Advanced code development on the modern HPC GPU architectures

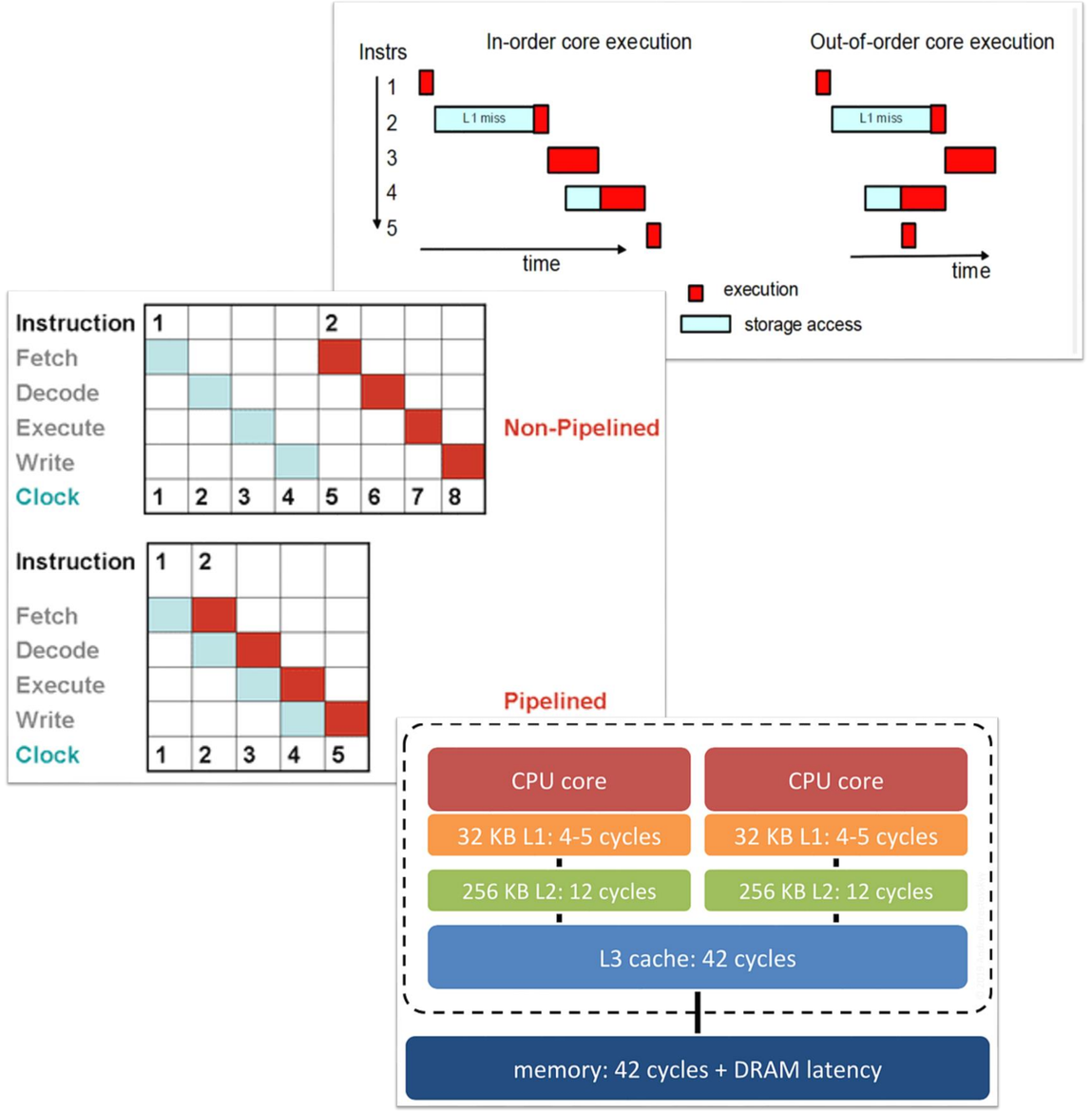
Dr. Dmitry Alexeev, NVIDIA

07/2025

Why the GPUs are the way they are?

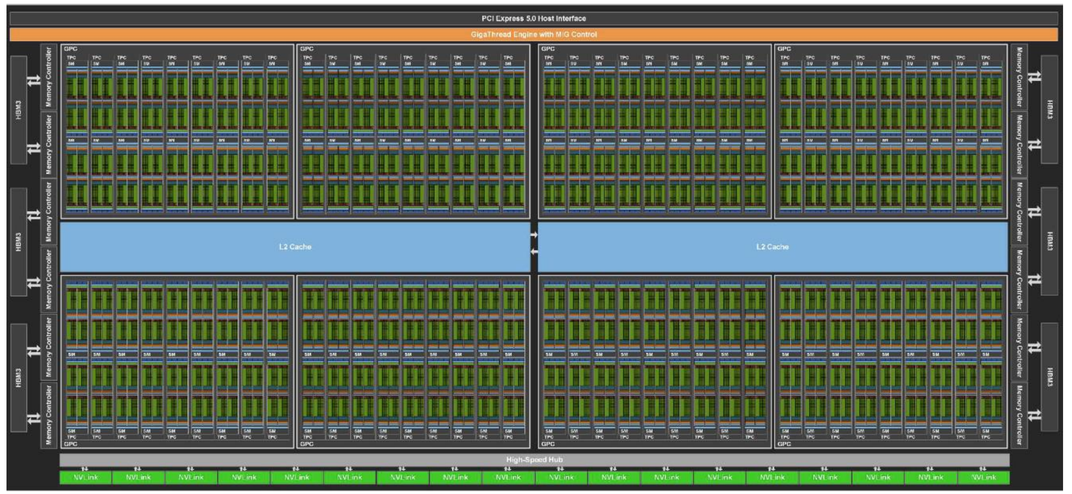
First the CPUs

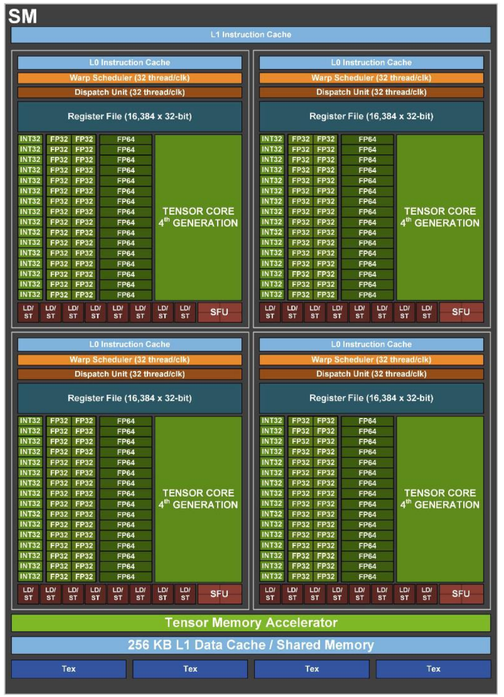
- A CPU (core) wants to execute a **stream** of instructions as fast as possible (think IPC)
- Every instruction takes a few cycles, 5-50-500 or more
- How can we reduce the latency?
  - Pipelining!
- But what if pipelining is not possible?
  - Instructions with too much latency (memory accesses) => cache
  - Data dependency => out-of-order execution
  - Branching => speculative execution and branch predictions
  - etc. etc. etc
- Now the CPU core has a very large **frontend**, and can execute instructions very very fast
- What else do we need?



Now the GPUs

- Can we somehow save transistors on the **frontend** and use them instead for the **backend**?
- Yes! But only for certain tasks
- The GPU **assumes** that the running task has a lot of data parallelism, i.e., very many items that can be processed *almost* independently
- But if we just multiply a CPU core and make each one process its own elements, we'll still need a wide frontend to reduce latency
- Instead, we'll use **oversubscription**

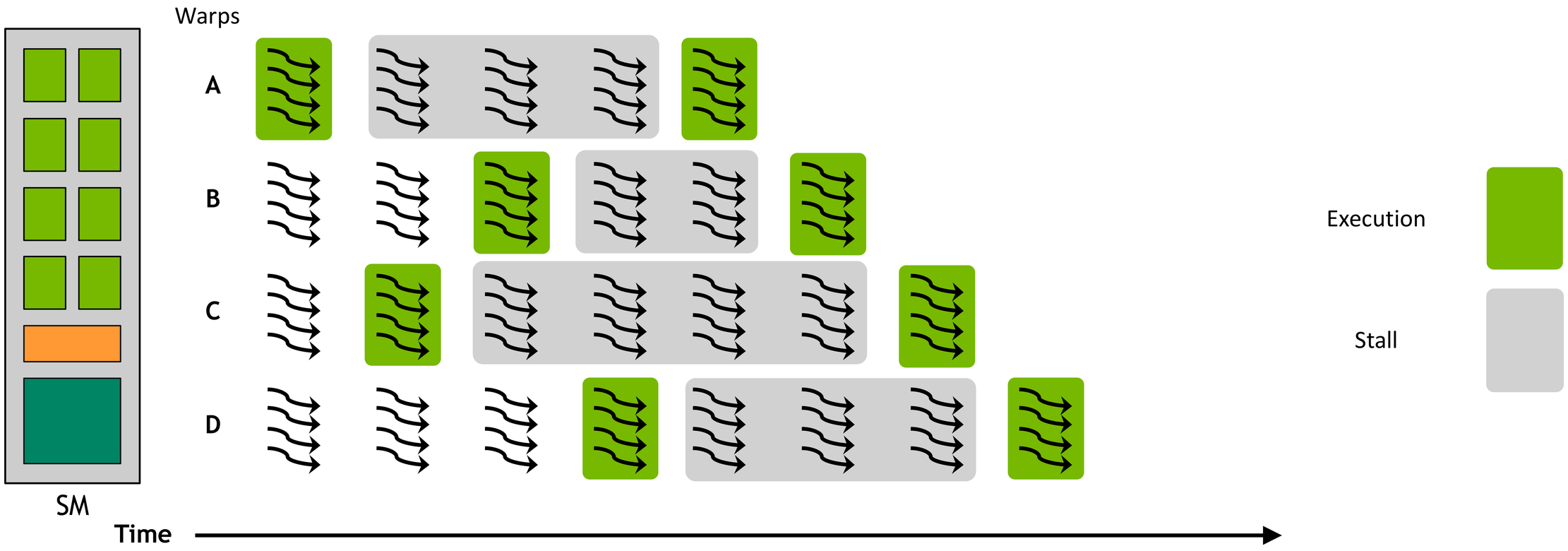




Now the GPUs

Oversubscription

- The idea is that we **additionally assume** that there are much more independent items than the hardware execution units
- Therefore, the execution units can switch between the different items in case of stalls
- Fast switching means that HW needs to store the state of each item it works on in fast memory: registers and cache
- These resources are limited, hence **occupancy**



Now the GPUs

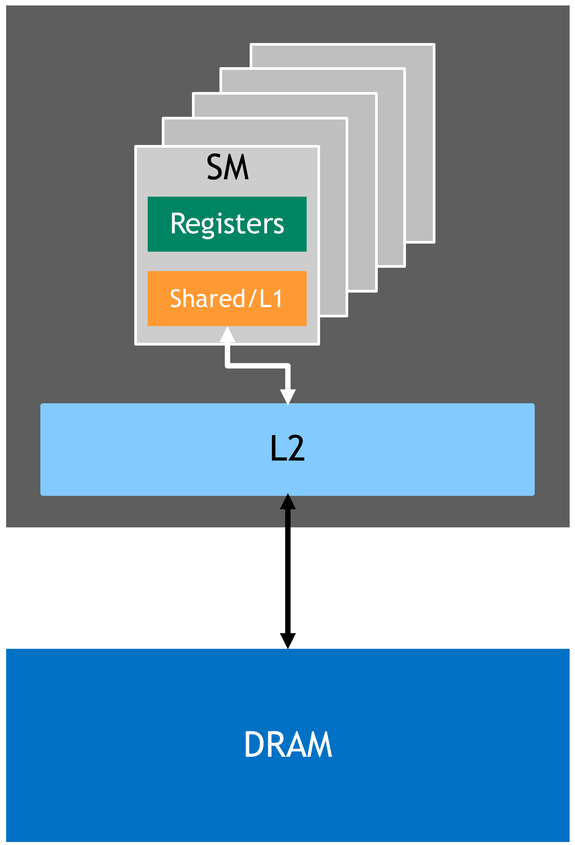
SMs

- Streaming multiprocessor is like a “core” of the GPU, and it implements the idea of oversubscription
- No OOO execution or branch prediction, frontend is very simple and predictable
- To save more frontend space: 1 instruction per 32 data elements, or **threads**. This is what’s called a **warp** and the concept is similar to CPU SIMD instructions, however, more flexible
- We need quite a few registers and cache, or shared memory, per each backend execution unit to enable fast context switching
- Added benefit: warps that reside on the same SM are physically close to each other and can synchronize/communicate quickly. This is exposed in CUDA as **thread blocks**
- Modern GPUs have more complexity with SM clusters, tensor cores connected to multiple SMs, separate instruction pointer per thread, etc. I only touched the basics!

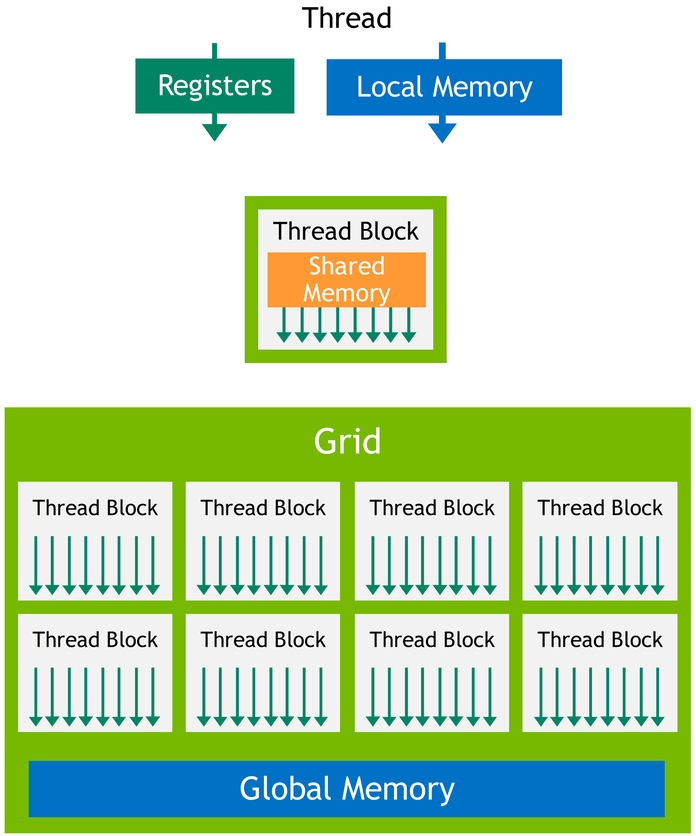
Now the GPUs

Memory hierarchy, because DRAM latency and bandwidth are still not great

Hardware



CUDA/Software



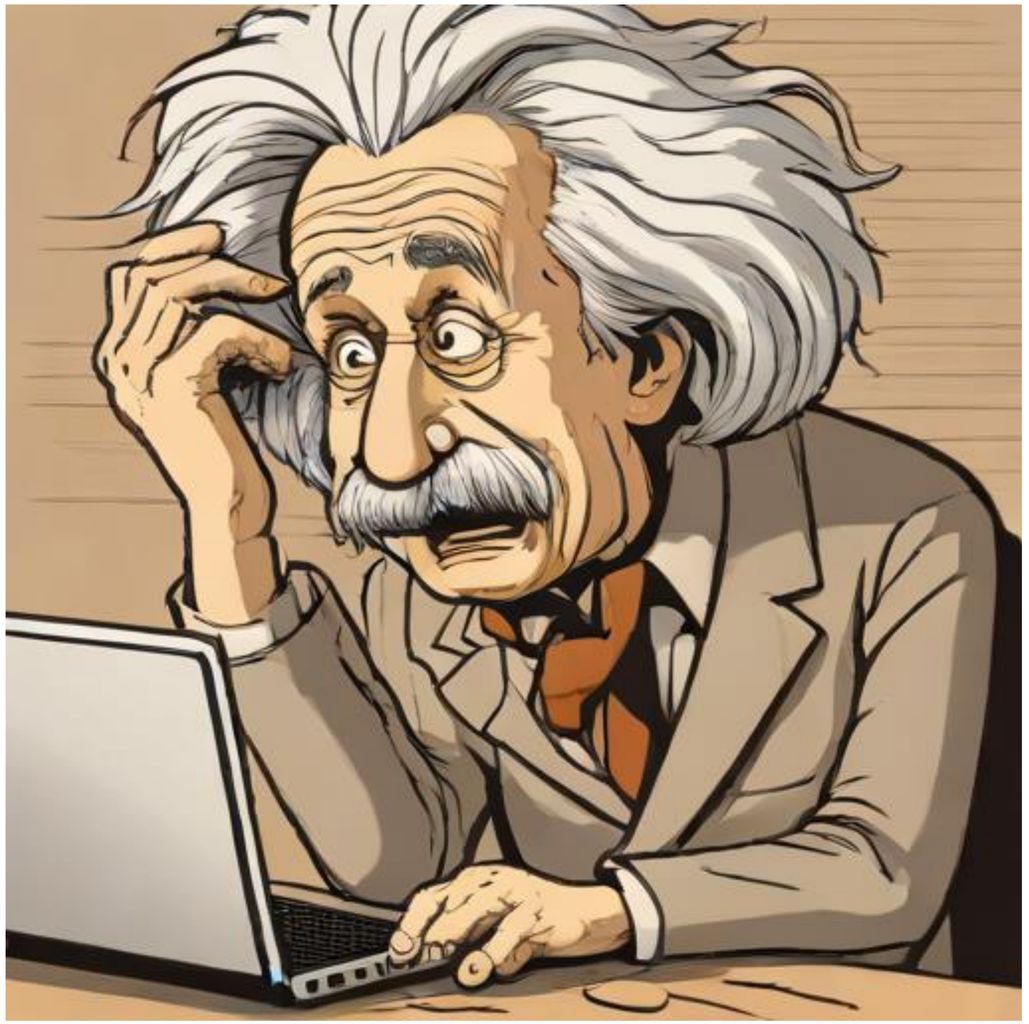
- Per-thread **registers**.
  - Lowest possible latency.
- Per-thread **local** memory.
  - Private storage.
  - Backed up by **global** memory (hence the color).
- Per-block **shared** memory.
  - Visible by all threads in a block.
  - Can be used to exchange data between threads in a thread block.
  - Very fast access.
  - Can serve as **L1** cache.
- **Global** memory.
  - Visible by all threads in a grid.
  - Slowest access.
  - Augmented by **L2** shared across all SMs and per-SM **L1**

How to program the GPUs?

Dispelling Some Myths

- GPU Programming...
  - …is hard
  - …is time-consuming
  - …isn't portable
  - …requires a special language or extensions
  - …can only be done in C++
  - …is only for ninja programmers

FALSE



Programming the NVIDIA Platform

Languages & Programming Models	Accelerated Standard Languages	Incremental Optimization	Platform Specialization
	  Fortran	OpenACC More Science, Less Programming 	 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

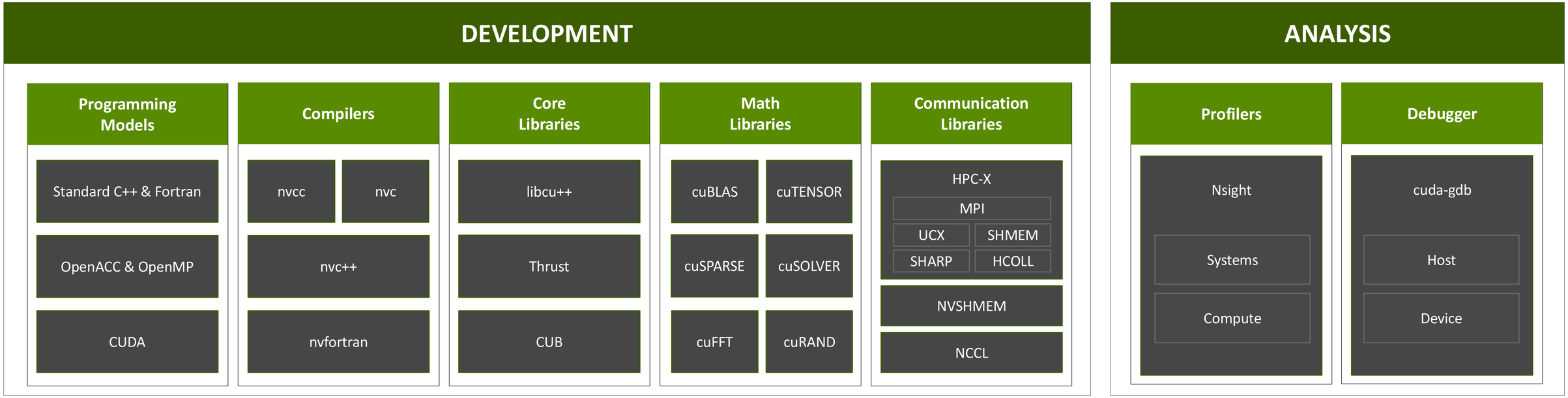
Programming the NVIDIA Platform

The best code is no code

- If your code uses an algorithm described in a detailed Wikipedia page, chances are the is a GPU library implementing it
- Python world has a lot of excellent GPU frameworks. Designed mainly for machine learning, they can be used for much more. Examples: Pytorch, JAX, Numba, Warp, CuPy, etc.
- Standard parallelism in Fortran and C++ gives you a compact way to express simple patterns, like parallel foreach
- OpenACC or OpenMP offload is a step up: more flexibility and potential performance, more code.
- CUDA is the top of the iceberg. Although, some algorithms are much easier to express directly in CUDA, especially if you can benefit from synchronizations and shared memory

NVIDIA HPC SDK

Available at developer.nvidia.com/hpc-sdk, on NGC, via Spack, preinstalled at CSCS



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

How to debug and profile the GPUs?

Debugging the GPU code

- You can use **printf** in the device code. This is all you need 😈
- Honestly, printf is really valuable when used properly:
  - Each thread can do a printf. To avoid overwhelming output, only print the data from a few threads (use if conditions)
  - Keep in mind that the order of printed lines is not specified, especially when the line come from different thread blocks. Always add an ID that can correlate work with its output
  - printf is slow and it may affect the execution flow. This may mask or exacerbate synchronization bugs
- **compute-sanitizer** is ideal to track illegal and uninitialized accesses and find race conditions.
- **cuda-gdb** is helpful, especially when you know that a particular kernel is faulty and when you can run your application with a single rank (i.e., no MPI parallelization)
- **Linaro Forge** (former Allinea DDT) can help with debugging at scale. Preinstalled at CSCS

Profiling the GPUs

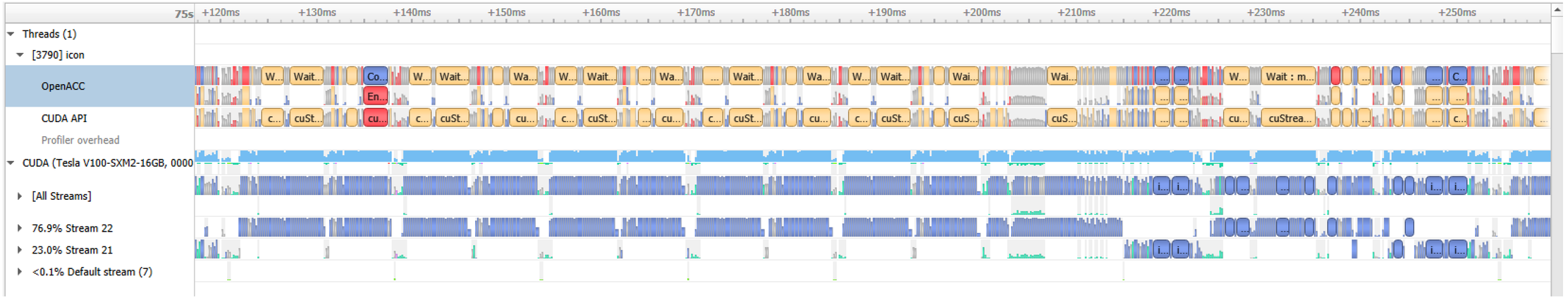
- There are two tools available: **Nsight Systems** and **Nsight Compute**
- Nsight Systems shows you an overview of the program as whole. It has a timeline with kernels running, API calls, CPU samples, etc. etc. Always start with Nsight Systems
- Nsight Compute is designed to help you optimize individual kernels. Can be a bit overwhelming, so go through On-Demand tutorials and demos
  - What I find very important is to check kernel memory traffic and FLOPs against expectations, probably roofline analysis
  - Then, warp stall reasons and occupancy calculator
  - And moreover, check the most sampled instructions in the Source tab
- Both tools are available in HPC SDK are require no special compilation. Reports can be viewed as text on the target machine or saved in a file. The file can be nicely visualized on your own computer

The Tools

GPU tools are good, use them!

- My main point: the tools are so easy to use, give them a try! Do not be afraid of complexity, the workflow is actually not complex at all
- Some of the tool reports could be hard to interpret, but it’s still much better that no reports at all

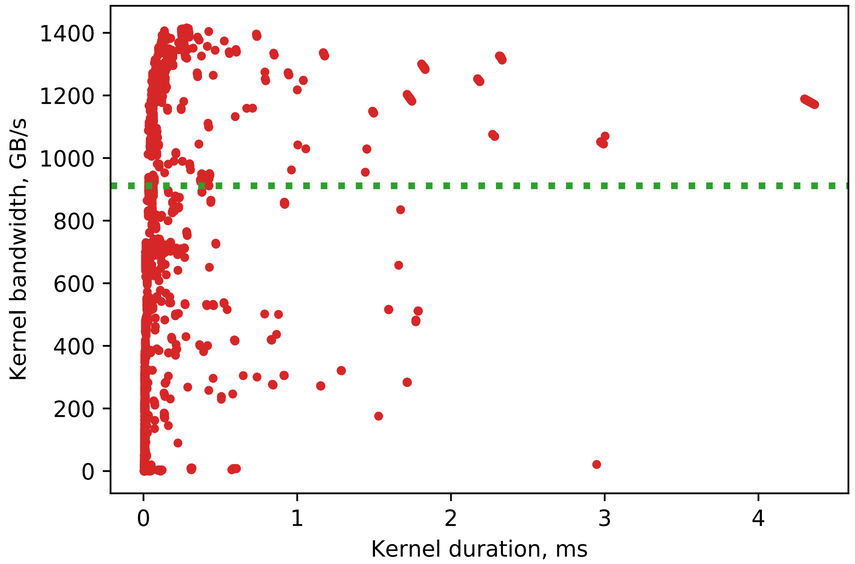
Nsight Systems example



- If we have time, I'll show a quick Nsys demo

ICON automated performance analysis

- I’m using Nsight Compute to extract bandwidth achieved by each kernel
- Required to perform detailed analysis of kernels bottlenecks
- Can identify kernels that yield low memory throughput or low overall GPU load
- Analysis can be easily automated with simple Python scripts



This kernel needs to improve

Kernel	Duration, ms	Bandwidth, GB/s
adding_1017_gpu	4.319616	1069.032674
lw_solver_noscat_136_gpu	3.002656	1028.267723
o3_pl2ml_219_gpu	2.947744	21.056265
inc_2stream_by_2stream_bybnd_576_gpu	2.325728	1041.890441
sw_two_stream_838_gpu	2.178112	1247.553819
lw_transport_noscat_576_gpu	1.826496	1328.899786
gas_optical_depths_major_339_gpu	1.788000	501.779672
inc_1scalar_by_1scalar_bybnd_456_gpu	1.771296	436.706152
compute_planck_source_578_gpu	1.716992	274.720473
sw_source_2str_942_gpu	1.714656	1253.359748

Grace Hopper and unified memory

NVIDIA GH200 Grace Hopper Superchip

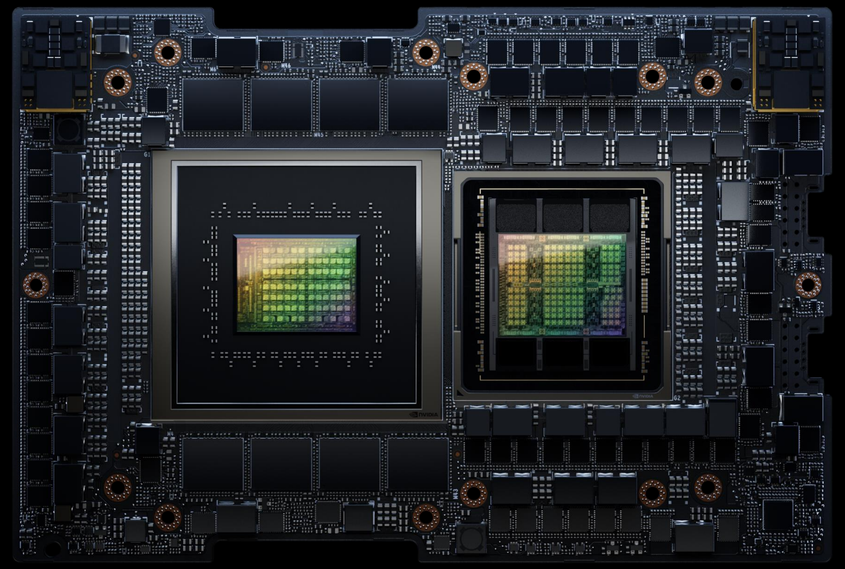
Built for the New Era of Accelerated Computing and Generative AI

Most versatile compute
Best performance across CPU, GPU or memory intensive applications

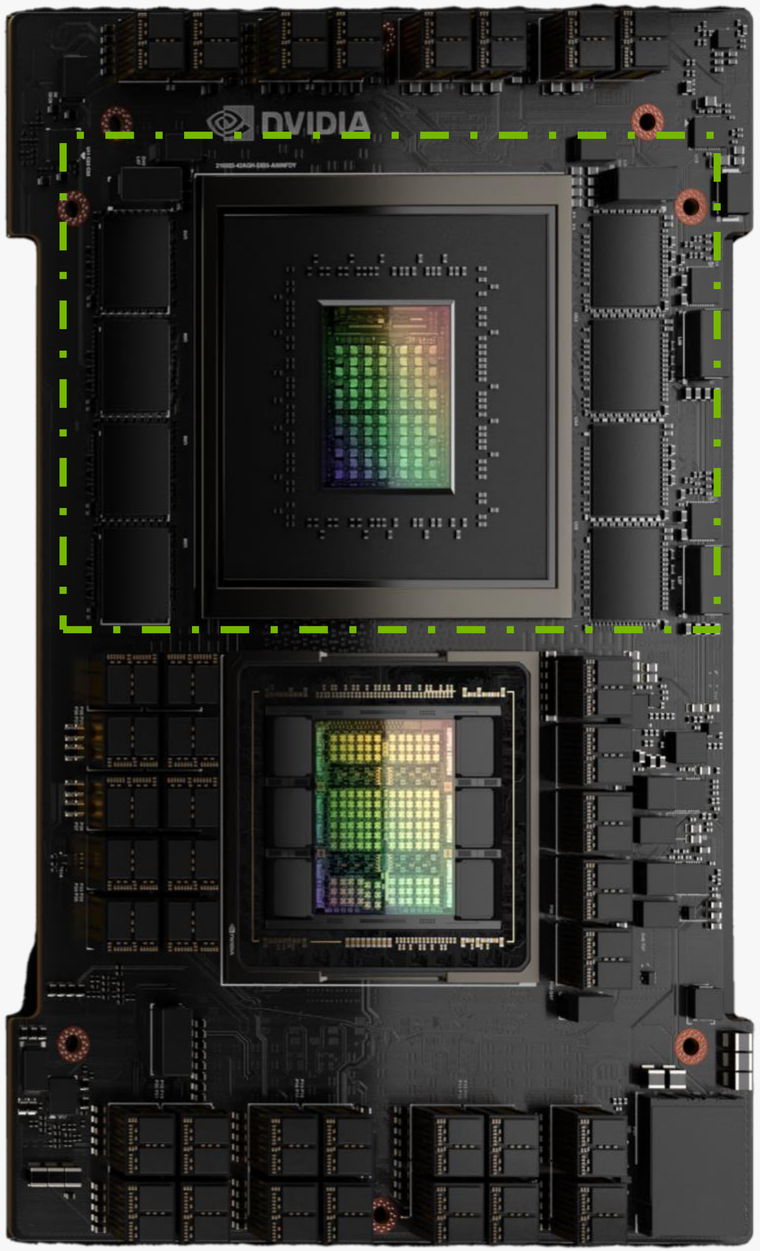
Easy to deploy and scale out
1 CPU:1 GPU node simple to manage and schedule for for HPC, enterprise, and cloud

Best Perf/TCO for diverse workloads
Maximize data center utilization and power efficiency

Continued Innovation
Grace and Hopper-Next in 2024



900GB/s NVLink-C2C | 576GB High-Speed Memory
4 PF AI Perf | 72 Arm Cores



NVIDIA Grace Hopper Superchip

"super" - more than a "chip"

NVIDIA CPU + NVIDIA GPU w/o compromises

- **NVIDIA Grace CPU**
  - 72 Arm v9 Neoverse V2 CPU cores with SVE2.
    - → Efficiency: 62pJ/DFMA (x86: ~99); 1.6x more efficient
    - → Throughput: 3.6 TFLOP/s, 3.2 TB/s bisection bandwidth
  - Memory:
    - → High capacity: ≤ 480 GB LPDDR5X (5pJ/bit vs 36 DDR)
    - → Reasonable bandwidth: ≤ 500 GB/s
    - → Very low latency

NVIDIA Grace Hopper Superchip

"super" - more than a "chip"

NVIDIA CPU + NVIDIA GPU w/o compromises

- **NVIDIA Grace CPU**
  - 72 Arm v9 Neoverse V2 CPU cores with SVE2.
    - → Efficiency: 62pJ/DFMA (x86: ~99); 1.6x more efficient
    - → Throughput: 3.6 TFLOP/s
  - Memory:
    - → High capacity: ≤ 480 GB LPDDR5X (5pJ/bit vs 36 DDR)
    - → Reasonable bandwidth: ≤ 500 GB/s
    - → Very low latency
- **NVIDIA Hopper GPU**
  - → High throughput: 67 TFLOP/s
  - Memory:
    - → Low capacity: 96 GB HBM3
    - → High latency
    - → Extreme bandwidth ≤ 4000 GB/s
  - ≤ 18x NVLink 4 → 900 GB/s and up to 256 GPUs.

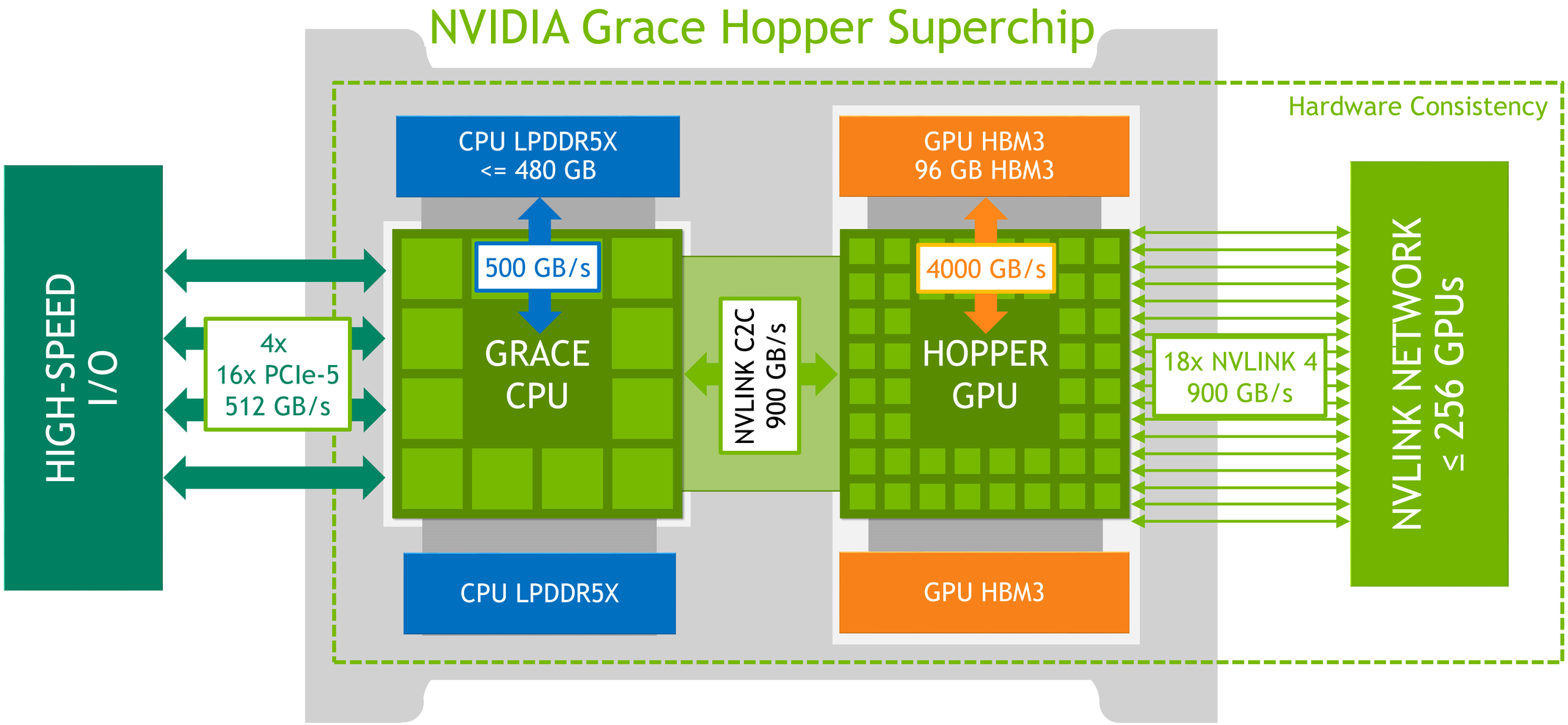
NVIDIA Grace Hopper Superchip

Soul is the new **NVLink-C2C** CPU ←→ GPU interconnect

- **Memory consistency**: ease of use
  - →*All* threads – GPU and CPU – access system memory: C++ `new`, `malloc`, `mmap`'ed files, atomics, …
  - → Fast automatic page migrations HBM3←→LPDDR5X.
  - → Threads cache peer memory → Less migrations.
- **High-bandwidth**: 900 GB/s (same as peer NVLink 4)
  - → GPU reads or writes local/peer LPDDR5X at ~peak BW
- **Low-latency**: GPU→HBM latency
  - →GPU reads or writes LPDDR5X at ~HBM3 latency

Grace Hopper Superchip

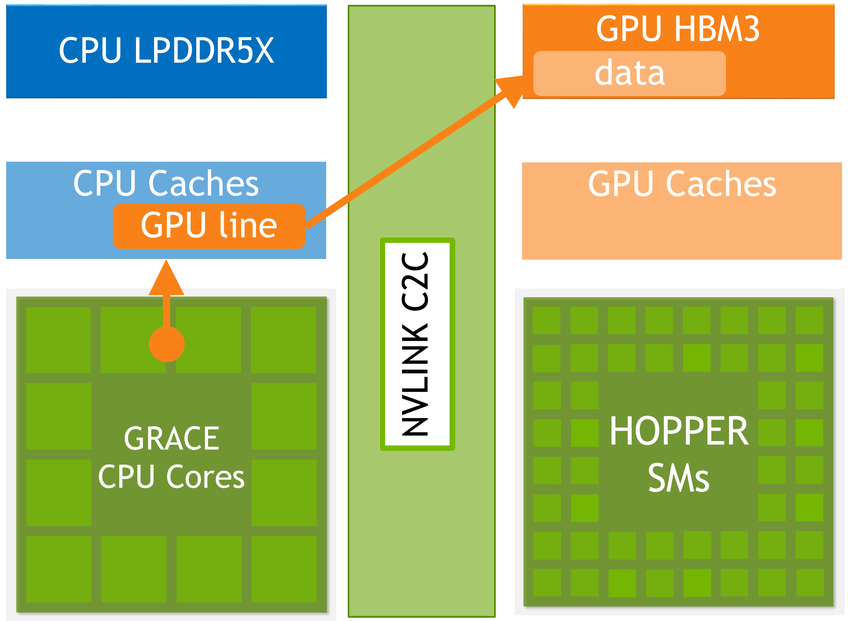
GPU can access CPU memory at CPU memory speeds



https://resources.nvidia.com/en-us-grace-cpu/nvidia-grace-hopper

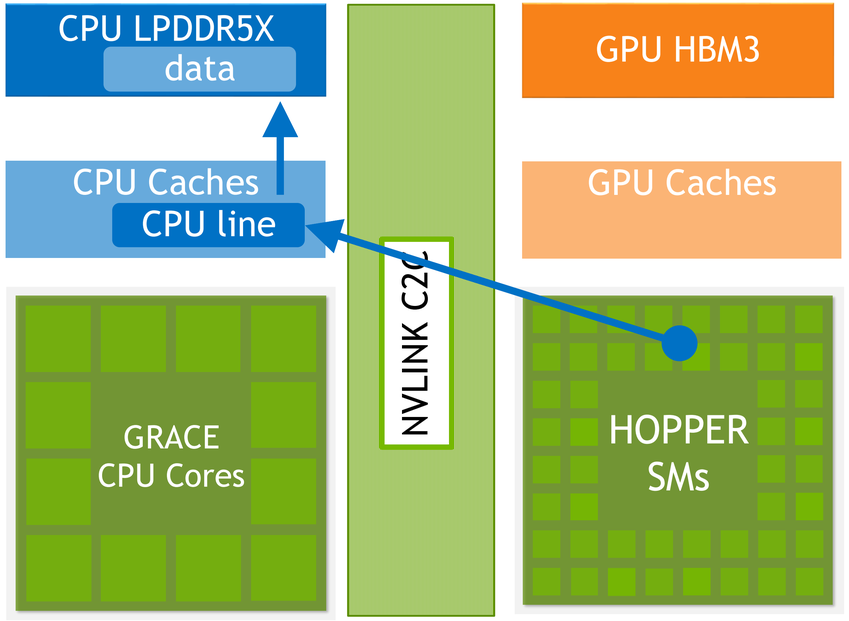
Global Access to All Data

Cache-coherent access via NVLink C2C from either processor to either physical memory



Grace directly reading Hopper's memory

CPU fetches GPU data into CPU L3 cache
Cache remains **coherent** with GPU memory
Changes to GPU memory **evict** cache line



Hopper directly reading Grace's memory

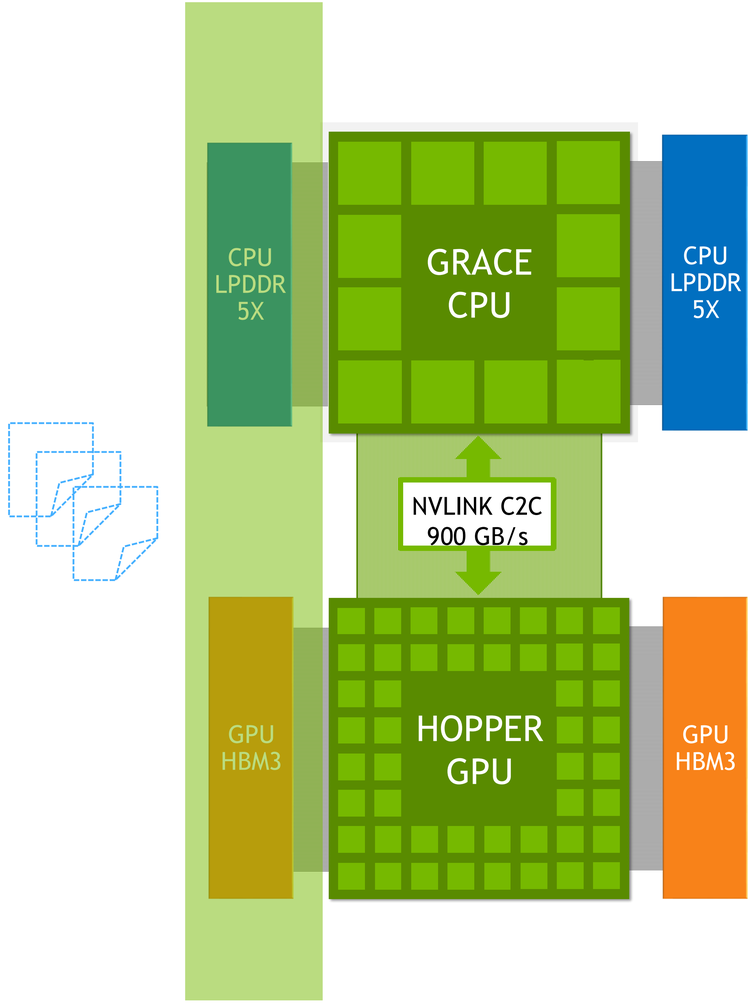
GPU loads CPU data via CPU L3 cache
CPU and GPU **can both hit** on cached data
Changes to CPU memory **update** cache line

System allocator

Easy to move codes to GPU. Uses Address Translation Service (ATS)

```
__global__ void kernel(int *data1, int *data2)
{
  data1[threadIdx.x] = threadIdx.x;
  data2[threadIdx.x] = threadIdx.x;
}

int main()
{
  int *data = (int *)malloc(128 * sizeof(int));
  int data2[128];

  kernel<<<1, 128>>>(data, data2);
  cudaDeviceSynchronize();

  for (int i = 0; i < 128; i++)
  {
    printf("(%d, %d)  ", data[i], data2[i]);
  }

  free(data);
  return 0;
}
```

Physical pages are not yet allocated

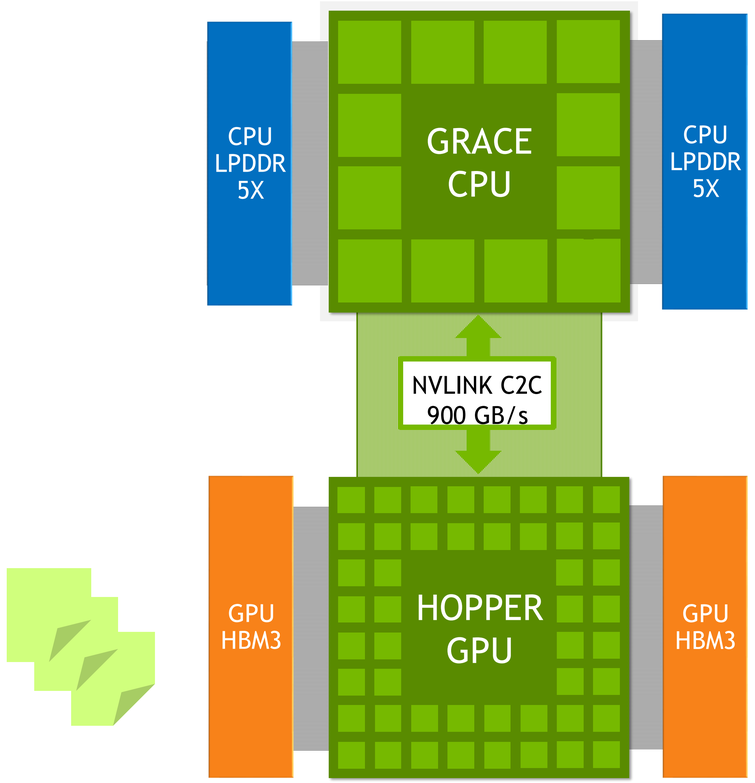


System allocator

Easy to move codes to GPU. Uses Address Translation Service (ATS)

```
__global__ void kernel(int *data1, int *data2)
{
  data1[threadIdx.x] = threadIdx.x;
  data2[threadIdx.x] = threadIdx.x;
}

int main()
{
  int *data = (int *)malloc(128 * sizeof(int));

  int data2[128];

  kernel<<<1, 128>>>(data, data2);
  cudaDeviceSynchronize();

  for (int i = 0; i < 128; i++)
  {
    printf("(%d, %d)  ", data[i], data2[i]);
  }

  free(data);
  return 0;
}
```

Physical pages are allocated on first touch

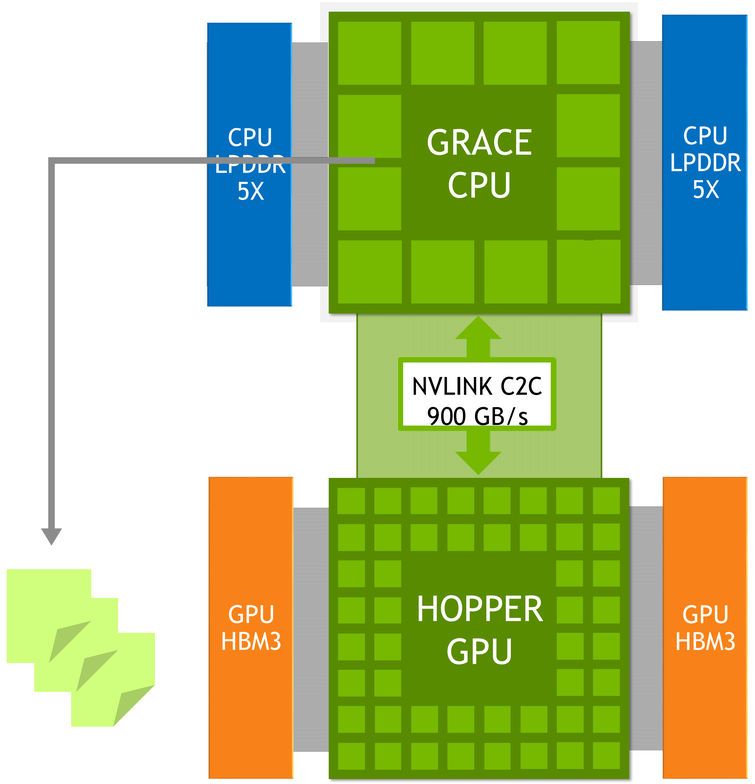


System allocator

Easy to move codes to GPU. Uses Address Translation Service (ATS)

```
__global__ void kernel(int *data1, int *data2)
{
  data1[threadIdx.x] = threadIdx.x;
  data2[threadIdx.x] = threadIdx.x;
}

int main()
{
  int *data = (int *)malloc(128 * sizeof(int));

  int data2[128];

  kernel<<<1, 128>>>(data, data2);
  cudaDeviceSynchronize();

  for (int i = 0; i < 128; i++)
  {
    printf("(%d, %d)  ", data[i], data2[i]);
  }

  free(data);
  return 0;
}
```

Direct access from CPU to GPU data



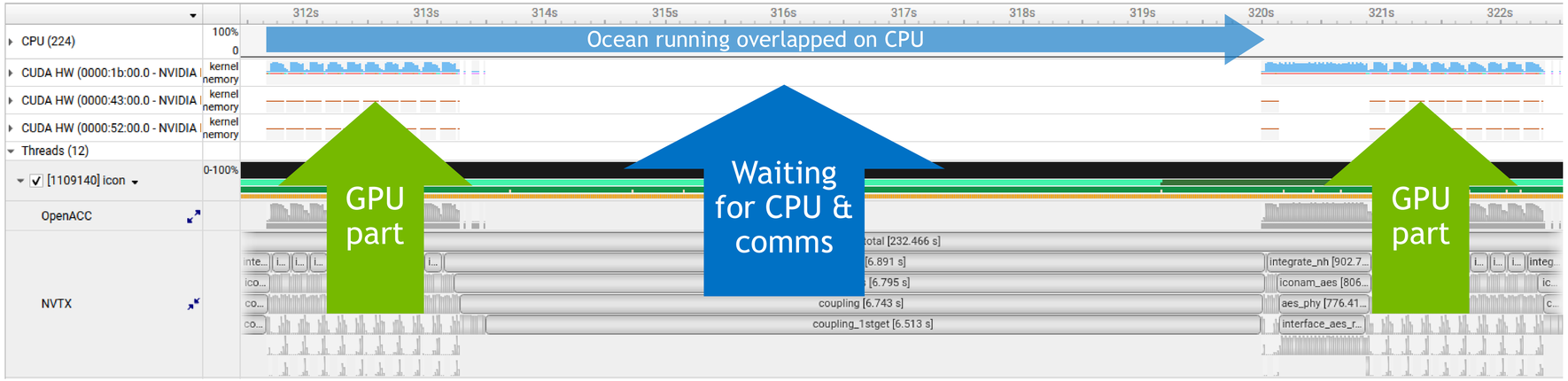
ICON Coupled Ocean

Profile

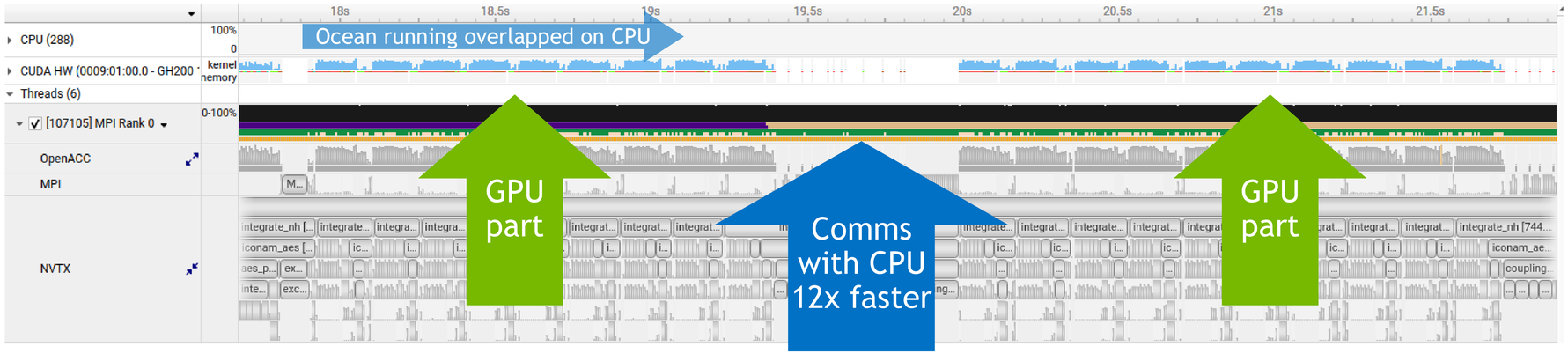
- Full globe coupled simulation at 10 km atmosphere resolution and 5 km ocean resolution. 90 vertical atmosphere layers, 72 vertical ocean layers. Atmosphere time-step is 90s, ocean time-step is 5 min and coupling time-step is 15 min. Atmosphere and ocean run in different ranks within the same MPI job. 64 GPUs and 512 (Eos) or 3008 (Alps) CPU ranks

Profiling atmosphere GPU process

Eos



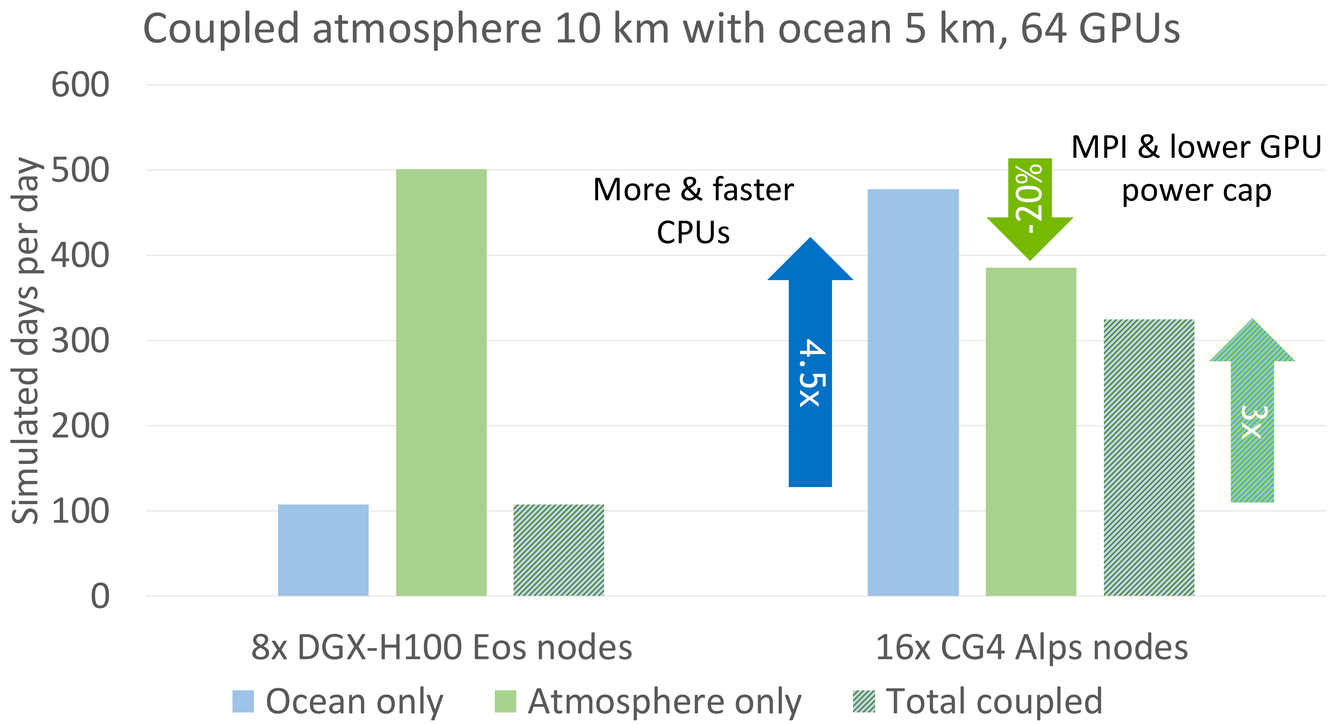
Alps



ICON Coupled Ocean

Grace-Hopper: 3x speedup

- On Eos:
  - Performance limited by Ocean running on the CPU
- On Alps:
  - Unleash full performance of Hopper GPUs
  - Grace is powerful enough to run the ocean in the background
  - Alps network is still in bring-up phase, which introduces some atmosphere-only and coupling overhead
  - 3x end-to-end performance

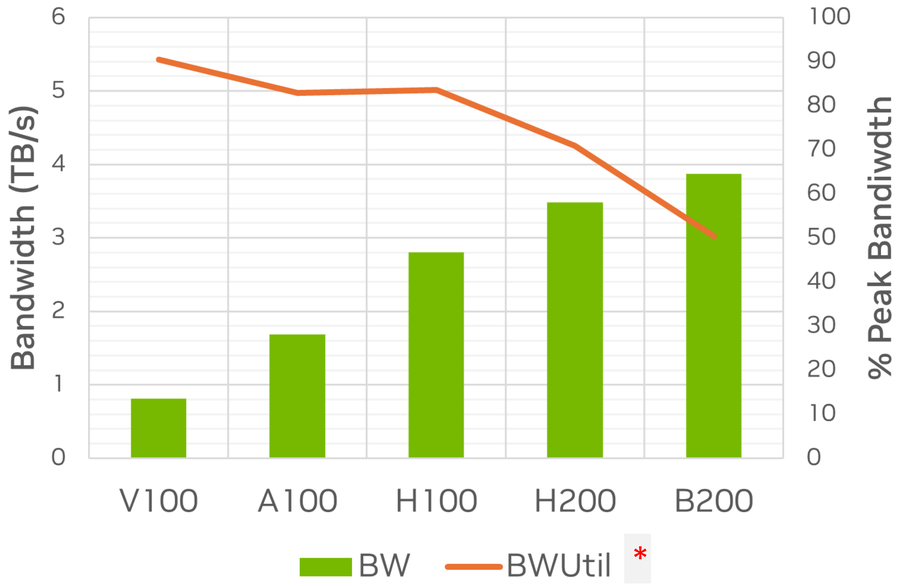


See also: Thomas Schulthess, CSCS – S62157

Bytes-in-flight and async copies

Simple kernels do not saturate bandwidth anymore!

```
__global__ void kernel(float *a, float *b, float *c)
{
  int i = blockIdx.x * blockDim.x + threadIdx.x;
  c[i] = a[i] + b[i];
}
```



* Any benchmark numbers provided for technical discussion only

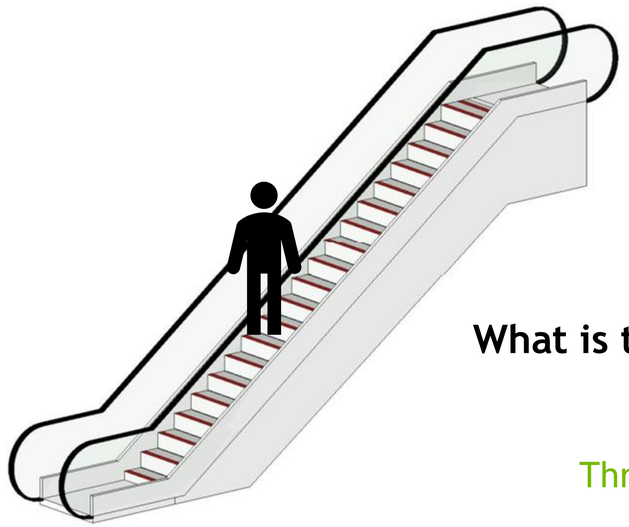
Little's Law

For escalators

mean number of units in a system = mean arrival rate * mean residency time

Our escalator specification:

- 1 person per step
- 20 steps tall
- A step arrives every 2 seconds
  - **Peak arrival rate** = 0.5 person/s
  - **Residency time** = 40 seconds



What is the achieved throughput with 1 person in-flight?

Throughput = # Persons / Residency time
= 0.025 person/s

Little's Law

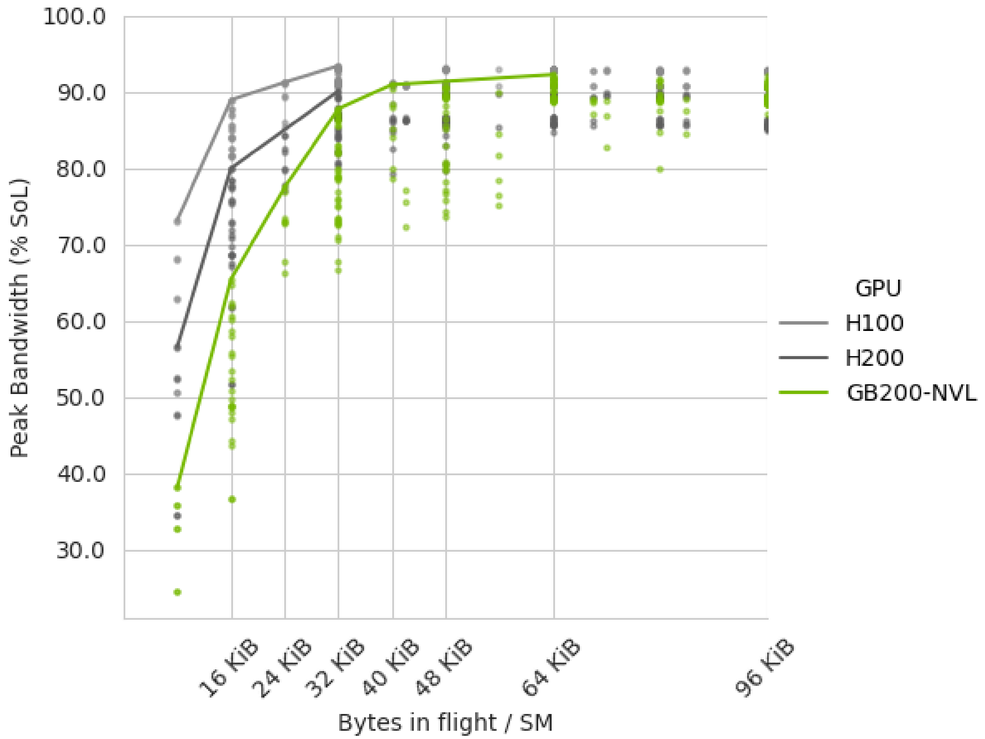
For GPU memory

bytes-in-flight = bandwidth * mean latency

controlled by software — determined by hardware

- Required bytes-in-flight to saturate DRAM bandwidth increases with every generation!
  - Mainly due to bandwidth increase.
  - ~2x moving from Hopper to Blackwell.
- However, bandwidth per SM also increases!
- Need more bytes-in-flight per SM to saturate bandwidth!

Adjust software 😇



* Any benchmark numbers provided for technical discussion only

* Dots in the plot represent STREAM-like kernels with varying operations, thread block dimensions, data types and number of parallel loads.

Little's Law

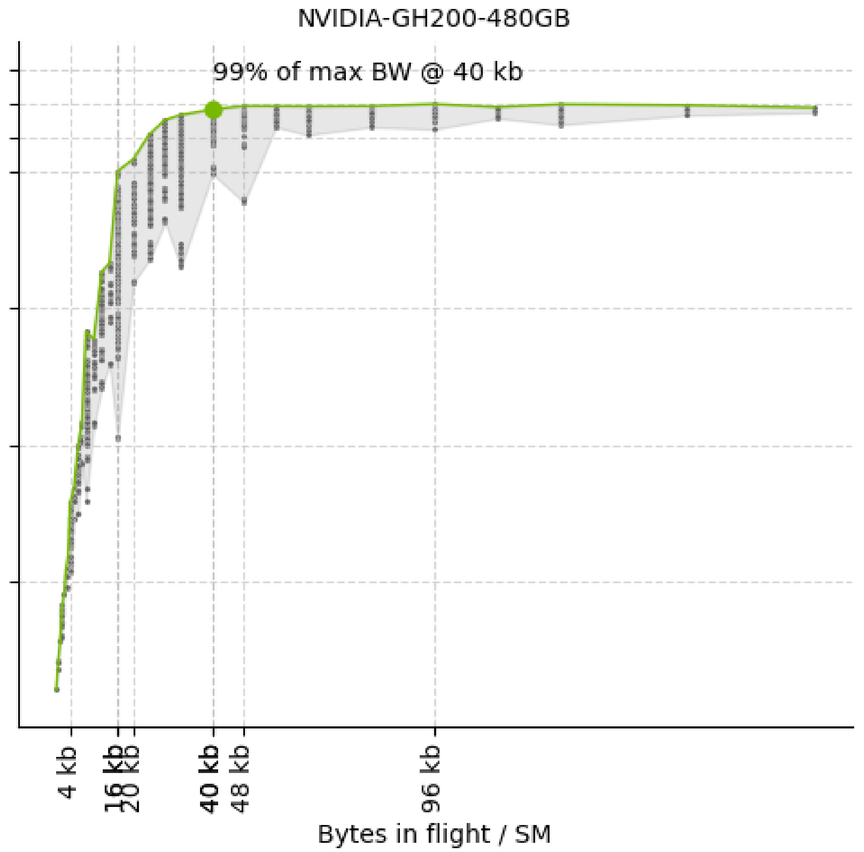
For GPU memory

bytes-in-flight = bandwidth * mean latency

controlled by software — determined by hardware

- Required bytes-in-flight to saturate DRAM bandwidth increases with every generation!
  - Mainly due to bandwidth increase.
  - ~2x moving from Hopper to Blackwell.
- However, bandwidth per SM also increases!
- Need more bytes-in-flight per SM to saturate bandwidth!

Adjust software 😇



* Any benchmark numbers provided for technical discussion only

* Dots in the plot represent STREAM-like kernels with varying operations, thread block dimensions, data types and number of parallel loads.

Little's Law

Can we saturate memory bandwidth with simple kernels?

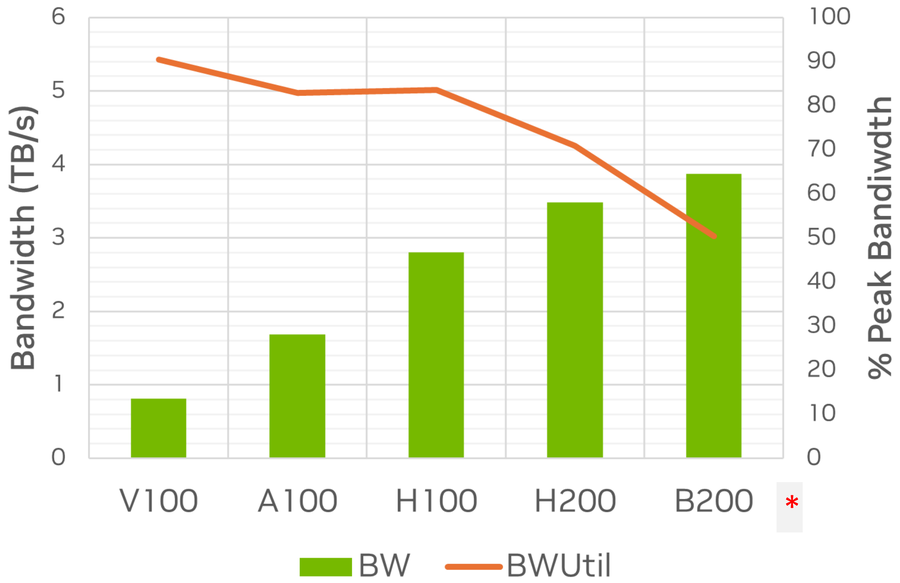
```
__global__ void kernel(float *a, float *b, float *c)
{
  int i = blockIdx.x * blockDim.x + threadIdx.x;
  c[i] = a[i] + b[i];
}
```

- Can we generate enough bytes-in-flight?

estimated bytes-in-flight / SM = # loads / thread * # bytes / load * # threads / block * # blocks / SM

= 2 * 4 * 256 * 8 = 16 KiB

100% occupancy



* Any benchmark numbers provided for technical discussion only

Little's Law

Can we saturate memory bandwidth with simple kernels?

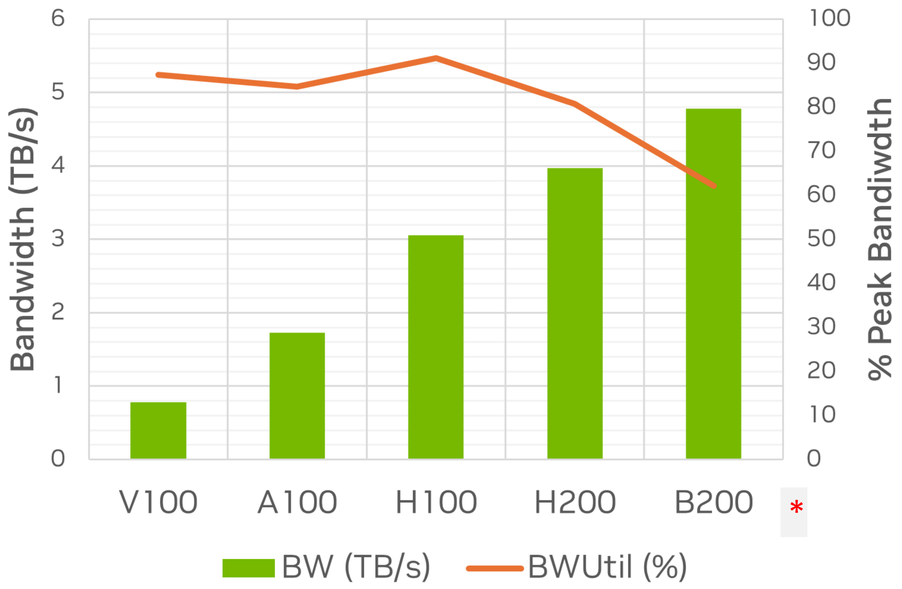
```
__global__ void kernel(float *a, float *b, float *c, float *d)
{
  int i = blockIdx.x * blockDim.x + threadIdx.x;
  d[i] = a[i] + b[i] + c[i];
}
```

- Can we generate enough bytes-in-flight?

**estimated bytes-in-flight / SM = # loads / thread *
bytes / load *
blocks / SM *
threads / block**

= 3 * 4 * 8 * 256 = 24 KiB

100% occupancy



* Any benchmark numbers provided for technical discussion only

Async Bulk Loads

Example using cuda::memcpy_async

- Completion mechanism: shared memory barrier

INIT

```
#include <cuda/barrier>;
__shared__ cuda::barrier<cuda::thread_scope_block> bar;
if (threadIdx.x == 0)
{
  init(&bar, blockDim.x);
}
__syncthreads();
```

Launch using a single thread.

FIRE

```
if (threadIdx.x == 0)
{
  cuda::memcpy_async(smem_a, a + idx, cuda::aligned_size_t<16>(num_bytes), bar);
  cuda::memcpy_async(smem_b, b + idx, cuda::aligned_size_t<16>(num_bytes), bar);
}
```

Uses TMA if src/dest are 16-byte aligned and size is a multiple of 16, otherwise it falls back to synchronous copies.

WAIT FOR COMPLETION

```
barrier::arrival_token token = bar.arrive();
bar.wait(cuda::std::move(token));

// alternatively:
// bar.arrive_and_wait();
```

Extra resources

- Nvidia on-demand: https://www.nvidia.com/en-us/on-demand/
- GTC recordings: https://www.nvidia.com/gtc/session-catalog/?regcode=no-ncid&ncid=no-ncid&search=#/
  - https://resources.nvidia.com/en-us-summer-of-learning-for-students/gtcspring22-s41496?ncid=no-ncid
  - https://www.nvidia.com/en-us/on-demand/session/gtc24-s62191/?playlistId=playList-d59c3dc3-9e5a-404d-8725-4b567f4dfe77
  - https://www.nvidia.com/en-us/on-demand/session/gtcspring23-s51120/
  - CUDA Techniques to Maximize Memory Bandwidth and Hide Latency S72683 | GTC 2025 | NVIDIA On-Demand

- Be careful when looking at old stuff like GPU Gems or 10-year-old stackoverflow answers. A lot has changed in CUDA platform and a lot more is possible now
- The best way to learn is to try, so get your hands dirty!