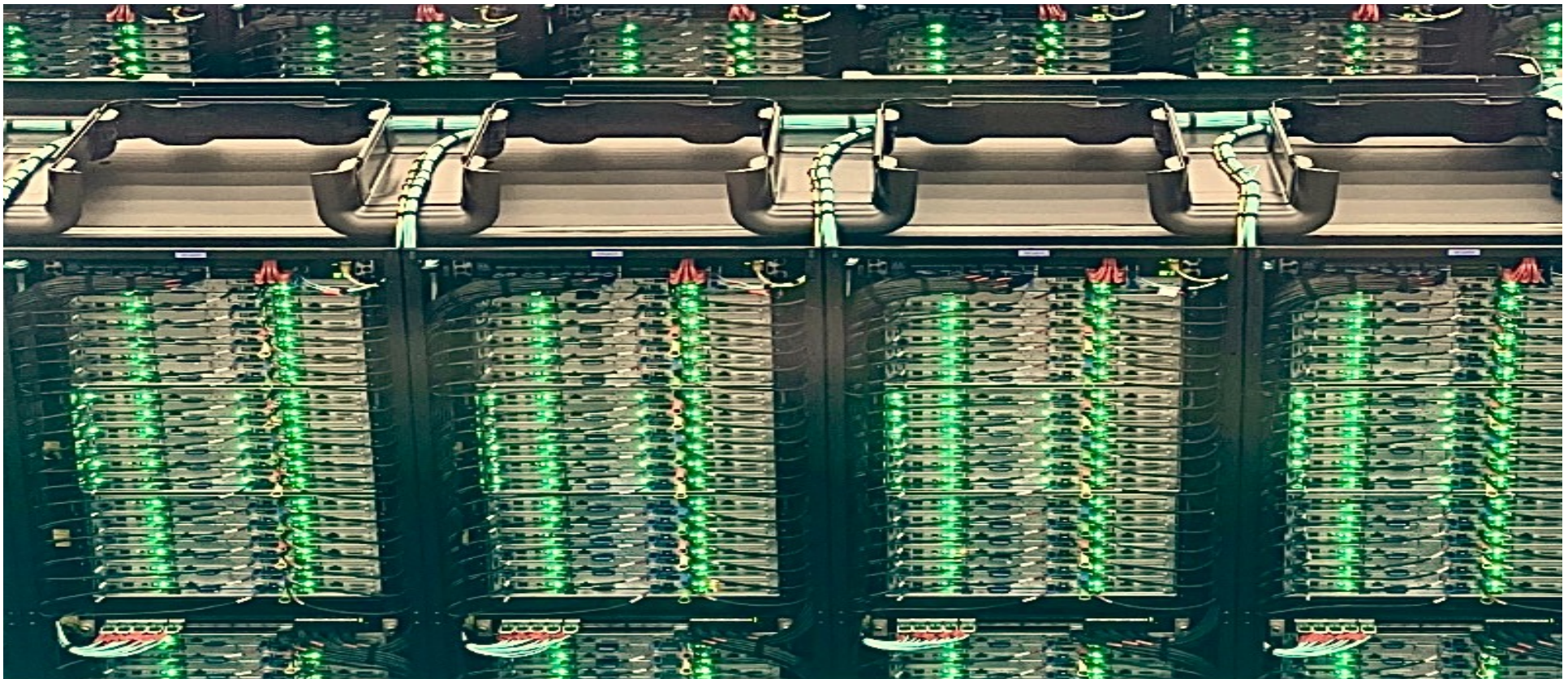


06. Distributed GPU Programming

Supercomputing for Artificial Intelligence
Foundations, Architectures, and Scaling Deep Learning Workloads

Jordi **TORRES.AI**



Content

6.1 H100: A Platform for Parallel and Distributed Computation

Streaming Multiprocessors (SMs)

Understanding the CUDA Execution Model: Threads, Warps and Blocks

Tensor Cores: Accelerating Matrix Operations for AI

HBM3: Ultra-Fast On-Board Memory

6.2 High-Speed Communication for Multi-GPU Systems

Inter-GPU Communication Challenges in Modern AI Supercomputing

NVLink: Fast GPU-to-GPU Communication

PCI Express Gen 5: Faster CPU-GPU Communication

Networking for Distributed GPU Systems: RDMA and GPUDirect

Other Interconnect Technologies and Communication Protocols

6.3 Distributed Computing with CUDA-aware MPI

CUDA-aware MPI

How does CUDA-aware MPI work?

6.4 NCCL: Collective Communication for GPUs

Limitations of MPI for GPU-Accelerated Workloads

Design Principles of NCCL

Other Software Level Communication Libraries

6.5 Case Study: Distributed GPU Computing

Jacobi Algorithm

Running the Jacobi Code

Performance Benchmarking and Scalability Analysis



6.1

H100: A Platform for Parallel and Distributed Computation

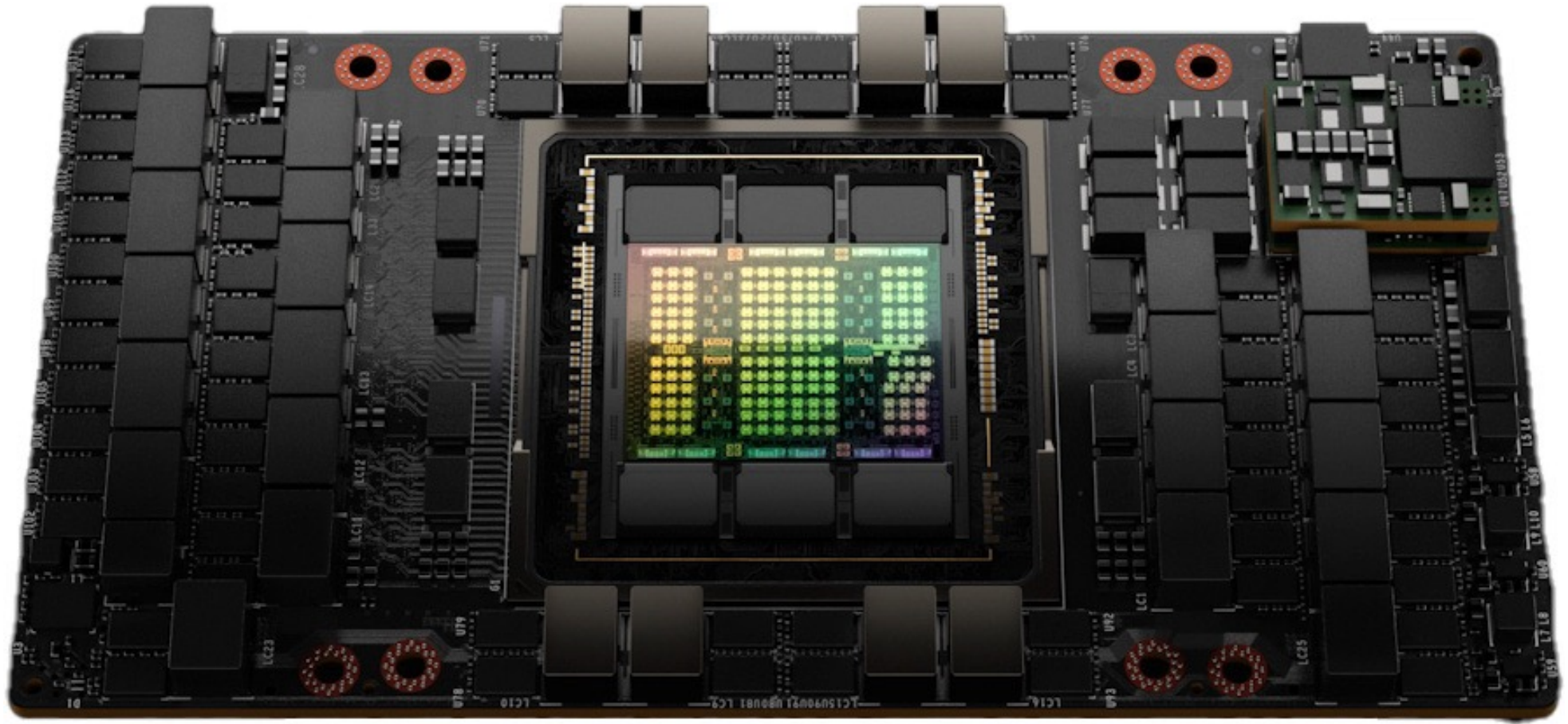
H100: A Platform for Parallel and Distributed Computation

- More than “a faster GPU”: a *data center processor* designed for large-scale AI and HPC.
- **Key architectural goals:**
 - Massive parallel execution
 - High-throughput memory access
 - Scalable multi-GPU communication
- **Enables distributed AI training with NVLink, HBM3, and Tensor Cores.**

H100: A Platform for Parallel and Distributed Computation

- Figure 6.1 – NVIDIA H100 GPU

(source: NVIDIA GTC22 Whitepaper – Hopper).



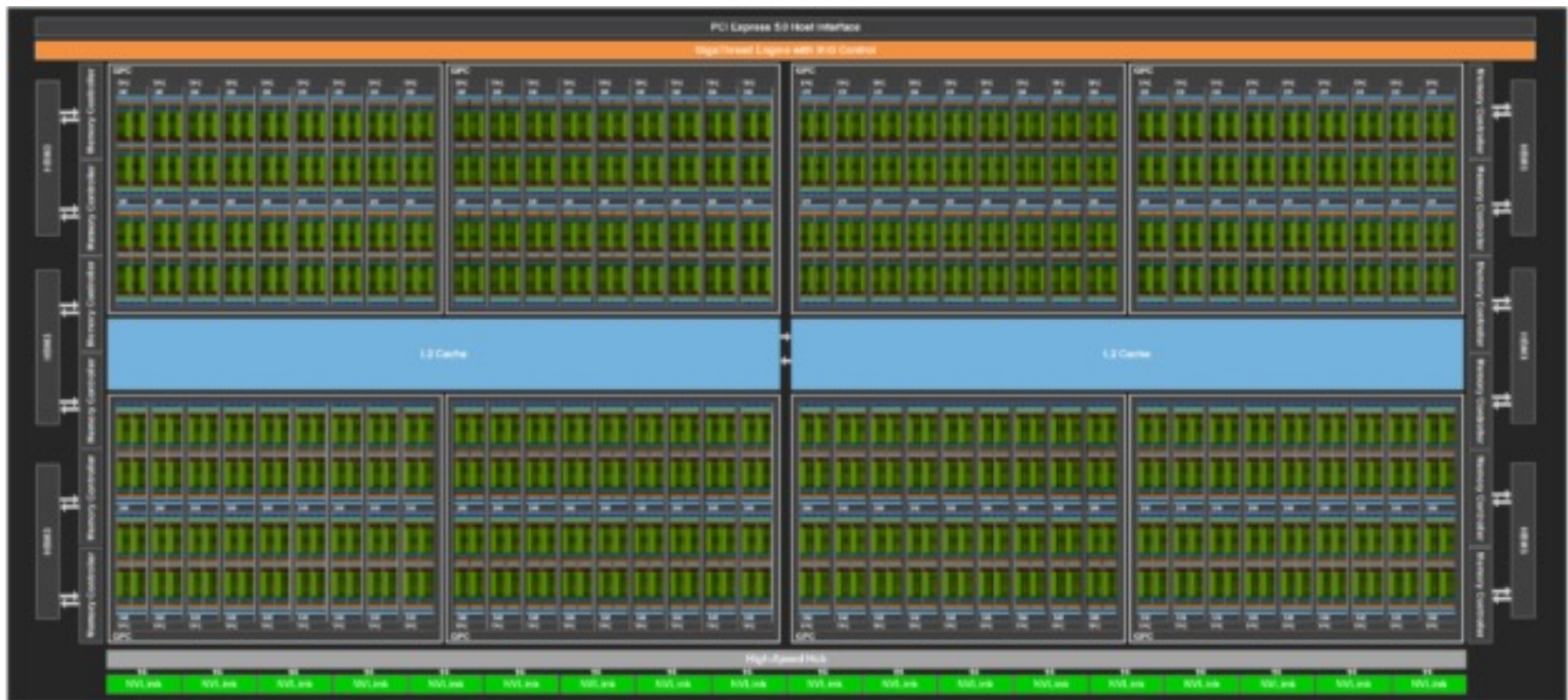
Streaming Multiprocessors (SMs): the heart of the GPU

- **Streaming Multiprocessors (SMs)**
 - SM = *self-contained compute unit* (arithmetic, memory, scheduling).
 - Each SM executes thousands of threads in parallel.
 - H100 contains 144 SMs (vs 108 SMs in A100).
 - Central L2 cache shared by all SMs reduces global memory traffic.
 - HBM memory controllers deliver extremely high bandwidth.
 - NVLink interfaces connect GPUs for large AI models.

Streaming Multiprocessors (SMs)

- Figure 6.2 – Floorplan of an H100 GPU showing 144 SMs and interconnect layout

(image source: NVIDIA)

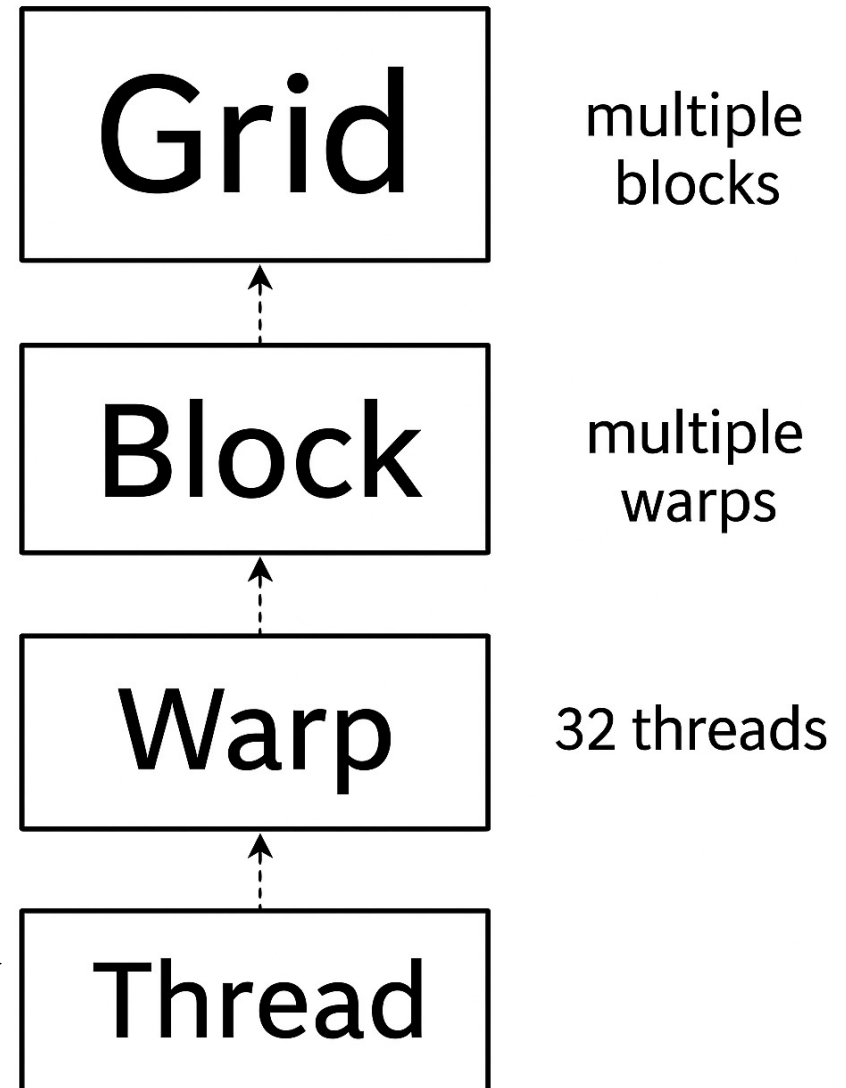


Inside a Streaming Multiprocessor

- **Functional View of an H100 SM**
 - **Each SM divided into 4 processing partitions:**
 - Each with its own dispatch unit, L0 instruction cache, and register file (16,384 × 32-bit registers).
 - Total of 65,536 registers per SM.
 - **Supports multiple warps (32 threads each) executing in parallel.**
 - **Hierarchical instruction caching:**
 - L0 cache per partition (very low latency)
 - L1 instruction cache shared across SM
 - **Execution units per partition:**
 - Arithmetic pipelines (FP32, FP64, INT32)
 - 4th-generation Tensor Cores (mixed precision)
 - Special Function Units (SFUs) for trig/exponential ops

Understanding the CUDA Execution Model

- CUDA provides a hierarchical execution model that abstracts hardware complexity.
- Enables developers to write scalable parallel code.
- Organizes threads into logical groups that map efficiently to GPU hardware.
- Thread execution hierarchy



Threads, Warps, and Blocks

■ Content:

- **Thread**: smallest unit of execution.
- **Warp**: group of 32 threads executing the same instruction in hardware.
- **Block**: collection of warps that share data via shared memory.
- **Grid**: full set of blocks launched for a kernel.

■ Key ideas:

- Programmers write thread and block code, but the hardware executes warps.
- Even partial warps (e.g., 20 threads) still consume 32-thread resources.
- Optimal performance aligns with warp boundaries.

■ **Warp = hardware scheduling unit**

→ understand it for performance tuning.

Logical vs Hardware Abstraction

- **IMPORTANT: Two Abstraction Levels in CUDA**

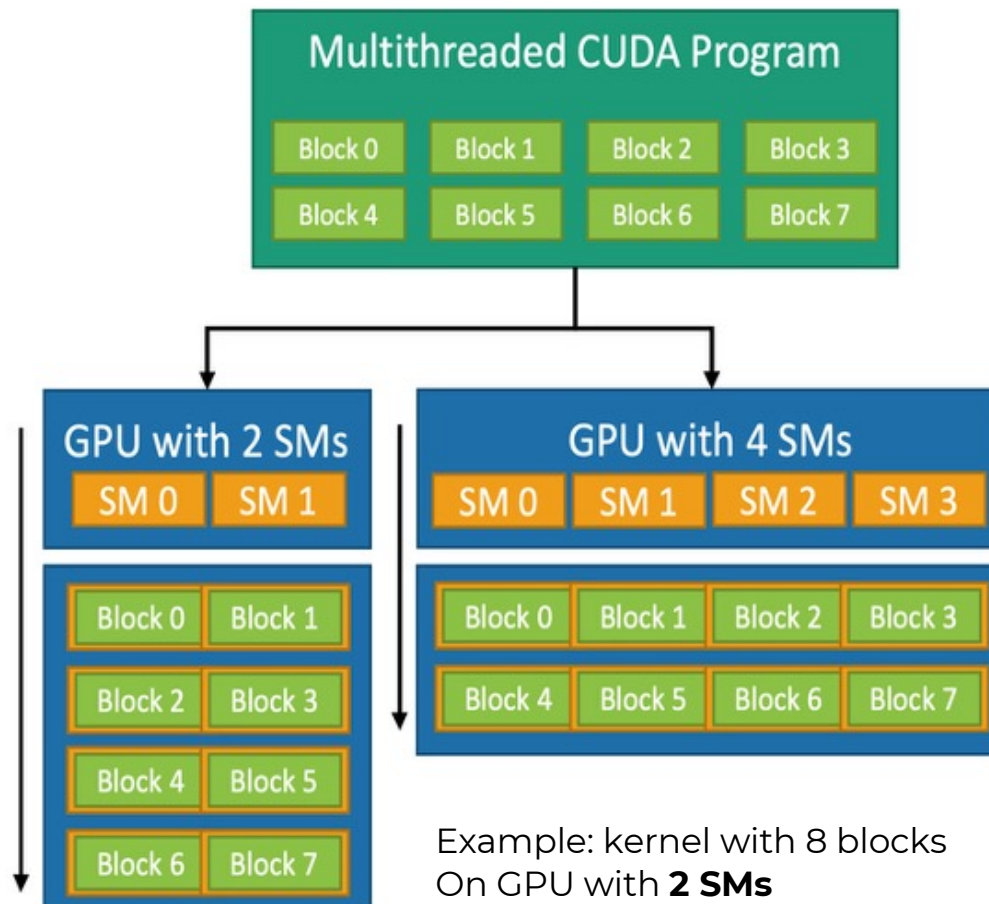
Logical Abstraction (Programmer view)	Hardware Abstraction (GPU view)
Grid, Block, Thread	Warp (32 threads)
Defined in code	Scheduled by SM hardware
Portable and scalable	Dependent on GPU architecture

- **Key takeaway:**

CUDA hides low-level scheduling — programmers focus on *what* to parallelize, not *how* threads are dispatched.

How CUDA Blocks Map to SMs

- Figure 6.5 – CUDA kernel blocks dynamically assigned to available SMs (source: NVIDIA)



Example: kernel with 8 blocks

On GPU with **2 SMs**

→ each SM executes 4 blocks sequentially.

On GPU with **4 SMs**

→ each SM executes 2 blocks → faster execution.

Task 6.1

■ Task 6.1 – Reflecting on CUDA’s Execution Model

We have explored how CUDA organizes the execution of threads through a hierarchical model of grids, blocks, warps, and threads. We have also seen how the CUDA runtime dynamically maps blocks to SMs, allowing programs to scale across different GPUs.

Reflect on the advantages of this model and explain, in your own words:

- Why can the same CUDA program run efficiently on GPUs with different numbers of SMs?
- How does the CUDA execution model simplify programming by hiding low-level scheduling details?

Tensor Cores: Accelerating Matrix Operations for AI

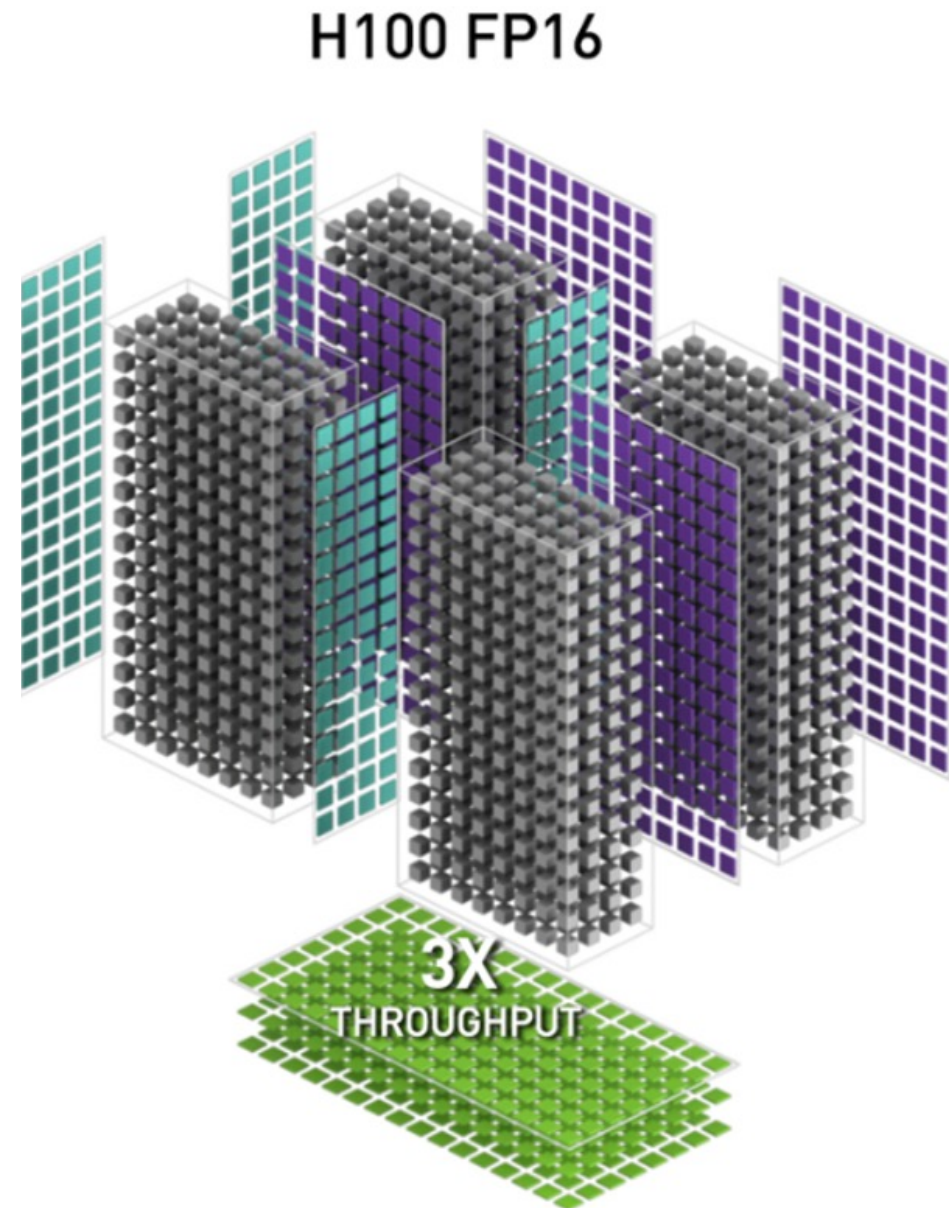
■ Tensor Cores

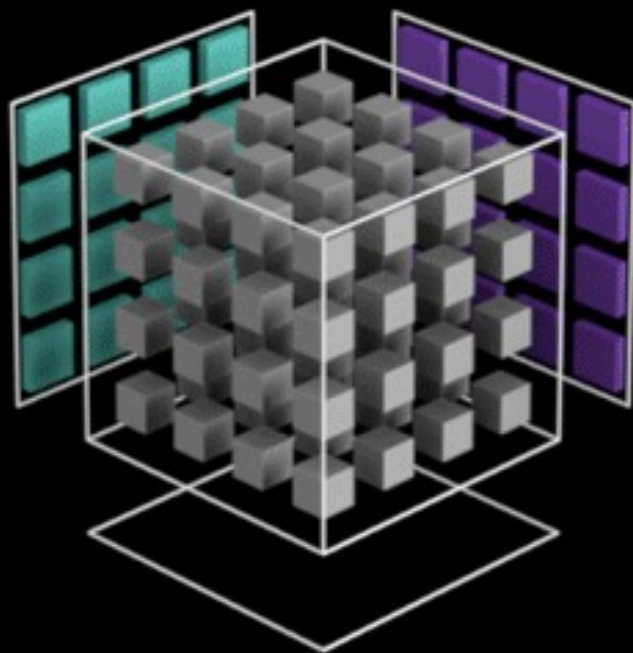
- One of the key innovations in modern NVIDIA GPUs.
- Specialized hardware for dense matrix operations, the core of deep learning workloads.
- Designed for massive parallelism and fused operations to boost throughput and reduce memory traffic.
- Operates on matrices as first-class data types (unlike scalar/vector CUDA cores).

$$D = A \times B + C$$

Tensor Cores: Accelerating Matrix Operations for AI

- **Figure 6.6 – Visual example of Tensor Cores (3D blocks) performing parallel FP16 operations,**
(achieving up to 3 × throughput over previous generations)
- (source: NVIDIA)





Supported Precision Formats

- **Precision Formats and Use Cases**

Format	Use Case
FP64	High-precision scientific computing
TF32	FP32 range + FP16 performance (default for AI training)
FP16 / BF16	Training deep neural networks with reduced memory use
INT8 / INT4	Quantized inference for high-speed deployment
FP8	New in H100: efficient training/inference for large models

- **Different precisions allow trading accuracy vs. performance depending on the workload.**

Using Tensor Cores

■ Implicit (most developers):

- Tensor Cores are used automatically by libraries/frameworks.
- No CUDA-specific coding needed.
- Examples:
 - **cuBLAS**: GEMM operations on FP16/BF16/TF32.
 - **cuDNN**: convolution and RNN primitives.
 - **NCCL**: collective ops across GPUs.
 - **PyTorch / TensorFlow / JAX**: use them transparently.

■ Explicit (advanced users):

- Use CUDA C++ APIs such as WMMA (Warp Matrix Multiply-Accumulate).
- Or write custom kernels with CUTLASS.
- Enables fine control of memory layout, tile size, and fusion.

Task 6.2

■ Task 6.2 – Precision Trade-offs: True or False?

Modern GPUs support reduced-precision formats such as FP16 or BF16. Choosing lower precision affects speed, memory, and numerical accuracy. Indicate whether each statement is True (T) or False (F) and explain briefly:

1. Using FP16 or BF16 can help a program run faster on GPUs that support Tensor Cores.
2. Switching from FP32 to FP16 always produces identical numerical results.
3. Reducing precision can lower the amount of memory required for computations.
4. All calculations in a GPU program must use the same precision format.
5. Developers can benefit from Tensor Cores without writing any specialized GPU code.

HBM3: Ultra-Fast On-Board Memory


- **HBM3 = *High Bandwidth Memory (version 3)*.**
 - Designed for massive parallel workloads like AI training and HPC.
 - Provides extremely high throughput and energy efficiency.
 - Integrated *on-package*, directly next to the GPU die → minimal latency.
- **Traditional DRAM is far → HBM3 is “on top” of the GPU.**

Bandwidth Comparison

■ Why HBM3 Matters

- H100 GPU:
 - Up to 6 HBM3 stacks.
 - Total bandwidth: >3 TB/s.
- Modern CPU (e.g., Intel Xeon):
 - DDR5 bandwidth: 100–300 GB/s per socket.
- Result:
 - 10 × –30 × higher bandwidth on GPU vs CPU.

■ HBM3 sustains the data flow needed by thousands of concurrent threads in the H100's 144 SMs.



6.2 High-Speed Communication for Multi-GPU Systems

NVLink: GPU-to-GPU Interconnect

■ NVLink: Fast GPU-to-GPU Communication

- Developed by NVIDIA to bypass CPU and system memory.
- Direct GPU-to-GPU transfers at very high bandwidth.
- H100 specs:
 - 18 NVLink4 connections
 - 50 GB/s per direction
 - → 900 GB/s one way / 1.8 TB/s bidirectional
- GPUs in MN5 nodes are fully interconnected via NVLink.

■ Advantages:

- Lower latency (no CPU or DRAM path)
- Higher bandwidth (up to 10 × PCIe)
- Lower CPU overhead

■ NVLink = dedicated GPU mesh

→ accelerates gradient synchronization.

PCI Express Gen 5

■ PCI Express Gen 5: CPU–GPU Communication

- Main interface between host (CPU) and device (GPU).
- 64 GB/s per direction → 128 GB/s bidirectional.
- Used for:
 - CPU–GPU data transfers.
 - GPU-to-GPU (if no NVLink).

■ Key point:

- PCIe = universal bridge
- NVLink = specialized GPU fabric.

InfiniBand and RDMA Networking

■ Networking for Distributed GPU Systems

- InfiniBand = dominant interconnect in supercomputing.
- In MN5 ACC nodes:
 - 4 × NDR200 interfaces → **800 Gb/s per node**.
- Supports:
 - **MPI**
 - **RDMA** (Remote Direct Memory Access)

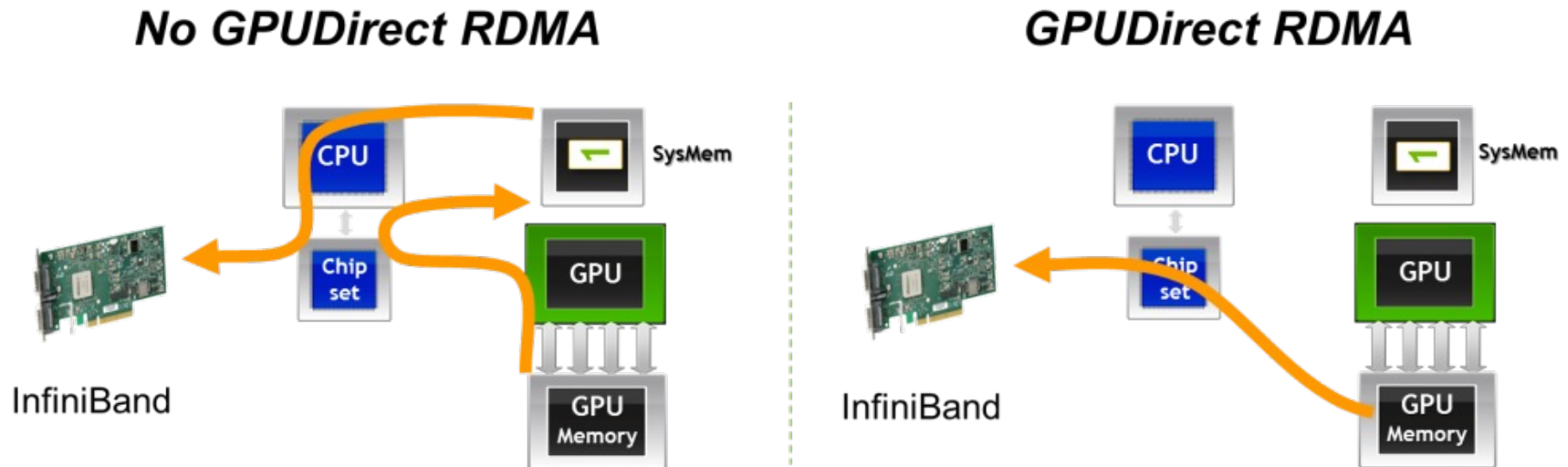
■ RDMA characteristics:

- Direct access to remote memory.
- No CPU intervention → ultra-low latency.
- Ideal for distributed AI training (many small, frequent exchanges).

■ “RDMA = memory-to-memory networking.”

GPUDirect RDMA

- **Figure 6.8 – GPUDirect RDMA schematic**
(source Jiri Kraus)



GPUDirect RDMA

- **GPUDirect RDMA: Direct GPU-to-GPU Transfers Across Nodes**
 - Allows direct network transfers between GPU memories.
 - Bypasses host CPU and DRAM → minimal latency.
 - Eliminates extra memory copies:
- **With/Without GPUDirect RDMA**
 - Without:
 - GPU → CPU memory → Network → CPU memory → GPU
 - With:
 - GPU → Network → GPU
- **Benefits:**
 - Lower latency.
 - Lower CPU load.
 - Higher scalability for distributed training.



6.3

Distributed Computing with CUDA-aware MPI

Why Combine MPI + CUDA?

■ Motivations for MPI + CUDA Integration

1. Data too large for a single GPU → distribute across GPUs/nodes.
2. Shorten runtime by parallelizing across many nodes.
3. Accelerate existing MPI CPU apps with GPU compute.
4. Scale single-node CUDA apps to multi-node clusters.

■ Leverage distributed parallelism (MPI) + device parallelism (CUDA).

– “MPI handles distribution — CUDA handles acceleration.”

Traditional MPI Communication

■ Standard MPI Requires Host Memory Buffers

- MPI_Send / Recv expect pointers to host memory.
- When using GPU memory: must stage through host.

```
//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);
```

■ Drawback:

- Extra memory copies → latency + CPU overhead.

GPU → Host → Network → Host → GPU

CUDA-aware MPI Simplifies Communication

- **Eliminating Staging with CUDA-aware MPI**
 - CUDA-aware MPI detects if a pointer refers to GPU memory.
 - Enables direct GPU communication without cudaMemcpy.

```
//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);
```

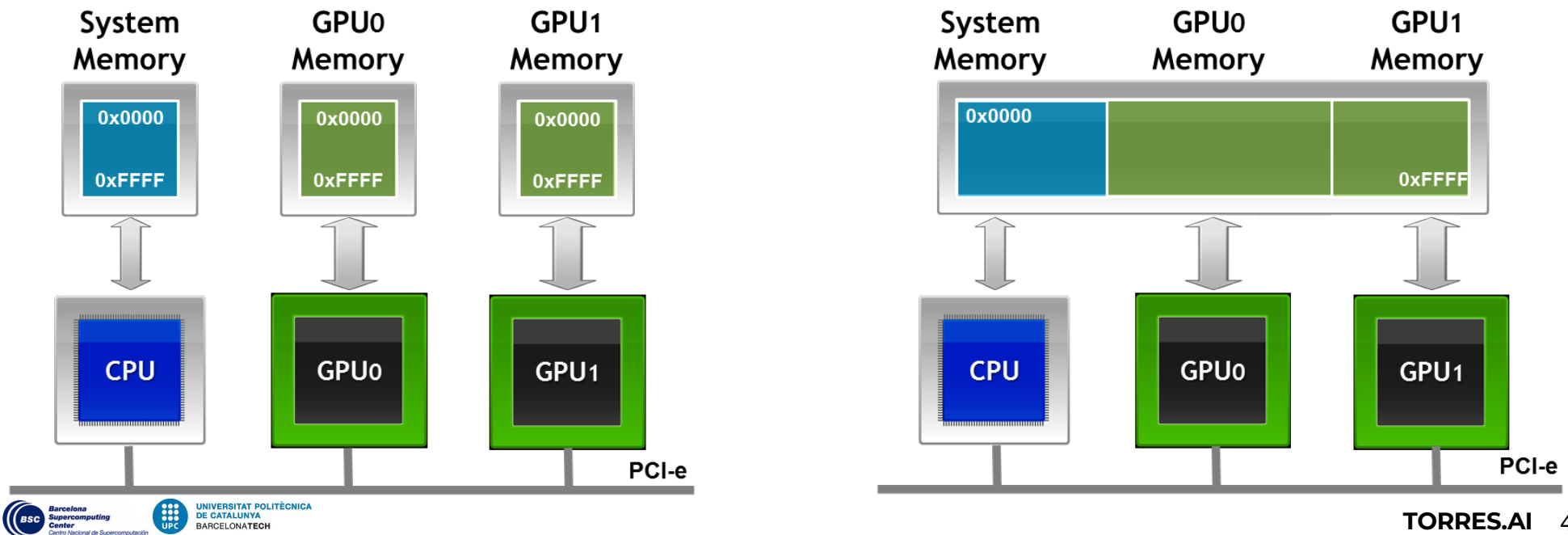
pointer to
GPU memory

pointer to
GPU memory

How CUDA-aware MPI Works

■ Unified Virtual Addressing (UVA)

- Introduced in CUDA 4.0 → now standard (CUDA 12.5+).
- Creates one unified virtual address space for:
 - Host memory
 - All GPU memories in the node
 - *MPI can infer memory type automatically.*
 - MPI checks pointer ranges → knows if it's host or device memory.





6.4

NCCL: Collective Communication for GPUs

Limitations of MPI for GPU Workloads

■ Why MPI Alone Is Not Enough

- Originally designed for CPU–CPU communication.
- Inefficient for GPU collectives because:
 - Requires staging through host memory.
 - Separate kernels for computation and communication.
 - Not aware of GPU interconnect topology.

→ **CUDA-aware MPI** adds GPU memory support, but still CPU-centric.

■ NCCL: NVIDIA Collective Communication Library

- Developed by NVIDIA for GPU-native communication.
- Optimized for deep learning and HPC workloads.
- Fully leverages:
 - NVLink, PCIe, InfiniBand, GPUDirect RDMA.

NCCL vs MPI: Key Differences

	MPI	NCCL
Target	General-purpose	GPU-accelerated deep learning
Memory model	Host or CUDA-aware	Fully GPU-native
Communication	CPU-managed collectives	GPU-executed collectives
Common use	HPC scientific apps	AI / Deep Learning training

- **NCCL achieves better scaling efficiency for GPU-dense systems.**

- Adapts to hardware topology:
 - **Intra-node:** NVLink / NVSwitch.
 - **Inter-node:** InfiniBand + GPUDirect RDMA.
- Chooses optimal collective algorithm dynamically.



6.5

Case Study: Distributed GPU Computing

Case Study

- **Applying CUDA-aware MPI to a real scientific workload.**
- **Example: Jacobi iterative solver**
 - a classic in numerical simulations.
- **Objective:**
 - understand computational patterns and measure scalability.
- **Based on the Jacobi code**
 - adapted from Jiri Kraus (NVIDIA).

Sequential Jacobi Algorithm

■ Overview of the Jacobi Iteration

- Solves a system of linear equations (2D-Laplace Equation on a rectangle) iteratively.
- Each point updated as the average of its four neighbors.
- Repeats until convergence (difference below tolerance).

Sequential Jacobi Algorithm

- Core loop structure in a single GPU:

```
While (not converged)
  Do Jacobi step:
    for (int iy = 1; iy < ny - 1; iy++)
      for (int ix = 1; ix < nx - 1; ix++)
        a_new[iy * nx + ix] = -0.25 *
          (a[iy * nx + (ix + 1)] + a[iy * nx + ix - 1]
           + a[(iy - 1) * nx + ix] + a[(iy + 1) * nx + ix]);

    Apply periodic boundary conditions

    Swap a_new and a
  Next iteration
```

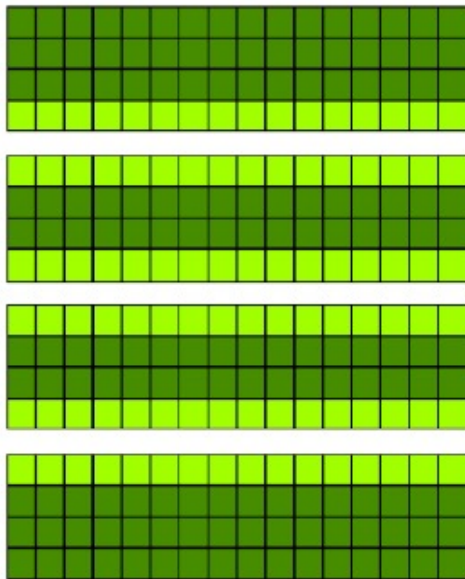
Local computation depends only on immediate neighbors

→ ideal for parallelism

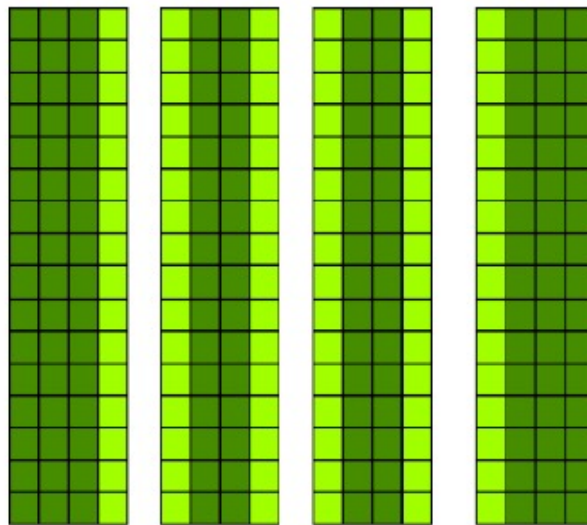
From Sequential to Parallel

■ Domain Decomposition for Parallelization

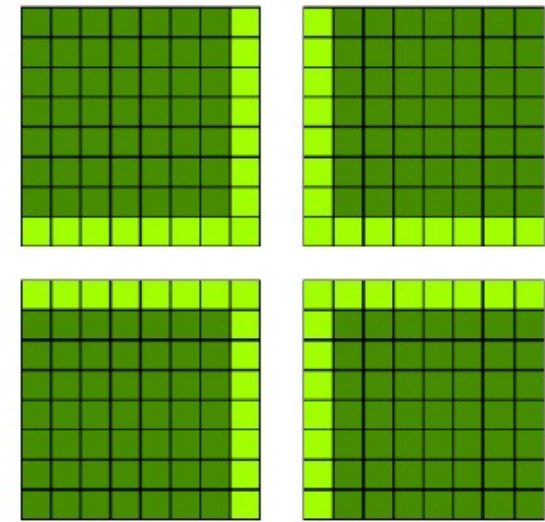
- To scale across GPUs, the 2D grid must be split into subdomains.
- Each process updates its local region and exchanges halo rows.
- Three decomposition strategies:



Horizontal Stripes



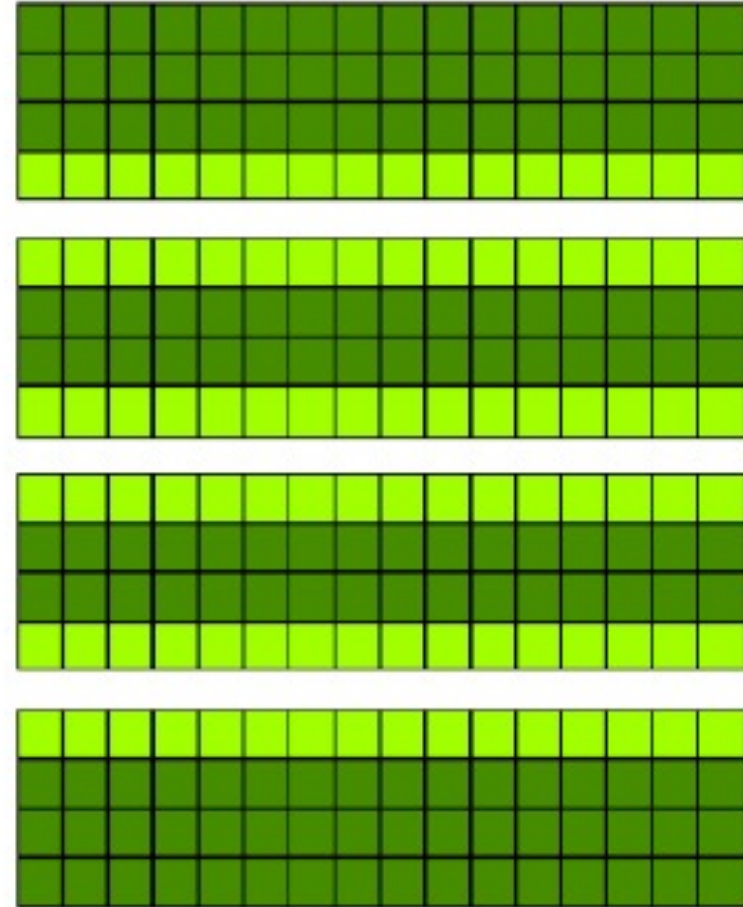
Vertical Stripes



Tiles

From Sequential to Parallel

- **Choice for this case study:**
 - Horizontal stripes
 - simplest to implement, minimal communication.



CUDA-aware MPI Implementation

- **After each Jacobi step:**
 - Each GPU sends its bottom row to the process below.
 - Each GPU receives the top row from the process above.
- **Efficient Communication with CUDA-aware MPI**
 - Data resides in GPU memory, sent directly between GPUs.
 - Uses **MPI_Sendrecv** instead of the separate MPI_Send and MPI_Recv functions
 - Simplifies code and improves performance.

MPI_Sendrecv

- The MPI_Sendrecv function allows us to send and receive data simultaneously in a single call, simplifying code management and reducing the risk of synchronization issues that can occur when using MPI_Send and MPI_Recv separately.

```
MPI_Sendrecv(send_buf_d, size, MPI_CHAR, down, 0,  
             recv_buf_d, size, MPI_CHAR, up, 0,  
             MPI_COMM_WORLD, MPI_STATUS_IGNORE);
```

- Avoids blocking dependencies between ranks and simplifies halo exchange.

CUDA Kernel for Jacobi Step

■ Kernel skeleton:

```
__global__ void jacobi_kernel(...) {  
    int iy = blockIdx.y * blockDim.y + threadIdx.y + iy_start;  
    int ix = blockIdx.x * blockDim.x + threadIdx.x + 1;  
    real local_l2_norm = 0.0;  
  
    if (iy < iy_end && ix < (nx - 1)) {  
        a_new[iy * nx + ix] = 0.25 * (  
            a[iy * nx + ix + 1]  
            + a[iy * nx + ix - 1]  
            + a[(iy + 1) * nx + ix]  
            + a[(iy - 1) * nx + ix]);  
    }  
}
```

Jacobi Computation on the GPU

- Each GPU computes updates only for its local stripe.
- Halo rows excluded from computation.
- Implemented as a CUDA kernel:

Convergence Control

- **Convergence and Benchmarking Loop**
- **Main control loop:**

```
while (l2_norm > tol && iter < iter_max)
```

- For benchmarking,
 - tolerance (`tol`) set very low
 - fixed number of iterations (`iter_max`).
- Guarantees identical workload across runs for fair performance comparison.



Running the Jacobi Code

Running the Jacobi Code

■ Overview:

- Practical case:
 - *Jacobi solver* implemented with CUDA-aware MPI.
 - Code and materials available on the course's GitHub repository.
- Content:
 - The exercise combines everything learned — CUDA, MPI, and performance analysis — into one reproducible workflow.
- Automatically reports key metrics:
 - Serial and parallel execution times
 - Speedup
 - Efficiency
- Experiments performed on Marenostrom 5 (MN5) using SLURM + MPI.

The Makefile

■ Three predefined build targets:

- `make optimized` → enables atomic operation optimizations.
- `make cub` → same + activates the CUB library.
- `make default` → no extra optimizations (baseline).

■ Notes:

- The `Makefile` is not modified.
- Only the version is selected according to the experiment.
- Different flags → different optimization strategies.
- Key message: Experimentation is controlled by compilation flags, not by code changes.

Jacobi.cu and Execution Arguments

- **Main program: jacobi.cu**

- No code modification required — just recompile.

- **Execution with mpirun:**

```
mpirun -np <num_GPUs> ./jacobi -niter <iters> -nx <x_size> -ny <y_size>
```

- **Default parameters:**

```
-niter 1000
```

```
-nx 16384
```

```
-ny 16384
```

Jacobi.cu and Execution Arguments

- **Each execution runs two simulations internally:**
 - Single-GPU reference
 - Multi-GPU run (-np value)
- **Each execution reports**
 - serial time,
 - parallel time,
 - speedup, and
 - efficiency.

SLURM Script Example

- **Content:**

Example job: 4 GPUs solving a 16384×16384 grid

```
#SBATCH ...
```

```
module load nvidia-hpc-sdk/23.11-cuda11.8
```

```
make clean
```

```
make default
```

```
mpirun -np 4 ./jacobi -niter 100000 -nx 16384 -ny 16384
```

- **Key ideas:**

- Always load the correct module environment.
- make clean → ensures reproducibility.
- Compilation inside the script → clarity and portability.
- mpirun chosen for compatibility with traditional MPI codes.

Performance Example (4 GPUs)

■ Output sample:

16384x16384: 1 GPU:648.91s, 4 GPUs:165.73s, speedup:3.92, efficiency:97.89%

■ Interpretation:

- $\sim 4\times$ faster \rightarrow near-perfect scaling.
- 97.9% efficiency \rightarrow excellent load balancing.
- Communication overhead is negligible.

Task 6.3

■ Task 6.3 – Submit and Validate

Instructions:

1. Download code from GitHub.
2. Prepare SLURM script (4 GPUs, 16384^2 grid).
3. Submit job.
4. Check .err → no runtime errors.
5. Check .out → verify metrics \approx reference values.

Purpose: Learn the full process: compilation → submission → validation.

Task 6.4

■ Task 6.4: Understanding the Metrics

Now that you have obtained valid performance results, revisit the Jacobi source code and review how these metrics are collected.

Notice that each execution internally runs two simulations: one using a single GPU and another using the number of GPUs specified via the `-np` argument.

This design allows the program to compute speedup and efficiency automatically for each execution.

If anything is unclear at this point, don't hesitate to ask your instructor for guidance.



Optimizing the code using compilation flags

Why Optimize?

- **Even with good speedup, performance can still improve.**
- **CUDA code can be optimized through compiler flags.**
 - Small compiler tweaks can yield large runtime improvements.

Understanding PTX and nvcc Flags

- **PTX (Parallel Thread Execution):**

- Intermediate assembly language in CUDA.

A virtual machine instruction set architecture that has been part of CUDA from its beginning

- Enables hardware abstraction and cross-GPU optimizations.

You can think of PTX as the assembly language of the NVIDIA CUDA GPU computing platform.

- **Compilation process:**

CUDA → PTX → device-specific binary.

- **Flag example:**

-Xptxas --optimize-float-atomics

→ Optimizes atomic operations on floating-point values.

Enabling Optimizations

- The Makefile already includes an optimization target.
- To enable it, use:
`make optimized`
 - Add this line inside your SLURM script before mpirun.
 - No code changes are required — only recompilation.

Task 6.5

■ Task 6.5 – Explore the Effect of Optimized Compilation Flags

Compile the Jacobi solver using the make optimized target, which enables the -Xptxas --optimize-float-atomics flag.

Run the program with the same configuration as before (grid size 16384 × 16384, using 4 GPUs), and obtain the performance metrics again.

Compare the new results to those from the default build.



Optimizing the code using the library CUB

Why Use CUDA Libraries?

- **Beyond compiler flags, CUDA libraries provide high-performance primitives.**
- **CUB (CUDA UnBound)**
 - header-only NVIDIA library for optimized parallel operations.
- **Offers ready-to-use, fine-tuned GPU kernels that outperform manual implementations.**
- **IMPORTANT: Libraries like CUB encapsulate years of GPU optimization research — use them!**

Introducing CUB

- To include CUB:

```
#include <cub/block/block_reduce.cuh>
```

- Compile using the predefined Makefile target:

```
make cub
```

- The macro `HAVE_CUB` activates CUB primitives.
 - Header-only → easy to integrate.
 - Controlled via `#ifdef HAVE_CUB`.

Introducing CUB

■ Code fragment using conditional compilation:

```
    if (calculate_norm) {  
#ifdef HAVE_CUB  
        real block_l2_norm = BlockReduce(temp_storage).Sum(  
                                           local_l2_norm);  
        if (0 == threadIdx.y && 0 == threadIdx.x)  
            atomicAdd(l2_norm, block_l2_norm);  
#else  
        atomicAdd(l2_norm, local_l2_norm);  
#endif    // HAVE_CUB  
    }
```

■ Explanation:

- BlockReduce performs in-block sum reduction using shared memory.
- Reduces global atomics → less contention → faster execution.
- Keeps only one atomicAdd per block (not per thread).

Task 6.6

■ Task 6.6 – Evaluate the Impact of the CUB Library

Compile the code using the make cub target to enable the use of the CUB library via the -DHAVE_CUB macro.

Run the Jacobi solver with the same problem size (16384×16384) and 4 GPUs, and measure the new execution metrics.

Compare the results with those obtained from both the default and optimized builds.



Benchmarking and Scalability

Overview

- Once the optimized CUDA-aware MPI Jacobi solver is ready, **we test scalability.**
- **Goals:**
 - Evaluate performance as GPU count increases.
 - Study how **problem size affects scalability.**
- **Benchmark setup:**
 - Grid sizes: **8192², 16384², 32768²**
 - GPUs: **1, 2, 4, 8, 16, 32**
 - Compiled with:
 - make cub

Automated Benchmark Script

- **SLURM loop to run all configurations automatically:**

```
make cub

sizes=(32768 16384 8192)
tasks=( 2 3 4 5 6 7 8 12 16 24 32)

for size in "${sizes[@]}; do
    for np in "${tasks[@]}; do
        mpirun -np $np ./jacobi -niter 100000 -nx $size -ny $size
    done
done
```

Automate testing to ensure consistent and reproducible benchmarking data.

Execution Time Results

GPUs	32768x32768	16384x16384	8192x8192
1	735.2762	186.0001	47,8084
2	368.7384	94.8350	25,8286
4	186.6551	49.5587	14,9914
8	95.7559	26.6790	9,3924
16	50.1935	15.5745	6,6131
32	27.9415	10.1108	4,9353

Table 6.1 – Execution time (in seconds) for three different problem sizes.

Execution Time Results

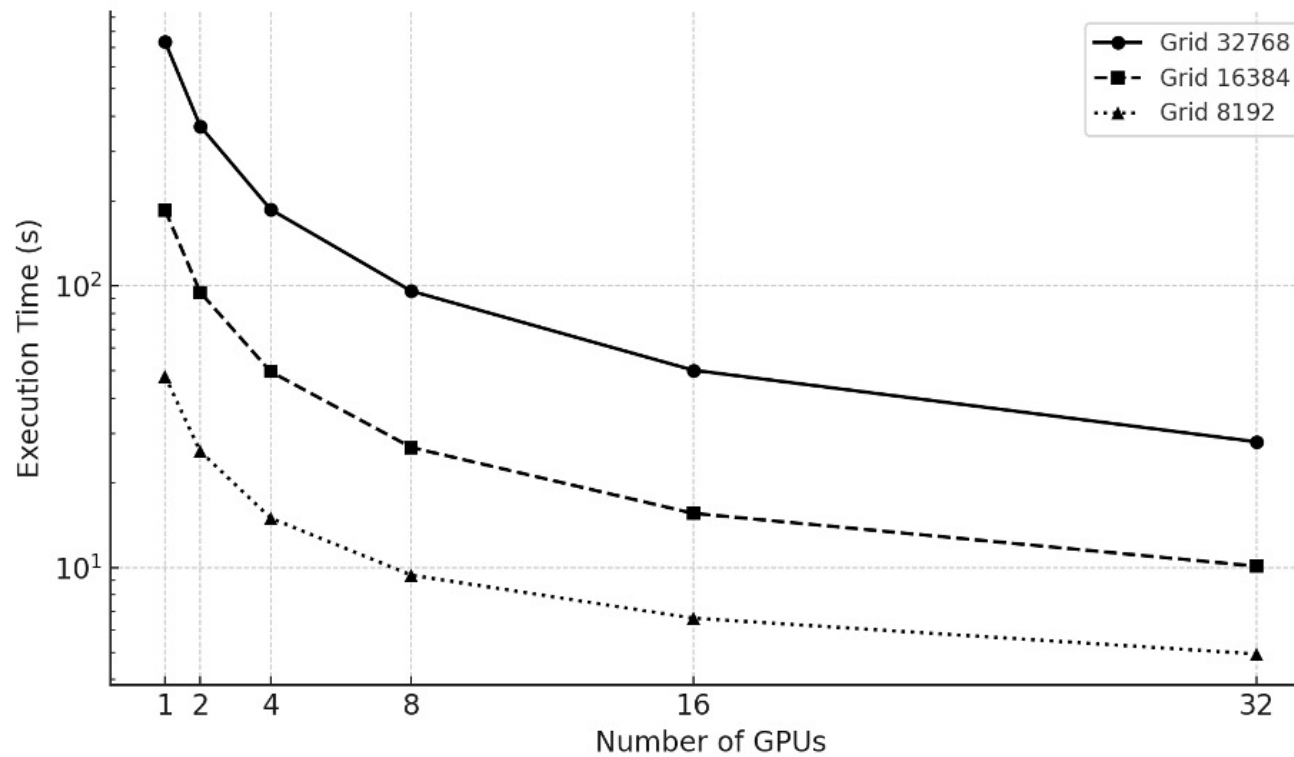


Figure 6.12 – Execution time of the Jacobi solver using 1 to 32 GPUs for three different problem sizes.

Parallel efficiency

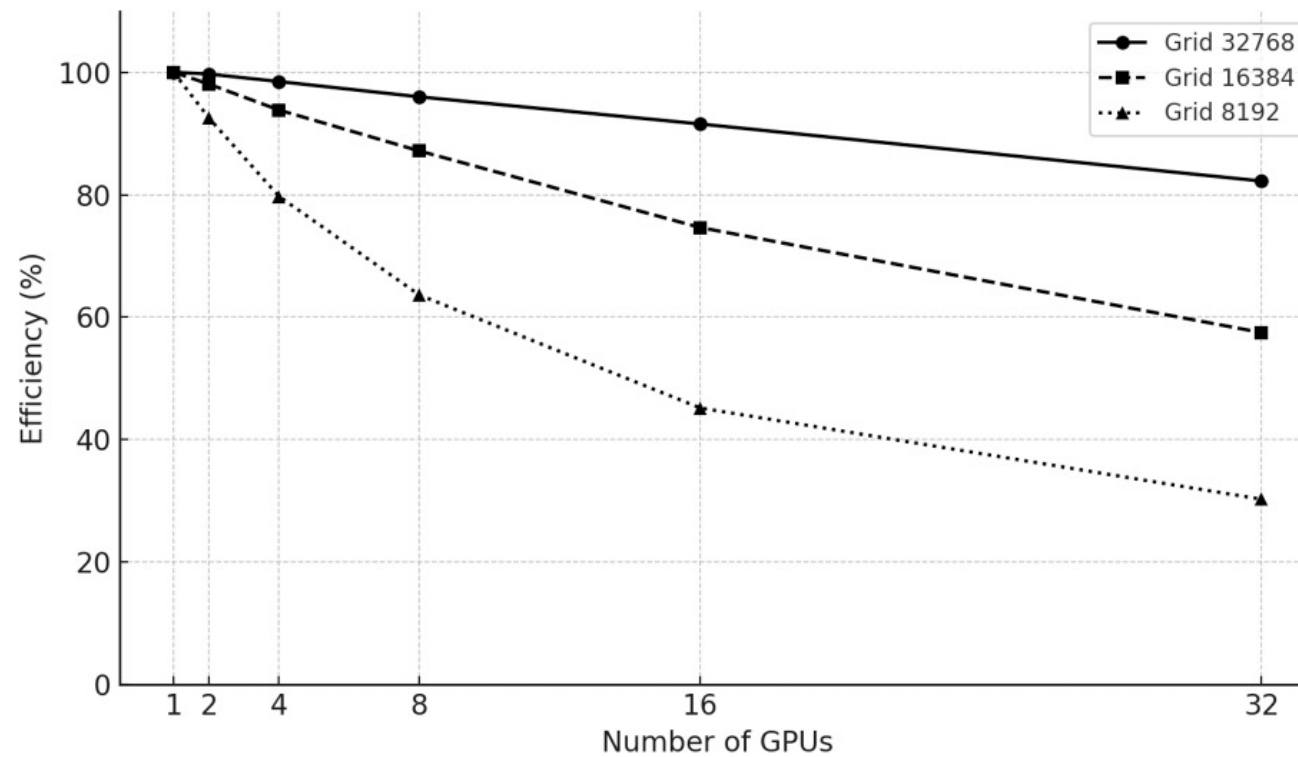


Figure 6.13 – Parallel efficiency vs. number of GPUs for different problem sizes.

Communication vs Computation

GPUs	Time (s)	Speedup	Efficiency (%)
1	47.8	1	100
2	25.8	~1.85	~92.5
4	14.99	~3.2	~80
8	9.39	~5.1	~64
16	6.61	~7.2	~45
32	4.93	~9.7	~30

Table 6.2 – Speedup and Efficiency for a mesh size 8192x8192.

Communication vs Computation

GPUs	Time (s)	Speedup	Efficiency (%)
1	735.2	1	100
2	368.7	1.99	99.5
4	186.6	3.94	98.5
8	95.7	7.68	96
16	50.2	14.6	91
32	27.9	26.3	82.2

Table 6.3 – Speedup and Efficiency for a mesh size 32768x32868..

Task 6.7

■ Task 6.7 – Benchmarking the Impact of GPU Count and Problem Size

- In this task, you will reproduce the scalability experiments shown earlier by running the optimized Jacobi solver using **1, 2, 4, 8, and 16 GPUs**, and three different problem sizes: 8192×8192 , 16384×16384 , and 32768×32768 .
- Your goal is to:
 - Modify your SLURM script to include a loop that automates execution over the different combinations of problem sizes and GPU counts.
 - Use make cub to compile the optimized code before launching your tests.
 - Collect the execution time for each combination from the output files.
 - Plot your own charts, One showing execution time vs. number of GPUs for each problem size and another showing speedup vs. number of GPUs for each problem size.
 - Analysis and recommendations from your data.

Pe: Distributed GPU Programming

- **Tasks included:**

- task 6.1 to task 6.7

- **Deliverable & Evaluation:**

- Upload a single PDF (per group) to the racó@FIB intranet, containing **as many slides as you need per task to clearly express what is requested in each one.**

- **In class (evaluation day):**

- One group (chosen at random) will give a clear, concise, and straight-to-the-point presentation.

However, **do not worry about timing — it does not need to follow a strict “elevator pitch” style as in previous labs.**

The goal here is pedagogical, allowing time to discuss results and reflect on what has been learned.

Lab presentation

- Remember: Good practical experience for students!
and ... a way to stimulate homework accomplishment
- 1 group **will be randomly chosen**
 - We'll sum 4 numbers from randomly chosen students and use the '%' function with the total number of students to find the winner in the list.

```
>>> nums_to_add = ...+...+...+...  
>>> winner= nums_to_add % num_students +1  
>>> print (winner)
```