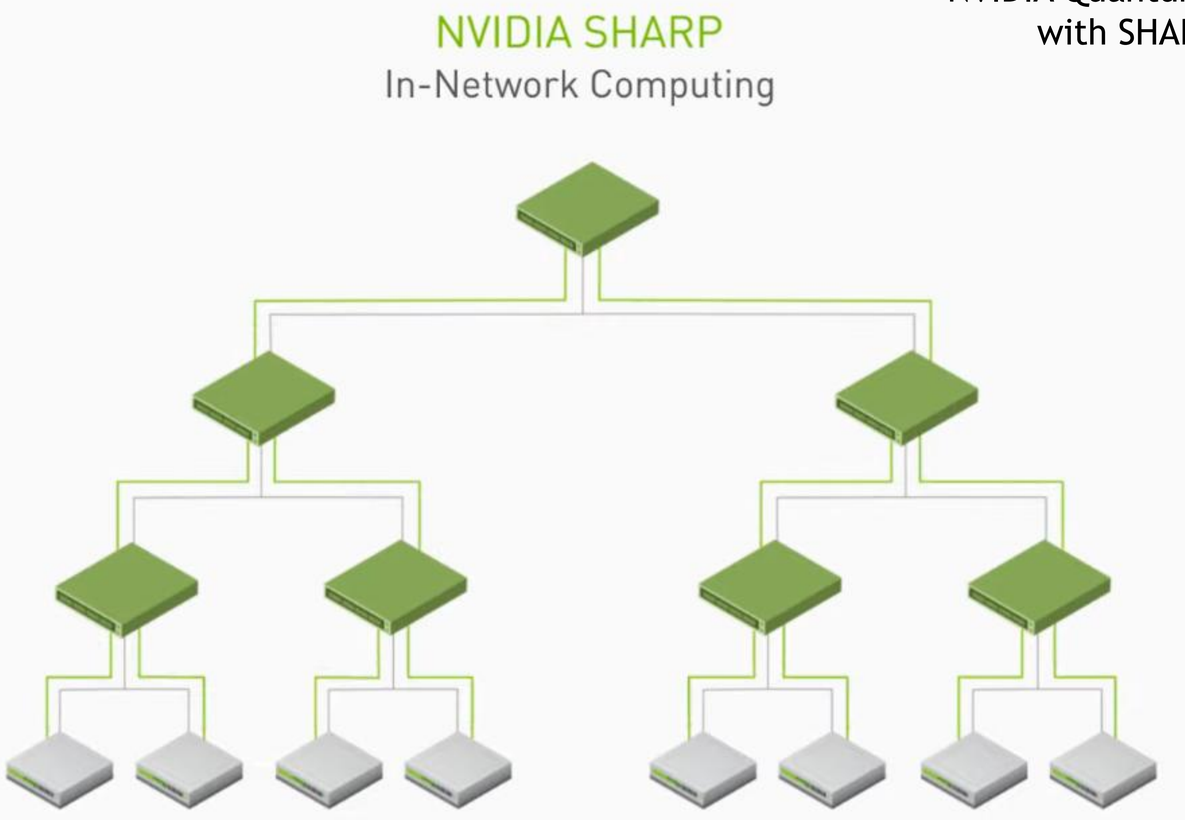


(w/ HCOLL, UCC)

SHARP

Scalable Hierarchical Aggregation and Reduction Protocol

NVIDIA Quantum InfiniBand Switch
with SHARP Technology

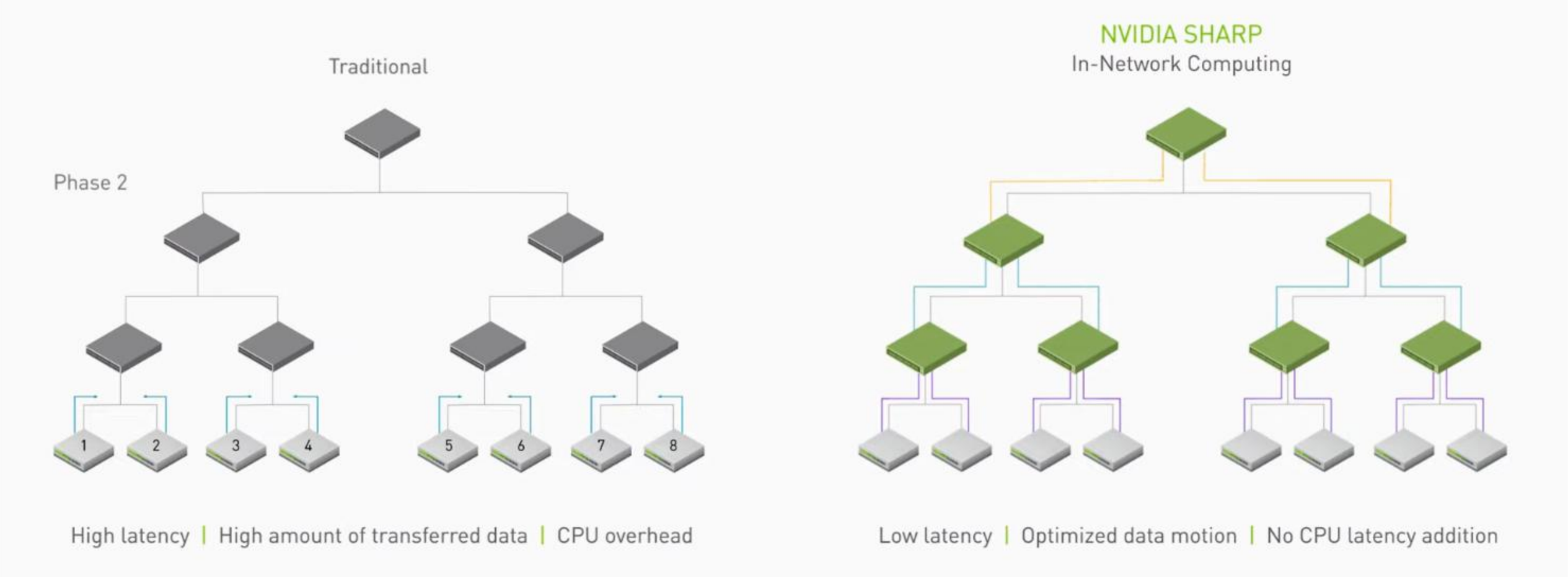


Low latency | Optimized data motion | No CPU latency addition

(w/ HCOLL, UCC)

SHARP

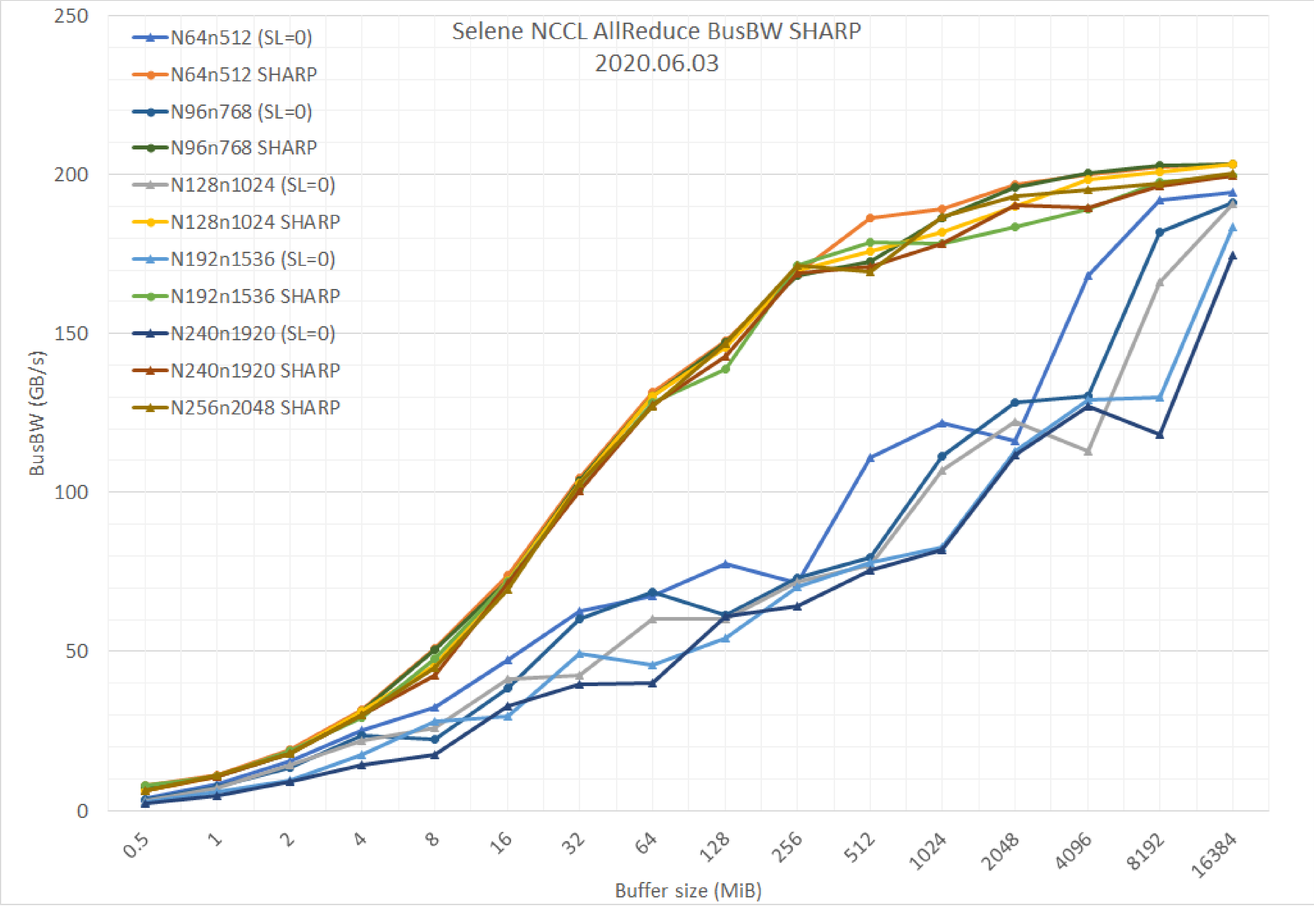
Scalable Hierarchical Aggregation and Reduction Protocol

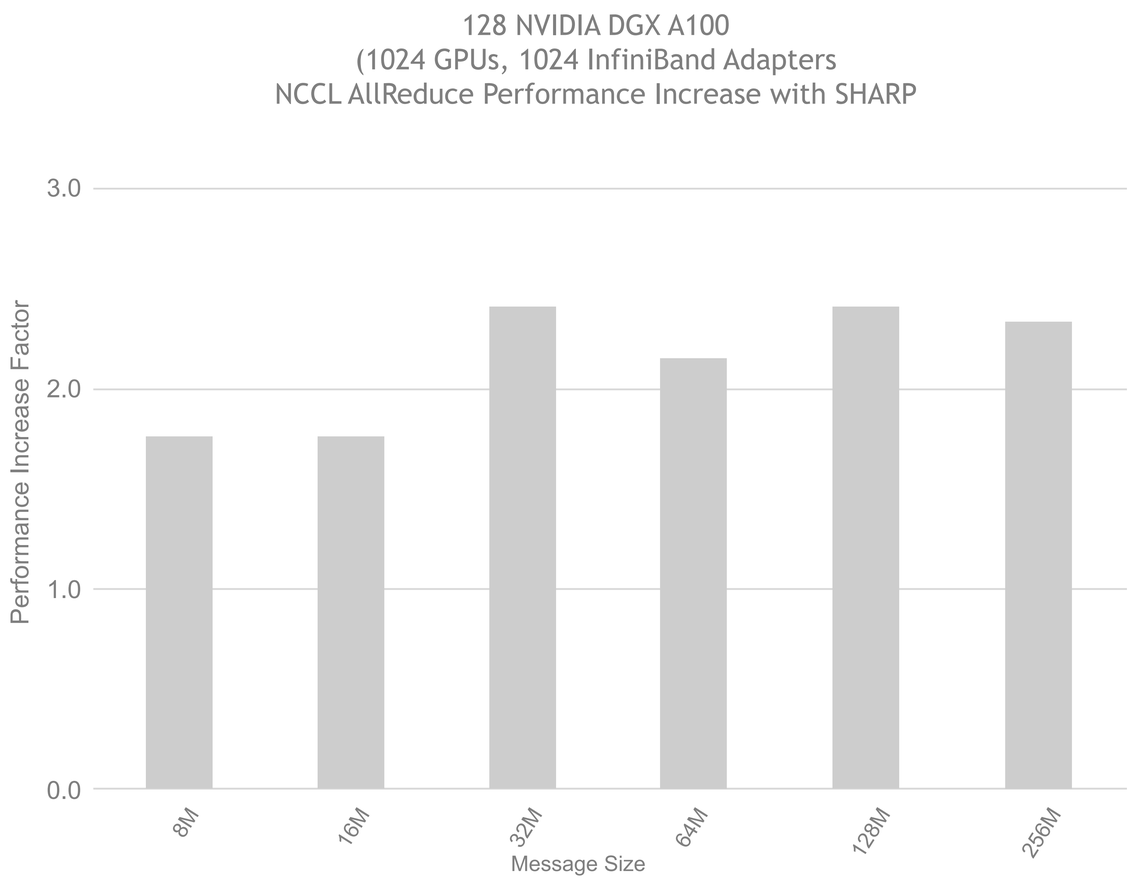


(w/ HCOLL, UCC)

NVIDIA HDR 200GB/S INFINIBAND

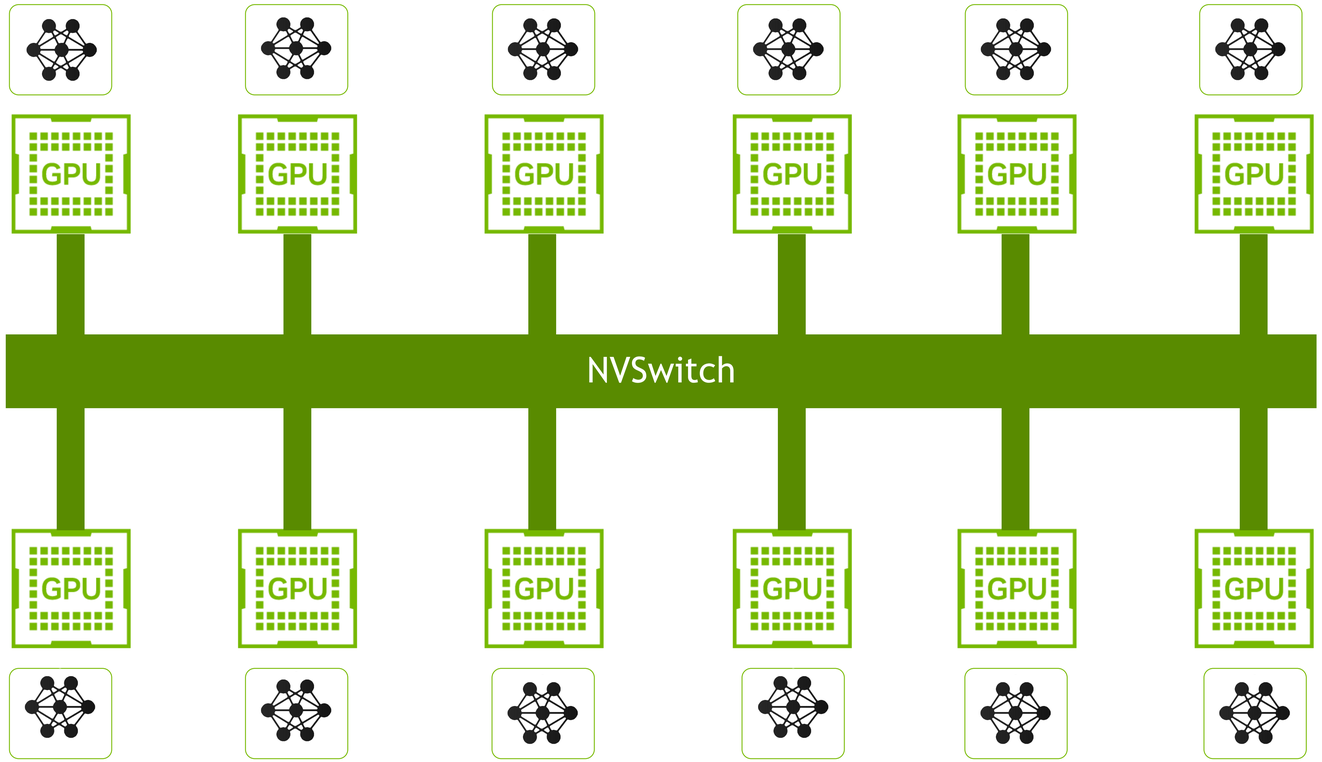
Results on Selene





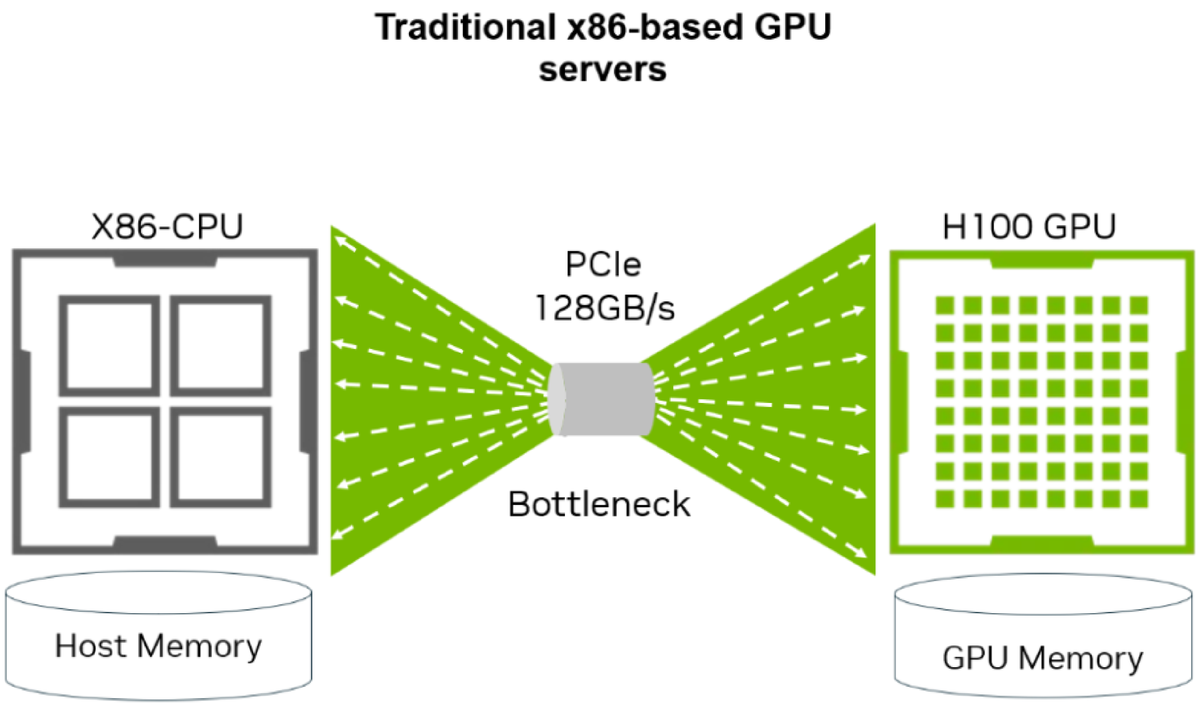
(w/ HCOLL, UCC)

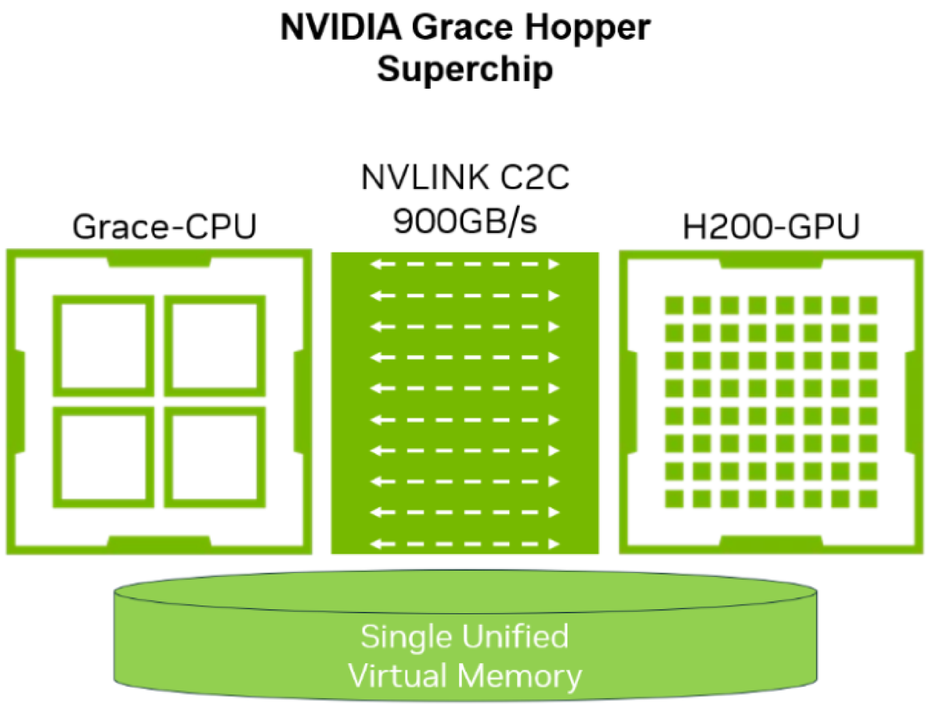
NVLink and NVSwitch (Recap)



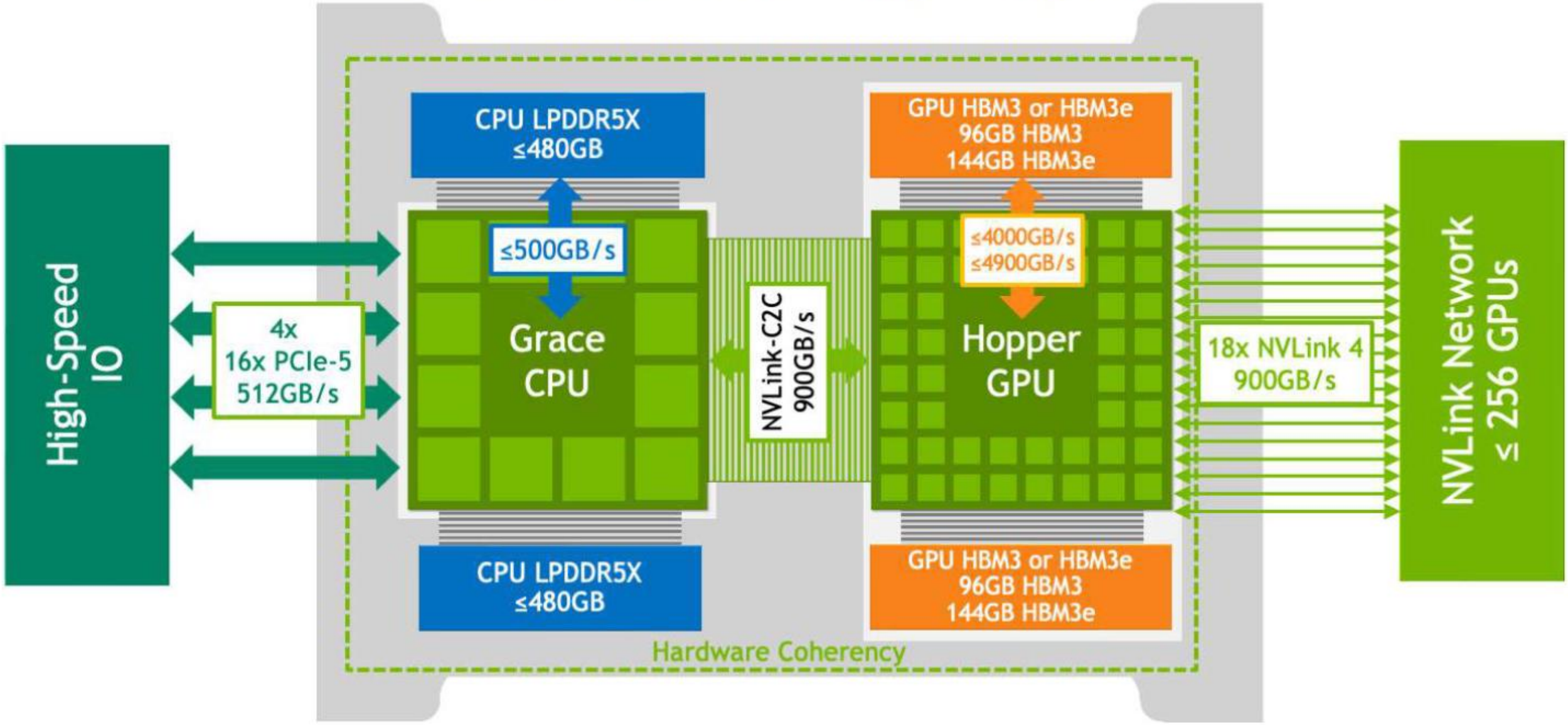
GH200 Grace Hopper Superchip

NVLink C2C



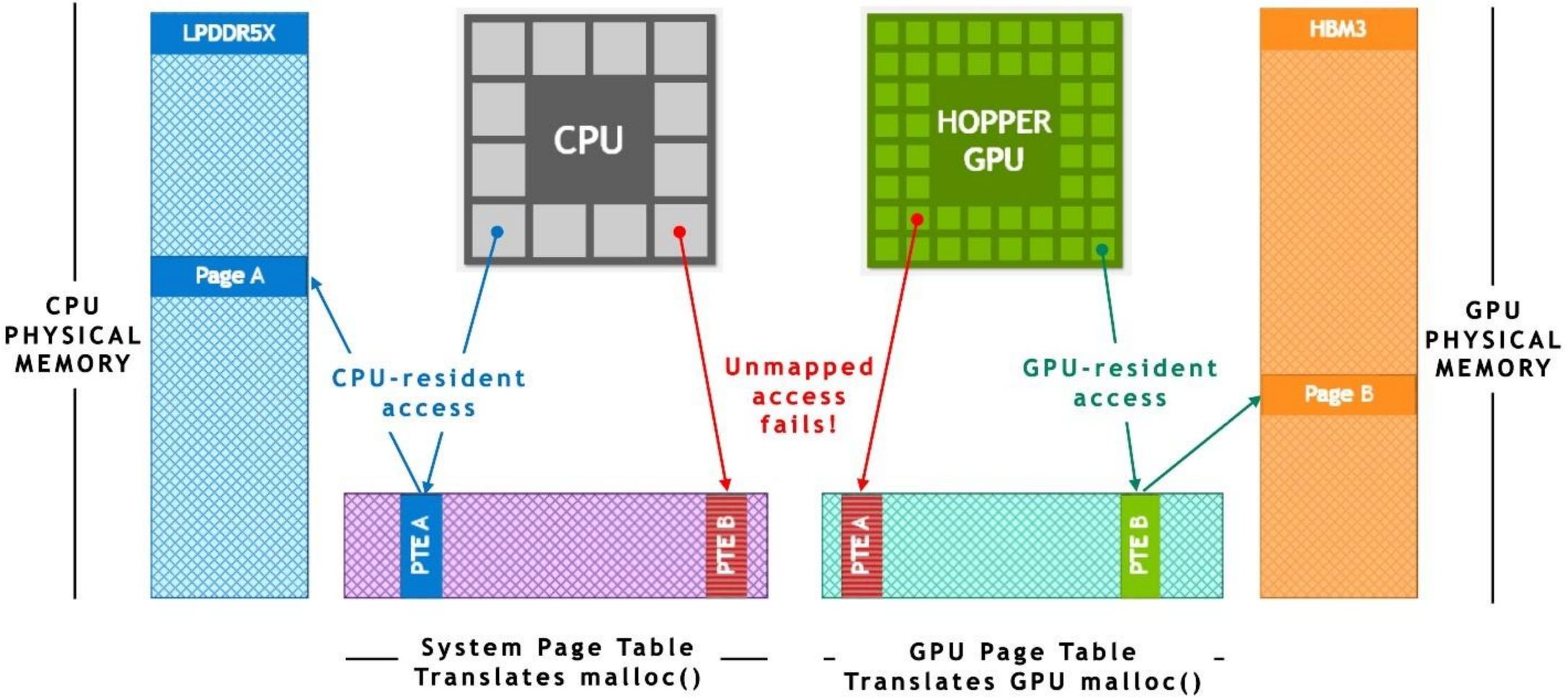


NVIDIA GH200 Grace Hopper Superchip



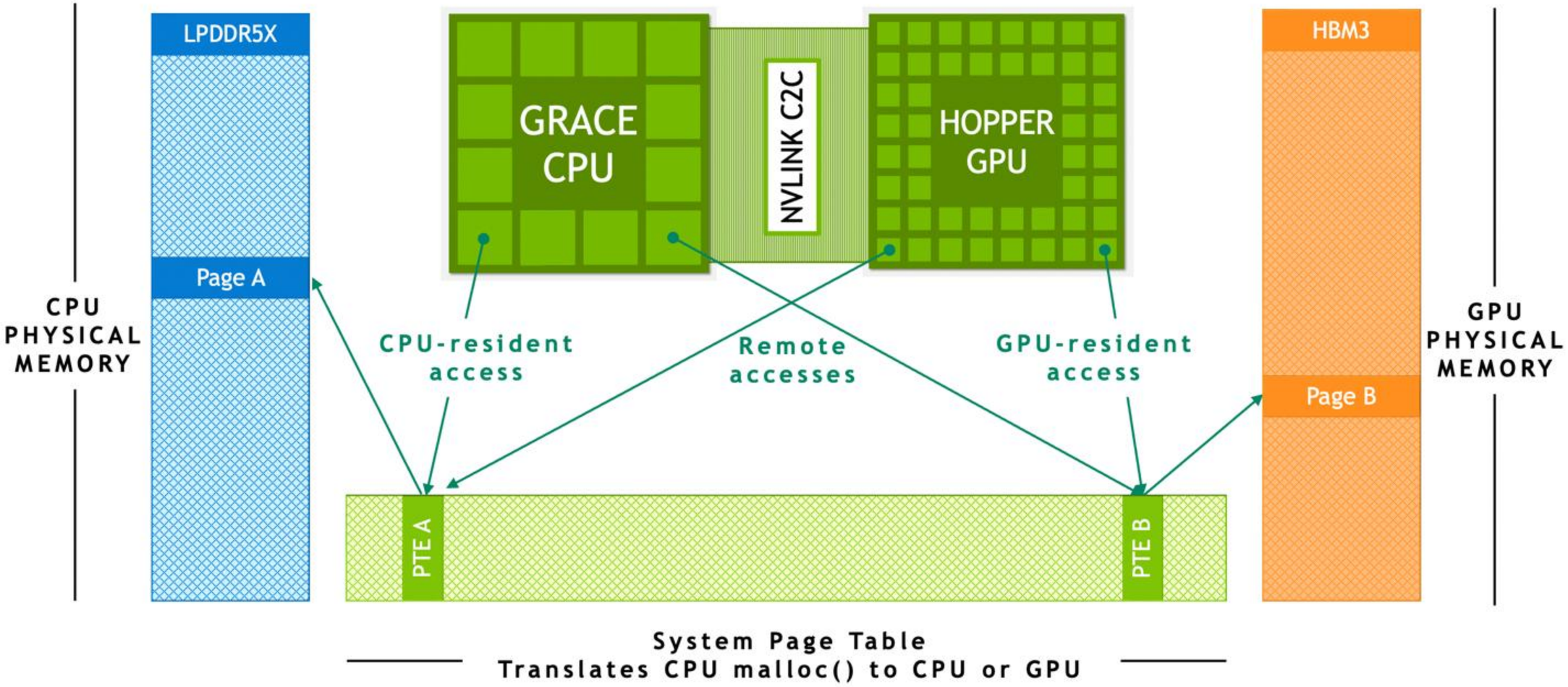
GH200 Grace Hopper Superchip

~~Unified Memory~~



GH200 Grace Hopper Superchip

Unified Memory



GH200 Grace Hopper Superchip

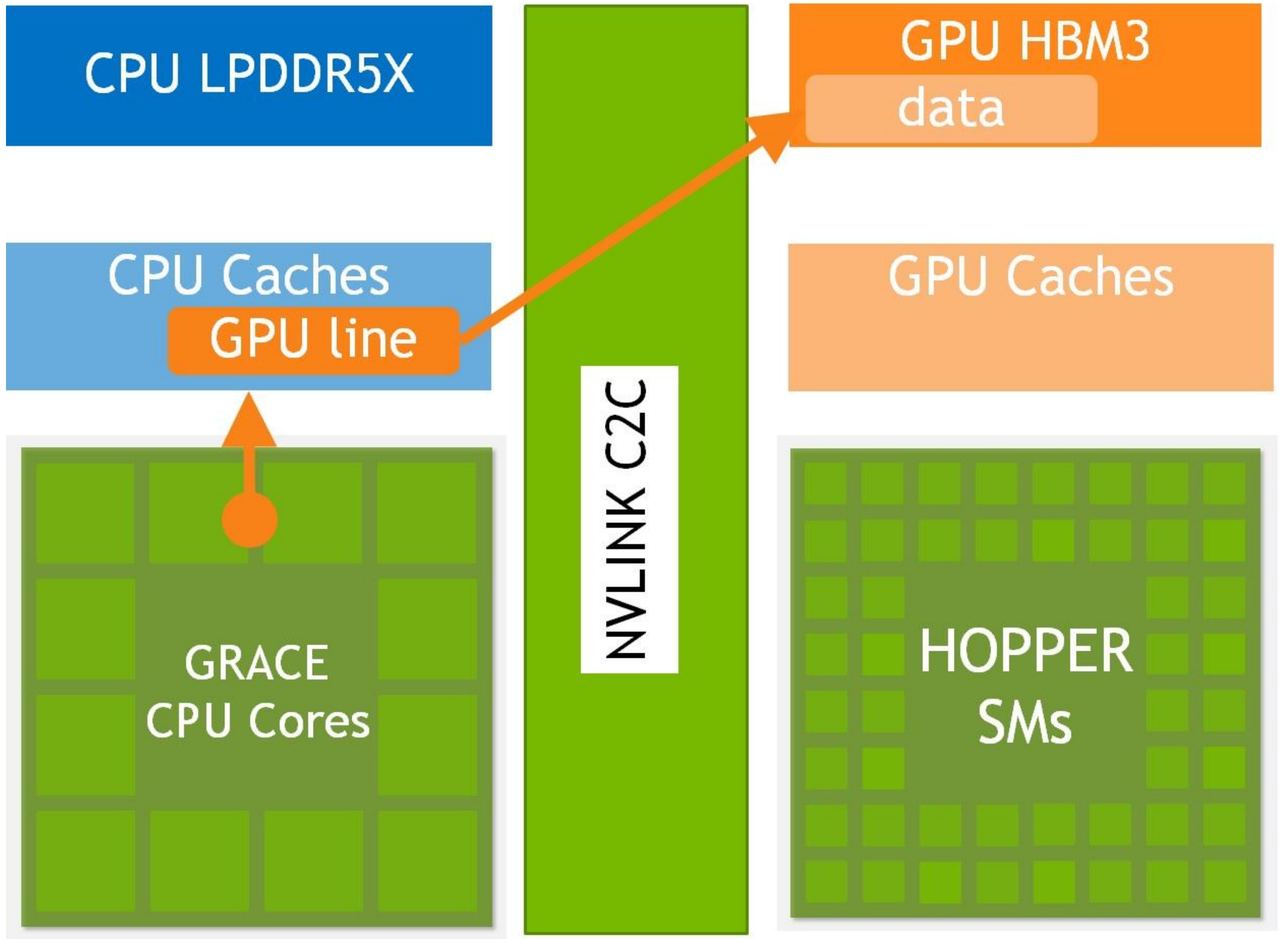
Memory Coherent Architecture

- Grace directly reading Hopper's memory:

  CPU fetches GPU data into CPU L3 cache.

  The cache remains **coherent** with GPU memory.

  Changes to GPU memory **evict** cache line.



GH200 Grace Hopper Superchip

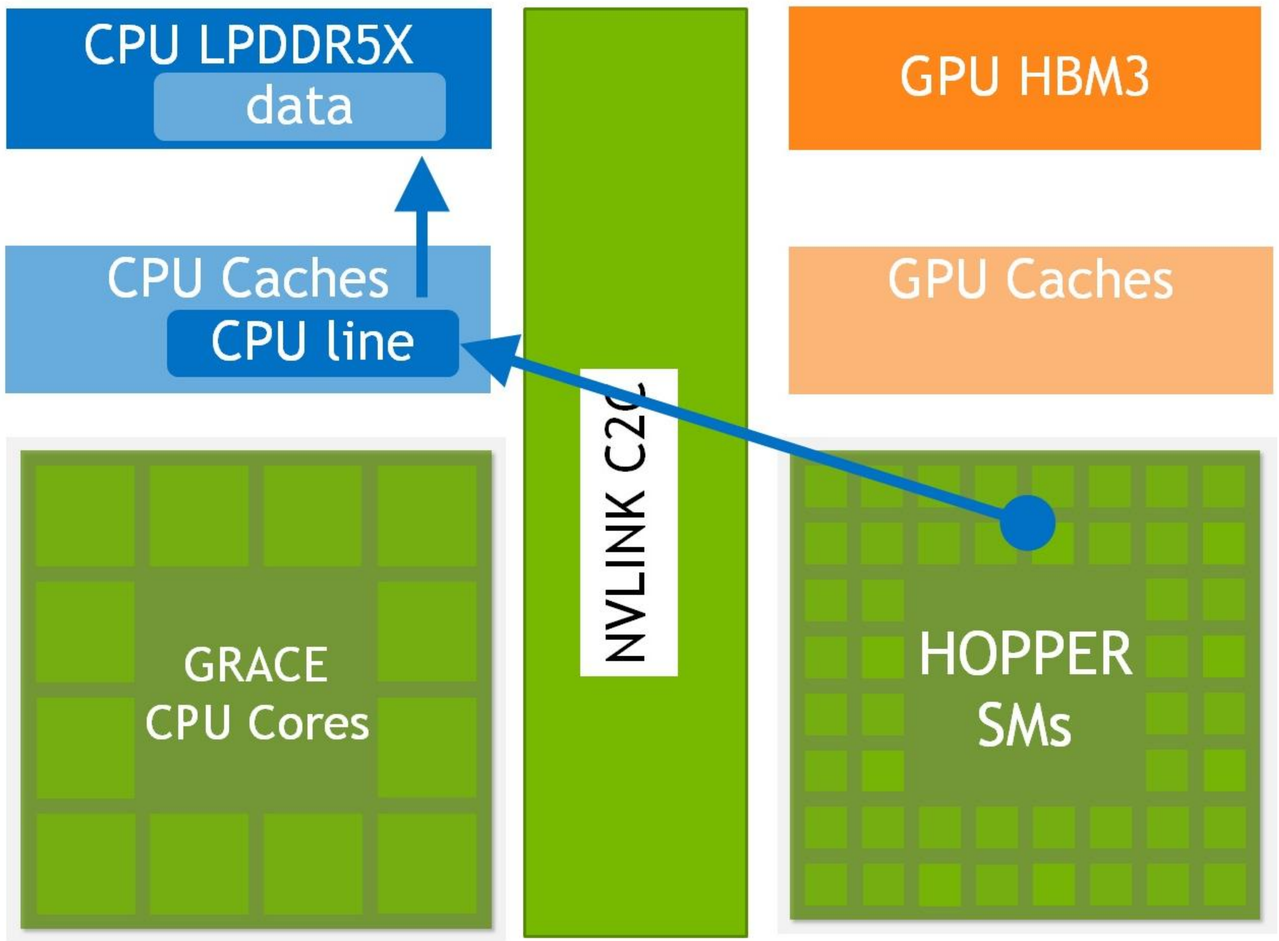
Memory Coherent Architecture

- Hopper directly reading Grace's memory:

  GPU loads CPU data via CPU L3 cache.

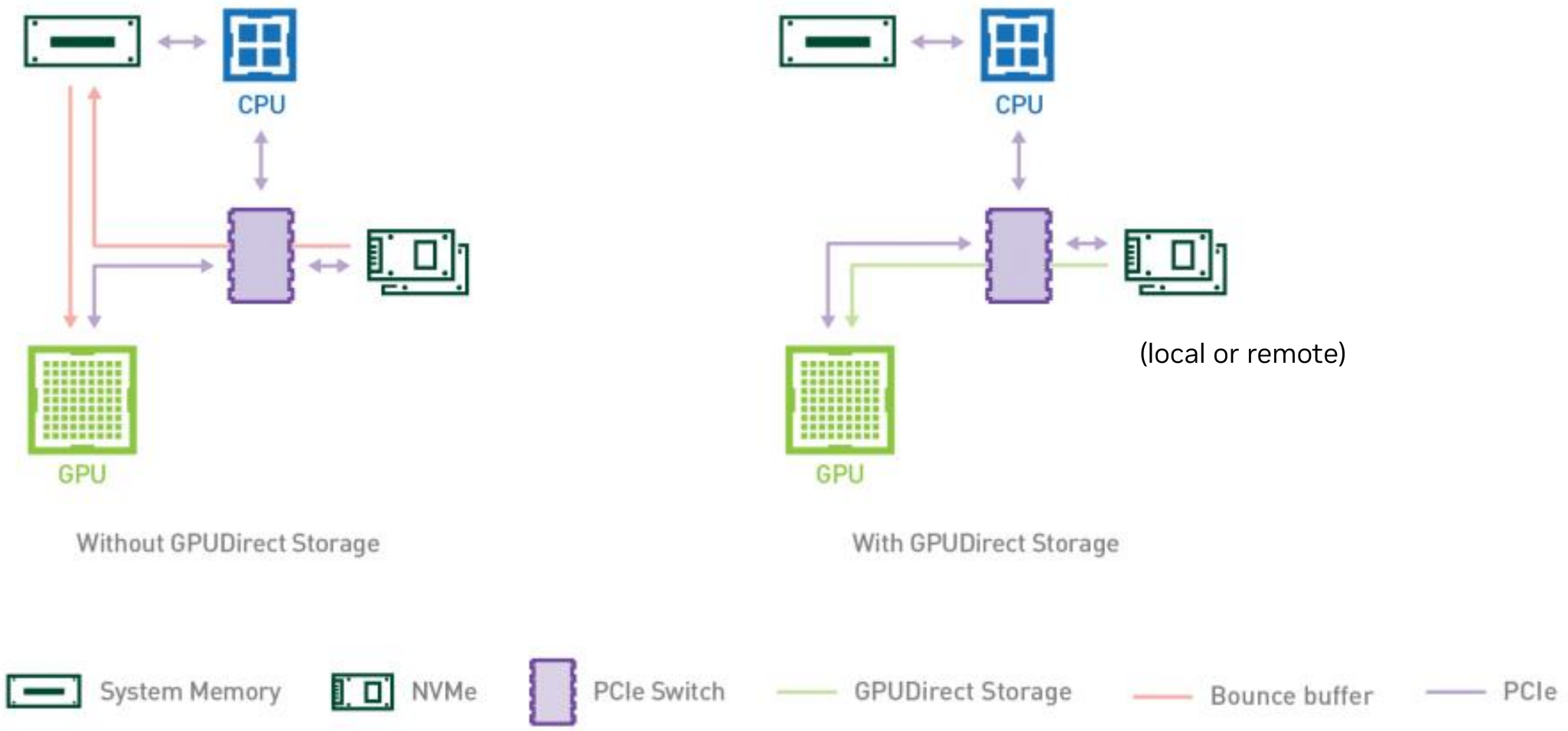
  CPU and GPU **can both hit** on cached data.

  Changes to CPU memory **update** cache line.



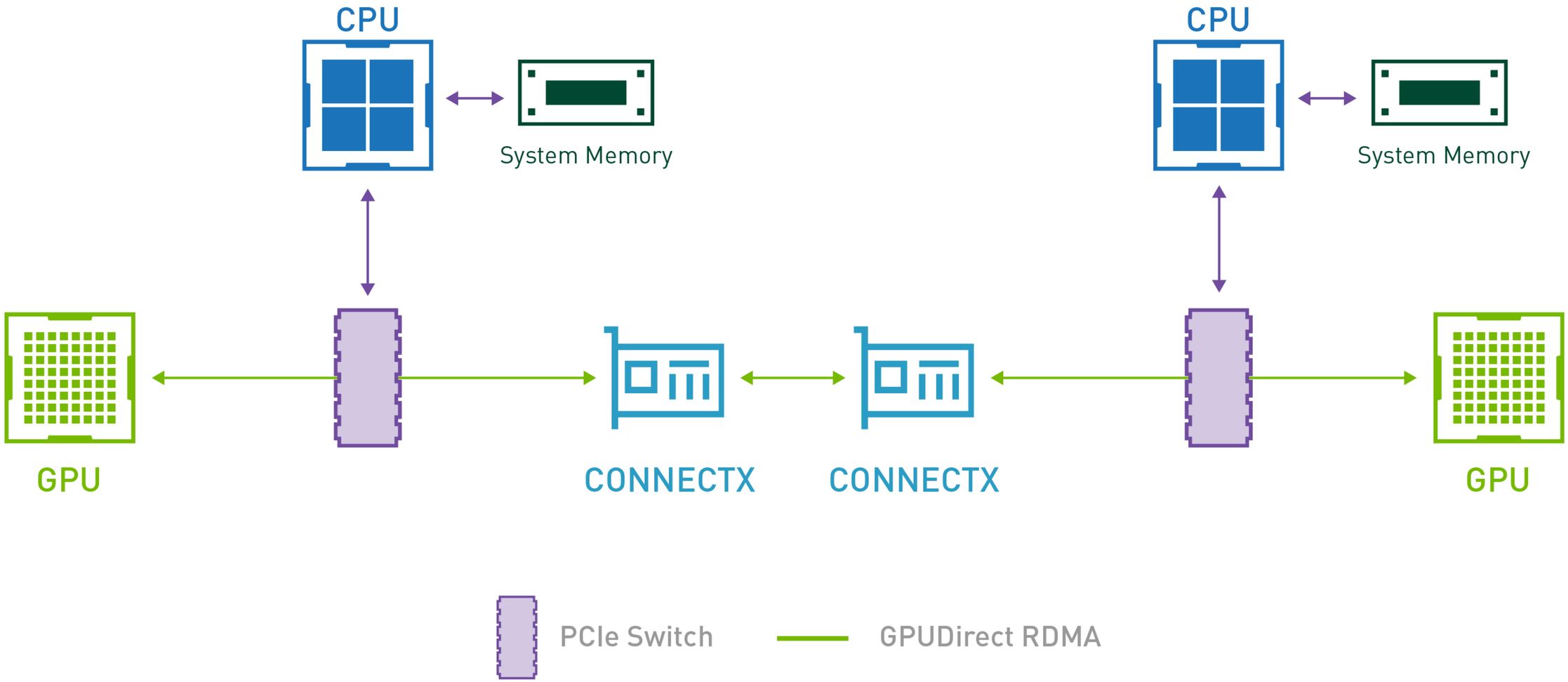
NVIDIA GPUDirect

GPUDirect Storage



NVIDIA GPUDirect

GPUDirect RDMA



Communication Libraries: HPC-X (Recap)

- CUDA-aware MPI (w/ UCX)
- NCCL (w/ NVLink)
- NVSHMEM, PGAS
- SHARP (w/ HCOLL or UCC)
- NVLink-C2C, Memory Coherence
- GPUDirect Storage/RDMA

Thank You!

Post-Talk Survey

Supplementary Materials

- Please complete this survey before leaving!

https://github.com/j3soon/advanced-topics-on-hpc-supp