E4
COMPUTER
ENGINEERING

# Deep Dive in Heterogenous Computing

Architecture, Applications, and Optimization

A comprehensive guide to solving the world’s biggest puzzles

# Lecture Agenda

## Module I

- **“What”:** High Performance Computing (HPC)
- **“Why”:** The physical limitations of CPUs and the rise of parallel computing
- **“How”:** Heterogenous computing changes the landscape of HPC
- **“Future”:** evaluation of HPC looks like the over the coming years

## Module II

- **“Understanding”:** HPC (CPU) and Device (GPU) architecture and relationship
- **“Program Flow”:** ways to program C/C++ to the device
- **“Vender”:** specific programming model and their limitations

## Module III

- **“CUDA”:** Accelerated computing C/C++ code aims achieve the best performance
- **“OpenACC”:** High abstract level of programming model combines performance and productivity
- **“Vender”:** specific to NVIDIA architecture

# Module I

## Part I

- **Introduction:** Definition, Importance, Evolution
- **HPC Systems:** Hardware Architecture
- **Memory Hierarchy:** Caches, NUMA, HBM
- **Storage & I/O:** Parallel Filesystems, Burst Buffers
- **Interconnects:** InfinitBand, Ethernet, Topologies

## Part II

- **Cluster Architecture:** Nodes, Scaling, Partitions
- **Programming Models:** MPI
- **Memory Hierarchy:** Caches, NUMA, HBM
- **Storage & I/O:** Parallel Filesystems, Burst Buffers
- **Interconnects:** InfinitBand, Ethernet, Topologies

# Introduction to HPC

# What is HPC?

> **High-Performance Computing (HPC)**
>
> The practice of aggregating computing power in a way that delivers much higher performance than one could get out of a typical desktop computer or workstation in order to solve large problems in science, engineering, or business

```
Performance = Node_Performance × Number_of_Nodes × Efficiency
```

**AGGREGATED POWER**

Solving problems that are too large for a single machine.

# Why HPC Matters?

## TIME-TO-SOLUTION

Reducing simulation time from months to days, or days to minutes. Critical for weather forecasting and financial trading.

## FIDELITY & RESOLUTION

Allows for finer mesh grids in simulations, capturing smaller physical phenomena (e.g., turbulence in airflow).

## IMPOSSIBLE EXPERIMENTS

Simulating scenarios that are dangerous, expensive, or impossible to test physically (e.g., nuclear aging, galaxy formation).

# Measuring Performance: FLOPS

## FLOPS

FLoating point OPerating per Second

| Unit | Value | Scale Example |
|---|---|---|
| **GigaFLOPS** | $10^9$ | Modern Laptop / Smartphone |
| **TeraFLOPS** | $10^{12}$ | High-end Workstation / Single GPU |
| **PetaFLOPS** | $10^{15}$ | Top 500 Supercomputers (2010s) |
| **ExaFLOPS** | $10^{18}$ | Current Frontier (Frontier of HPC) |

# Systen Architecture: Core Components

# Processor Architecture: CPUS

### Majority of silicon is dedicated to

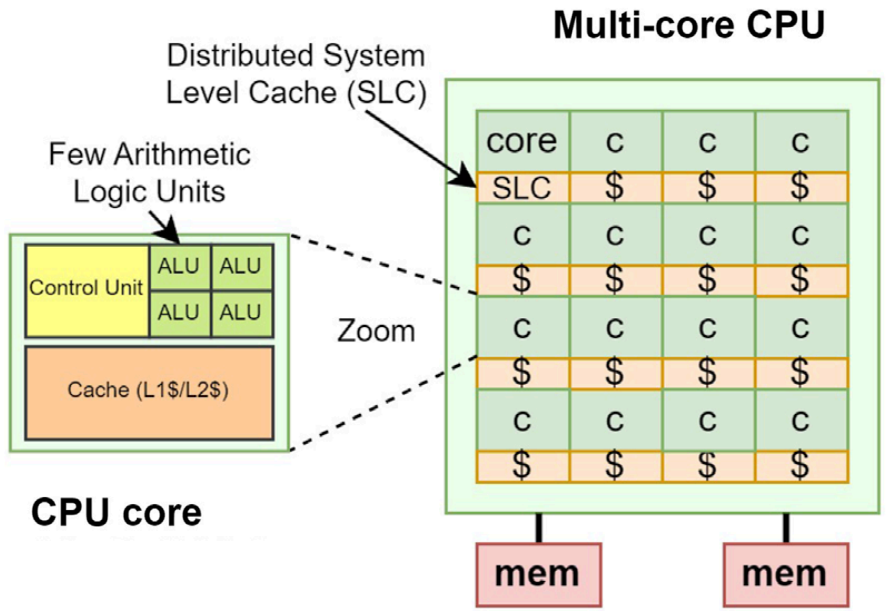
- Just a few cores (8-32)
- Large cache, low latency
- Tens of software threads at a time
- Taks parallelism

### Memory hierarchy

- Very large Dynamic Random Memory Access (DRAM)
- Fast cache memory (Registers, L1-L3 cache)

CPU architecture must minimize latency within each thread

CPUs are designed to minimize latency



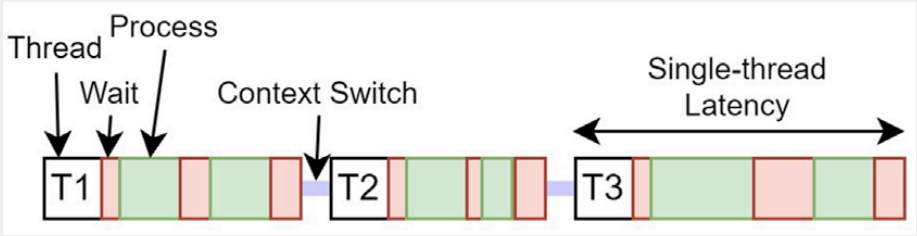
# Processor Architecture: CPUS

CPUs optimized for latency and complex serial logic

- **x86_64:** Dominant player (Intel Xeon, AMD EPYC)
- **ARM (AArch64):** Rising star (Fugaku A64FX, NVIDIA Grace). Focus on power efficiency and memory bandwidth
- **Role:** OS management, 1/0 handling, complex branching code

Designed for diverse tasks.

```
[Core 1] [Core 2] ... [Core 64]
| |
[ L3 Cache (Shared) ]
|
[ Memory Controller ]
```



# Memory Hierarchy

### Bandwidth

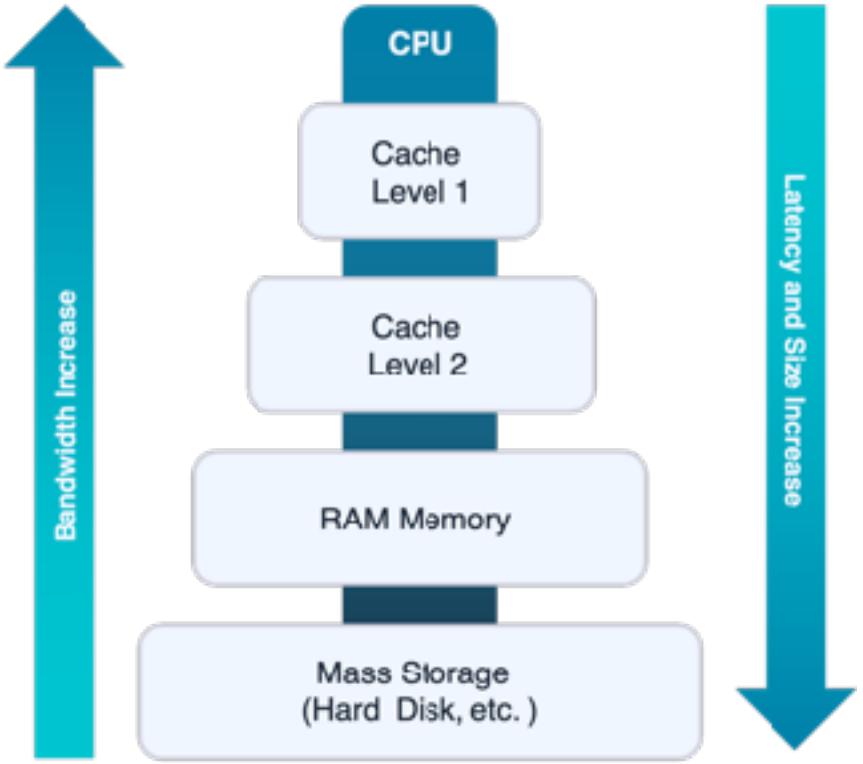
- Refers to the maximum amount of data that can be transferred between the CPU and memory in a given amount of time, usually measured in GB/s

### Latency

- Refers to the delay between when the CPU requests data or instructions and when it receives them to start processing

### Throughput

- Refers to the number of processes completed by the hardware in a given per time unit

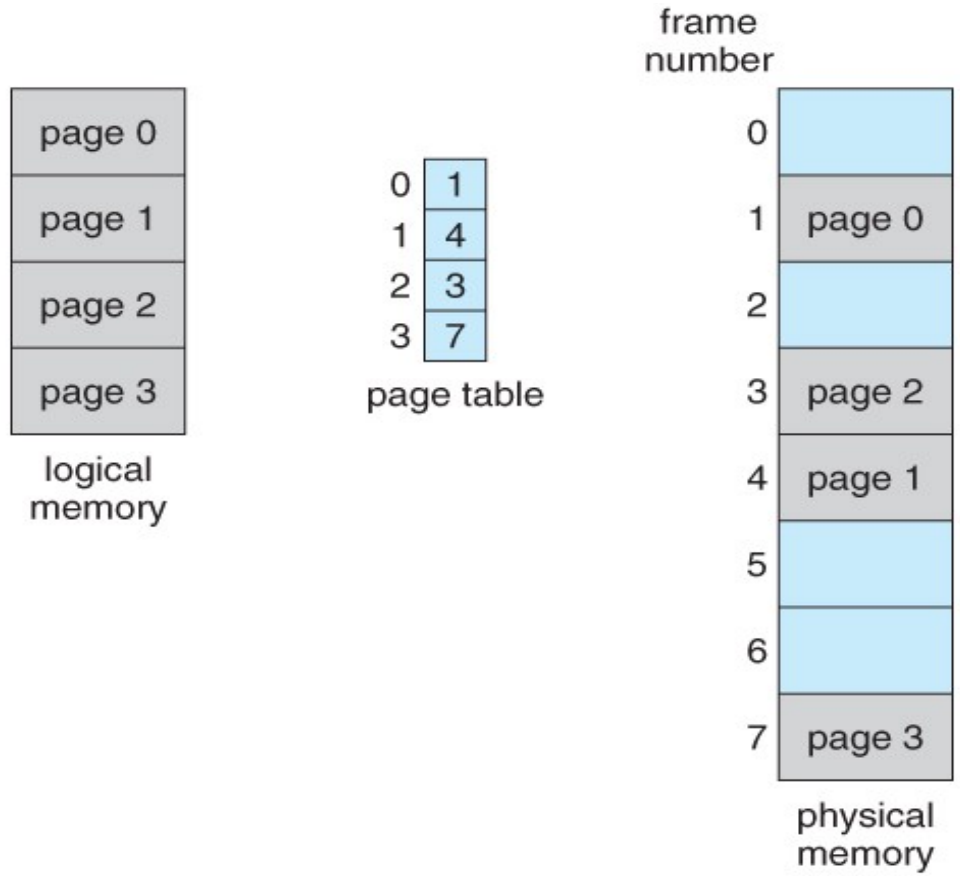



High latency = CPUs sitting idle = wasted resources

# Virtual Memory: The Great Abstraction

A memory page is a fixed-length contiguous block of virtual memory that serves as the fundamental unit of memory management in modern computing systems.

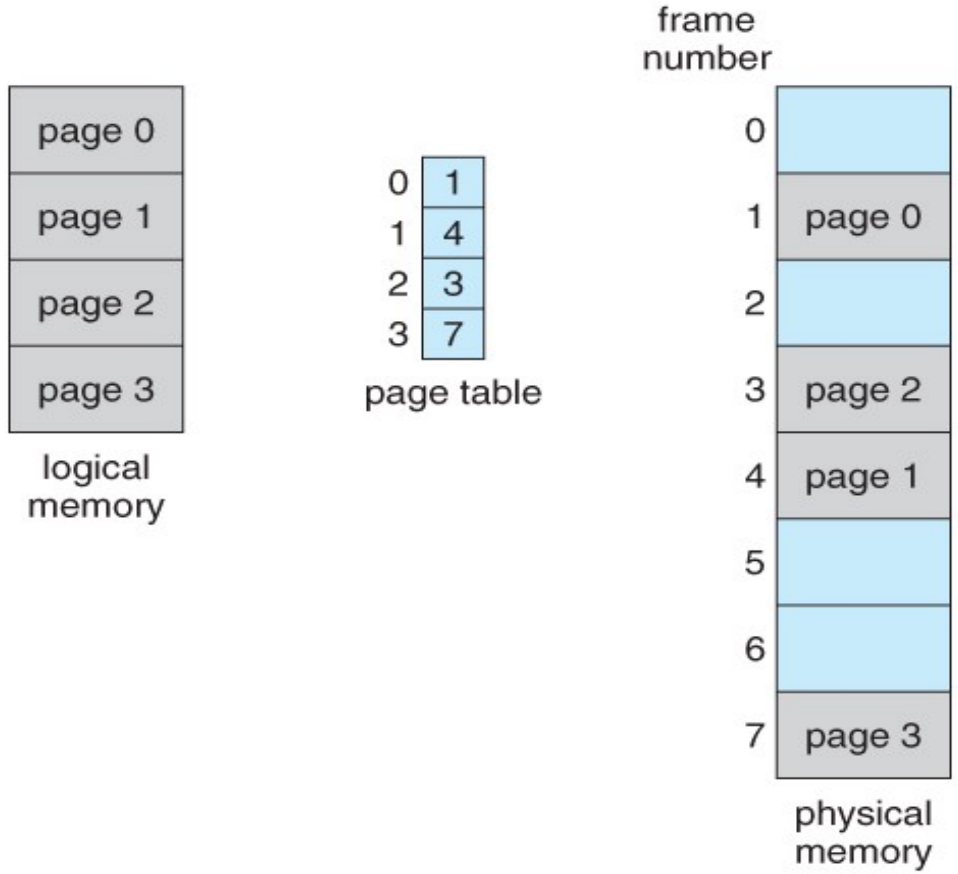
- Typically 4 KB in size, though this can vary based on operating system and hardware architecture
- With paged memory each program has its own logical memory, which can be broken into consecutive pages
- The program will read and write data to the pages , which are mapped to the physical memory via page table

# Virtual Memory: The Great Abstraction

A memory page is a fixed-length contiguous block of virtual memory that serves as the fundamental unit of memory management in modern computing systems.

- Although main memory is utilised better with paged memory, its size is limited
- When main memory does't have enough space, and more data needs to be written, some pages will be moved to the hard drive, called **pages out or swapping**
- The process of moving pages from hard drive back to memory is called ***page in***

# Cluster Architecture

# What is a HPC cluster?

### Your Laptop

Good for: Browsing, Word, Netflix.

Processors: 1 (with 4-12 cores)

**Personal Use**


**Supercomputer**

Processors: 10,000+

Scientific scale

It's not just a **'Fast'** computer, It's a **Cluster** of computers

### Hardware

Thousands of individual servers connected together.

### Network

A high-speed fabric allowing them to talk.

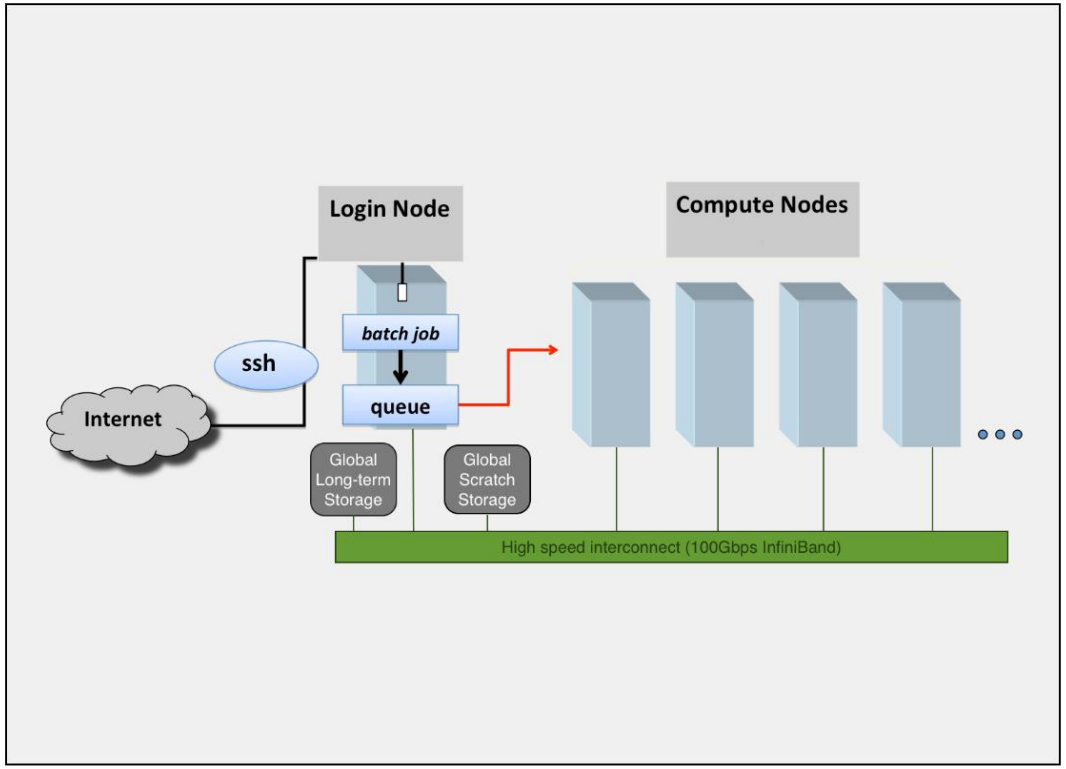
### Software

A stack that makes them act as one machine.

# The Anatomy

We will dissect the system layer by layer

- **Physical Layer:** The room & Racks
- **Network Layer:** The Interconnect, i.e. low-latency network enabling fast node-to-node communication
- **Head/Login Node:** The gateway for users to submit jobs and manage resources
- **Compute Layer:** The workhorses of the cluster where processing happens
- **Storage Layer:** High-performance parallel file system (e.g., Lustre, GPFS)
- **The Software**
  - **Operating Systems:** Why Linux?
  - **Environment:** Module systems (Lmod/TCL)
  - **Scheduling:** Commands, Scripts, Management
  - **Workflows:** Interactive, Arrays, Dependencies



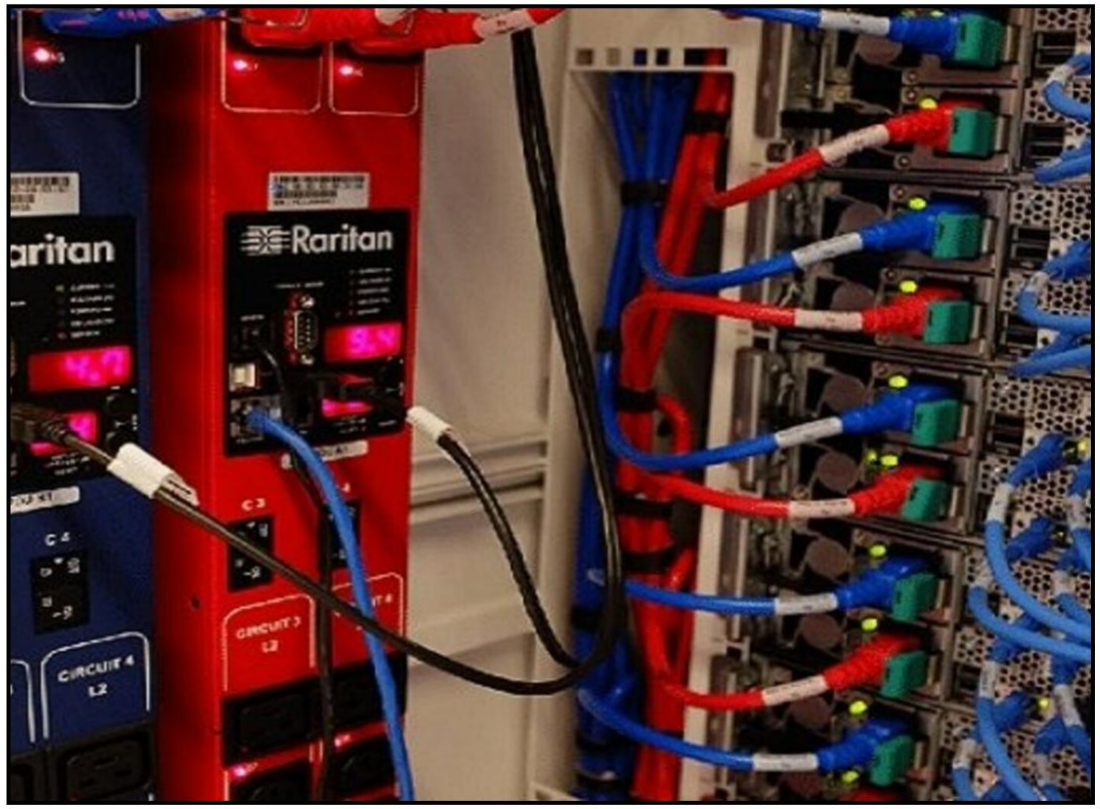
# The nodes classification: the workers

**Login Node:** The front door

- **This is where you SSH into ('ssh user@hpc.edu')**
- **Shared by All users**
- **Purpose:** Edit files compile code, submit jobs
- **WARNING:** Never run heavy calculations here. You will slow it down for everyone

**Compute Node:** the workhorse

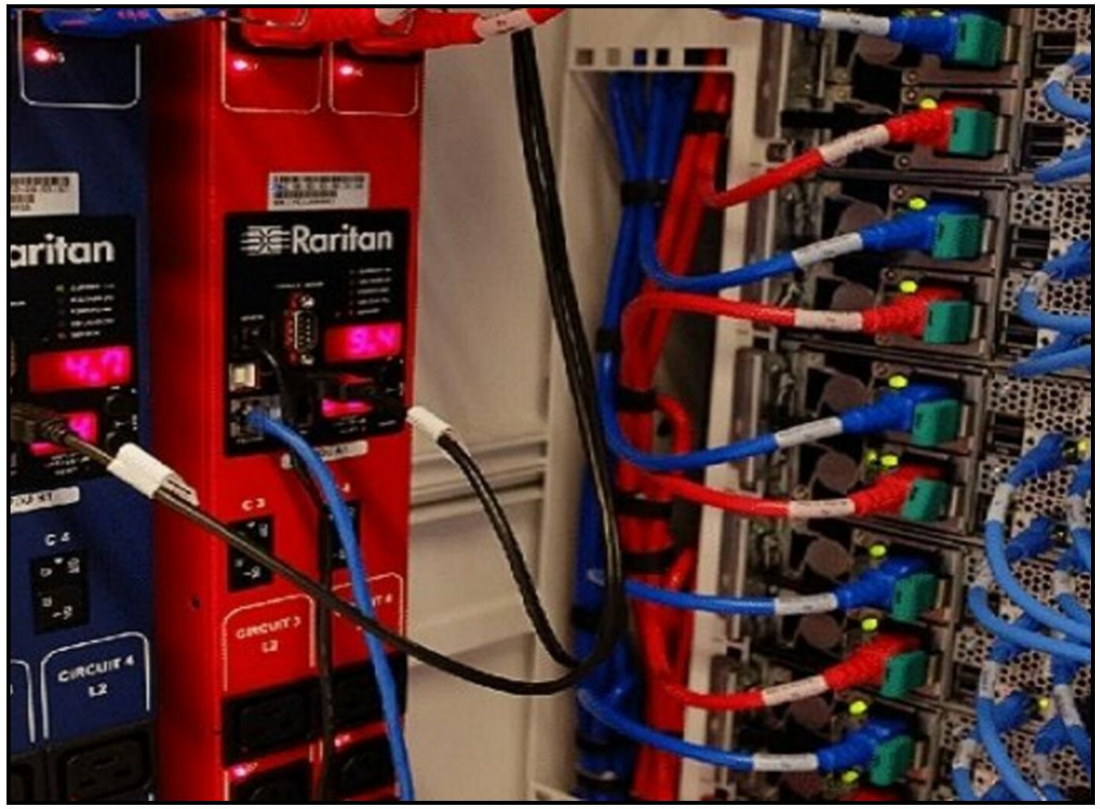
- **You cannot SSH here directly (usually)**
- Access is granted only via the Scheduler (Slurm)
- **Purpose:** Purpose: Crunch numbers 24/7
- **Configuration:** Minimal OS services t o save CPU cycles



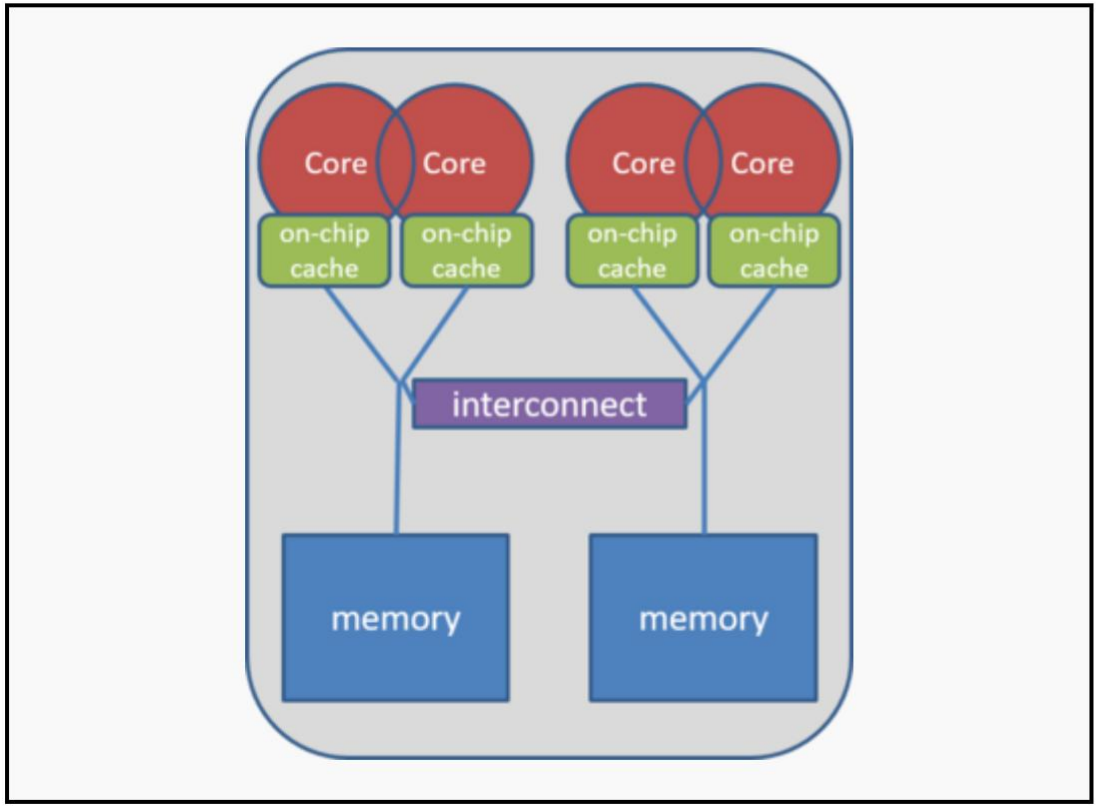
# Inside a Compute Node

Typical Specs (2025 Standard):

- CPU: Dual socket (2 Physical chips)
- Cores: 64-128 Cores total
- RAM: 256 GB - 1 TB DDR5
- DISK: Small SSD (480 GB) for local scratch
- Network: 1x InfiniBand, 1x Ethernet



# NUMA Architecture



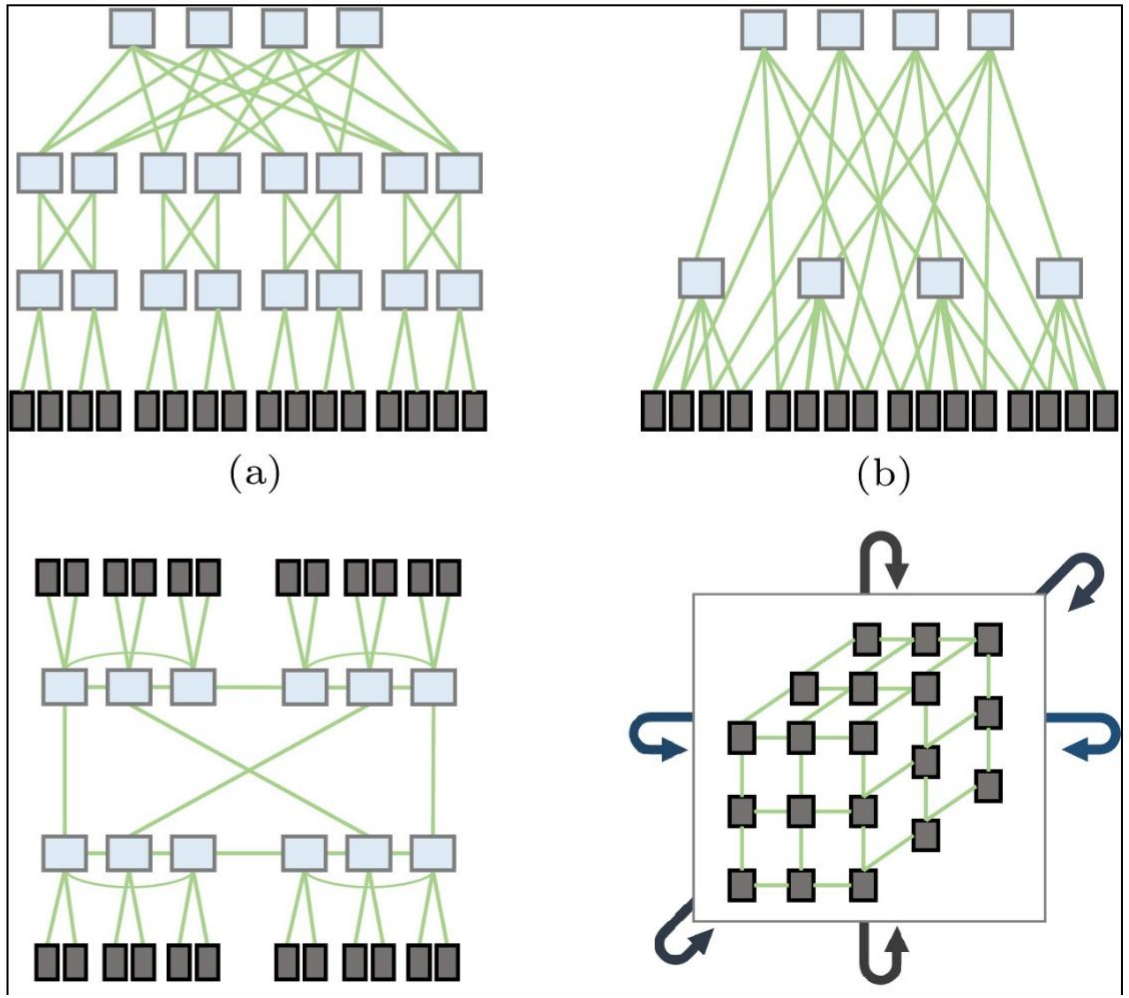

### Non-Uniform Memory Access

- A dual-socket node has 2 CPUs
- Half the RAM is attached to CPU1, half to CPU2
- CPU1 accessing CPU2's RAM is **slower**
- **Tip:** Pin your tasks to cores to avoid crossing the bridge

# NETWORK TOPOLOGY

How do we connect 5,000 nodes together efficiently?

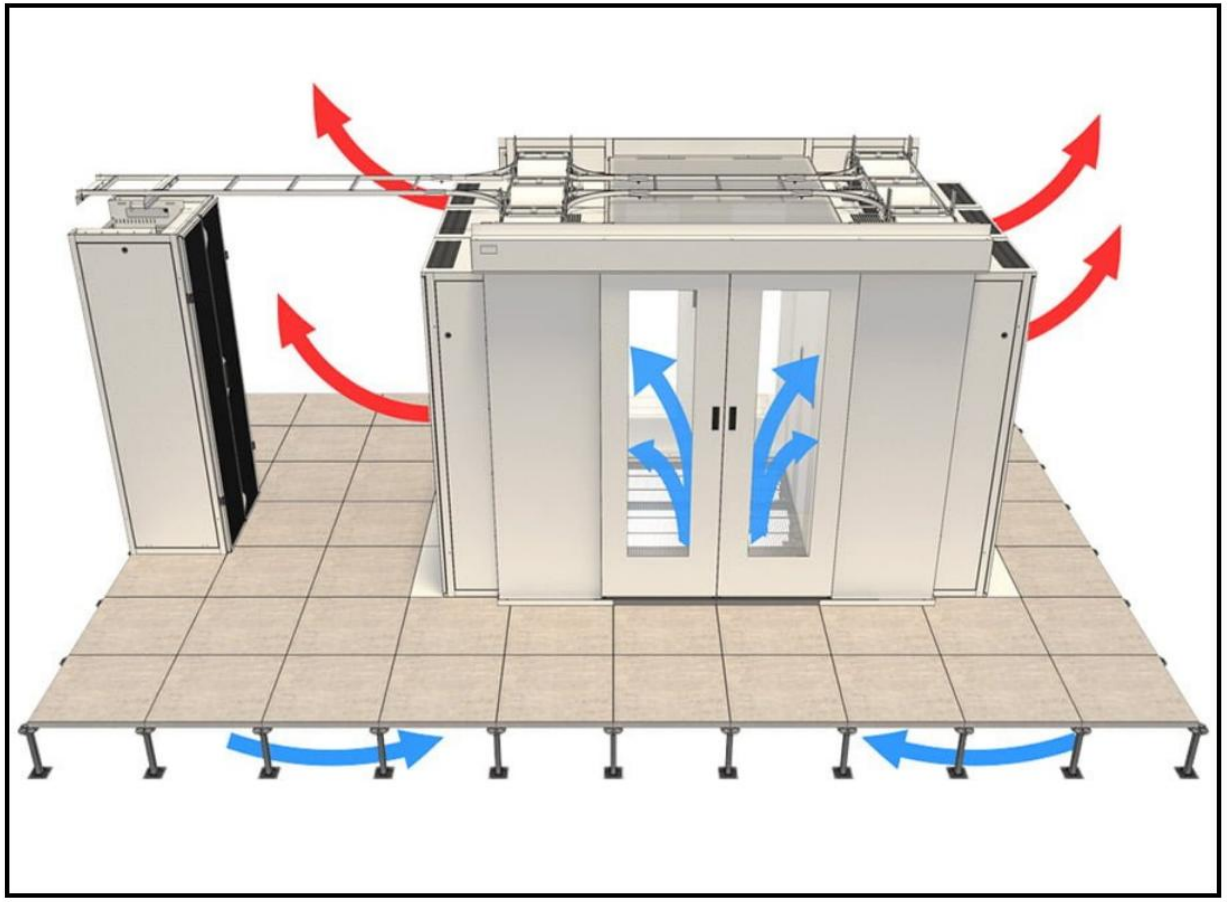
- **Goal:** Minimize "Hops" (switch jumps) between any two nodes.
- **Blocking Factor:** Ratio of bandwidth available within a rack vs. bandwidth leaving the rack.
- **Non-Blocking (1:1):** Full bandwidth available for all nodes simultaneously (Expensive).



# Physical Layer Summary

## The data center

- **High density racks:** A “rack” is a standardised metal frame (1,100 kg)
- **Power distribution (PDU):** Smart Power Strips running vertically down the back of the rack
- **Cooling (Hot/Cold Aisles):**
  - Front of racks face each other. Cold air is pumped
  - Back of racks face each other. Hot exhaust is sucked away here
  - **Liquid Cooling DLC:** Water pipes run inside the server, directly touching
  - the CPU/GPU, dissipating heat 24x better than air (Efficiency 95% Heat capture rate)
- **Cable Management: The “Speghetti” problem**
  - An HPC rack has ~40 nodes. Each node has: 2 Power, 1 IPMI, 1 Ethernet, 1 Infiniband.
  - Total: ~200 cables per rack
  - Cabling must be pristine to allow airflow and maintenance



# The network: the nervous system

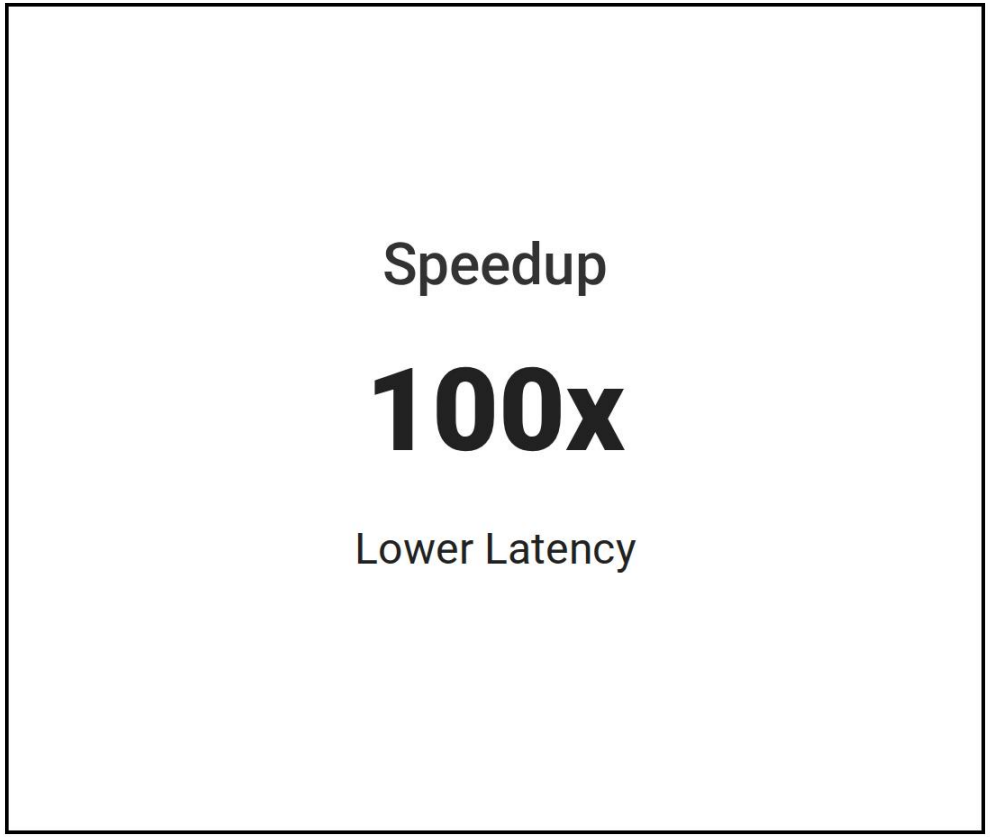
Why not just use WiFi or standard Ethernet?

## Latency is King

In HPC, computers talk millions of times per second.

Standard Ethernet latency: **20-50 microseconds**.

HPC Interconnect latency: **0.5 microseconds**.



# Infiniband (IB)


## The gold standard for HPC

- Manufacturer: Mostly NVIDIA (Mellanox)
- Current Speed: NDR 400 GBps per cable
- Feature: RDMA Remote Direct Memory Access)

**Remote Direct Memory Access**

**Without RDMA:** Node A CPU -> OS Kernel -> Network Card -> Wire -> Network Card -> OS Kernel -> Node B CPU.

**With RDMA:** Node A Memory -> Network Card -> Wire -> Network Card -> Node B Memory.

It bypasses the CPU entirely.

# Topology

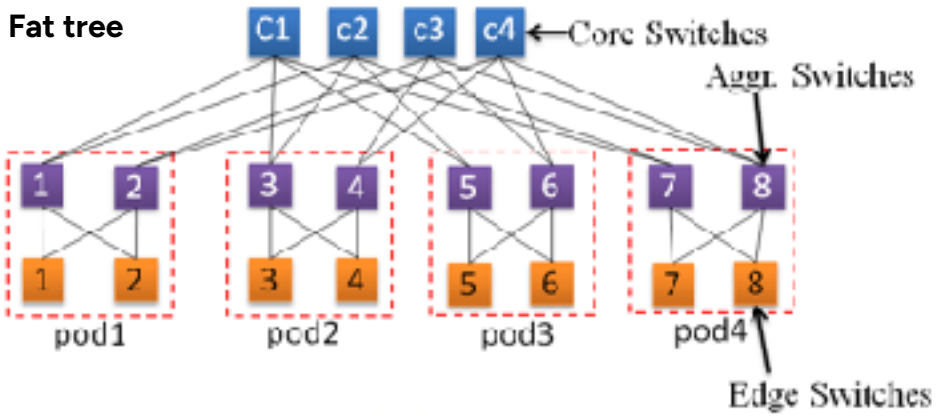
### **The Switches:** device that connects the cables

- **Leaf Switch:** Sits in the rack (Top of Rack). Connects to nodes
- **Spines Switch:** Sits in a central row. Connect Leaf switches together
- **Director Class:** Massive chassis switches the size of a fridge

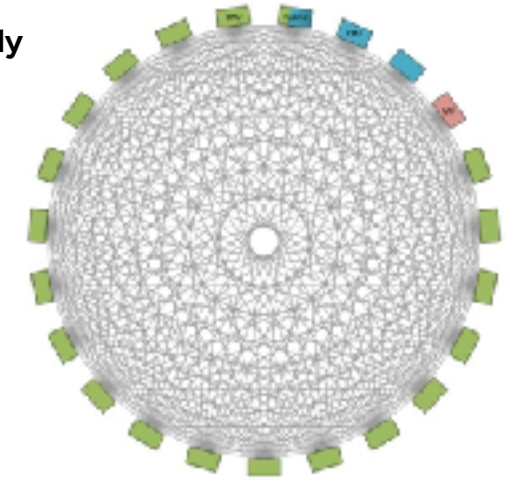
### How do we connect 100 switches?

- **Fat Tree:**
  - Like a real tree, branches get thicker near the trunk, requires tons of cable
  - Ensures Non-Blocking bandwidth (any node can talk to any node at full speed)
- **Dragonfly:**
  - Groups routers into electrical “groups
  - Uses optical cables to connect groups in a "all-to-all" pattern
  - **Benefit:** Lower cost, fewer optical cables
  - **Drawback:** Complex routing algorithms needed

**Fat tree**




**Dragonfly**



# Ethernet (The Other Network)

Every cluster has TWO networks.

## 1. High Speed Fabric

Infiniband / Slingshot.

Used for **MPI & Storage**.

## 2. Management Ethernet

1GbE or 10GbE.

Used for **Login, SSH, Admin, Monitoring**.

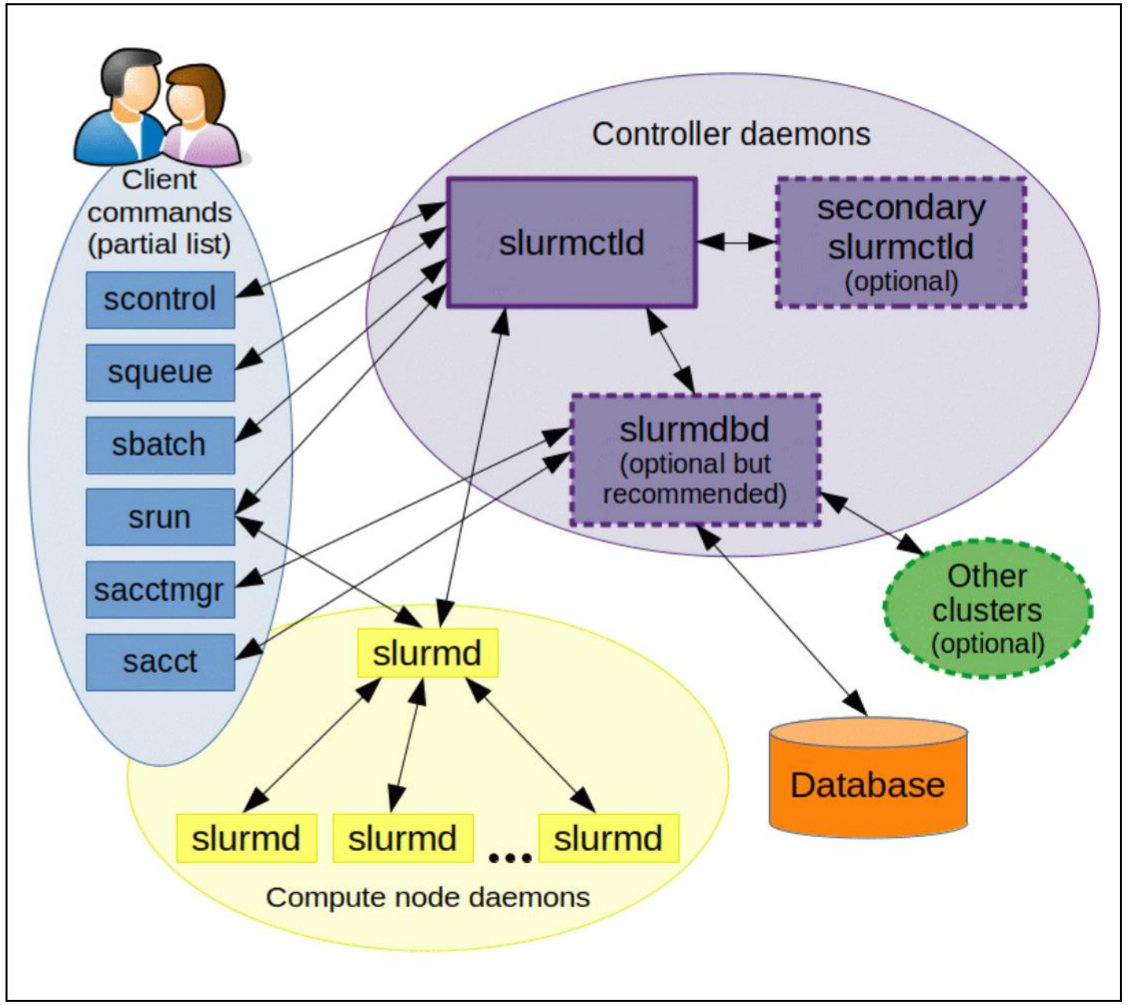
How to machine

# Slurm workload manager

# What is a Scheduler?

The "Brain" of the cluster allocation.

- **Software:** Slurm, PBS Pro, LSF, Moab.
- **Function:** Matches requested resources (Nodes, GPUs, Time) with available hardware.
- **Fairshare:** Algorithm to ensure no single user hogs the machine.
- **Backfill:** Fitting small short jobs into gaps left by large jobs waiting for resources.



# Useful command

```
$ sinfo #check the status of the cluster
  PARTITION AVAIL TIMELIMIT NODES STATES NODELIST
  debug* up 30:00 2 idle node [01-02]
  batch up 24:00:00 40 alloc node node [03-42]
```

State: Idle (Empty), Alloc (Full), Mix (Partial), Down (Broken).

```
$ scancel 12345 #Oops I made a mistake
  Kill specific job

$ scancel -u username #kill all my jobs
```

```
$ sbatch my_script.sh #Submit a job
  Submitted batch job 12345
```

The system reads your script, finds resources, and runs it when available

```
$ squeue -u username #check the list of jobs
  JOBID PARITION NAME USER ST TIME NODES nodelist
  12345 batch my_test alice R 10:00 1 idle node05
  12345 gpu ai_train bob PD 0:00 1 idle (Resources)
```

ST (State): R (Running), PD (Pending), CG (Completing).

# Slurm Accounting

How much did I use?

```
$ sacct -j 12345 --format=JobID,JobName,MaxRSS,Elapsed
```

- **MaxRSS:** Maximum RAM used.
- **Elapsed:** Wall clock time.
- Check this to optimize your next submission.

# Anatomy of a Job Script

A Slurm script is just a Bash script with special comments.

```
#!/bin/bash
#SBATCH --job-name=my_simulation
#SBATCH --nodes=1
#SBATCH --time=01:00:00

echo "Hello World"
```

# Directives: Resources

Lines starting with `#SBATCH` tell the scheduler what you need.

- `--nodes=N`: Number of physical computers.
- `--ntasks=n`: Total number of processes (MPI ranks).
- `--cpus-per-task=c`: Threads per process (OpenMP).
- `--mem=M`: Memory required (e.g., 10G).

# Directives: Logistics

- `--time=d-hh:mm:ss`: Wall clock limit. Job is killed if it exceeds this.
- `--partition=name`: Which queue to use.
- `--output=file_%j.out`: Where to write STDOUT (%j = jobID).
- `--error=file_%j.err`: Where to write STDERR.

# Module (Environment)

HPC systems have hundreds of software versions installed.

We use **Lmod** to manage them.

```
$ module avail # List all software
$ module load python/3.9
$ module load cuda/11.2
$ module list # Show what is loaded
```

# Sample Script: Serial Job

Running a simple python script on 1 core.

```
#!/bin/bash
#SBATCH --nodes=1
#SBATCH --ntasks=1
#SBATCH --time=00:10:00

module load python
python my_script.py
```

# Sample Script: MPI Job

```
#!/bin/bash
#SBATCH --job-name=mpi_gpu_job
#SBATCH --output=mpi_gpu_job.out
#SBATCH --error=mpi_gpu_job.err
#SBATCH --partition=gpu
#SBATCH --nodes=2
#SBATCH --ntasks-per-node=4
#SBATCH --gpus=gpu:1    ## GRES: Generic Resource. This asks for 1 GPU card.
#SBATCH --cpus-per-task=4
#SBATCH --time=02:00:00
#SBATCH --mail-type=END, FAIL
#SBATCH --mail-user=user@uni.edu ## email you when the job finishes or crashes.

module purge
module load mpi
module load cuda

export OMP_NUM_THREADS=$SLURM_CPUS_PER_TASK

srun ./your_mpi_gpu_executable
```

# Why srun?

`srun` is Slurm's parallel launcher.

- It knows exactly which nodes were allocated.
- It propagates signals (like Kill) to all processes.
- It handles accounting and resource limits better than mpirun.

## Interactive Jobs

Sometimes you need to debug or explore data.

```
$ salloc --nodes=1 --time=00:30:00 --partition=debug
```

This command waits for a node, then gives you a shell on the allocated node. You are now "logged in" to a compute node legally.

# You are now HPC Ready.

You understand the hardware, the network, the storage, and the scheduler.

Respect the Node

Optimize Time

Script Smart

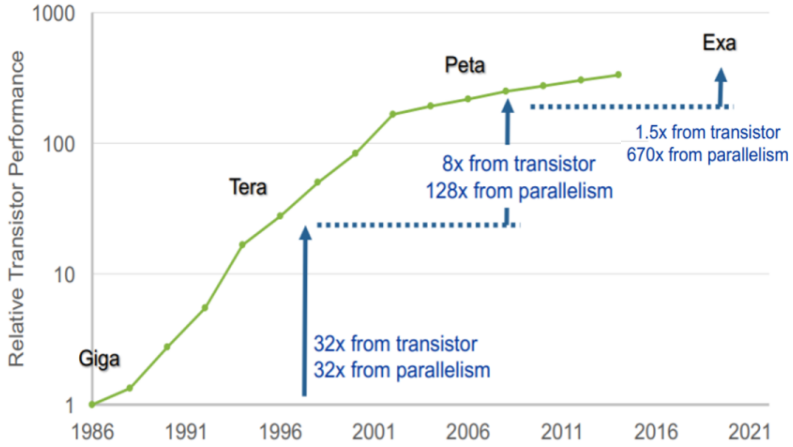
# Programming for HPC

# The Golden Era (1975-2005)

## Faster, Every Year

For 30 years, computer performance improved automatically. If your code was slow, you just waited for the next Intel CPU.

- **Free Lunch:** Software developers didn't need to change their code.
- **Clock Speed:** Went from 5 MHz (8086) to 3.8 GHz (Pentium 4).
- **Strategy:** Shrink transistors to make them switch faster.




*S. Borkar, Intel

Performance from parallelism

**50%**

Annual performance growth for nearly three decades.

# Key Concept: Parallelism

## DATA PARALLELISM

Distributing data across different cores. Each core performs the *same* operation on its piece of data.

*Example: Increasing brightness on an image pixels [1-100] on Core A, pixels [101-200] on Core B.*

## TASK PARALLELISM

Distributing distinct tasks across cores. Each core performs a *different* operation.

*Example: Core A handles UI, Core B handles Physics, Core C handles AI.*

# Memory models

## SHARED MEMORY

All threads can access the same variables.

**Scope:** Within a single node.

**Standard:** OpenMP.

**Pros:** Easy to program.

**Cons:** Doesn't scale beyond one node.

## DISTRIBUTED MEMORY

Each process has private memory. Must send messages to share data.

**Scope:** Across the whole cluster.

**Standard:** MPI.

**Pros:** Massive scalability.

**Cons:** Harder to program (explicit comms).

# MPI (Message Passing Interface)

The de-facto standard for HPC.

```
// Concept Code
MPI_Init();
rank = MPI_Comm_rank();
if (rank == 0) {
MPI_Send(data, dest=1);
} else {
MPI_Recv(data, source=0);
}
MPI_Finalize();
```

- Processes are independent.
- Communication is explicit (Send/Recv, Broadcast, Reduce).
- Works over InfiniBand/Ethernet.

# OpenMP (OPEN MULTI-PROCESSING)

Directive-based parallelism for Shared Memory.

```
// Concept Code
#pragma omp parallel for
for (i=0; i<1000; i++) {
a[i] = b[i] + c[i];
}
```

- Uses compiler directives (#pragma).
- Spawns threads (fork-join model).
- Best for loops within a single node.

# Vectorization (SIMD)

**Single Instruction Multiple Data.**

Instead of adding A[0]+B[0], then A[1]+B[1]...

Modern CPUs (AVX-512) load 8 doubles (64-bit) into a single wide register and add them all in ONE clock cycle.

**Impact:** Theoretical 8x speedup for math-heavy loops.

**Requirement:** Data must be contiguous in memory (Structure of Arrays vs Array of Structures).

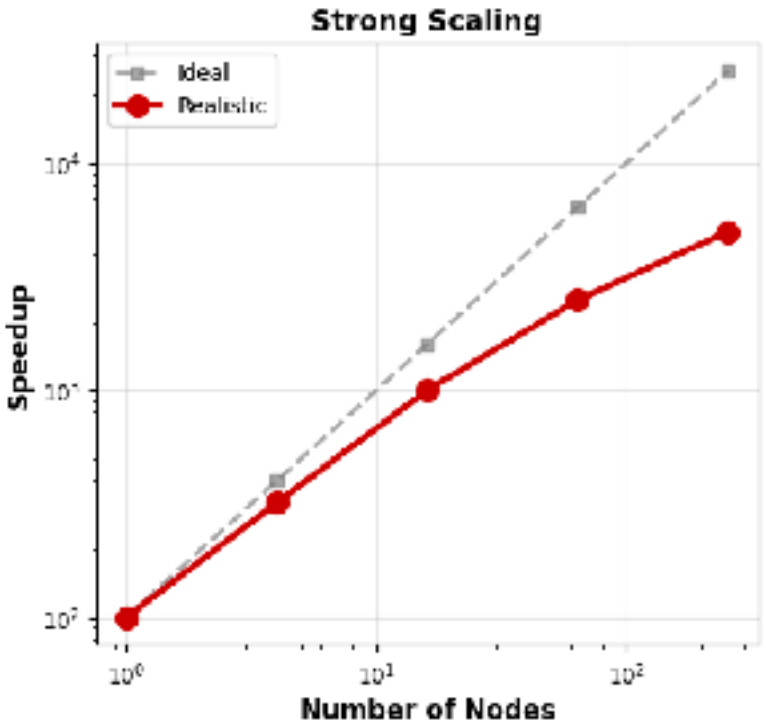
# What is Parallel Computing?

## The Shift to Parallel Computing

- Sequential: Single processor executes tasks serially
- Parallel: Multiple processors solve problem simultaneously
- Goal: Reduce total execution time
- Speedup = Time_sequential / Time_parallel
- Efficiency = Speedup / Number_of_processors
- Challenge: Communication overhead, load imbalance

## Scalability Limits: Amdahl's Law

- Formula: Speedup = 1 / ((1-P) + P/N)
- P = Parallel fraction, N = Number of processors
- Example: 95% parallel code on 8 cores → Speedup ≈ 4.6x
- Serial bottleneck limits maximum speedup
- For 100x speedup on 100 cores: Need 99% parallelism
- Lesson: Remove serial bottlenecks first before scaling



# Scaling tests of a parallel program

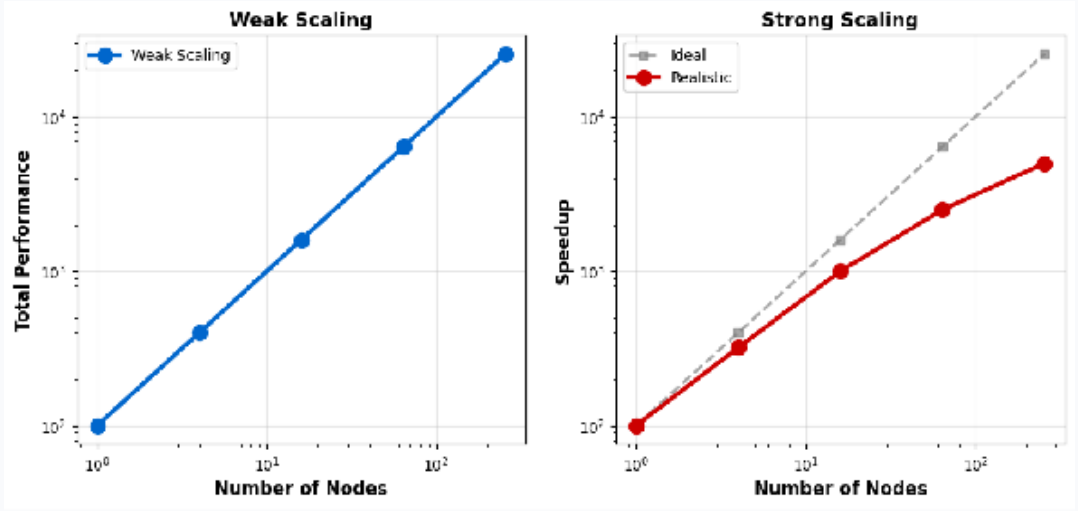
## Strong Scaling

- Fixed problem size, increase cores
- Metric: Time(1) / Time(N)
- Limited by: Communication overhead, dependencies

## Weak Scaling

- Problem grows with cores
- Metric: Maintain constant time as N increases
- More realistic for production workloads

Typical: 50-80% efficiency at 256 cores, drops at 1000+




Speedup & Efficiency Analysis

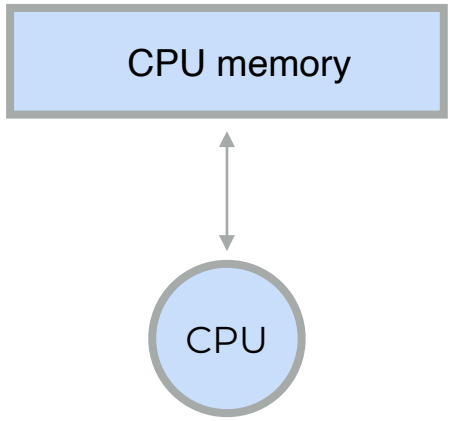
**Ideal speedup:** S(N) = N (linear), **Efficiency** = 100%: Reality: Communication reduces speedup

**Typical efficiency: E** = S(N)/N ≈ 70% at N=64; **Communication cost:** MPI allreduce, barriers, latency

**Load imbalance:** Fast processors wait for slow ones; **Solution:** Overlap computation and communication

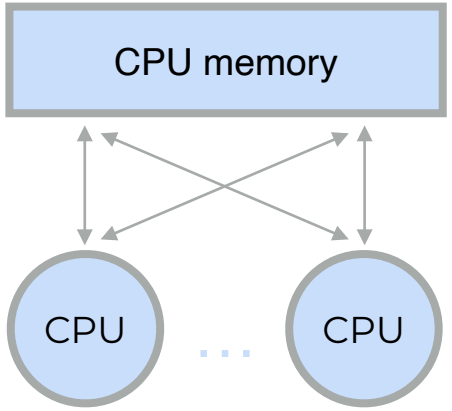
# Summary: System Evolution

Changes in the architecture level also requires changes in the programming models



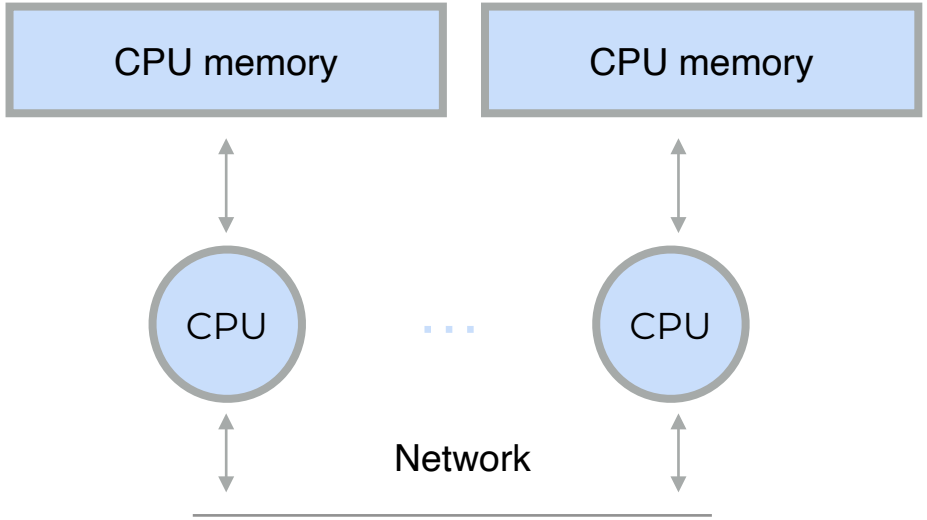

**Single CPU**

- Sequential computing
- Performance in Mflop/sec




**Multiple CPU**

- Shared memory
- Requires change in hardware and software architecture
- New standard: pThread or OpenMP
- Performance in Gflop/sec




**Multi-node system**

- Connected with high-speed and low latency network ( 200 GB/s via Infi band network)
- Requires change in hardware and software architecture
- New standard: MPI+OpenMP
- Performance in Tflop/sec

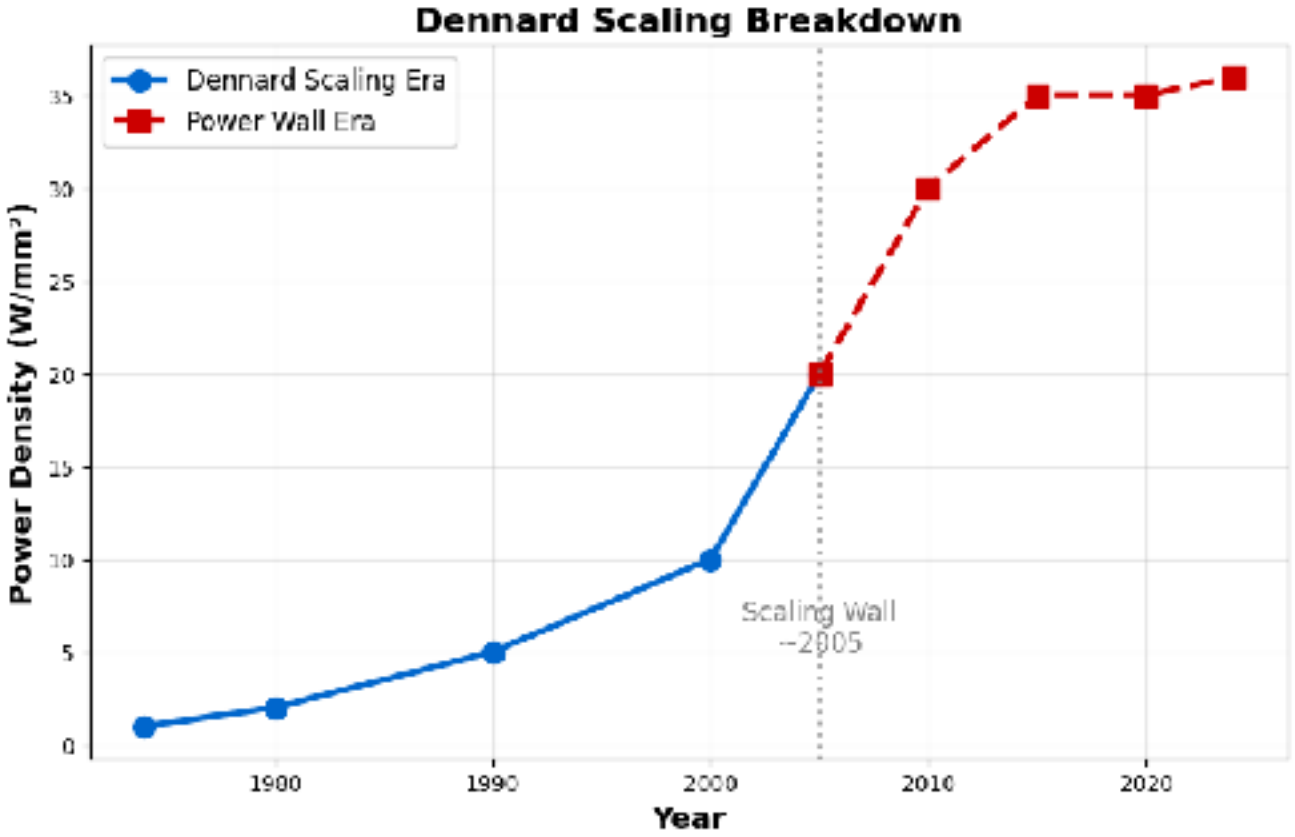
# The End of Frequency Scaling

**The “Power Wall”**
For decades, we got faster computers simply increasing the clock speed (MHZ -> GHZ)

- Heat: Chips become too hot to cool effectively
- Leakage: As transistors got smaller, power leakage increased
- Result: Single-core performance levelled off

**Gustafson's Law**

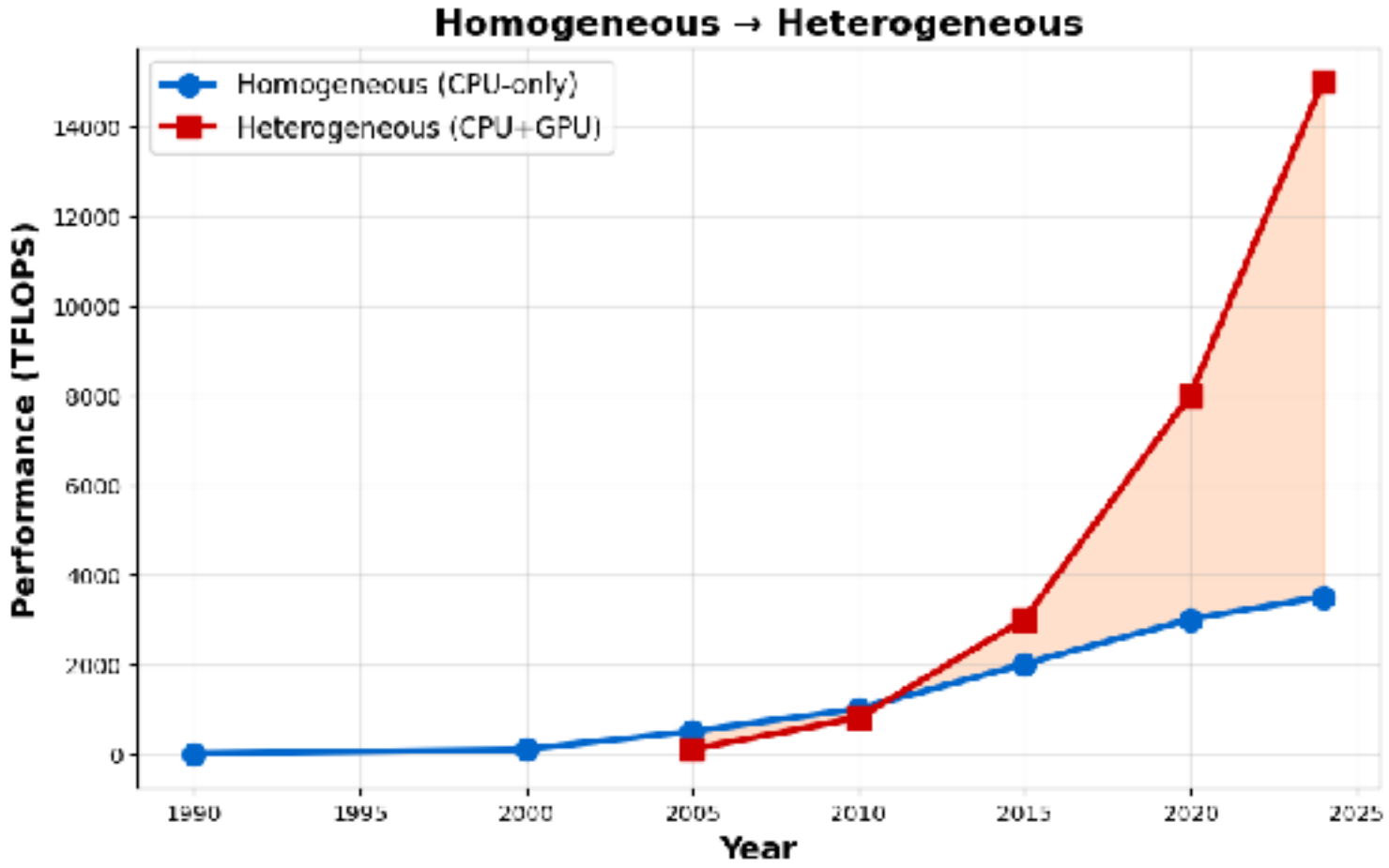
- Amdahl’s law assumes the problem size is fixed
- Gustafson's Law argues that as we get more computing power, we tackle larger problems
- The parallel part ($P$) grows, while the serial part ($S$) stays constant



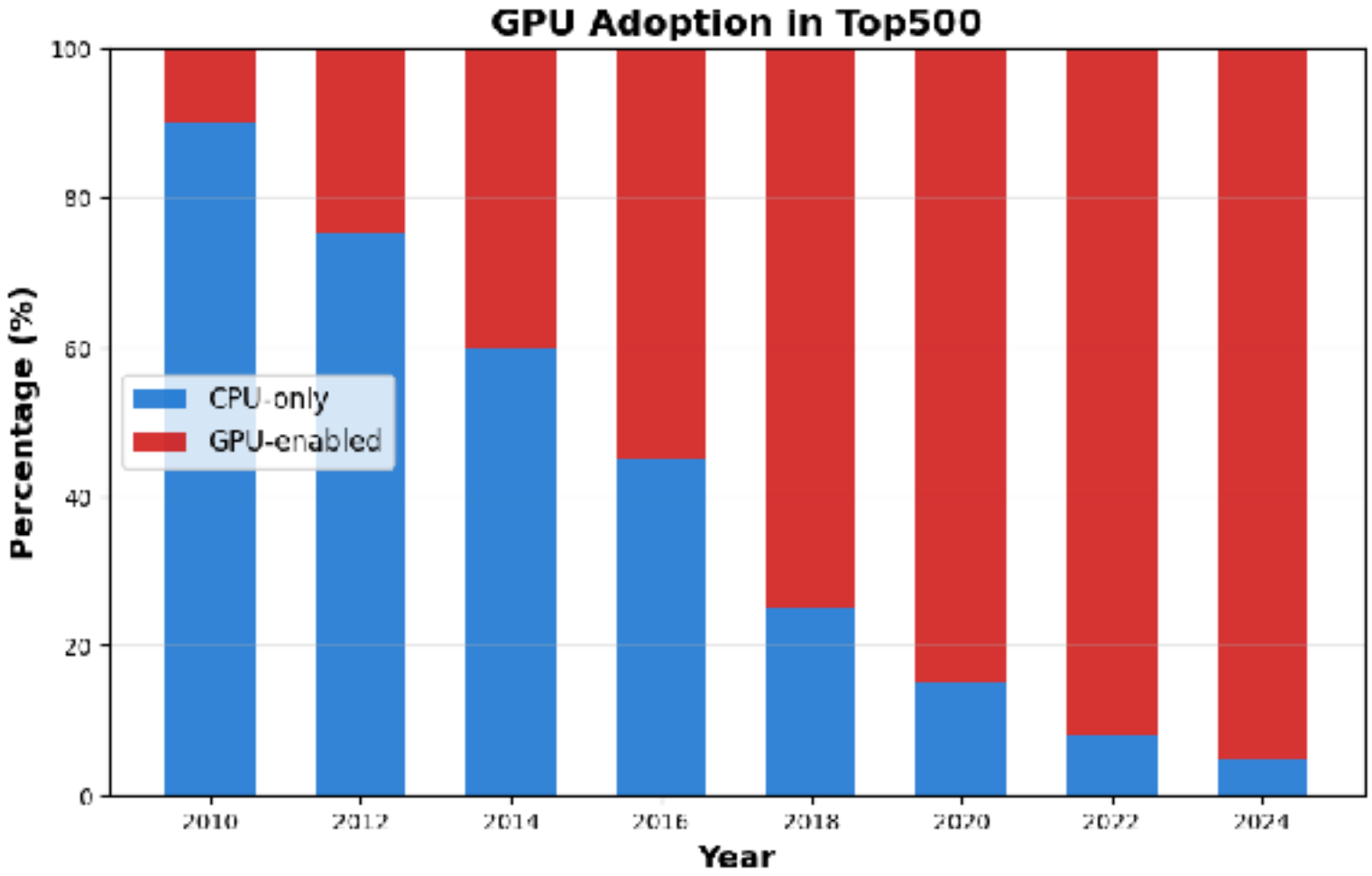
# The Rise of Heterogenous systems

### GPU Emergence (2007+)

- Nvidia CUDA launch (2007): First practical GPU computing
- Originally: Graphics cards (image processing)
- Key insight: Massive parallelism for simple operations
- Architecture: Thousands simple cores vs few complex
- Clock: 1-2 GHz (slower than CPUs)
- Throughput: 10-100x higher than CPU for suitable workloads



# The Rise of Heterogenous systems



# Why GPU Computing?

## Parallel Computing when Perf/Watt Matters

- **Data Growth:** Processing massive datasets (TB to PB scale)
- **Real-time Requirements:** Video processing, autonomous vehicles
- **Scientific Computing:** Climate models, molecular dynamics
- **AI/ML:** Training deep neural networks with billions of parameters
- **Financial Modeling:** Risk analysis, options pricing

**Performance Example**

| Processor | Cores | Performance |
|---|---|---|
| CPU (Intel i9) | 15 cores | ~500 GFLOPS |
| GPU (NVIDIA A10C) | 6,912 cores | ~19,500 GFLOPS |

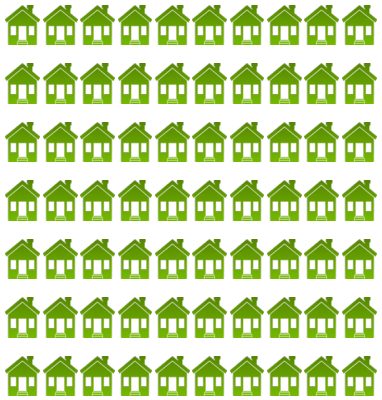
**GPU = 40x faster for parallel workloads!**

Traditional CPUs are not economically feasible

**2.3 PFlops**


**7.0 Megawatts**

**7000 homes**



**7.0 Megawatts**

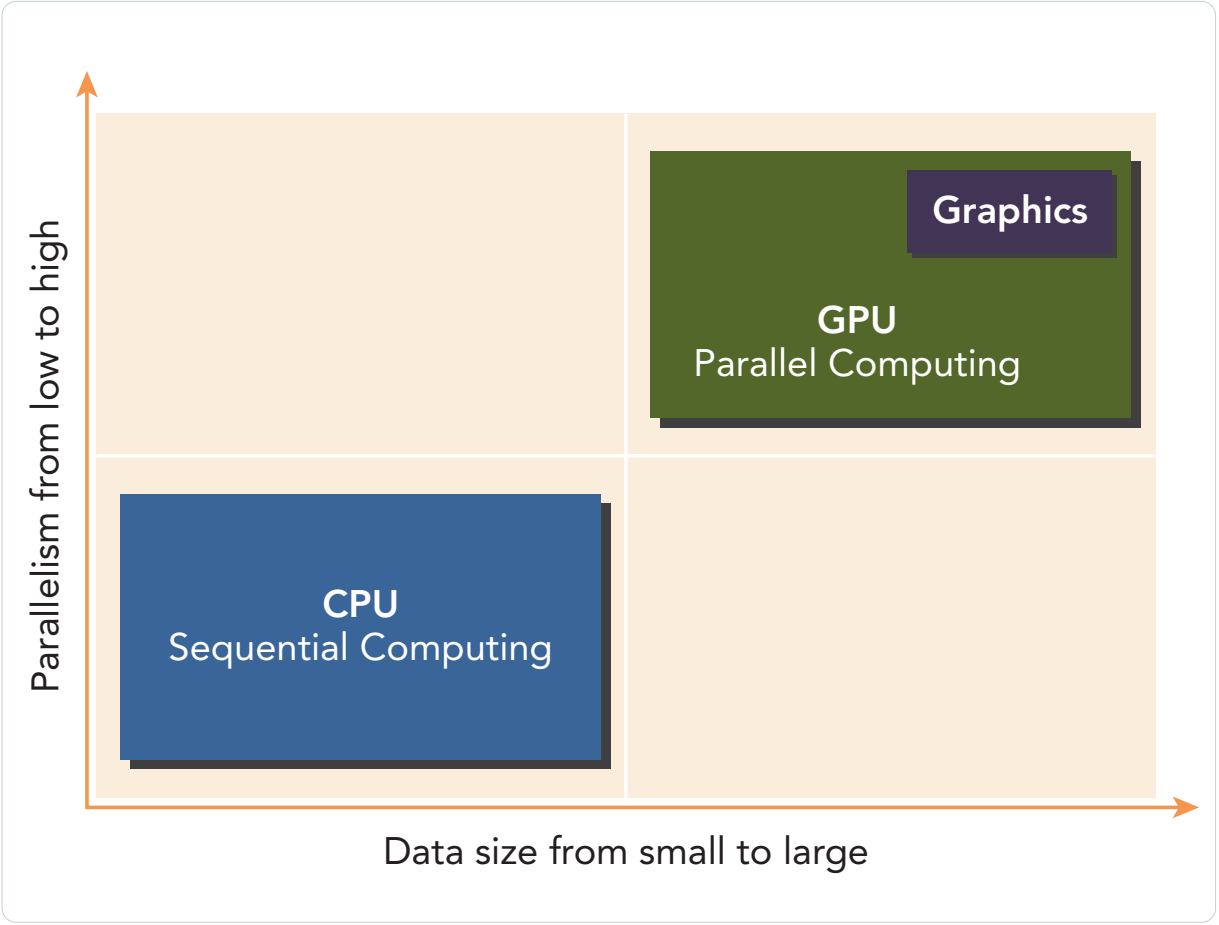
# Why Heterogenous Computing?

## Different tasks need different hardware

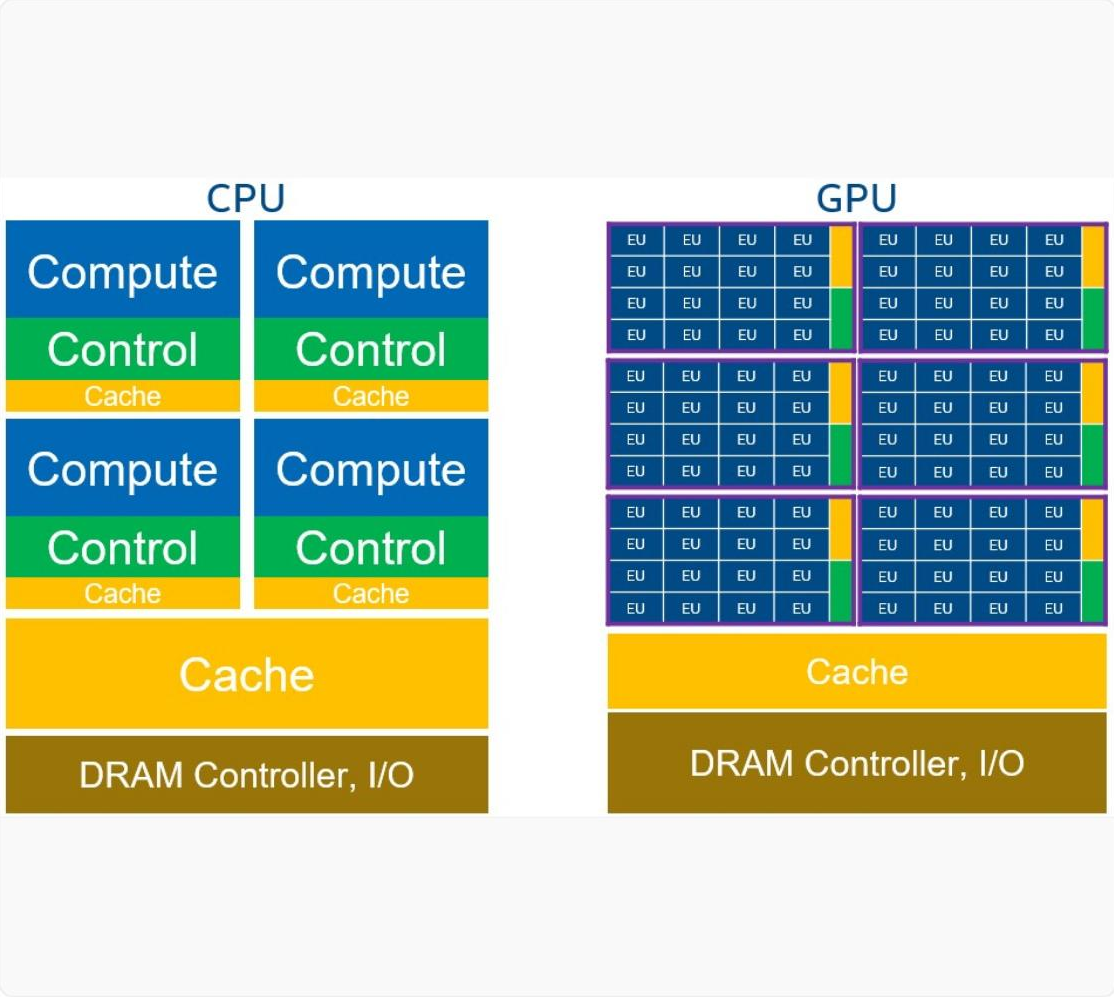
- Sequential control → CPU
- Massive parallelism → GPU
- Energy efficiency: GPUs deliver more GFLOPS/Watt
- Cost: GPUs amortize across many parallel tasks

## Fundamentals types of parallelism

- **Task parallelisms:** multiple independent tasks can run simultaneously, distributing functions across multiple cores
- **Data parallelisms:** multiple data items can be processed simultaneously, distributing the data across multiple cores



# CPU vs. GPU: Silicon Budget




**Where do the transistors go?**

- **CPU (Latency):** Huge caches (L1/L2/L3) and complex Control Units (Branch Prediction, Out-of-Order execution). Small area for math (ALU).
- **GPU (Throughput):** Almost entire chip is ALUs (Green). Tiny caches. Simple control (SIMT).

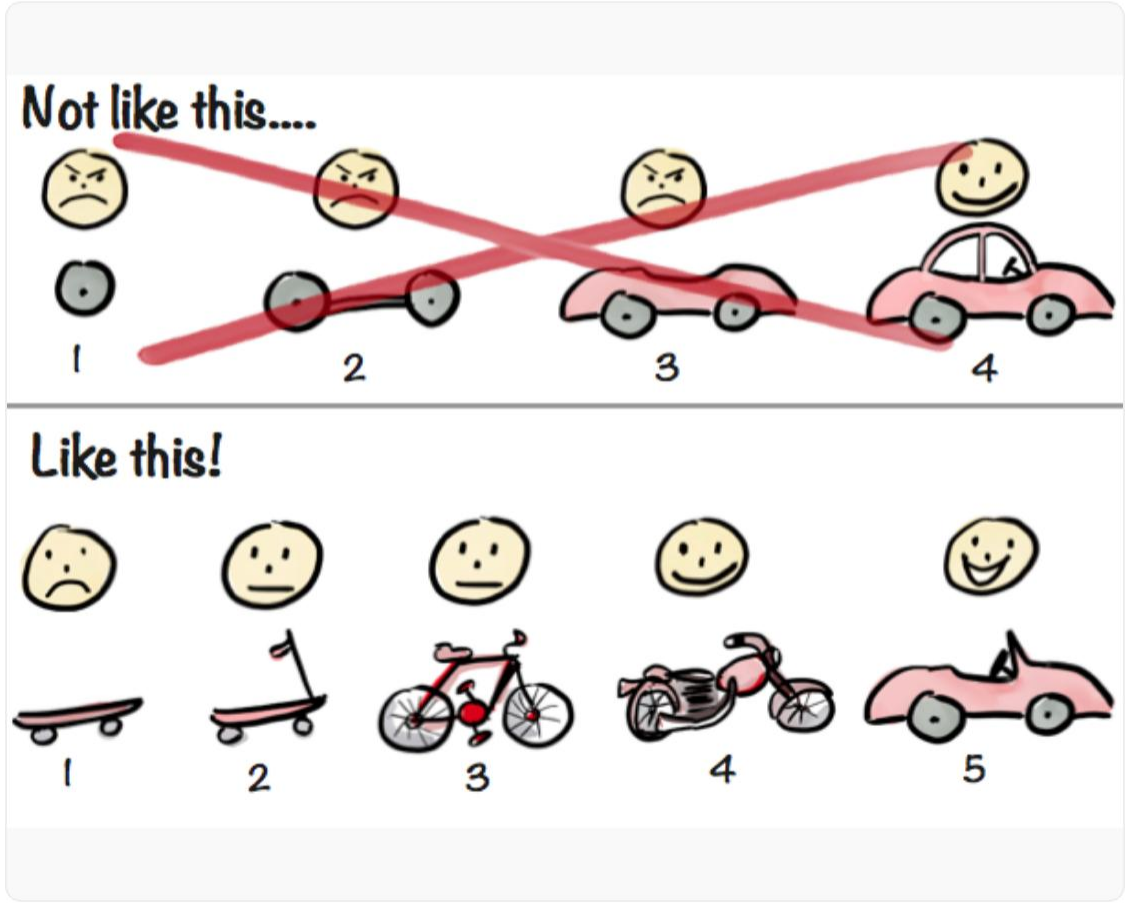
**"The GPU devotes more transistors to data processing than data caching."**

# Latency vs. Throughput

**The Race Car vs. The Bus**

- **CPU is a Ferrari:** It moves one person (task) extremely fast. Low latency.
- **GPU is a City Bus:** It is slow (lower clock speed), but it moves 50 people at once. High throughput.

**Implication:** If you have 1 task, use CPU. If you have 1 million tasks, use GPU.

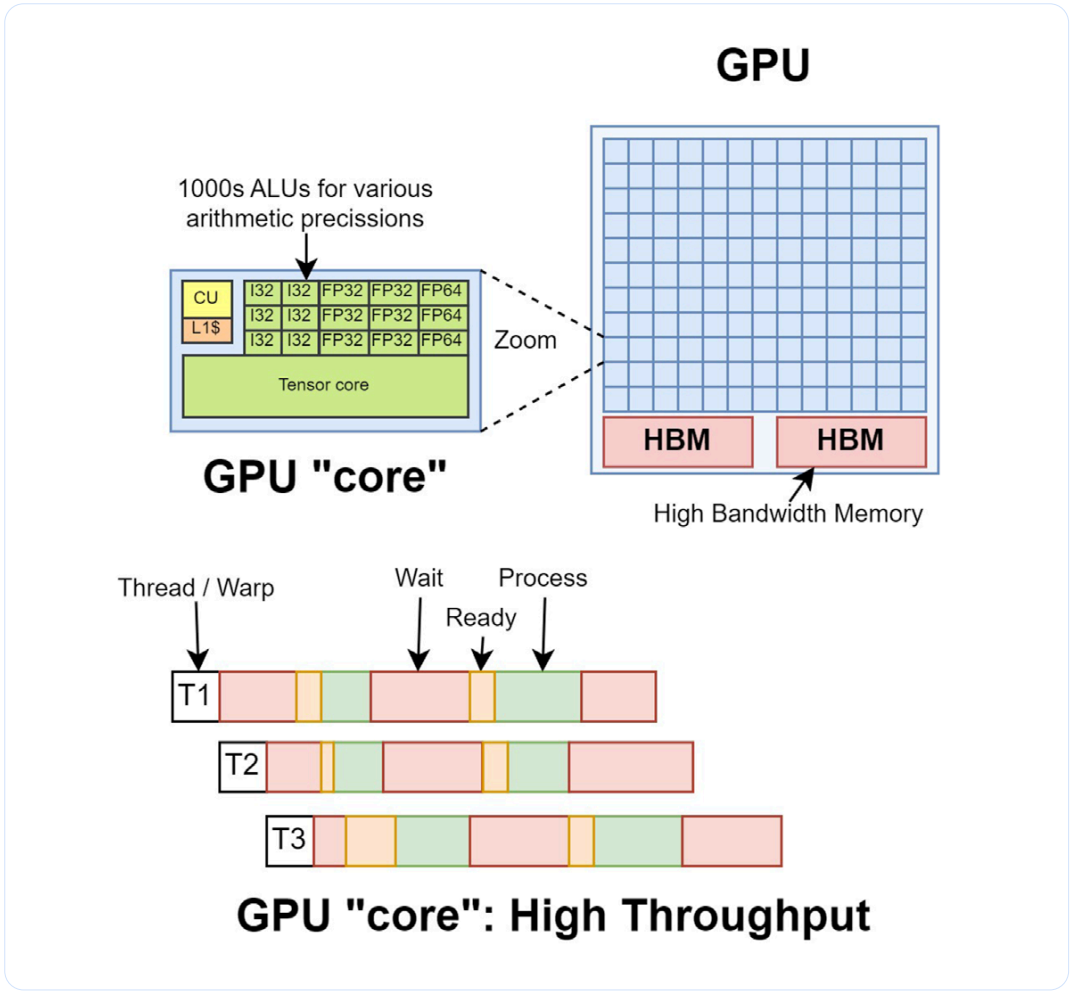


# GPU Design Philosophy

## Maximizing Throughput

GPUs execute thousands of threads slowly, but simultaneously.

- **Throughput Oriented:** Focus on total work done per second.
- **Latency Hiding:** Instead of big caches, GPUs instantly switch to other threads when one is waiting for memory.
- **SIMT (Single Instruction, Multiple Threads):** One instruction controls many functional units.



# Comparison: Cores

## CPU Cores

- **Count:** Few (4 - 128).
- **Strength:** Heavy, complex, high frequency (3-5 GHz).
- **Task:** Logic, OS, IO, serial math.

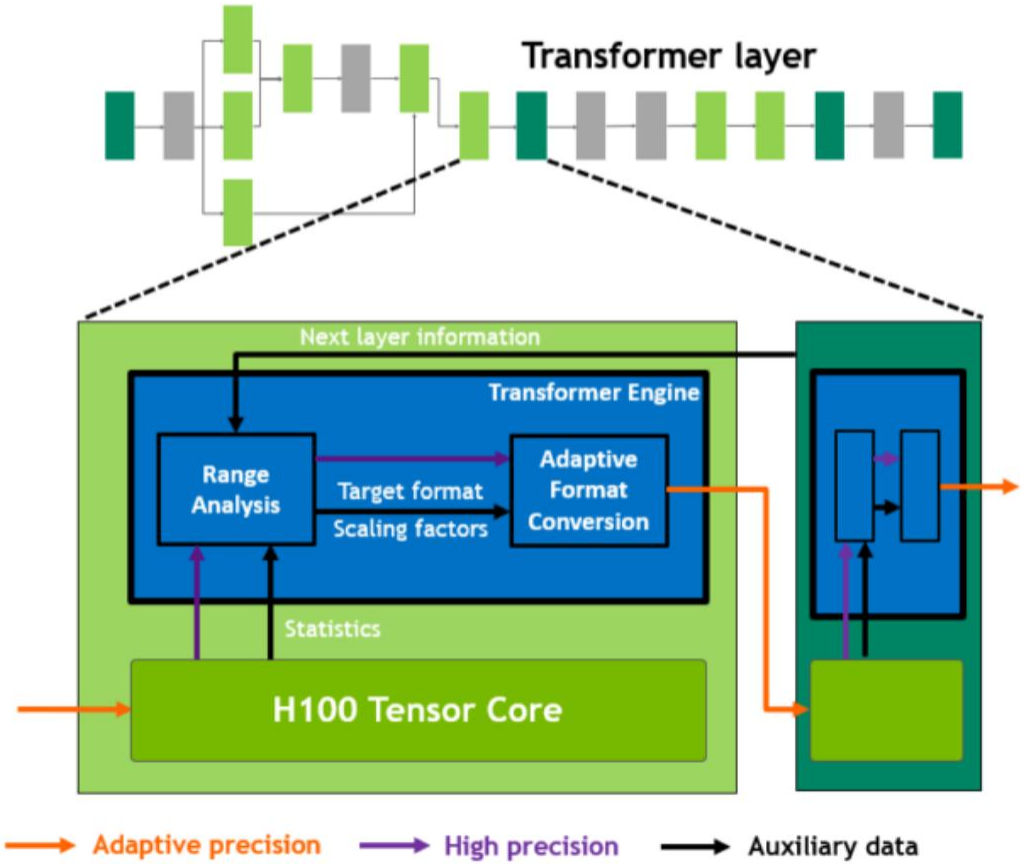
## GPU Cores

- **Count:** Many (1,000 - 18,000).
- **Strength:** Lightweight, simple, lower frequency (1-2 GHz).
- **Task:** Parallel math (FP32, INT32).

# NVIDIA GPU Architecture

A modern NVIDIA GPU (e.g., H100) is a hierarchical system.

- **GPC (Graphics Processing Cluster):** High-level grouping of resources.
- **SM (Streaming Multiprocessor):** The main building block.
- **Memory Controllers:** Interface with HBM memory stacks.

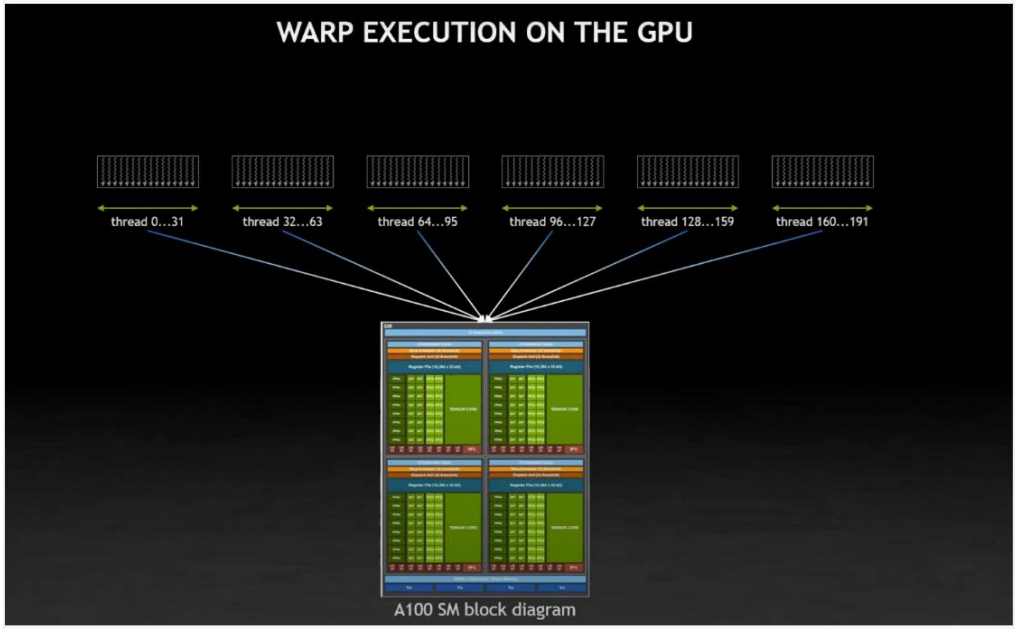


# The Streaming Multiprocessor (SM)

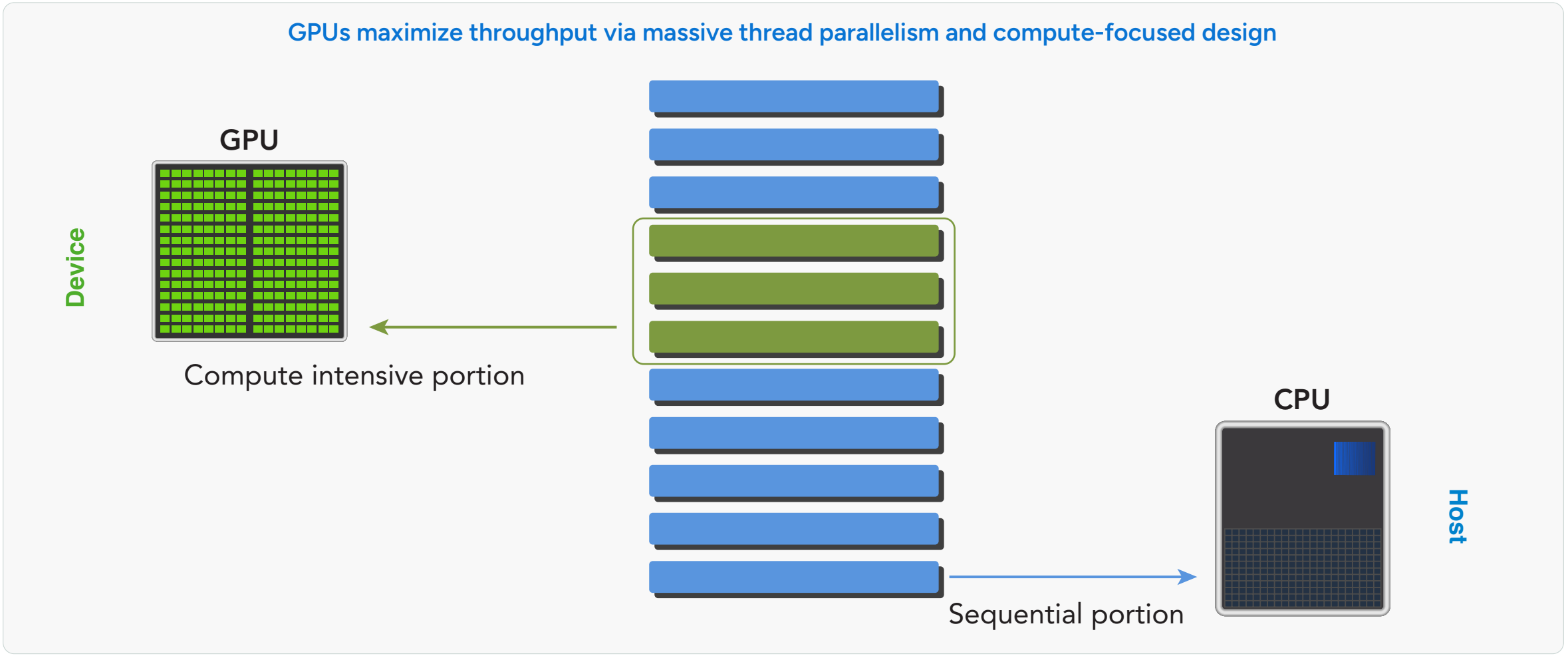
The GPU is built from building blocks called **Streaming Multiprocessors (SMs)**. A modern H100 GPU has 144+ SMs.

### Inside an SM:

- **CUDA Cores:** INT32, FP32, FP64 units.
- **Tensor Cores:** Specialized matrix math (AI).
- **RT Cores:** Ray Tracing units.
- **Shared Memory:** Fast, manual cache (L1).
- **Register File:** Massive fast storage for threads.



# GPUs serve as a co-processor, not a standalone platform



# Heterogenous Computing

**Host & Device**

- **Host (CPU):** The "Manager". Runs the OS, serial code, and orchestrates the GPU.
- **Device (GPU):** The "Worker". Executes parallel tasks (kernels) launched by the host.

The system memory (RAM) and device memory (VRAM) are physically separate (usually) and connected via PCIe.

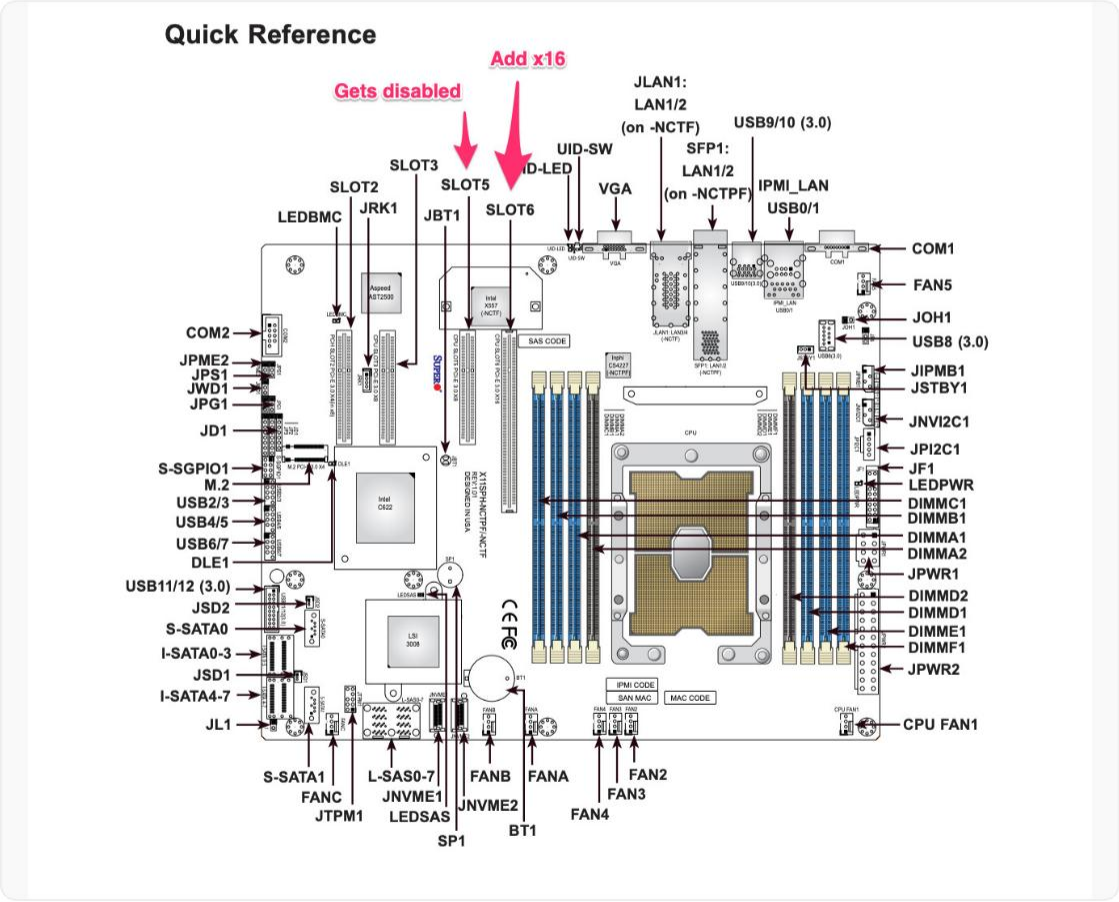
**PCIe Bus**

The bridge between Host and Device.

**Bottleneck Alert!**

Transfer speed: ~32-64 GB/s.
Compute Speed: ~2000 GB/s.

# The Physical Connection




## PCI Express (PCIe)

GPUs plug into x16 slots. This serial link is the lifeline for data.

- **Gen 3:** ~16 GB/s
- **Gen 4:** ~32 GB/s
- **Gen 5:** ~64 GB/s

**Optimization Tip:** Minimizing data transfer across this bus is the #1 rule of CUDA optimization.

# Scale-UP VS. Scale-Out

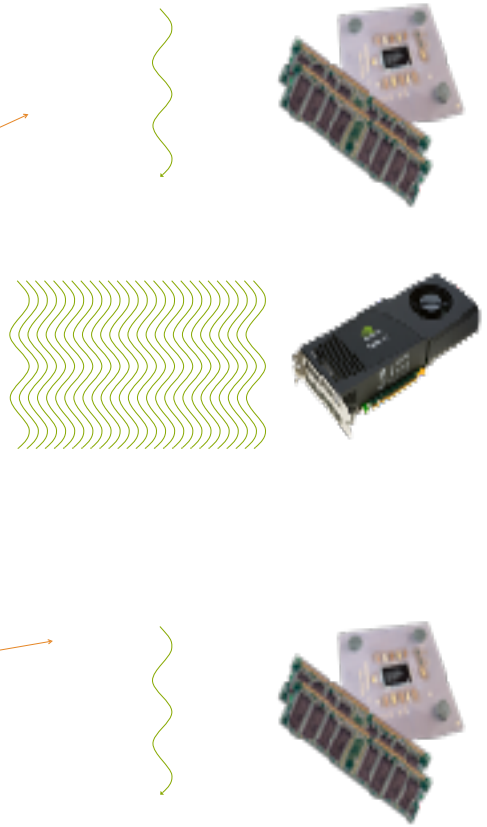
```
#include <iostream>
#include <algorithm>

using namespace std;

#define N          1024
#define RADIUS     3
#define BLOCK_SIZE 16

__global__ void stencil_1d(int *in, int *out) {
        __shared__ int temp[BLOCK_SIZE + 2 * RADIUS];
        int gindex = threadIdx.x + blockIdx.x * blockDim.x;
        int lindex = threadIdx.x + RADIUS;

        // Read input elements into shared memory
        temp[lindex] = in[gindex];
        if (threadIdx.x < RADIUS) {
                temp[lindex - RADIUS] = in[gindex - RADIUS];
                temp[lindex + BLOCK_SIZE] = in[gindex + BLOCK_SIZE];
        }

        // Synchronize (ensure all the data is available)
        __syncthreads();

        // Apply the stencil
        int result = 0;
        for (int offset = -RADIUS ; offset <= RADIUS ; offset++)
                result += temp[lindex + offset];

        // Store the result
        out[gindex] = result;
}

void fill_ints(int *x, int n) {
        fill_n(x, n, 1);
}

int main(void) {
        int *in, *out;          // host copies of a, b, c
        int *d_in, *d_out;      // device copies of a, b, c
        int size = (N + 2*RADIUS) * sizeof(int);

        // Alloc space for host copies and setup values
        in  = (int *)malloc(size); fill_ints(in,  N + 2*RADIUS);
        out = (int *)malloc(size); fill_ints(out, N + 2*RADIUS);

        // Alloc space for device copies
        cudaMalloc((void **)&d_in,  size);
        cudaMalloc((void **)&d_out, size);

        // Copy to device
        cudaMemcpy(d_in,  in,  size, cudaMemcpyHostToDevice);
        cudaMemcpy(d_out, out, size, cudaMemcpyHostToDevice);

        // Launch stencil_1d() kernel on GPU
        stencil_1d<<<N/BLOCK_SIZE,BLOCK_SIZE>>>(d_in + RADIUS,
d_out + RADIUS);

        // Copy result back to host
        cudaMemcpy(out, d_out, size, cudaMemcpyDeviceToHost);

        // Cleanup
        free(in); free(out);
        cudaFree(d_in); cudaFree(d_out);
        return 0;
}
```

parallel fn

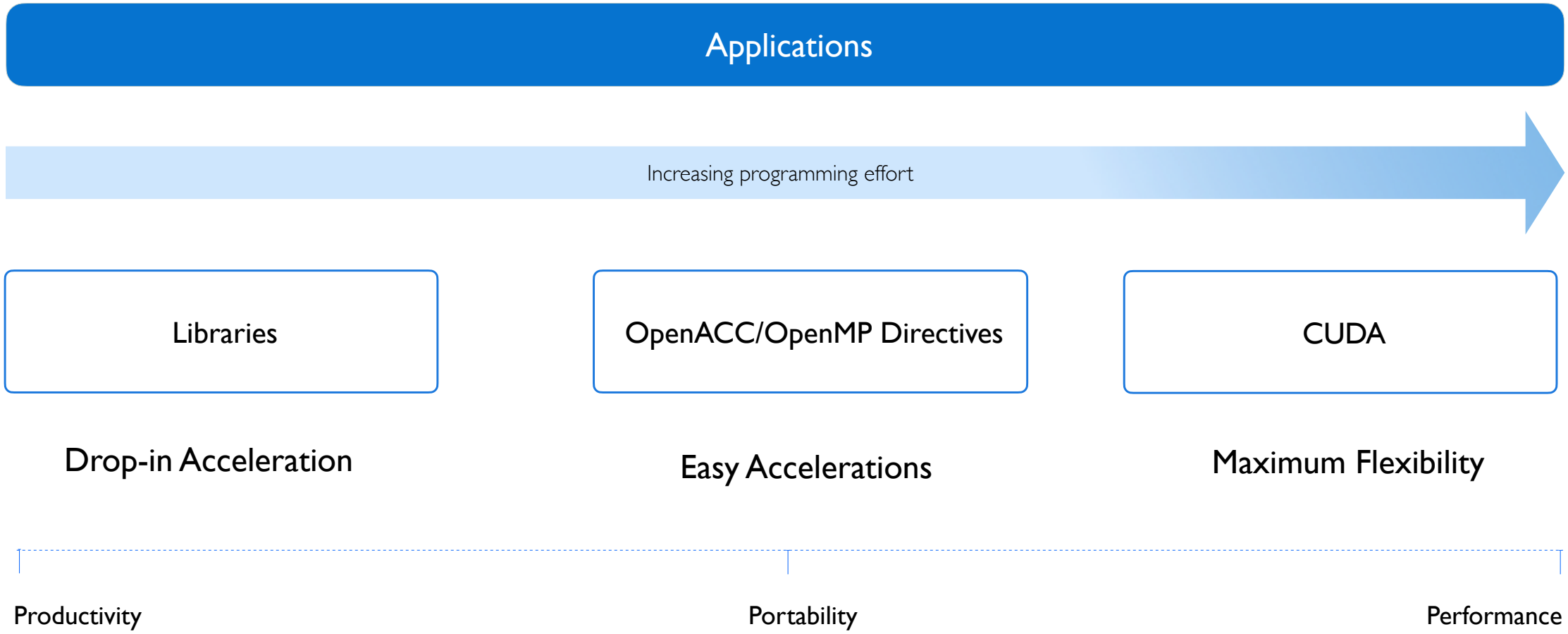
serial code

parallel code

serial code



# Ways to parallels an applications on Nvidia GPUs



# Accelerating C/C++ Applications with NVIDIA GPU Programming Using OpenX

# Topics explored

1

Why OpenACC

Low learning and maximum gain

2

OpenACC directives

Parallelising and profiling with OpenACC Toolkit on NVIDIA GPUs

3

Data Management

Explicit data management, Data regions and clause, Unstructured Data Lifetimes

4

Loop Optimisation

Overlapping kernel execution & data transfer on Single/Multi GPU

0

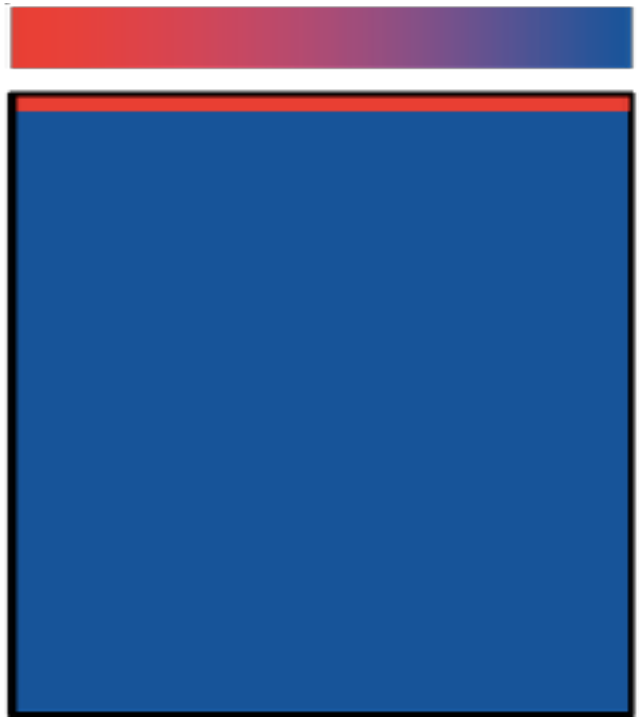
# Case study: Parallelise a serial Laplace 2D

# Laplace Heat Transfer

We will observe a simple simulation of heat distributing across a metal plate.

We will apply a consistent heat to the top of the plate.

Then, we will simulate the heat distributing across the plate.

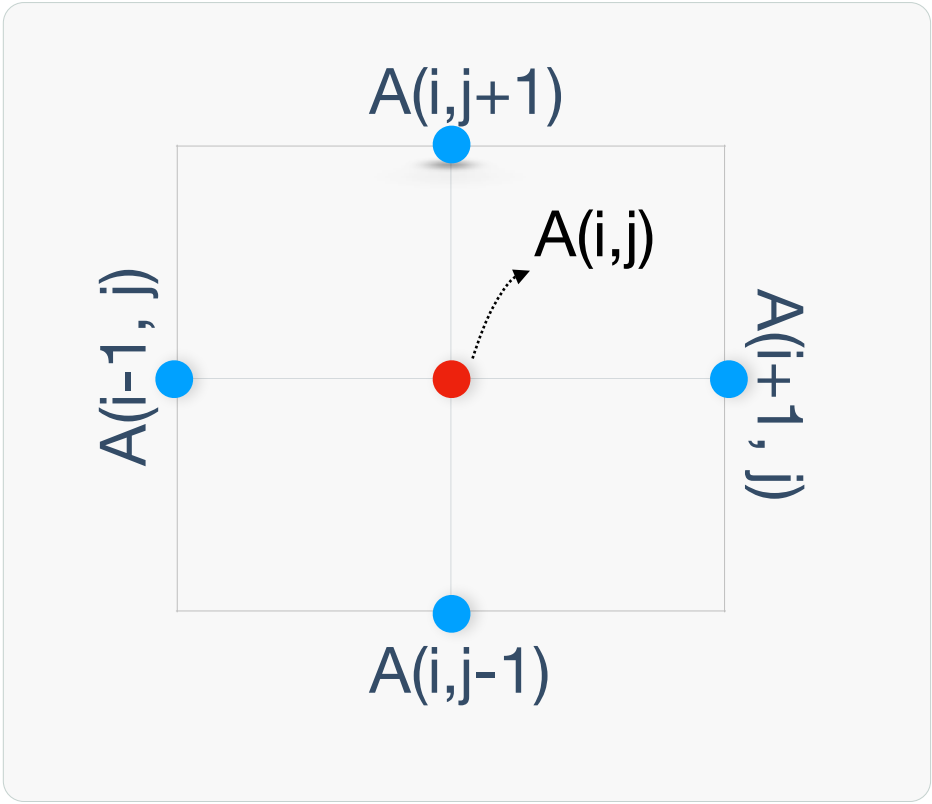


# Code description

- Iteratively converges to correct value (e.g. Temperature)
- by computing new values at each point from the average of neighboring points.
- Example: Solve Laplace equation in 2D

$$\nabla^2 f(x,y) = 0$$

$$A_{k+1}(i,j) = \frac{A_k(i-1,j) + A_k(i+1,j) + A_k(i,j-1) + A_k(i,j+1}{4}$$



# Code description

```
while (error > tol && niter < niter_max)
error = 0.0;

  for (int j = 1; j < n-1; ++j) {
    for (int i = 1; i < m-1; ++i) {
      Anew[idx] = 0.25 * ( A[idx+1] + A[idx-1] + A[idx-m] + A[idx+m]);

      error = fmax(error, fabs(Anew[idx] - A[idx])); }
      }
  for (int j = 1; j < n-1; ++j)
    for (int i = 1; i < m-1; ++i)
      A[j][i] = Anew[j][i]
```

Iterate until converged

Iterate cross matrix elements

Compute new quantity

Compute max error for converge

Swap input/output arrays

# Compiling code with NVHPC

# Prerequisites & Set up

- **Programming Language:** Basic comfort with C/C++ or Fortran.
- **Concepts:** Loops, Arrays, Functions, Pointers.
- **Environment:**
  - NVIDIA HPC SDK (contains `nvc`, `nvc++`, `nvfortran`).
  - Supported GPU (NVIDIA Tesla, Quadro, GeForce).
  - Linux Environment (Recommended).

```
# Verify installation
$ nvc --version
$ nvidia-smi
```

# NVIDIA's HPC Compilers (AKA PGI)

Instruct compiler to print feedback about the compiled code

- -Minfo = accel informs about what parts of the code were accelerated via OpenX
- -Minfo = opt informs about all code optimisations
- -Minfo=all gives all code feedback, whether positive or negative

```
nvc -fast my_program.c
nvc++ -fast  my_program.cpp
nvfortran -fast  my_program.cf90
```

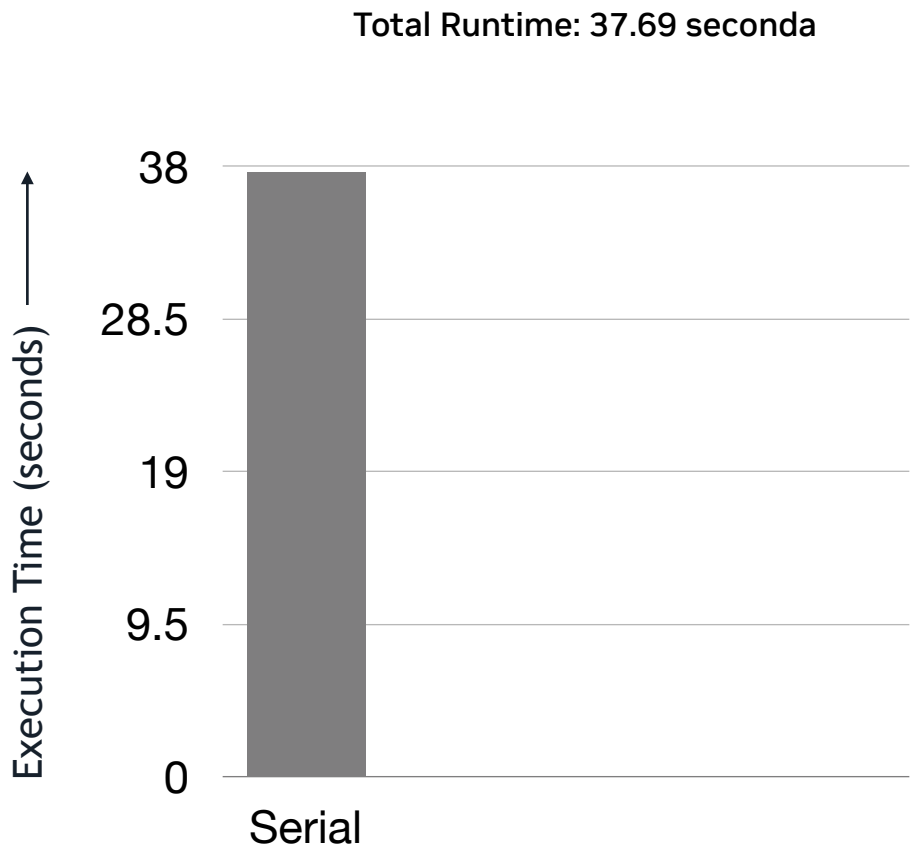
# NVIDIA's HPC Compilers (AKA PGI)

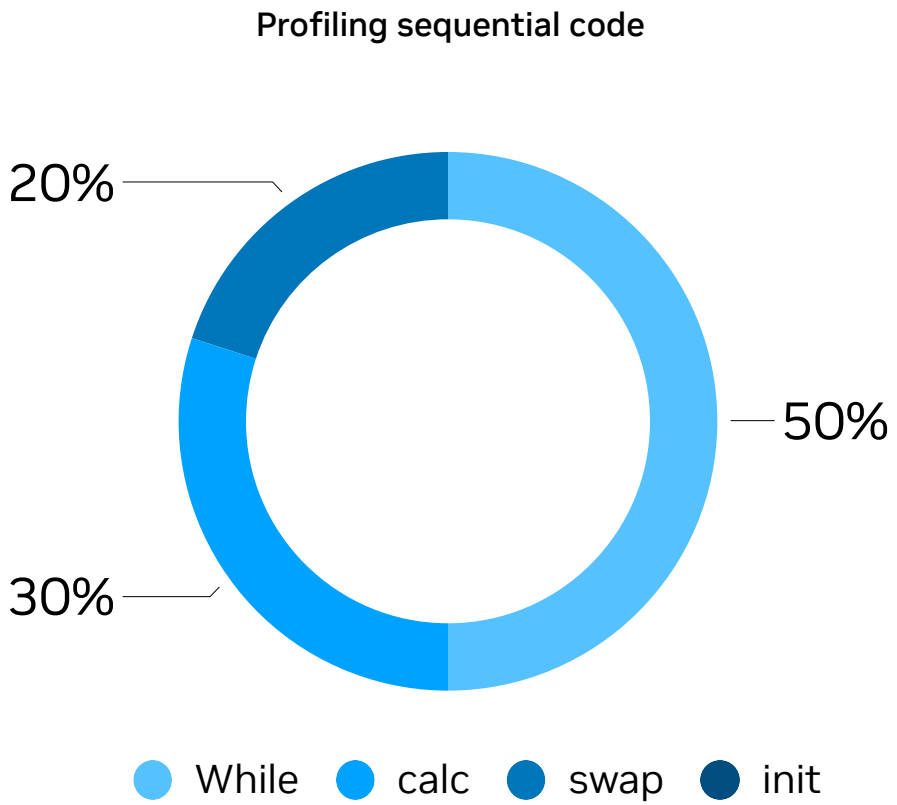
Instruct compiler to print feedback about the compiled code

- -Minfo = accel informs about what parts of the code were accelerated via OpenX
- -Minfo = opt informs about all code optimisations
- -Minfo=all gives all code feedback, whether positive or negative

```
nvc -fast -Minfo=all my_program.c
nvc++ -fast -Minfo=all  my_program.cpp
nvfortran -fast -Minfo=all  my_program.cf90
```

# Simulation was performed 1000 Iterations

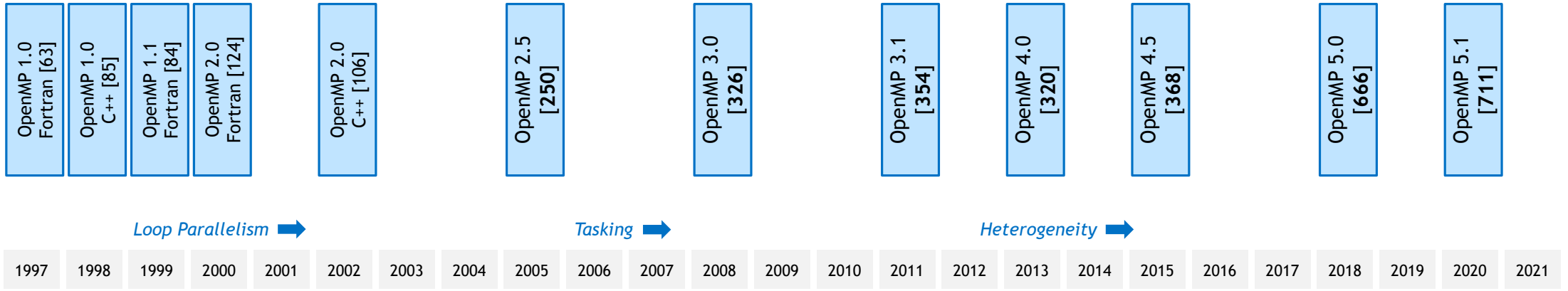


Are you familiar with directive based parallelism?

# OpenX (X = OMP, ACC): 1997-2021

Looking at TIME (pace of innovation) and SPACE (specification length)

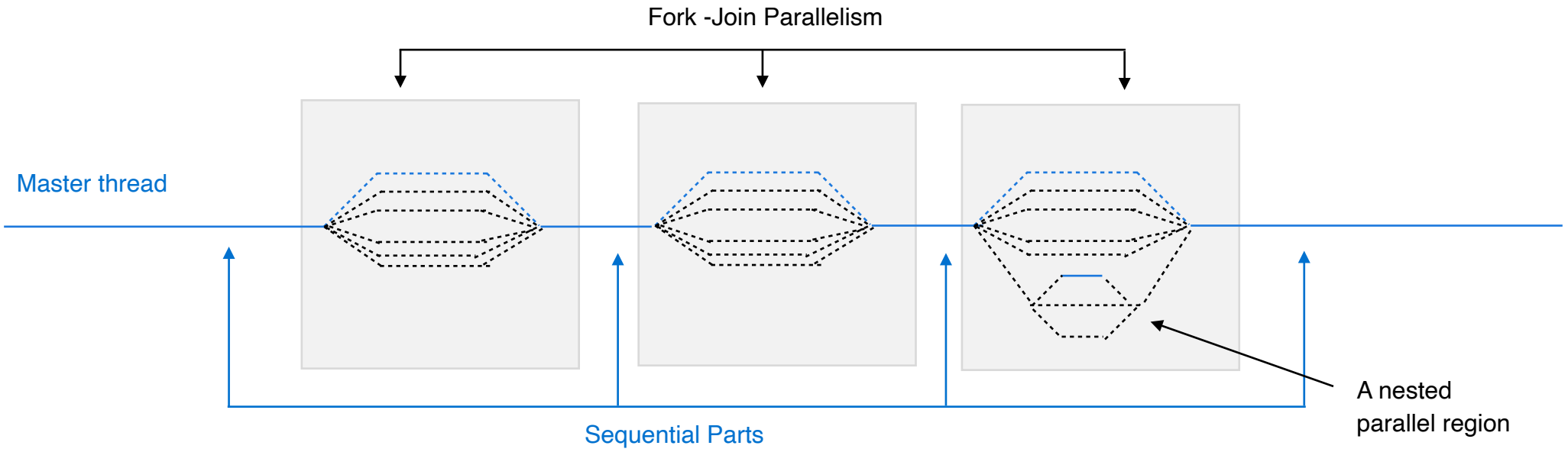



**Members of OpenMP ARB = 33**

# Revisit: OpenMP Application Program Interface (API)

Allows programmers to develop threaded parallel codes on shared memory computational units

- Directives are understood by OpenMP aware compilers (others are free to ignore)
- Generates parallel threaded code
  - Original thread becomes thread “0”
  - Share resources of the original thread (or rank)
  - Data-sharing attributes of variables can be specified based on usage patterns

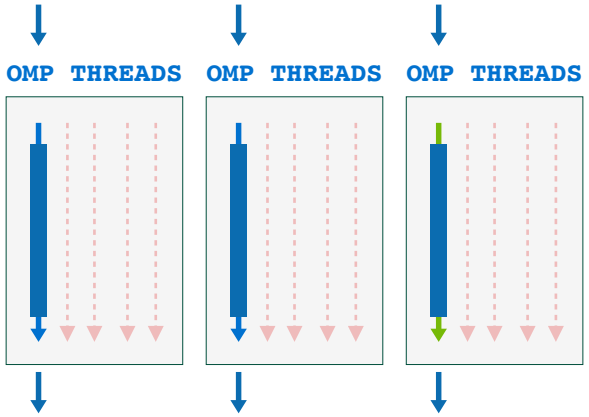


# Revisit: OpenMP Application Program Interface (API)

Allows programmers to develop threaded parallel codes on shared memory computational units

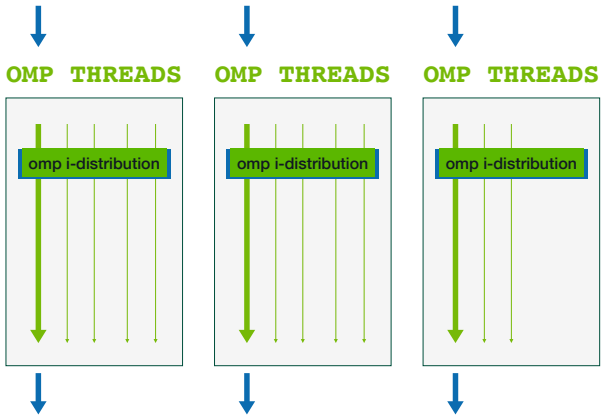
- Creates a team of OpenMP threads that execute the structured-block that follows
- Number of threads property is generally specified by OMP_NUM_THREADS

*#pragma omp parallel*




All threads will execute the region

*#pragma omp parallel for*




All threads will execute a part of the iterations

# Revisit: OpenMP Application Program Interface (API)

## Serial

```
for (int i = 0; i < N; ++i)
{
  C[i] = A[i] + B[i];
}
```

- 1 thread/process will execute each iteration sequentially
- Total time = time_for_single_iteration * N

## Parallel

```
#pragma omp parallel
for (int i = 0; i < N; ++i)
{
  C[i] = A[i] + B[i];
}
```

- Say, OMP_NUM_THREADS = 4
- 4 threads will execute each iteration redundantly (overwriting values of C)
- Total time = time_for_single_iteration * N

## Parallel worksharing

```
#pragma omp parallel for
for (int i = 0; i < N; ++i)
{
  C[i] = A[i] + B[i];
}
```

- Say, OMP_NUM_THREADS = 4
- 4 threads will execute each iteration (roughly N/4 per thread)
- Total time = time_for_single_iteration * N/4

# Checkpoint-0:  00-laplace2d_serial: parallelize with OpenMP

# Exercise0 00-laplace2d_serial: parallelize with OpenMP

```
while ( error > tol && iter < iter_max ) {
  error=0.0;

for( int j = 1; j < n-1; j++) {
    for(int i = 1; i < m-1; i++) {
      Anew[j][i] = 0.25 * (A[j][i+1] + A[j][i-1] +
                   A[j-1][i] + A[j+1][i]);
      error = max(error, abs(Anew[j][i] - A[j][i]);
    }
  }

for( int j = 1; j < n-1; j++) {
    for( int i = 1; i < m-1; i++ ) {
      A[j][i] = Anew[j][i];
    }
  }
  iter++;
}
```

## Things to do

- Make these two loops parallel with OpenMP directives
- by changing the number of OpenMP threads, repeating the execution with: export OMP_NUM_THREADS = (1 to 8 )
- Do not forget to record the time

# NVIDIA's HPC Compilers (AKA PGI)

Building and running enabled OpenMP code

- nvc -mp -fast -Minfo=all my_program.c -o bin
- nvc++ -mp  -fast -Minfo=all my_program.cpp. -o bin
- nvfortran -mp  -fast -Minfo=all my_program.cf90.  -o bin

# Solution

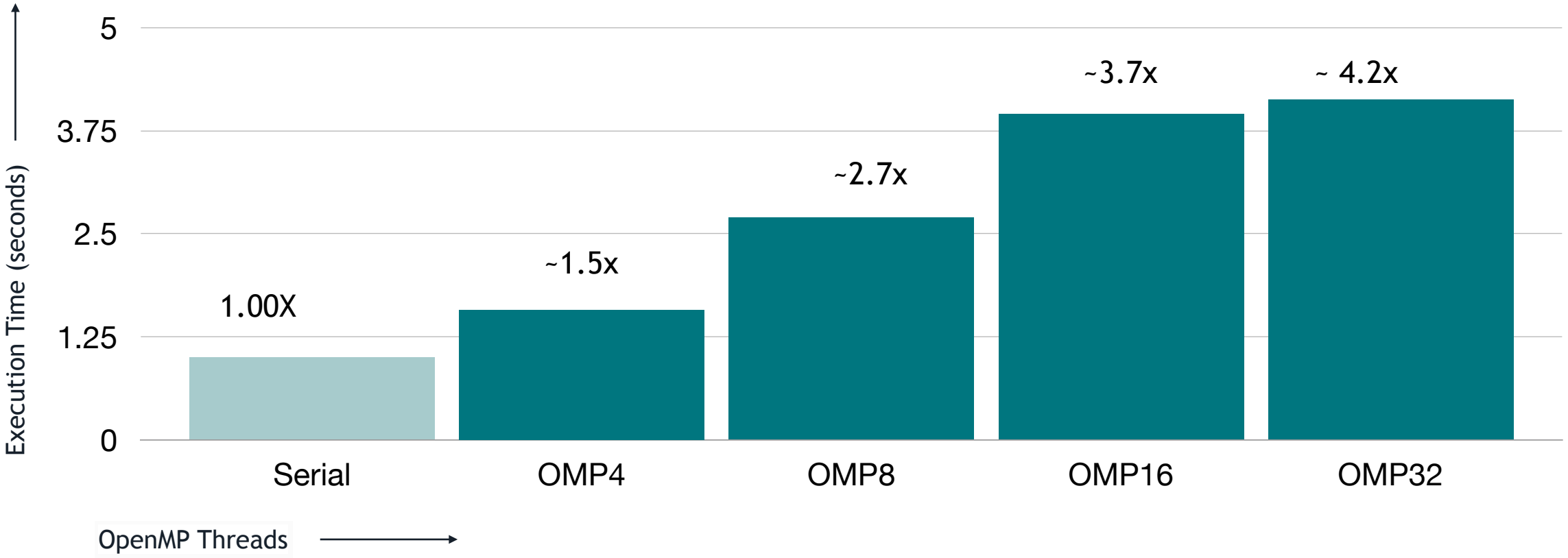
# Solution0: parallelize with OpenMP

```
while ( error > tol && iter < iter_max ) {
  error=0.0;
#pragma omp parallel for shared(m, n, Anew, A) reduction(max:error)
  for( int j = 1; j < n-1; j++) {
    for(int i = 1; i < m-1; i++) {
      Anew[j][i] = 0.25 * (A[j][i+1] + A[j][i-1] +
                   A[j-1][i] + A[j+1][i]);
      error = max(error, abs(Anew[j][i] - A[j][i]);
    }
  }
#pragma omp parallel for shared(m, n, Anew, A)
  for( int j = 1; j < n-1; j++) {
    for( int i = 1; i < m-1; i++ ) {
      A[j][i] = Anew[j][i];
    }
  }
  iter++;
}
```

# Solution0: parallelize with OpenMP

```
while ( error > tol && iter < iter_max ) {
  error=0.0;
#pragma omp parallel for collapse(2) shared(m, n, Anew, A) reduction(max:error)
  for( int j = 1; j < n-1; j++) {
    for(int i = 1; i < m-1; i++) {
      Anew[j][i] = 0.25 * (A[j][i+1] + A[j][i-1] +
                   A[j-1][i] + A[j+1][i]);
      error = max(error, abs(Anew[j][i] - A[j][i]);
    }
  }
#pragma omp parallel for collapse(2) shared(m, n, Anew, A)
  for( int j = 1; j < n-1; j++) {
    for( int i = 1; i < m-1; i++ ) {
      A[j][i] = Anew[j][i];
    }
  }
  iter++;
}
```

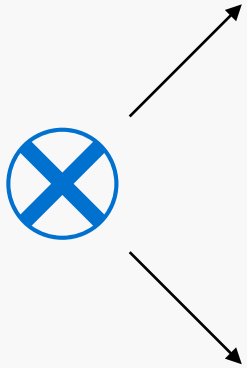
# Performance speed up (higher is better)



Will your legacy OpenMP code perform well on the GPU?

# What is OpenX offloadings?

# What is OpenX?



OpenACC — Main focus is to target to offloading code onto GPUs

**Designed for performance and portability**

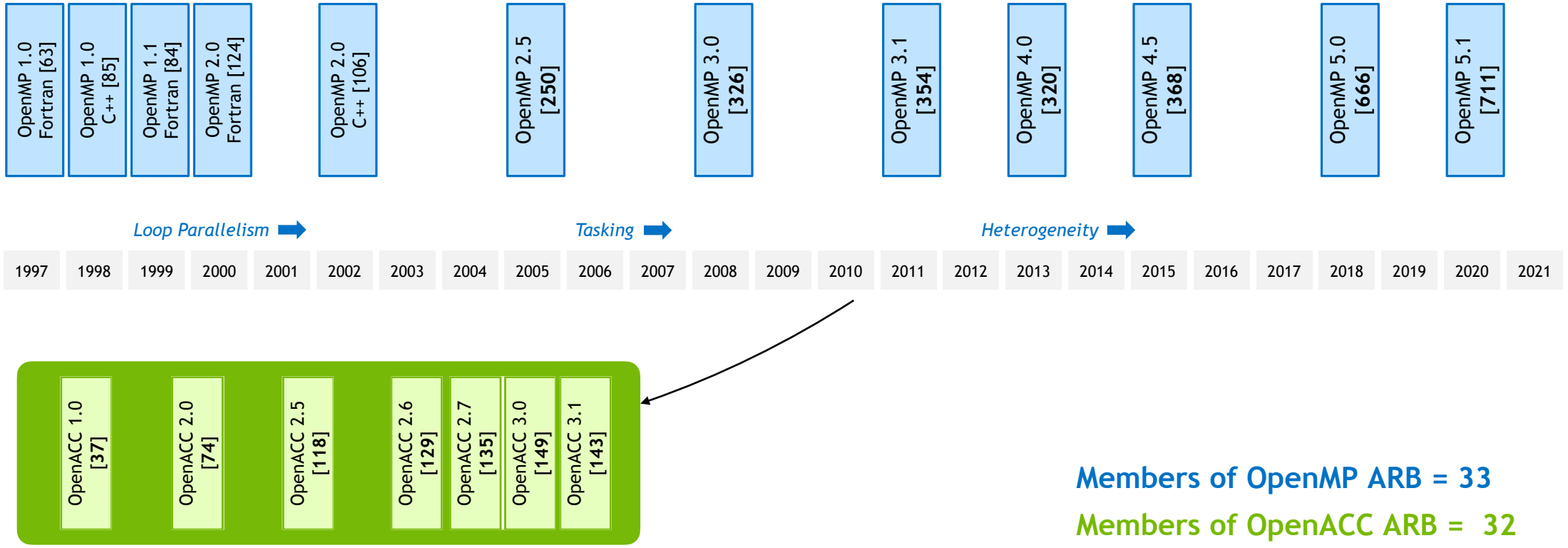
OpenMP — from v 4.0 allows offloading of tasks onto GPUs

**IMPORTANT**

This course will not give you any concrete hints if OpenMP is better than OpenACC

# OpenX (X = OMP, ACC): 1997-2021

Looking at TIME (pace of innovation) and SPACE (specification length)




**Members of OpenMP ARB = 33**

**Members of OpenACC ARB = 32**

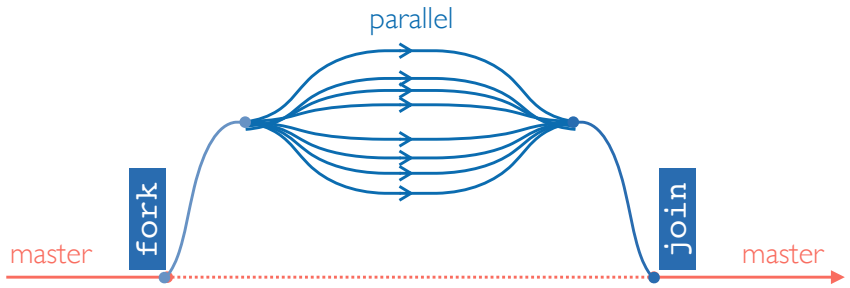
# Same basic principle: Fork-Join model

## OpenACC

Specifically targets GPU accelerators

It started after OpenMP

Basic principle: fork-join model




OpenACC is more descriptive

**Compiler support**

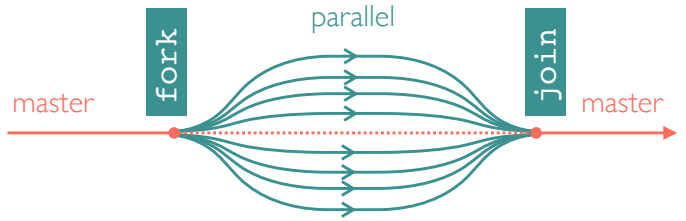
PGI, Cray, NVIDIA

## OpenMP

Designed to replace low-level and multi-threaded programming solutions like POSIX, threads or Pthreads

Intend to target independent processor (shared memory)

Basic principle: fork-join model




OpenMP 4./4.5 onwards; offloading capabilities

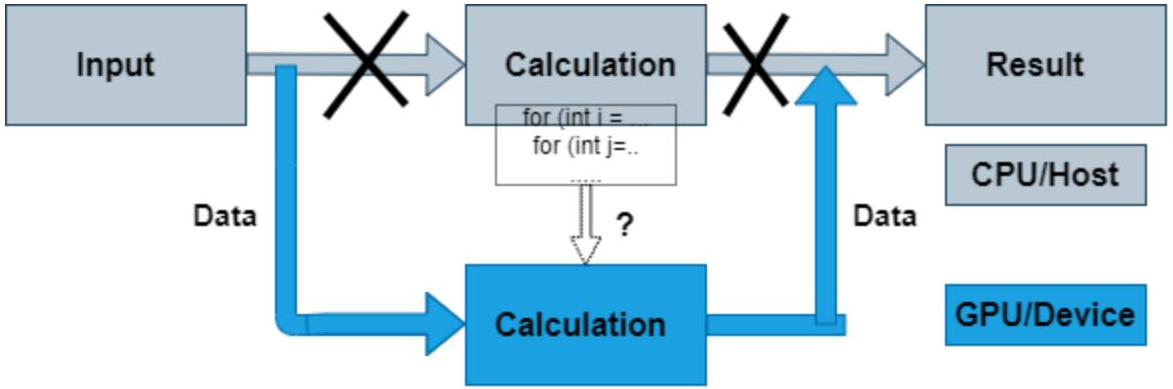
OpenMP is more prescriptive

**Compiler support**

GCC, Intel, IBM XL, LLVM/Clang etc

# The accelerator model

- **Host (CPU):** Orchestrates the application. Handles I/O, serial logic, and operating system tasks.
- **Device (GPU):** Coprocessor. Executes heavy compute kernels.
- **Offloading:** The host sends data and code to the device, waits for completion, and reads results back.

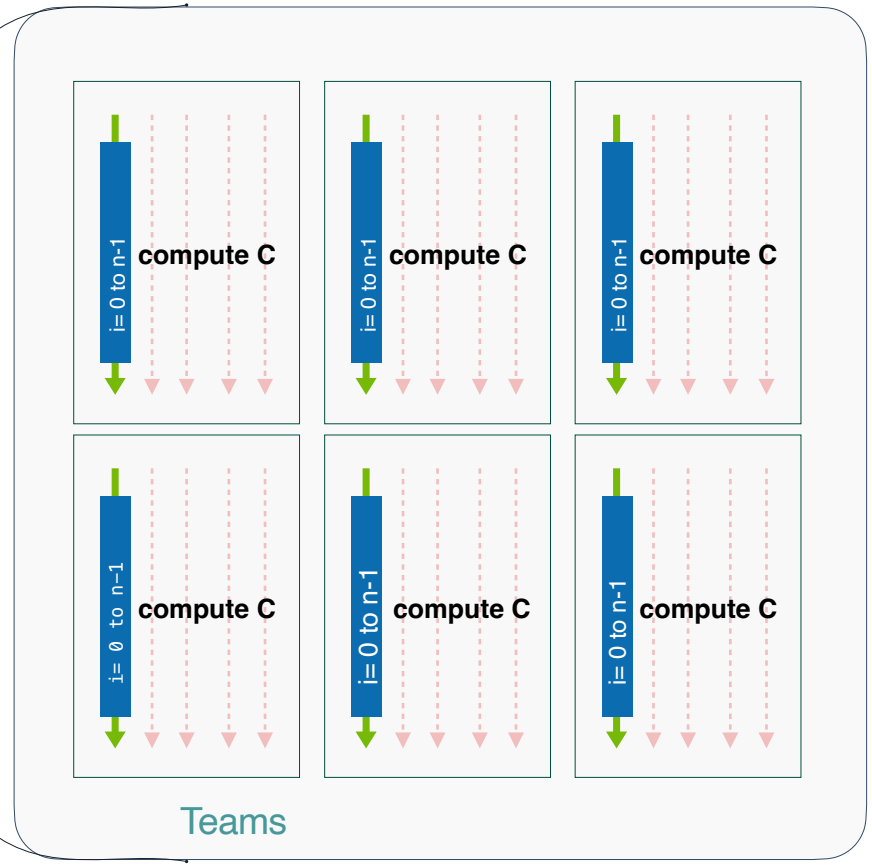


# Offloading serial code with OpenX

Instructions to the compiler on how to compile with code

```
#pragma acc parallel
#pragma omp target teams
{
  Compiler will generate 1 or more parallel GANGS or Teams of threads
  which execute block of code redundantly
}
```

```
!$acc parallel
!$omp target teams
DO I = 1, N
  Compiler will generate 1 or more parallel GANGS or Teams of threads which
  execute block of code redundantly
END DO
!$omp end target teams
!$acc end parallel
```



# What and why OpenACC directives?

# What is OpenACC?

**OpenACC** is …

a directive-based

parallel programming model

designed for

performance and portability

Add Simple Compiler Directive

```
main()
 {
  <serial code>
  #pragma acc kernels
  {
    <parallel code>
  }
}
```

# OpenACC Directives

Simple I Powerful I Portable

- Incremental
- Single Source
- Interoperable
- Performance portable: CPU, GPU, MIC

1 Management Data Movement

2 Initiate Parallel Execution

3 Optimize Loop Mappings

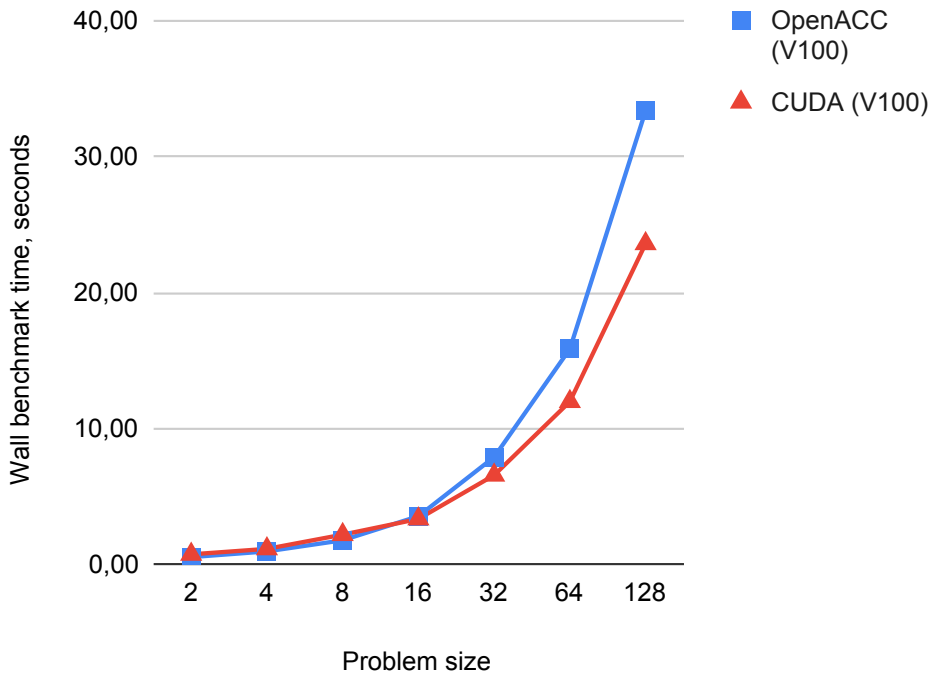
```
#pragma acc data copyin(a,b) copycout(c)
{
  ...
  #pragma acc parallel
  {
  #pragma acc loop gang vector
    for (int idx=0; idx<N; idx++)
        C[idx] = A[idx] + B[idx];
  }
}
```

# OpenACC Directives: Simple, Powerful and Portable

```
main()
 {
   <serial code>
   #pragma acc kernels // Automatically runs on GPU
   <parallel code>
}
```

## OpenACC VS CUDA

- Powerful: Could reach to 98 % speed in comparison to CUDA
- A rough estimate 10,0000 +  Developers using OpenACC
- Scientific research, Industry adoption
- Tool and Ecosystem Support: NVHPC, CRAY, LLVM




Comparison of OpenACC and CUDA performance on Cloverleaf application (doi:10.1088/1742-6596/1740/1/012056)

# OpenACC Directives: Simple, Powerful and Portable

### #pragma acc <directive> <clauses>

**#pragma** in C/C++ is what's known as a "compiler hint." These are very similar to programmer comments, however, the compiler will actually read our pragmas. Pragmas are a way for the programmer to "guide" the compiler, without running the chance damaging the code. If the compiler does not understand the pragma, it can ignore it, rather than throw a syntax error.

**acc** is an addition to our pragma. It specifies that this is an **OpenACC pragma**. Any non-OpenACC compiler will ignore this pragma. Even the nvc/nvc++ compiler can be told to ignore them. (which lets us run our parallel code sequentially!)

**directives** are commands in OpenACC that will tell the compiler to do some action. For now, we will only use directives that allow the compiler to parallelize our code.

**clauses** are additions/alterations to our directives. These include (but are not limited to) optimizations. The way that I prefer to think about it: directives describe a general action for our compiler to do (such as, paralellize our code), and clauses allow the programmer to be more specific (such as, how we specifically want the code to be parallelized).

# Parallelism identified by programer

**Parallel:** a parallel region of code. The compiler generates a parallel kernel for that region. Each gang will execute the entire loop

```
#pragma acc parallel
{

    for (int i=0; i<N; i++)
      c[i] = a[i] + b[i]
}
```

```
#pragma acc parallel
{
    for (int i=0; i<N; i++)
      a[i] = 0;

    for (int i=0; i<N; i++)
      a[i]++;
}
```

# Parallelism identified by programer

**Parallel:** a parallel region of code. The compiler generates a parallel kernel for that region. Each gang will execute the entire loop

**Loop:** identifies a loop that should be distributed across threads. Parallel and loop are often placed together

```
#pragma acc parallel
{
  #pragma acc loop
    for (int i=0; i<N; i++)
      c[i] = a[i] + b[i]
}
```

```
 #pragma acc parallel
 {
     for (int i=0; i<N; i++)
       a[i] = 0;

   #pragma acc loop
     for (int i=0; i<N; i++)
       a[i]++;
 }
```

Similar to OpenMP, compiler translates the parallel region into a kernel that runs in parallel on the GPU

# Parallelism identified by programer

**Kernels:** a parallel region of code. It allows the programmer to step back, and rely solely on the compiler

**Loop:** identifies a loop that should be distributed across threads. Parallel and loop are often placed together

```
#pragma acc kernels
{
    for (int i=0; i<N; I++) {c[I] = a[i] + b[I]}

    for (int i=0; i<N; I++) {d[I] = a*x[I] + y[I]}
}
```

```
#pragma acc kernels loop independent
{
    for (int i=0; i<N; I++) {c[I] = a[i] + b[I]}

}
```

No explicit counterpart

# Kernels Vs Parallel Construct

## Kernel compute directives

- Compiler's leeway parallelising and optimising a loop
- Only provide by OpenACC
- Compiler guarantees correctness
- Or at least it does two things:
  - Fuse the two loop nests into one loop nest
  - Generates two kernels
- Implicit barrier at the end and between each loop

## Parallel compute directives

- Programmer's responsibility to identify region for  parallelization
- Programmer guarantees correctness
- Parallel just run the same code on multiple threads  redundantly
- Guarantees that no dependency occurs iterations
- Implicit barrier at the end of the parallel region

When fully optimized, both will give similar performance

# Development cycle

### Analyze

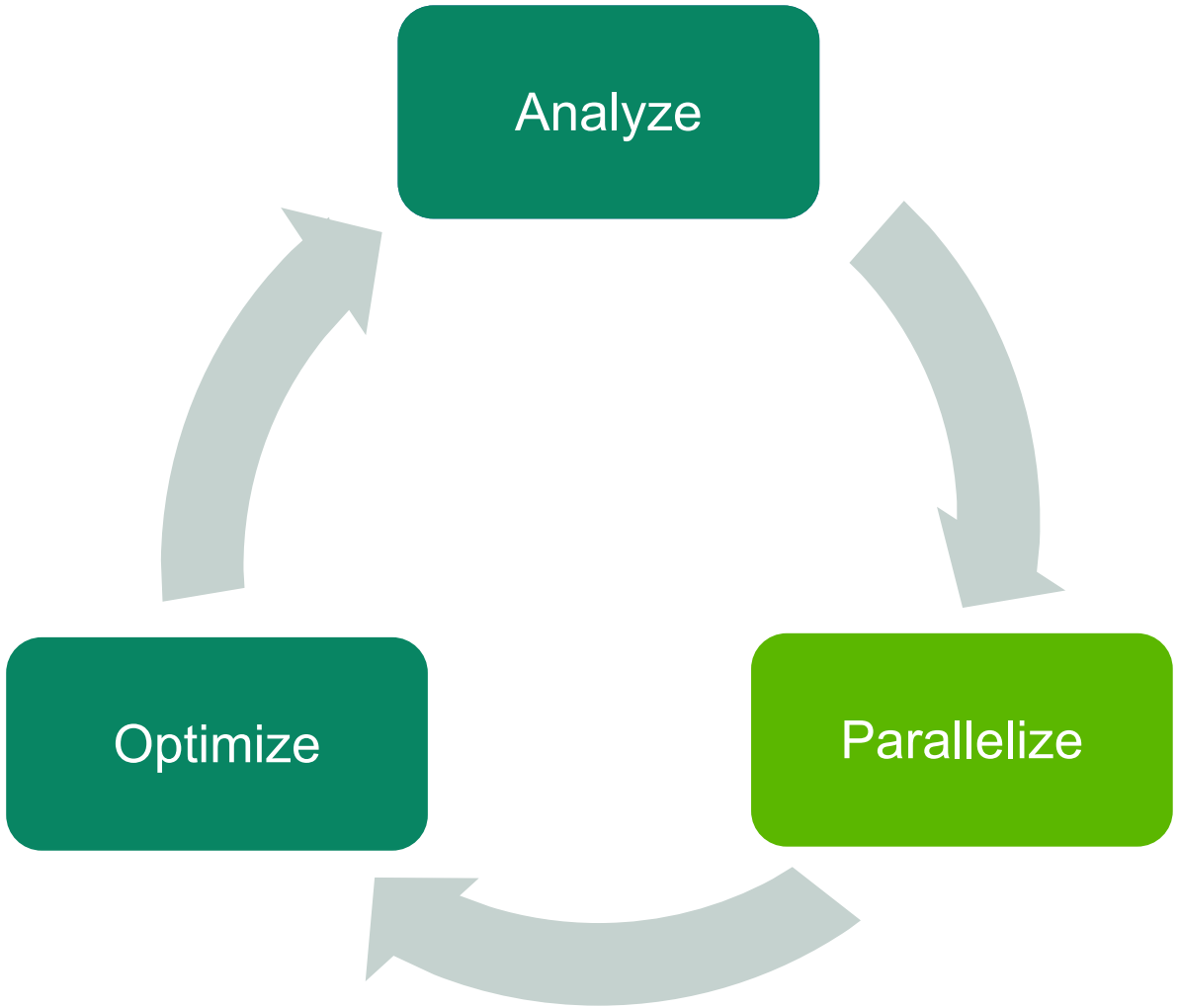
- Your code to determine most likely places needing parallelisation or optimisation

### Parallelise

- Your code by starting with the most time consuming parts, check the correctness and analysis again

### Optimize

- Your code to improve and observed speedup from parallelisation



# Compiling OpenX code with NVHPC

# Compiling OpenACC code

To compile OpenACC code with the NVIDIA HPC SDK, use the `-acc` flag.

## Basic Command:

```
$ nvc -acc -gpu=managed -Minfo=accel main.c -o app
```

## Flags Explained:

- `-acc` : Enable OpenACC directives.
- `-gpu=managed` : Enable Unified Memory (easier for beginners).
- `-Minfo=accel` : Output compiler feedback (Crucial!).

# Compiler Feedback (-Minfo)

Always compile with `-Minfo=accel`. It tells you what happened.

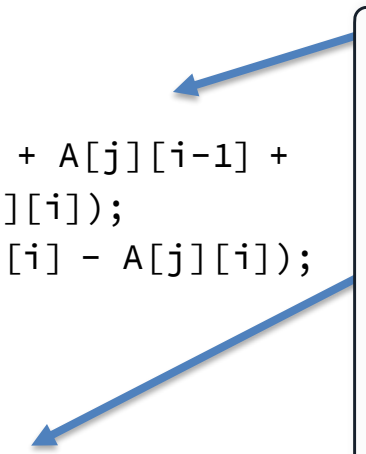
```
main:
     15, Generating copyin(a[:],b[:]) [if not already present]
         Generating copyout(c[:]) [if not already present]
     16, Loop is parallelizable
         Generating Tesla code
         16, #pragma acc loop gang, vector(128) /* blockIdx.x threadIdx.x */
```

- **Generating Tesla code:** Success! Code generated for GPU.
- **Loop is parallelizable:** No dependencies found.
- **Loop carried dependence:** Failed to parallelize.

# Checkpoint-1:  laplace2d_serial: parallelize with OpenACC

# Building and running enabled OpenACC code

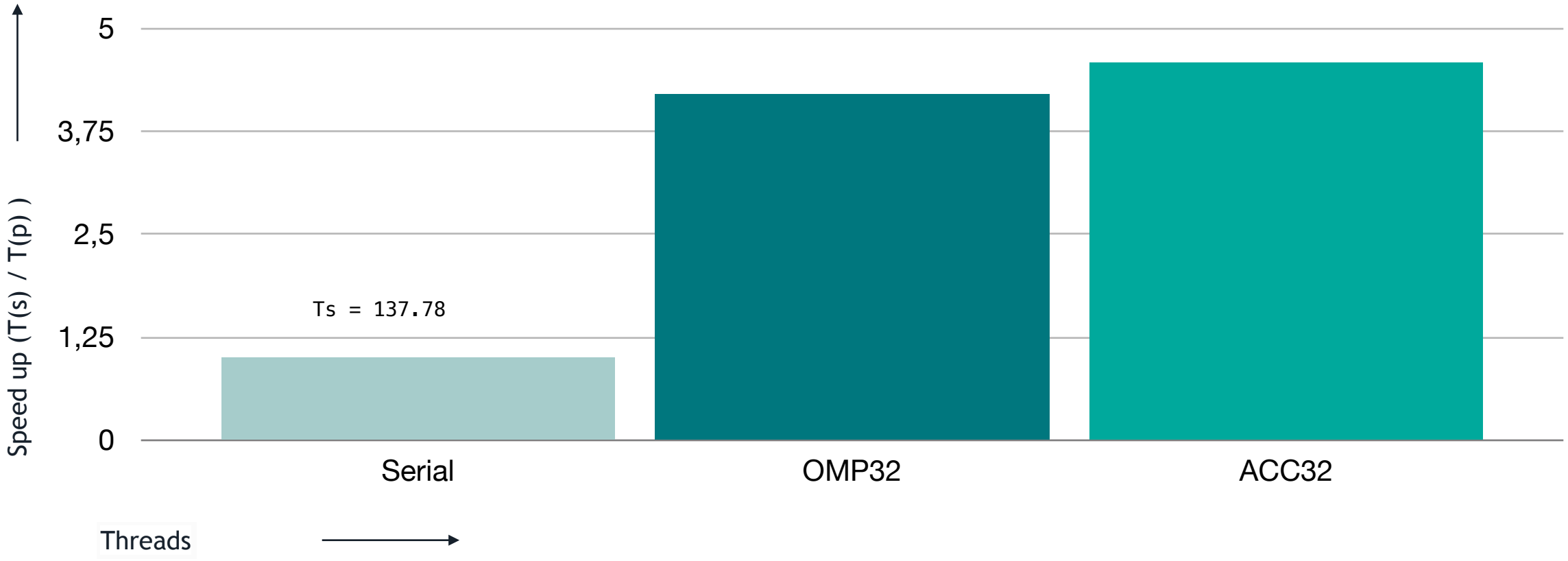
```
while ( error > tol && iter < iter_max ) {
  error=0.0;

for( int j = 1; j < n-1; j++) {
    for(int i = 1; i < m-1; i++) {
      Anew[j][i] = 0.25 * (A[j][i+1] + A[j][i-1] +
                           A[j-1][i] + A[j+1][i]);
      error = max(error, abs(Anew[j][i] - A[j][i]);
    }
  }

for( int j = 1; j < n-1; j++) {
    for( int i = 1; i < m-1; i++ ) {
      A[j][i] = Anew[j][i];
    }
  }
  iter++;
}
```



## Things to do

- Make these two loops parallel with OpenACC directives
- Add Parallel and Kernel construct
- Hint you might want to use reduction as well

# Performance speed up (higher is better)



# Laplace2D_kernels_report

Set export PGI_ACC_TIME=1

```
main:
     34, Loop unrolled 8 times
     44, Loop not vectorized/parallelized: potential early exits
     49, Loop is parallelizable
         Generating implicit copyin(A[:][:]) [if not already present]
         Generating implicit copy(error) [if not already present]
         Generating implicit copyout(Anew[1:4094][1:4094]) [if not already present]
     50, Loop is parallelizable
         Generating Tesla code
         49, #pragma acc loop gang, vector(128) collapse(2) /* blockIdx.x threadIdx.x */
             Generating implicit reduction(max:error)
         50,   /* blockIdx.x threadIdx.x auto-collapsed */
     58, Loop is parallelizable
         Generating implicit copyin(Anew[1:4094][1:4094]) [if not already present]
         Generating implicit copyout(A[1:4094][1:4094]) [if not already present]
     59, Loop is parallelizable
         Generating Tesla code
         58, #pragma acc loop gang, vector(128) collapse(2) /* blockIdx.x threadIdx.x */
         59,   /* blockIdx.x threadIdx.x auto-collapsed */
     69, FMA (fused multiply-add) instruction(s) generated
```

Implicit reduction

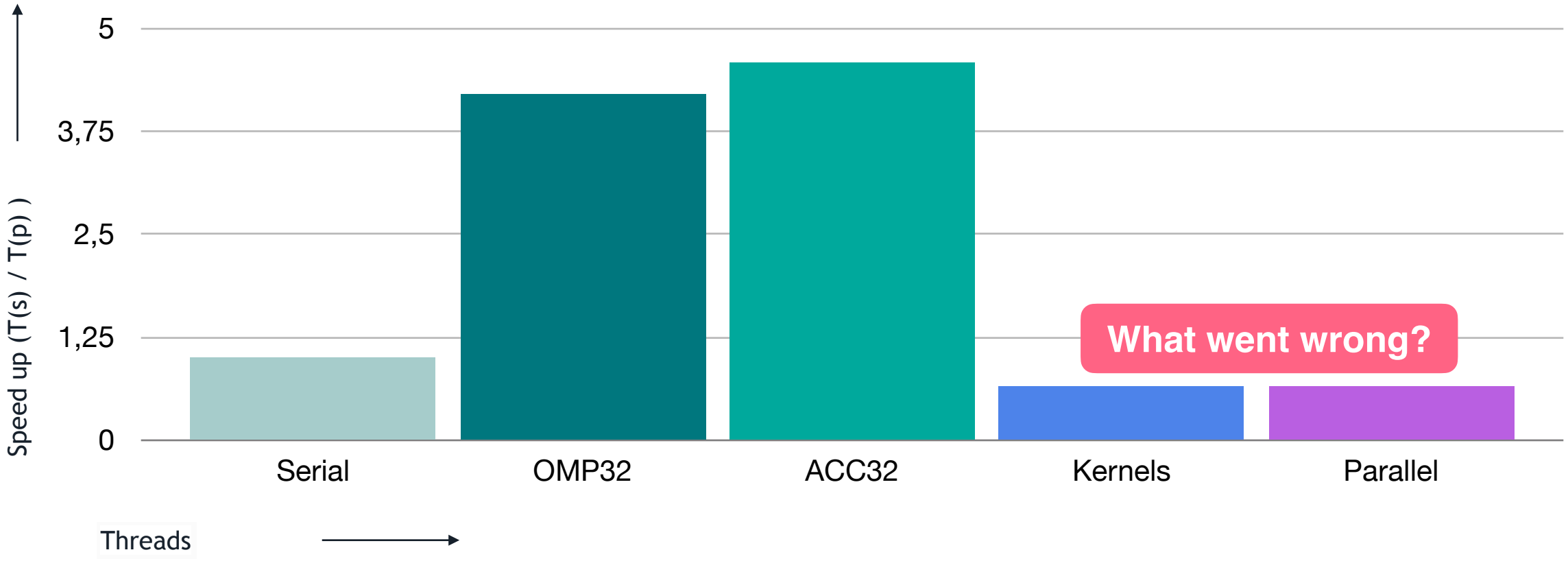
# Laplace2D_kernels_report

```
main:
     34, Loop unrolled 8 times
     44, Loop not vectorized/parallelized: potential early exits
     49, Loop is parallelizable
         Generating implicit copyin(A[:][:]) [if not already present]
         Generating implicit copy(error) [if not already present]
         Generating implicit copyout(Anew[1:4094][1:4094]) [if not already present]
     50, Loop is parallelizable
         Generating Tesla code
         49, #pragma acc loop gang, vector(128) collapse(2) /* blockIdx.x threadIdx.x */
             Generating reduction(max:error)
         50,   /* blockIdx.x threadIdx.x auto-collapsed */
     58, Loop is parallelizable
         Generating implicit copyin(Anew[1:4094][1:4094]) [if not already present]
         Generating implicit copyout(A[1:4094][1:4094]) [if not already present]
     59, Loop is parallelizable
         Generating Tesla code
         58, #pragma acc loop gang, vector(128) collapse(2) /* blockIdx.x threadIdx.x */
         59,   /* blockIdx.x threadIdx.x auto-collapsed */
     69, FMA (fused multiply-add) instruction(s) generated
```

Compiler Generates reduction

Set NV_ACC_TIME=1 for lightweight profiler on time of data movements and kernels
NV_ACC_NOTIFY=1 gives a detailed breakdown of kernel launches and data transfers (bit field)

# Performance speed up (higher is better)



# Profilers provide deep insight

# OpenX (X = OMP, ACC) porting strategies

Identify the compute kernels

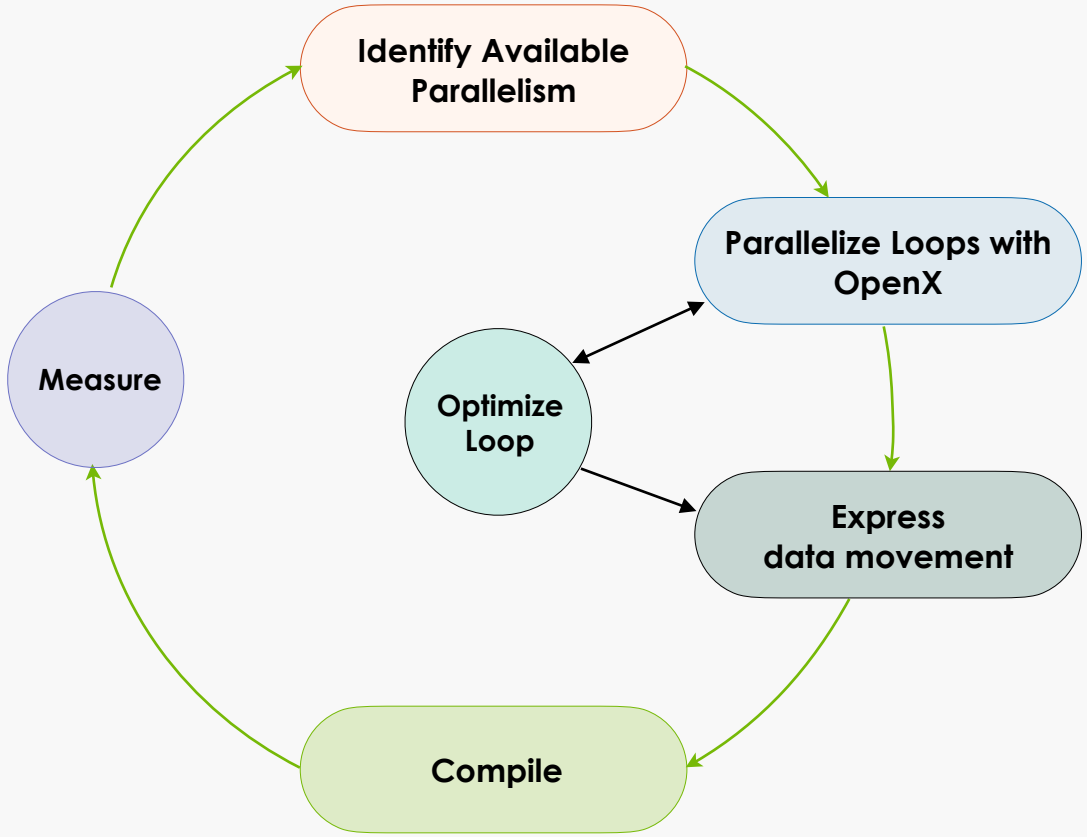
- most compute intensive code
- use performance analysis tools to find bottlenecks

Expressing parallelism within the kernel

- Use performance analysis tools to find bottlenecks
- Track independent work units with well define data access

Managing data transfer between CPU and Device

- relevant data needs to be moved from the host memory to device memory
- kernel executes using device memory
- relevant data needs to be moved from device to host memory



# Tools We Will Use: NSIGHT SUITE

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

# Tools We Will Use: NSIGHT SUITE

## Nsight product family

**Nsight System**

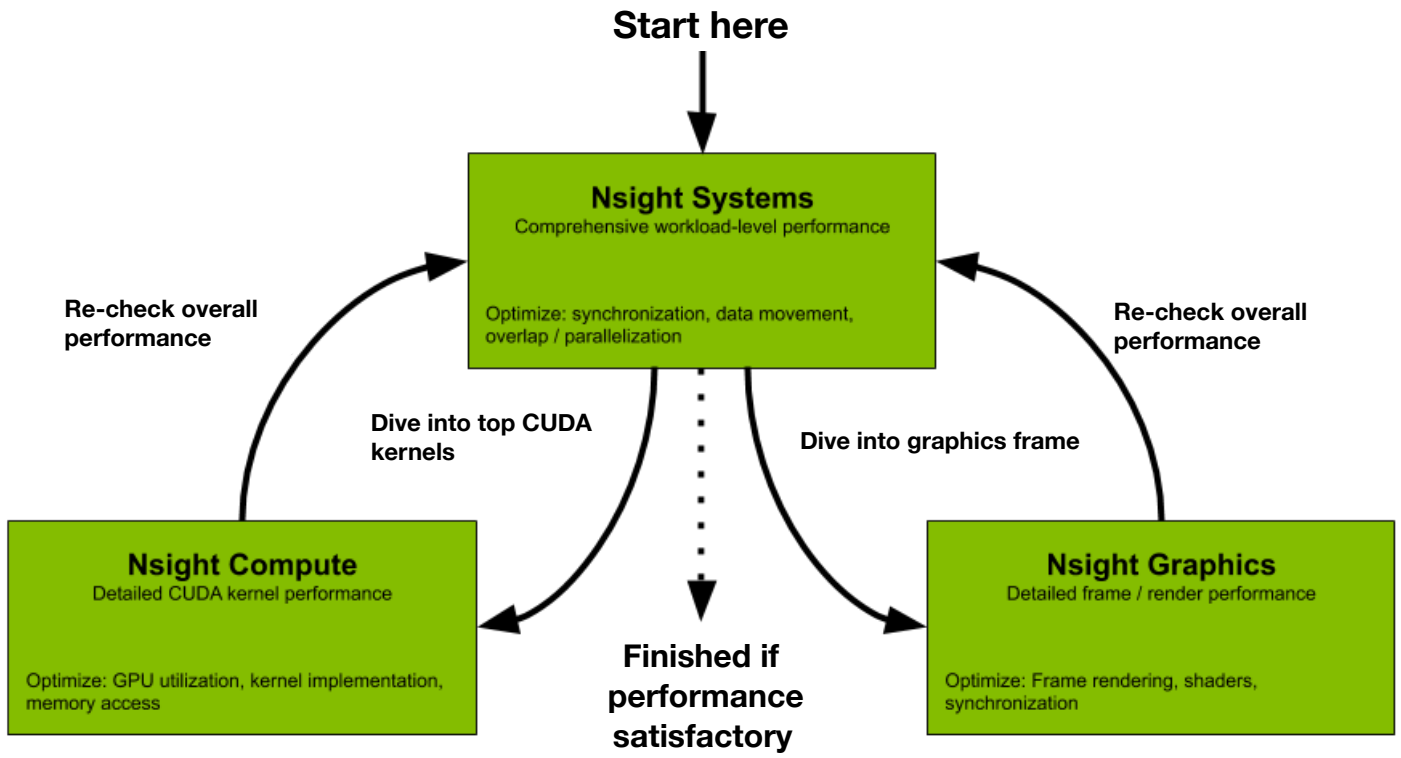
Analyze application algorithm system-wide

**Nsight Compute**

Debug/ Optimise CUDA kernel

**Nsight Graphics**

Debug/ Optimise graphics workloads




Courtesy: NVIDIA

# NSIGHT: Recording an Application Timeline

Notable flags for nays profile

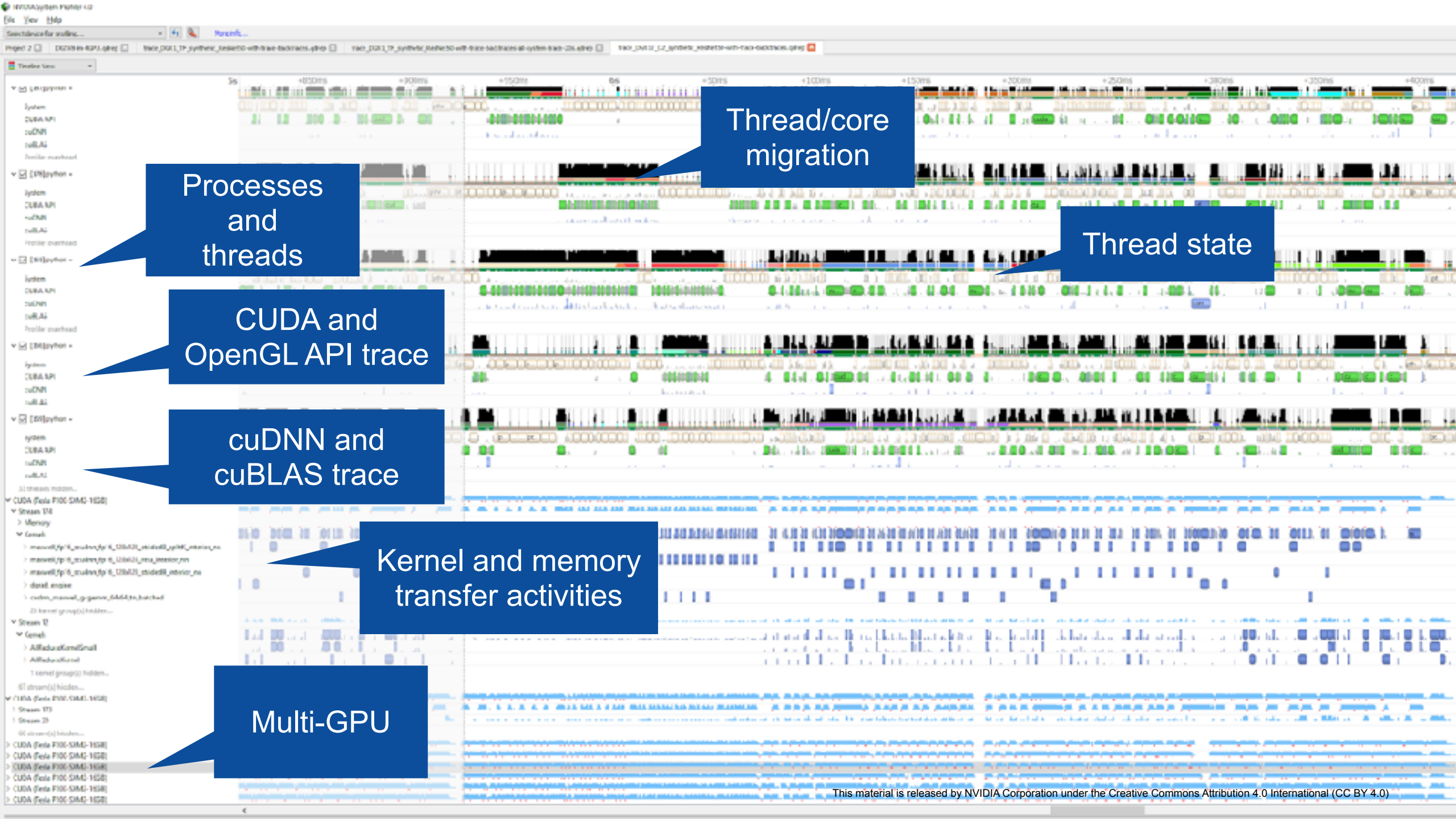
**nsys profile** -t **cuda,nvtx,mpi,openmp** --stats=true --force-overwrite true -o my_report ./myapp

- profile – start a profiling session
- -t: Selects the APIs to be traced ( cuda, cublas, nvtx, mpi openmp and openacc in this example)
- —cuda-memory-usage = true or false
- --stats: if true, it generates summary of statistics after the collection
- --force-overwrite: if true, it overwrites the existing generated report
- -o – name for the intermediate result file, created at the end of the collection (.qdrep filename)

nsys --help or nsys [specific command] --help

Inspect results: Open the report file in the GUI

See also https://docs.nvidia.com/nsight-systems/



# Why do we have to care about data transfers?

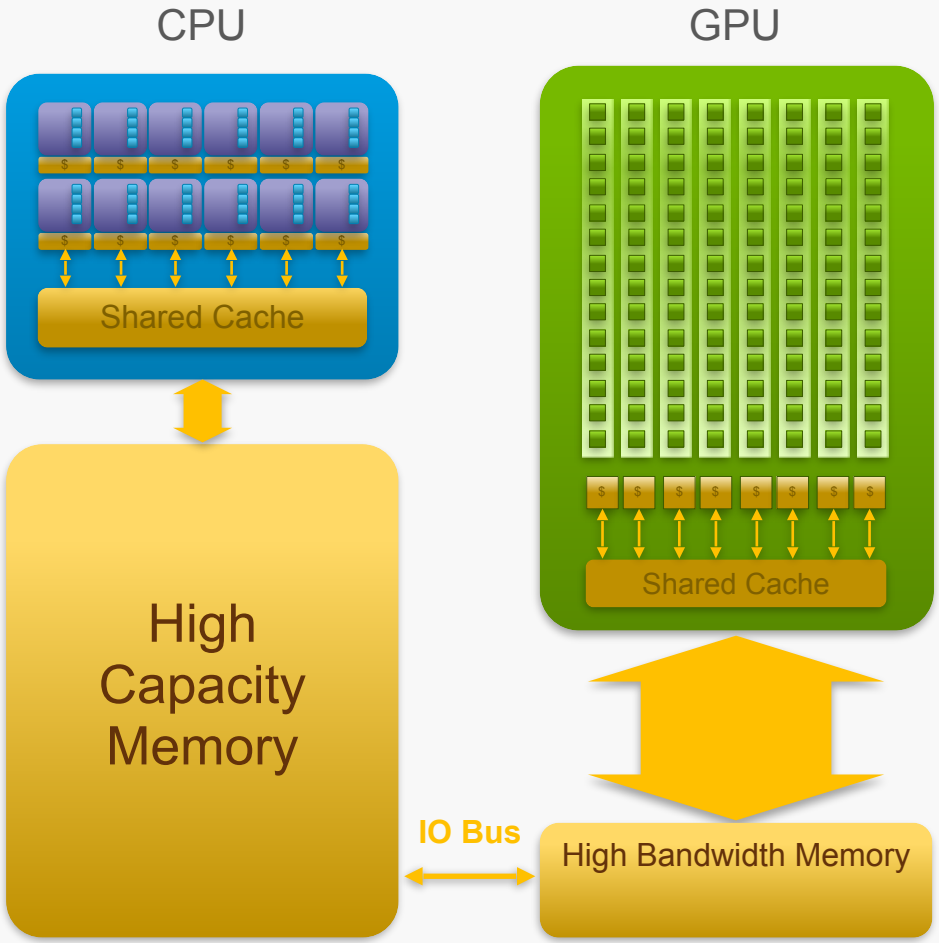
# Data movement is a bottleneck in GPU programming

Between the host and device

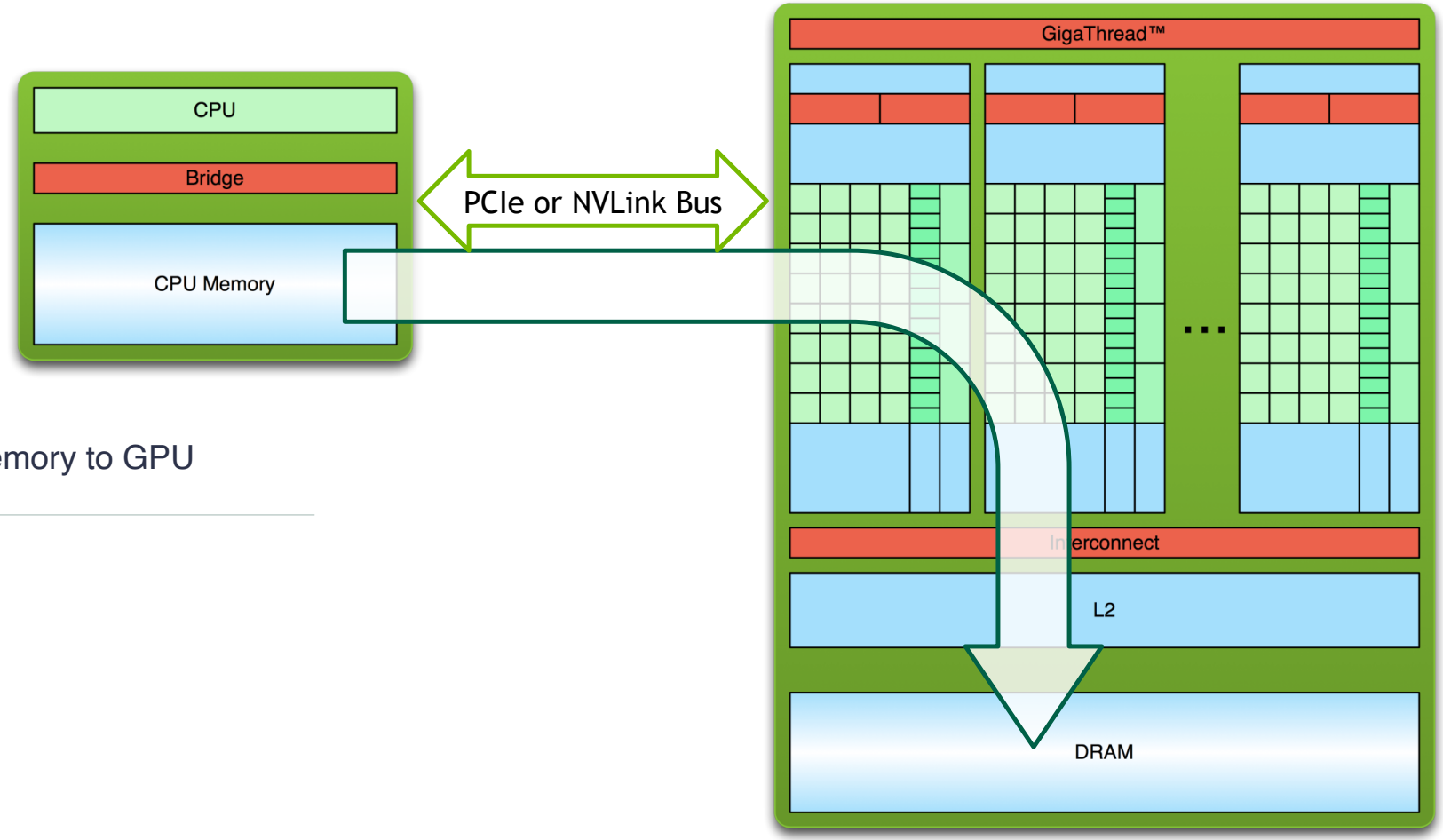
- GPU to its internal memory (HBM2): 900 GB/s
- GPU to CPU via PCIe: 16 GB/s
- GPU to GPU via NVLink: 25 GB/s
- CPU to RAM (DDR4): 128 GB/s

Explicit memory management

- Data must be visible on the device when the kernel is launched
- To maximize performance, data movement needs to be minimise



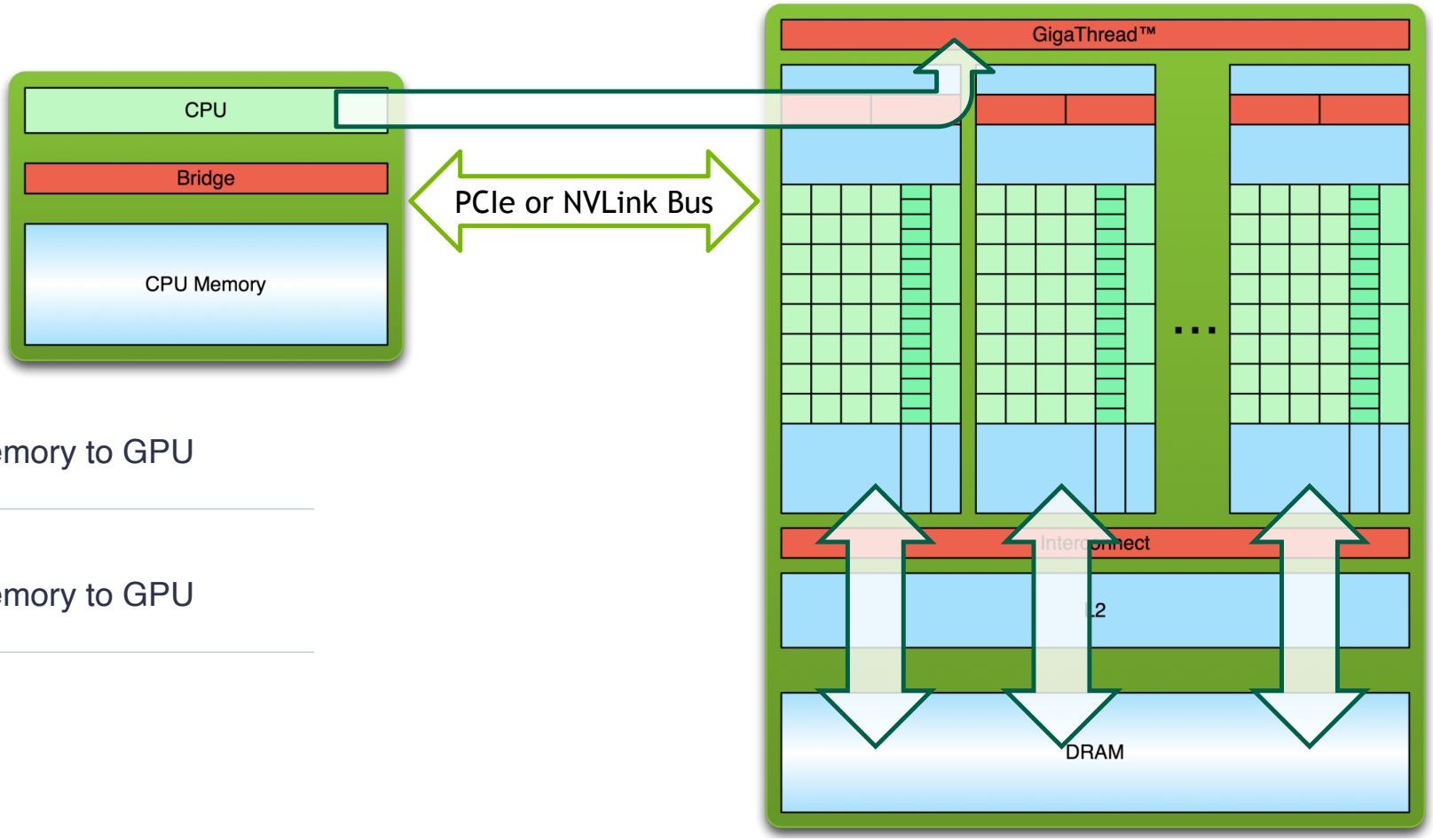
# Three simple processing steps




1 Copy input data from CPU memory to GPU

Courtesy: NVIDIA

# Three simple processing steps

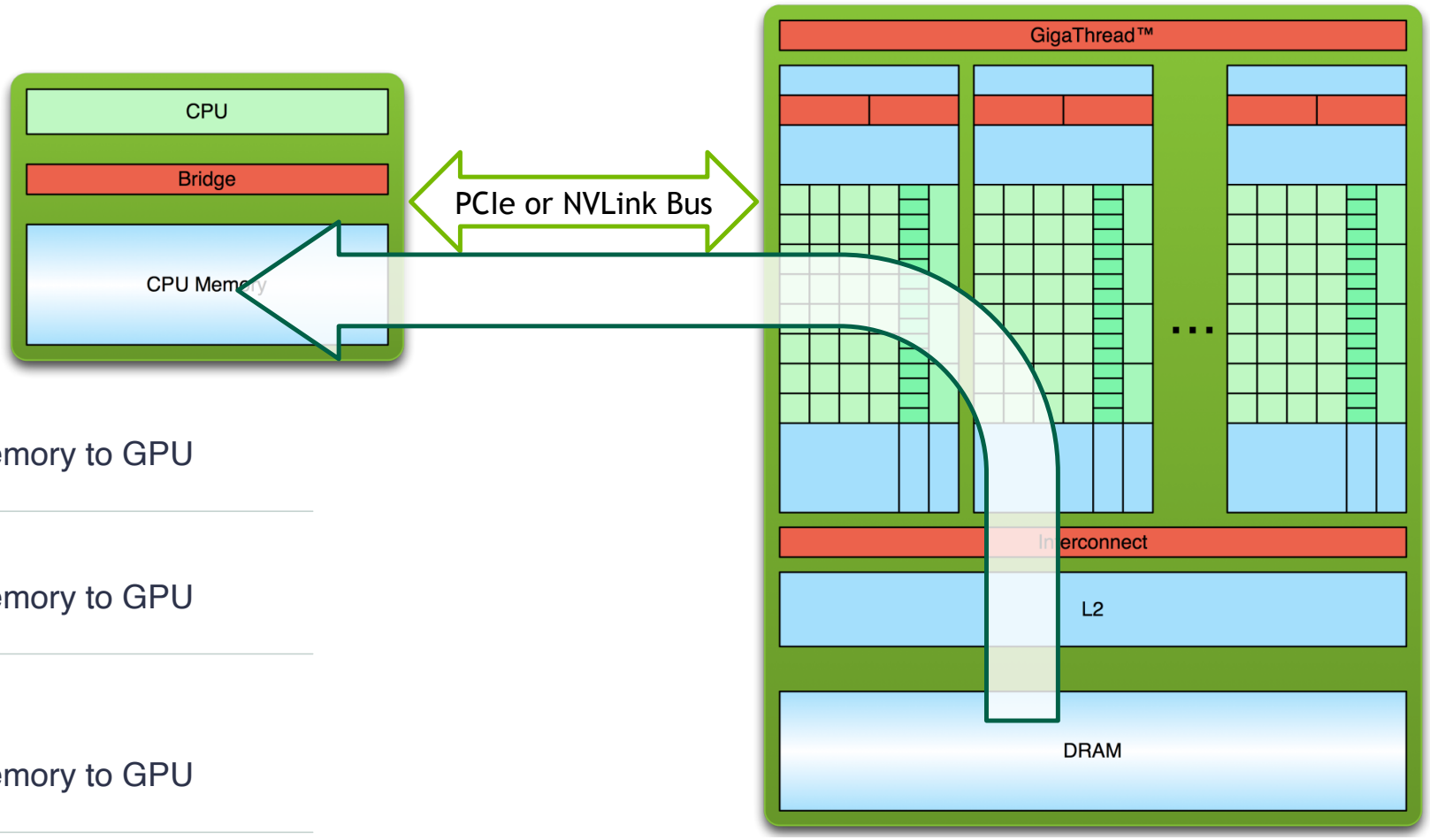



1. Copy input data from CPU memory to GPU

2. Copy input data from CPU memory to GPU

Courtesy: NVIDIA

# Three simple processing steps

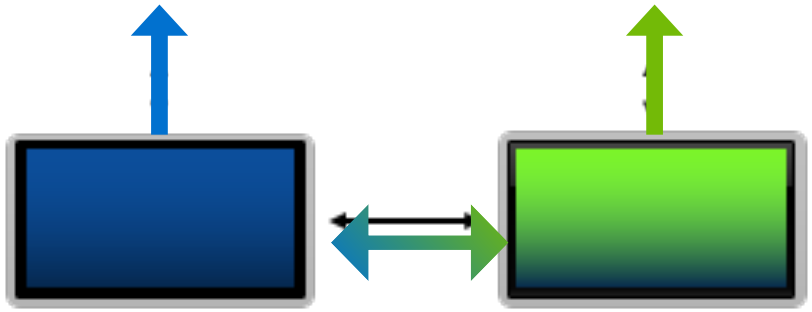



1. Copy input data from CPU memory to GPU
2. Copy input data from CPU memory to GPU
3. Copy input data from CPU memory to GPU

Courtesy: NVIDIA

# Managed Memory

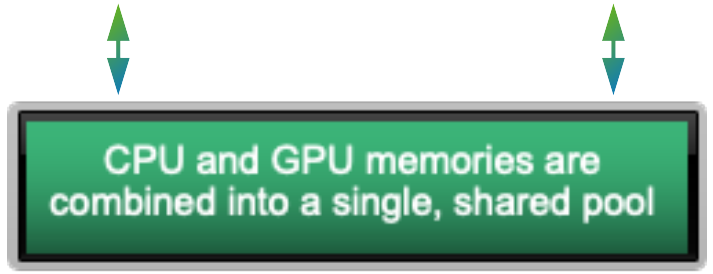
Without Managed memory



System memory GPU memory

$ nccx -gpu=cc80 -Minfo=acc -o binary Code.c

With Managed memory




Managed memory

$ nccx -gpu=cc80,managed -Minfo=acc -o

binary Code.c

# OpenACC with Managed memory

```
while ( error > tol && iter < iter_max )
{
  error = 0.0;
#pragma acc kernels
  {
    for( int j = 1; j < n-1; j++)
    {
      for( int i = 1; i < m-1; i++ )
      {
        Anew[j][i] = 0.25 * ( A[j][i+1] + A[j][i-1]
                            + A[j-1][i] + A[j-1][i]);
        error = fmax( error, fabs(Anew[j][i] - A[j][i]));
      }
    }

    for( int j = 1; j < n-1; j++)
    {
      for( int i = 1; i < m-1; i++ )
      {
        A[j][i] = Anew[j][i];
      }
    }
  }
}
```

Without Managed Memory the compiler must determine the size of A and Anew and copy their data to and from the GPU each iteration to ensure correctness

With Managed Memory the underlying runtime will move the data only when needed

# Data Management with OpenACC

1. Allocate ‘A’ on GPU
2. Copy From CPU to GPU
3. Copy back to GPU and deallocate A from GPU

```
int *A = (int*) malloc(N * sizeof(int));
#pragma acc parallel loop      copy(A[0:N])
{
  for (int i=0; i<N; i++) A[i] = 0;
}
```

```
for (int i=0; i<N; I++) A[i] = 0;
#pragma acc parallel loop copy(A[0:N])
{
    for (int i=0; i<N; I++) A[i] = 1;
}
```

```
#pragma acc parallel loop copyin(A[0:N]) copyout (B[0:N])
{
    for (int i=0; i<N; I++) A[i] = B[I];
}
```

# Managing the data on the device

**Data region constructs**

- A data region is the dynamic scope of a structured block associated with an implicit or explicit data construct.
- Facilities the sharing of data between multiple parallel regions (kernels, parallel, loop *etc*)
- Must start and end in the scope of the same function or subroutine – it's a structure construct

The data directive defines a region of code in which GPU arrays remain on the GPU and are shared among all kernels in that region.

Syntax:

**C:**

```
#pragma acc data [clause]
{
    code region ...
    including compute related pragmas
}
```

**Fortran:**

```
!$acc data
    code region ...
    including compute related pragmas
!$acc end data
```

# Managing the data on the device

provides the programmer additional control over how and when data is created on and copied to or from the device

### Clause: copyin / map(to)

- Copy the variable data to the device at the beginning of the region, and
- release the space on the device when done without copying the data back to the host

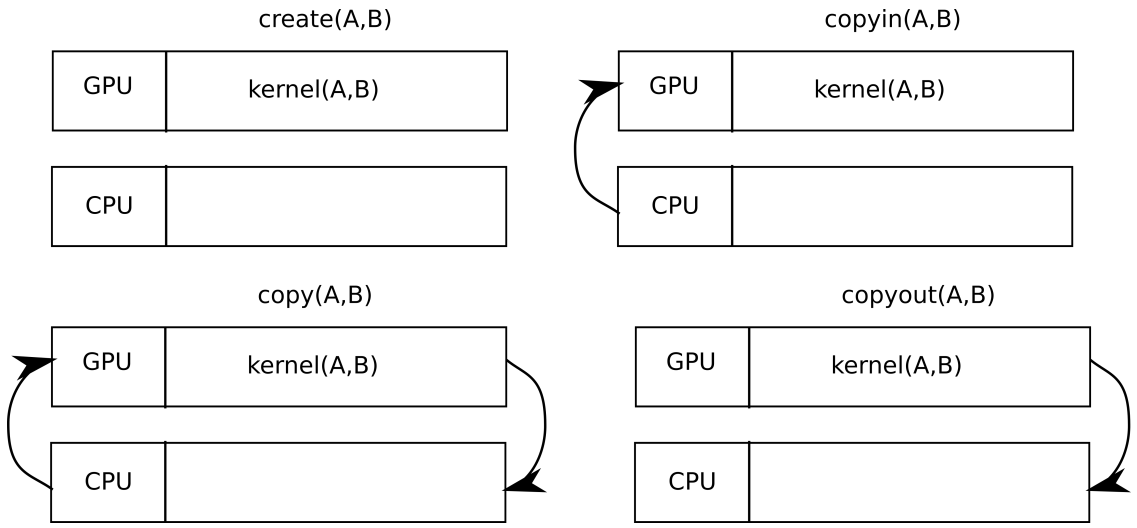
### Clause: copyout / map(from)

- Create space for the listed variables on the device but do not initialize them
- At the end of the region, copy the results back to the host and release the space on the device

### Clause: copy / map(from)

- Create space for the listed variables on the device, initialize the variable by copying data to the device at the beginning of the region
- copy the results back to the host at the end of the region, and release the space on the device when done

and create(list), delete(list), present(list), present_or_copy[inlout], present_or_create, device_ptr




**C and C++:**

```
#pragma omp target data [clause [[,] clause]…]
#pragma acc data [clause [[,] clause]…]
```

**Fortran:**

```
!$omp target data [clause [[,] clause]…]
!$acc data [clause [[,] clause]…]
```

# Data clause: Moving data between Host to Device

used with the kernels, parallel, data and declare constructs

```
#pragma acc data copyin(a[0:N],b[0:N]) copyout(c[0:N])
{
        #pragma acc parallel loop
        for(int i = 0; i < N; i++){
                c[i] = a[i] + b[i];
        }
}
```

| Action |
| --- |
| [illegible] |


# Array Shaping

Compiler sometimes cannot determine size of arrays

- Must specify explicitly start/end point
- Memory only exits within the data region
- Must be within a single function

```
/* C/C++ code to offload on the device*/

    #pragma acc data copyin(a[0:N]) copyout(b[s/4:3*s/4])

/* Fortran code to offload on the device*/

    !$acc data copyin(a[1:N]) copyout(b(s/4:3*s/4))
```

- Fortran uses *start:end* and C uses *start:count*
- Data clauses can be used on data, kernels or parallel

Note: data clauses can be used on data, parallel, or kernels

# Encompassing Multiple Compute Regions

```
#pragma acc data copyin(A[0:N], B[0:N]) create(C[0:N])
{

    #pragma acc parallel loop
    for( int i = 0; i < N; i++ )
    {
        C[i] = A[i] + B[i];
    }

    #pragma acc parallel loop
    for( int i = 0; i < N; i++ )
    {
        A[i] = C[i] + B[i];
    }

}
```

```
void copy(int *A, int *B, int N)
{
#pragma acc parallel loop copyout(A[0:N]) copyin(B[0:N])
    for( int i = 0; i < N; i++ )
    {
        A[i] = B[i];
    }
}
```

```
#pragma acc data copyout(A[0:N],B[0:N]) copyin(C[0:N])
{
    copy(A, C, N);

    copy(A, B, N);
}
```

# Encompassing Multiple Compute Regions

**DAXPY in C**

```
#pragma acc data create( D[0:ARRAY_SIZE], Y[0:ARRAY_SIZE] ) copyin(A) copyout(D[0:ARRAY_SIZE])
 {
#pragma acc parallel loop
 for (size_t i=0; i<ARRAY_SIZE; i++)
 {
        D[i] = 0.0;
        X[i] = 1.0;
        Y[i] = 2.0;
 }

 start_time = omp_get_wtime();
 #pragma acc parallel loop
 for ( size_t i=0; i<ARRAY_SIZE; i++ )
        D[i] = A*X[i] + Y[i];
 end_time = omp_get_wtime();
 }
```

Compute region

Data region

# Checkpoint-2:  laplace2d_parallel: data clause

# Managing the data on the device

## Analyze the code and refactor the code by following these steps

- **Step1:** **Parallelize the code with**
  - Add parallel loop
- **Step2:** **Including data clause in our Laplace code**
  - Use acc data to minimize transfers
  - Add a **structured data directive** to properly handle the arrays **A** and **Anew**
    Run the Code (With Managed Memory)

## Try to understand the compiler report to be sure about what the compiler is doing

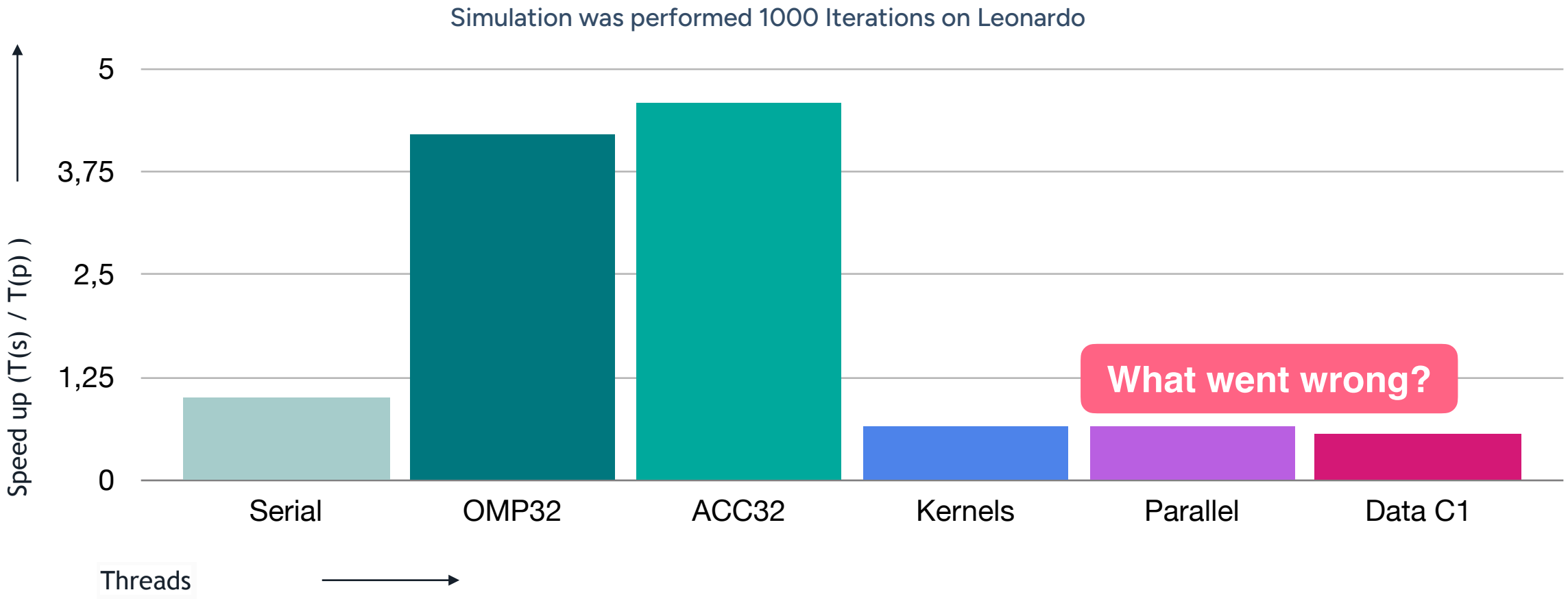
- nsys profile -t nvtx,openacc --stats=true --force-overwrite true -o laplace ./laplace

# 02-laplace2d: Data clause with OpenACC

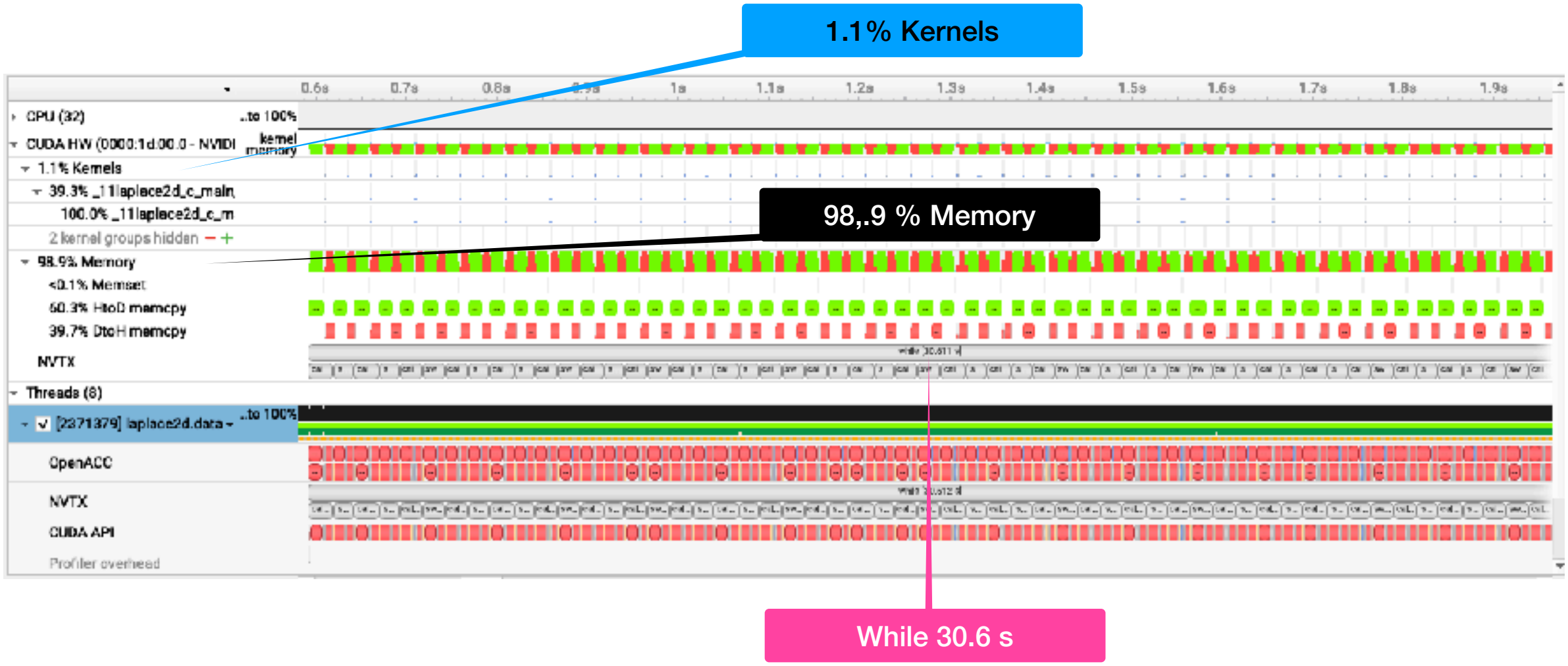
```
while ( err > tol && iter < iter_max ) {
  err=0.0;

#pragma acc parallel loop reduction(max:err) copyin(A[0:n*m]) copy(Anew[0:n*m])
  for( int j = 1; j < n-1; j++) {
    for(int i = 1; i < m-1; i++) {

      Anew[j][i] = 0.25 * (A[j][i+1] + A[j][i-1] +
                           A[j-1][i] + A[j+1][i]);

      err = max(err, abs(Anew[j][i] - A[j][i]));
    }
  }

#pragma acc parallel loop copyin(Anew[0:n*m]) copyout(A[0:n*m])
  for( int j = 1; j < n-1; j++) {
    for( int i = 1; i < m-1; i++ ) {
      A[j][i] = Anew[j][i];
    }
  }
  iter++;
}
```

Data clauses provide necessary “shape” to the arrays.

# Performance ~~speed up~~ (higher is better)



# NSIGHT: Recording an Application Timeline



# Excessive Data Transfers

```
while ( error > tol && iter < iter_max ) {
  error=0.0;
```

A, Anew resident on host

COPY
HOST / DEVICE


```
#pragma acc parallel loop reduction(max:error)
```

A, Anew resident on accelerator

```
for( int j = 1; j < n-1; j++) {
    for(int i = 1; i < m-1; i++) {
      Anew[j][i] = 0.25 * (A[j][i+1] + A[j][i-1] +
                           A[j-1][i] + A[j+1][i]);
      error = max(error, abs(Anew[j][i] - A[j][i]);
    }
  }
```

```
    ...
}
```

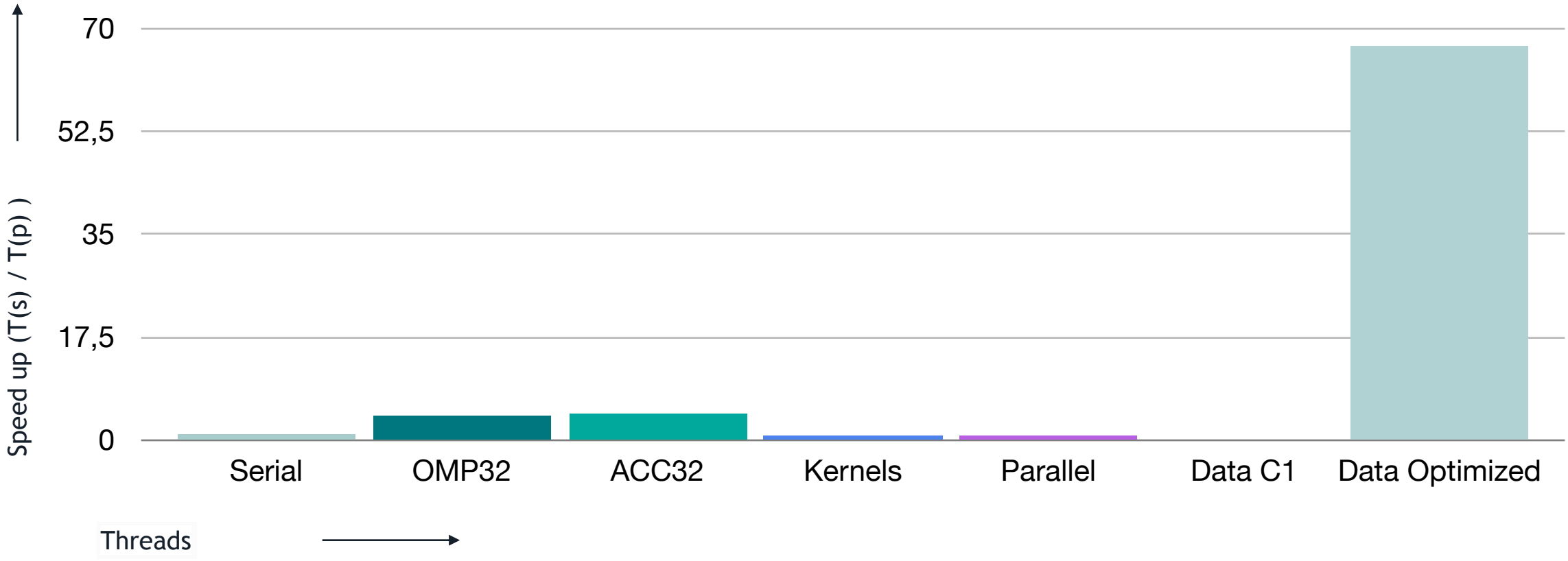
A, Anew resident on host


COPY
DEVICE / HOST

A, Anew resident on accelerator

These copies are performed every iteration of the while loop.
NB: in case of two #pragma acc parallel there are 4 copies per while loop iteration.

# Performance ~~speed up~~ (higher is better)



# Summary Data Clause

- Data clauses allow the programmer to tell the compiler which data to move and when
- Data clauses may be added to kernels or parallel regions
- Advanced data clause: enter data, and exit data, which will learn tomorrow
- In addition:
    - Declare clause
    - Update clause
    - Routine clause and much more …

# OpenACC Loop Optimization

# SEQ Clause

SEQ Clause: short for sequential

- the compiler will tell to run the loop
- Compiler will parallelise the outer loop across the parallel threads, but each thread will run the inner-most loop sequentially
- It may automatically apply the seq clause to loops that have too any dimensions

```
#pragma acc parallel loop
    for (int i = 0; i < size; I++)

      #pragma acc loop
            for (int j = 0; j < size; j++)

              #pragma acc loop seq
                    for (int k = 0; k < size; k++)

                       c[i][j] += a[I][j] * b[I][j];
```

# Private and Firstprivate Clause

The Private Clause: allows the programmer to define a list of variables as “thread-private”

- Each thread will be given a private copy of every variable in comma-separated list

The Firstprivate Clause: like private except

- The private values are initialised to the same value used on the host. Private values are uninitizlised

```
double tmp[3]
 #pragma acc kernels loop privarte(tmp[0:3]
    for (int i = 0; i < size; i++)
       tmp[0] += <value>;
       tmp[1] += <value>;
       tmp[2] += <value>;
#pragma acc parallel loop
    for (int j = 0; j < size; I++)
              array[I][j] = tmp[0] + tmp[1] + tmp[2]
```

tmp array is private to each iteration of the outer loop

Shared among the threads in the inner loop

# Scalars and Private clause

By default, scalars are firstprivate when used in a parallel region and private when used in a kernels region

Except in some cases, scalars do not need to be added to a private clause. These cases may include but are not limited to:

- Scalars with global storage such as global variables in C/C++, Module variables in Fortran
- When the scalar is passed by reference to a device subroutine
- When the scalar used as an rvalue after the compute region, aka “live-out”

Note: putting scalars in a private clause may actually hurt performance !

# The declare directive (Global Variables)

For global or static variables that are constant or accessed repeatedly across many regions, use `declare` to manage their existence on the device.

```
static float PI = 3.14159f;

// Create PI on the device only once, available to all regions
#pragma acc declare create(PI)
```

- **`create`**: Allocates space, but doesn't initialize it.
- **`copyin`**: Allocates space and copies the Host value once.
- **`link`**: Assumes the variable is already defined and linked to the device (advanced interop).

# Collapse Clause

- Collapse(N)
- Combine the next N tightly nested loops
- Can turn a multidimensional loop nest into a single-dimension loop
- Extremely useful for increasing memory locality, as well as creating larger loop to expose more parallelism

```
#pragma acc parallel loop collapse(2)
for ( int i = 0; i < N; i++)
    for ( int j = 0; j < N; j++)
        structured-block
```

# The Tile clause

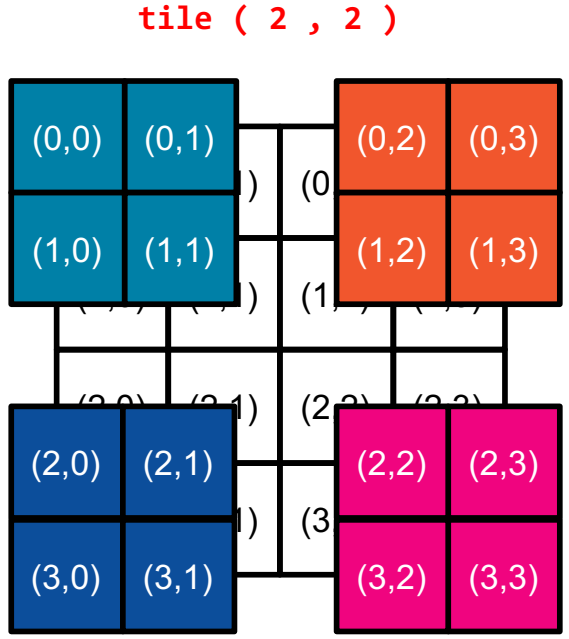
**tile( x, y, z . . . .)**

- Breaks multidimensional loop into “blocks ”
- Can increase data locality in some codes
- Will be able to execute multiple tiles simultaneously

```
#pragma acc kernels loop tile(32,32)
for ( int i = 0; i < 128; i++)
    for ( int i = 0; i < 128; i++)
     structured-block
```

Similar to the `collapse` clause, the inner loops should not have the `loop` directive.

```
#pragma acc kernels loop tile(32,32)
for ( int i = 0; i < 128; i++)
    #pragma acc loop
    for ( int i = 0; i < 128; i++)
```



# Reduction Clause

- Takes many values and “reduces” to a single value
- Each thread calculates its part
- Perform a partial reduction on the loop iterations the they compute
- After the loop, the compiler will perform a final reduction to produce a single result using the specified operation

```
for (int i = 0; i < size; i++)
        for (int j = 0; j < size; j++)
        #pragma acc parallel loop reduction(+:tmp)
                for (int k = 0; k < size; j++)
                    tmp += a[i][j] + b[i][j];
                    c[i][j] = tmp;
```

| Operator | Example | Description |
|---|---|---|
| + | reduction(+:sum) | Mathematical summation |
| * | reduction(*:product) | Mathematical product |
| max | reduction(max:maximum) | Maximum value |
| min | reduction(min:minimum) | Minimum value |
| & | reduction(&:val) | Bitwise AND |
| \| | reduction(\|:val) | Bitwise OR |
| && | reduction(&&:bool) | Logical AND |
| \|\| | reduction(\|\|:bool) | Logical OR |

# The update Directive

Used to synchronize data between Host and Device *within* a data region.

```
#pragma acc data copy(a[0:N])
{
    // Modify 'a' on GPU
    #pragma acc parallel loop
    for( ... ) {  ...  }

    // Need intermediate result on CPU for logging
    #pragma acc update self(a[0:1])
    printf("Current val: %f\n", a[0]);
}
```

- `update self(list)`: GPU -> CPU
- `update device(list)`: CPU -> GPU

# GPU hardware hierarchy

# Execution model: three levels of parallelism

| PLATFORM | GANG | WORKER | VECTOR |
|---|---|---|---|
| MULTICORE CPU | Entire CPU (NUMA domain) | Core | SIMD vector |
| MANYCORE CPU (e.g. Xeon Phi) | NUMA domain (whole chip) | Core | SIMD vector |
| NVIDIA GPU | Thread block | WARP | Thread |
| AMD GPU | Workgroup | Wavefront | Thread |

Number of threads in a gang $N_{threads} = L_{vector} \times N_{workers}$

# A simplified representation of a NVIDIA GPU-architecture

**Peak performance NVIDIA A100**

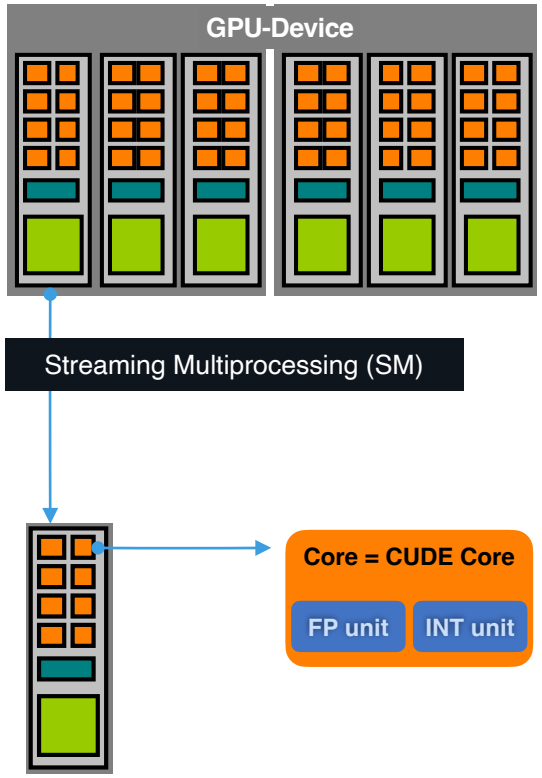
**FLOPS = Clock speed × cores × FLOP/cycle**

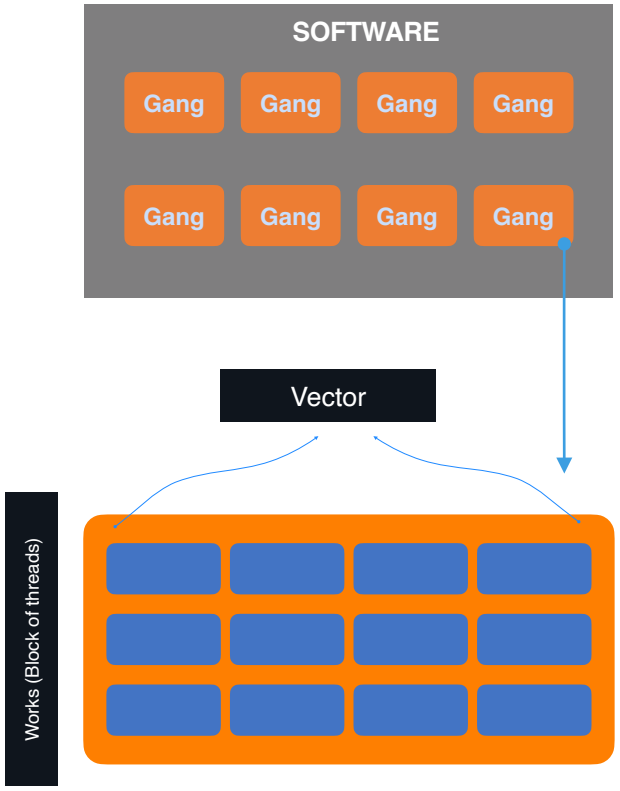
**FLOPS <=> FP64 or FP32 or ...**

**Clock speed (or GPU Boost Clock) = 1.41 GHz**

**Total FLOPS = 250.0 TFLOPS**

(Thread ∈ Block ∈ Grid)



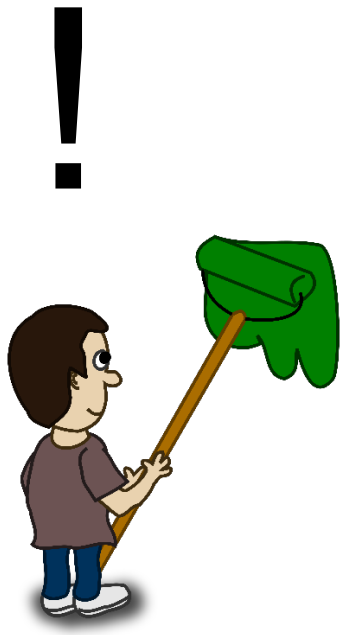



# Gang Worker Vector demystified

OpenACC More Science, Less Programming

NVIDIA

# Gang Worker Vector demystified



OpenACC NVIDIA

How much work 1 worker can do is limited by his speed.

A single worker can only move so fast.

# Gang Worker Vector demystified


Even if we increase the size of his roller, he can only paint so fast.

We need more workers!

# Gang Worker Vector demystified

OpenACC More Science, Less Programming  NVIDIA

# Gang Worker Vector demystified

By organizing our workers into groups (gangs), they can effectively work together within a floor.

Groups (gangs) on different floors can operate independently.

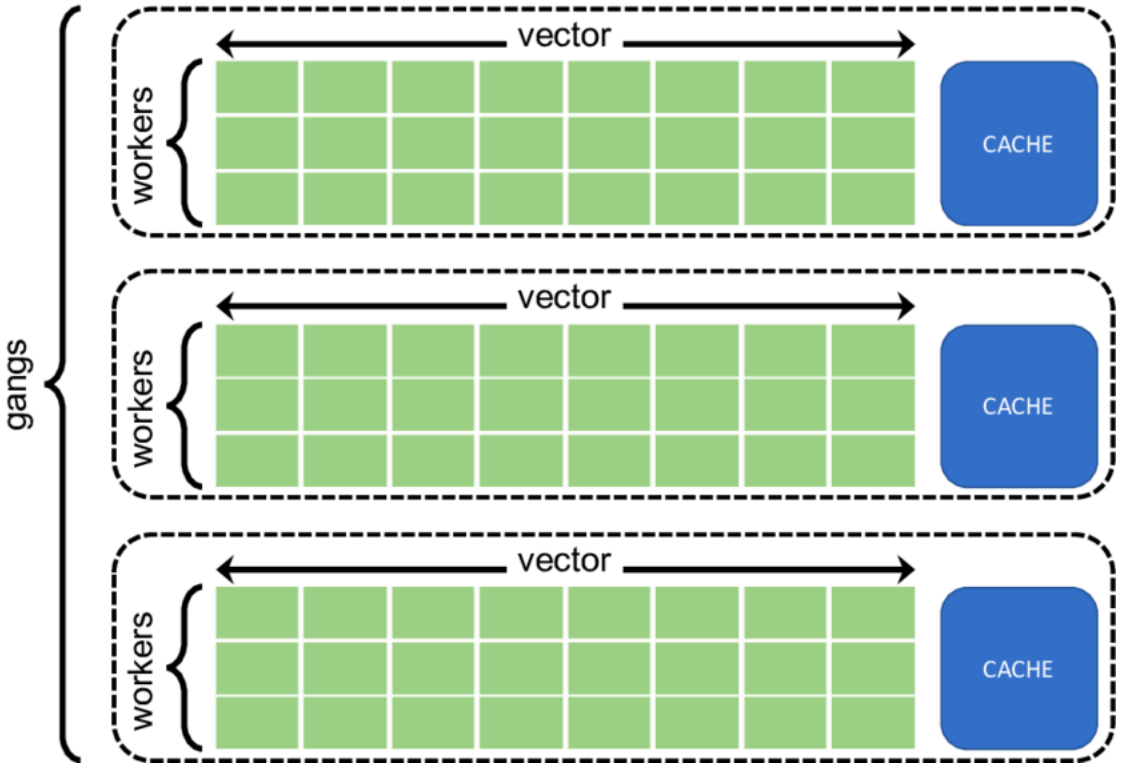
Since gangs operate independently, we can use as many or few as we need.

# Gang Worker Vector demystified

Our painter is like an OpenACC ***worker***, he can only do so much.

His roller is like a ***vector***, he can move faster by covering more wall at once.

Eventually we need more workers, which can be organized into ***gangs*** to get more done.




OpenACC More Science, Less Programming

NVIDIA

# Gang Worker Vector demystified

## Gang

- Multiple gangs will be generated, and loops iterations will be spread across the gangs
- Gangs are independent of each other
- There is no way to for the programmer to know exactly how many gangs are running at a given time

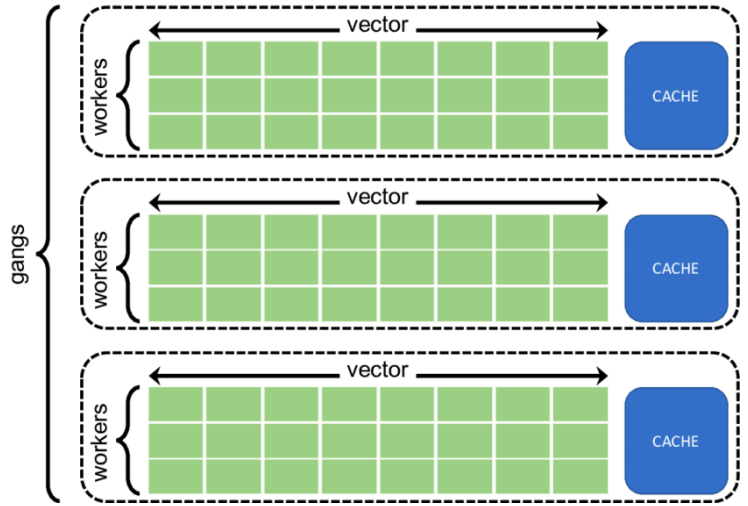
## Worker

- To have **multiple vectors** within a gang
- Splits up one large vector into multiple smaller vectors
- Intermediate level  between the low-level parallelism implemented in vector and group of threads
- useful when our inner parallel loops are very small, and will not benefit from having a large vector

## Vector parallelism:

- Lowest level of parallelism
- Every gang will have at least 1 vector
- Threads work in lockstep (SIMD/SIMT parallelism)

```
#pragma acc loop gang
for ( int i = 0; i < N; i++)
 #pragma acc loop worker
  for ( int j = 0; j < N; j++)
  #pragma acc loop vector
   for ( int k = 0; k < N; k++)
          structured-block
```



# Controlling the size of Gang, worker and Vectors

The compiler will choose a number of gangs, workers, and a vector length for you, but you can change it with clauses

num_gangs(N)

- Generate N gangs for this parallel region

num_workers(M)

- Generate N gangs for this parallel region

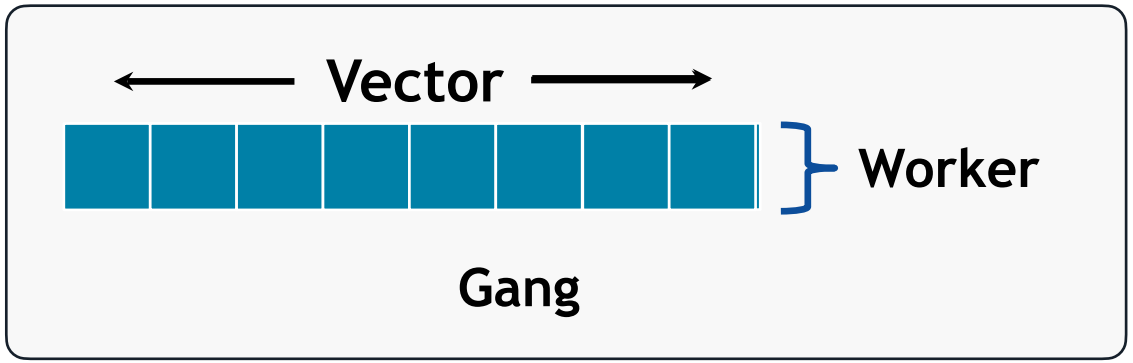
vector_length((P):

- Use a vector length of P for this parallel region

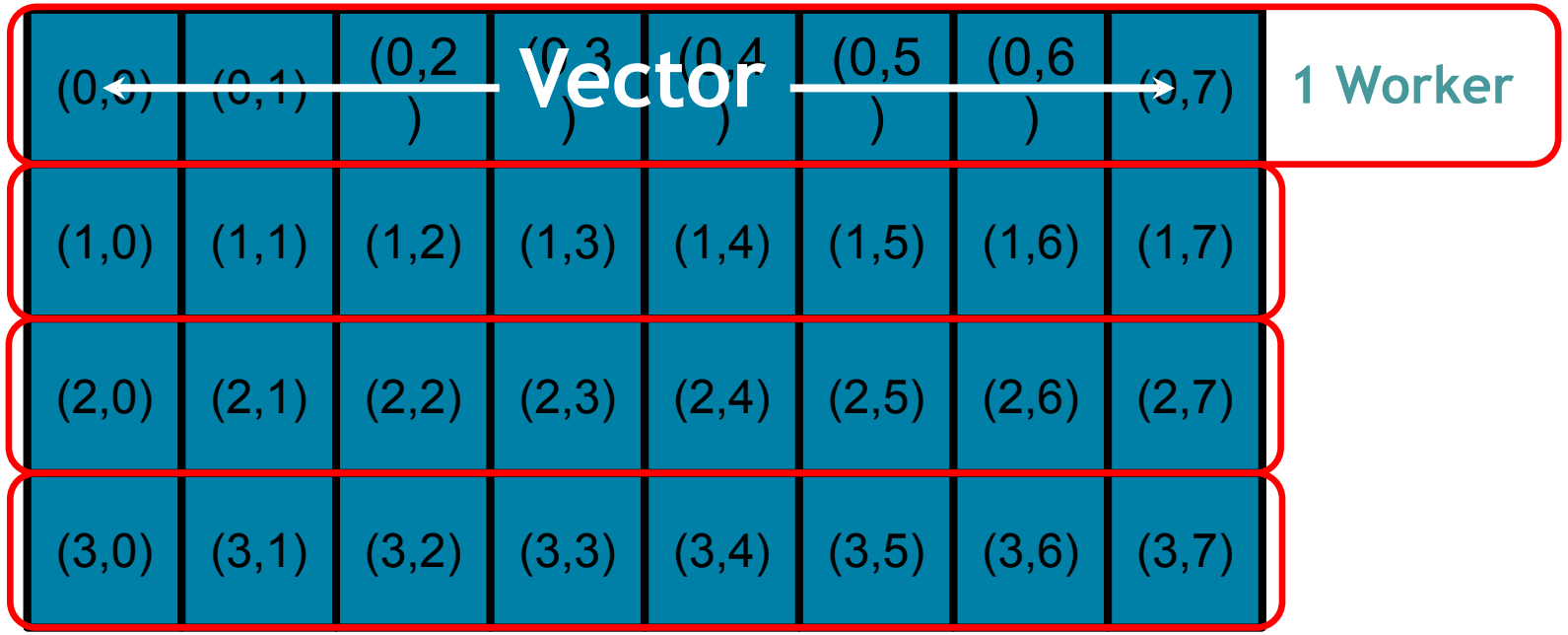
```
#pragma acc paralle nums_gangs(2) num_workers(4) vector_length(32)
 {
  #pragma acc loop worker
   for ( int i = 0; i < N; i++)
     #pragma acc loop vector
     for ( int j = 0; j < N; j++)
            structured-block
 }
```

Rule of 32: general rule of thumb for programming for NVIDIA GPUs is to always ensure that your vector length is a multiple of 32 (which means 32, 64, 96, 128, … 512, … 1024… etc.)

# Gang Worker Vector demystified



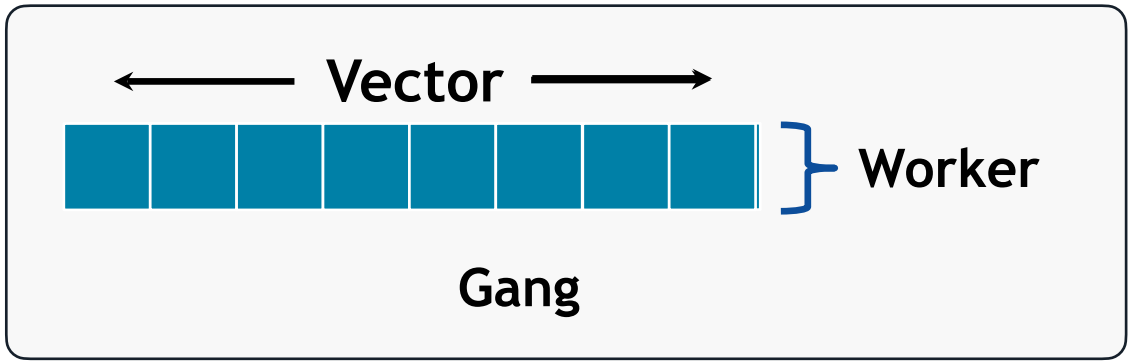

```
#pragma acc kernels loop gang worker(1)
for(int x = 0; x < 4; x++){
   #pragma acc loop vector(8)
    for(int y = 0; y < 8; y++){
        array[x][y]++;
     }
}
```



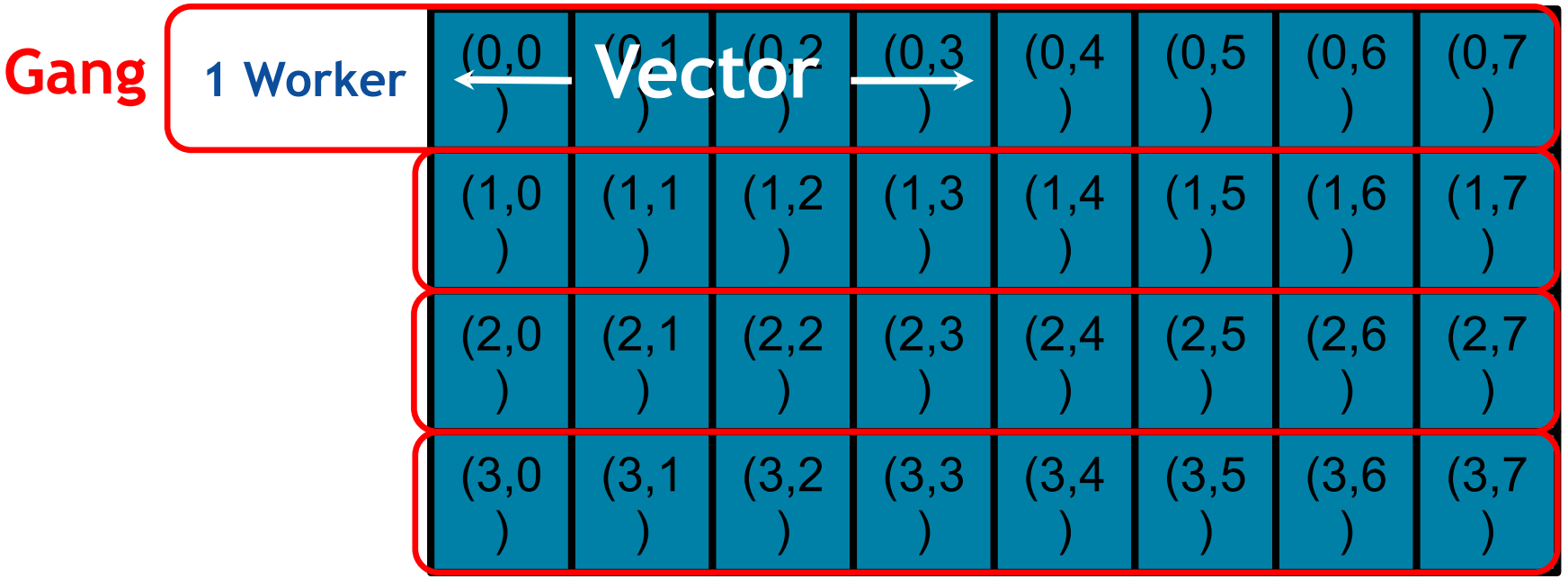

## Gang

- The vectors are colored, so that we can observe which loop iterations they are being applied to
- Based on the size of this loop nest, the compiler will (theoretically) generate **4 gangs**

# Gang Worker Vector demystified

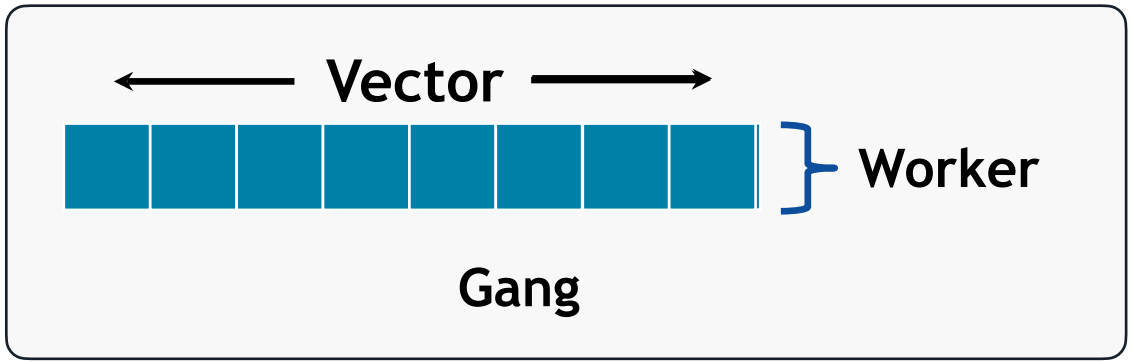



```
#pragma acc kernels loop gang worker(1)
for(int x = 0; x < 4; x++){
   #pragma acc loop vector(4)
    for(int y = 0; y < 8; y++){
        array[x][y]++;
     }
}
```

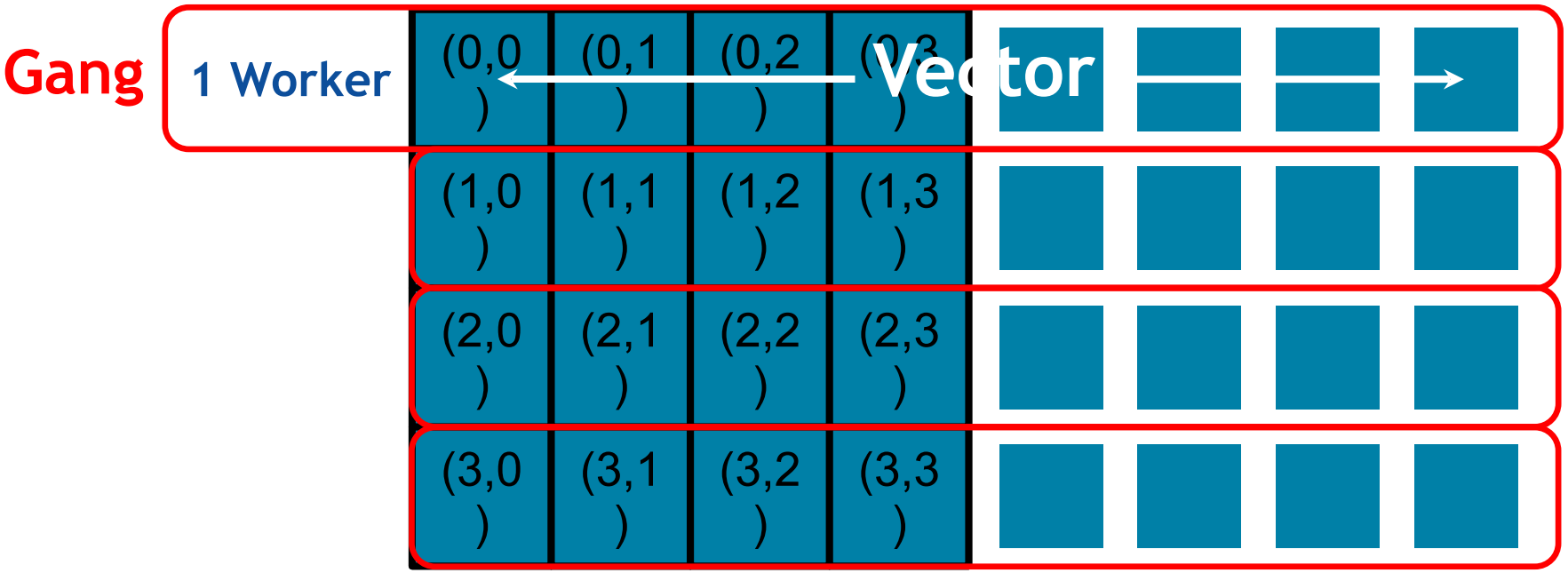



- We are still generating 4 gangs, but now each vector is computing two loop iterations
- If we wanted to generate **more gangs**, we would need to increase the size of the outer-loop

# Gang Worker Vector demystified

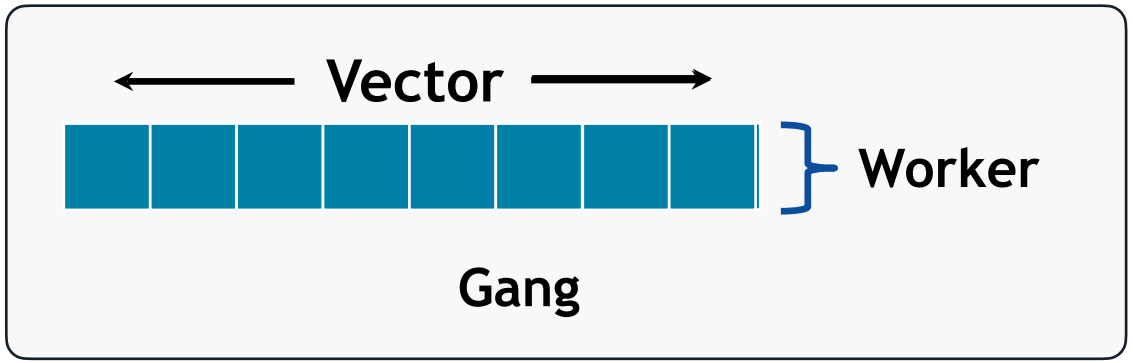



```
#pragma acc kernels loop gang worker(1)
for(int x = 0; x < 4; x++){
   #pragma acc loop vector(8)
    for(int y = 0; y < 4; y++){
        array[x][y]++;
     }
}
```



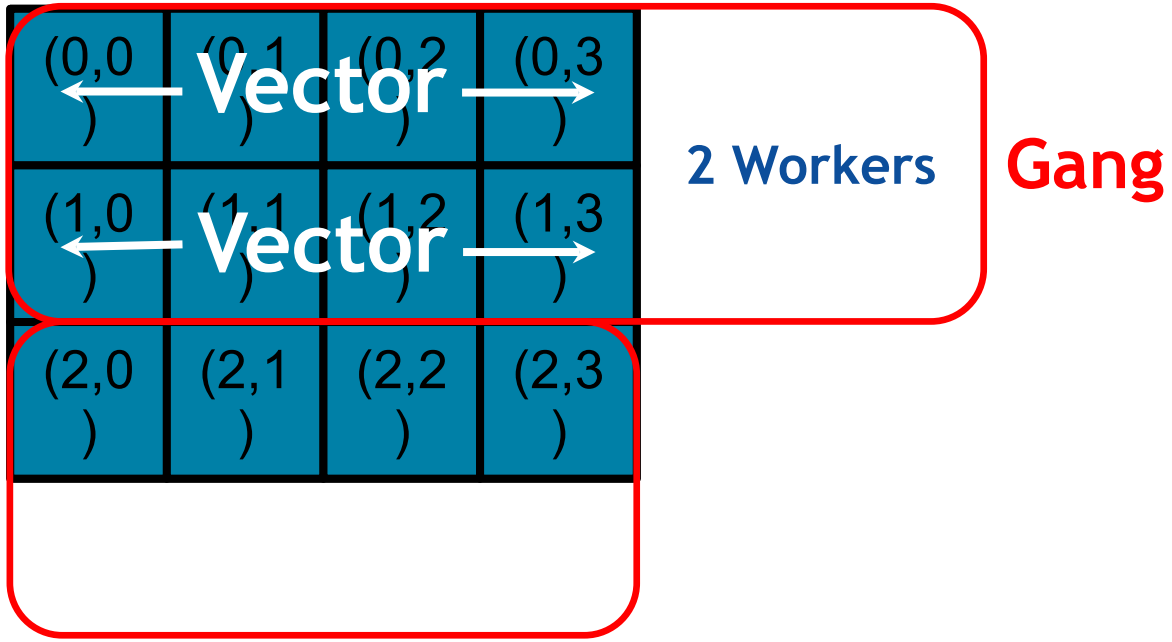

- We can see that our vector length is **much larger** than our inner-loop
- We are **wasting** half of our vector, meaning our code is performing half as well as it could

# Gang Worker Vector demystified




We can fix this by **breaking our vector** up among **2 workers**

```
#pragma acc kernels loop gang worker(2)
for(int x = 0; x < 4; x++){
   #pragma acc loop vector(4)
    for(int y = 0; y < 4; y++){
        array[x][y]++; }
}
```




- We are no longer wasting a portion of our vectors, since the smaller vector size now fits our loop properly
- We always need to consider the size of the loop when choosing the gang worker vector dimensions

# Checkpoint-3:  laplace2d_parallel: loop optimization

# Optimize the  laplace2d_OpenACC

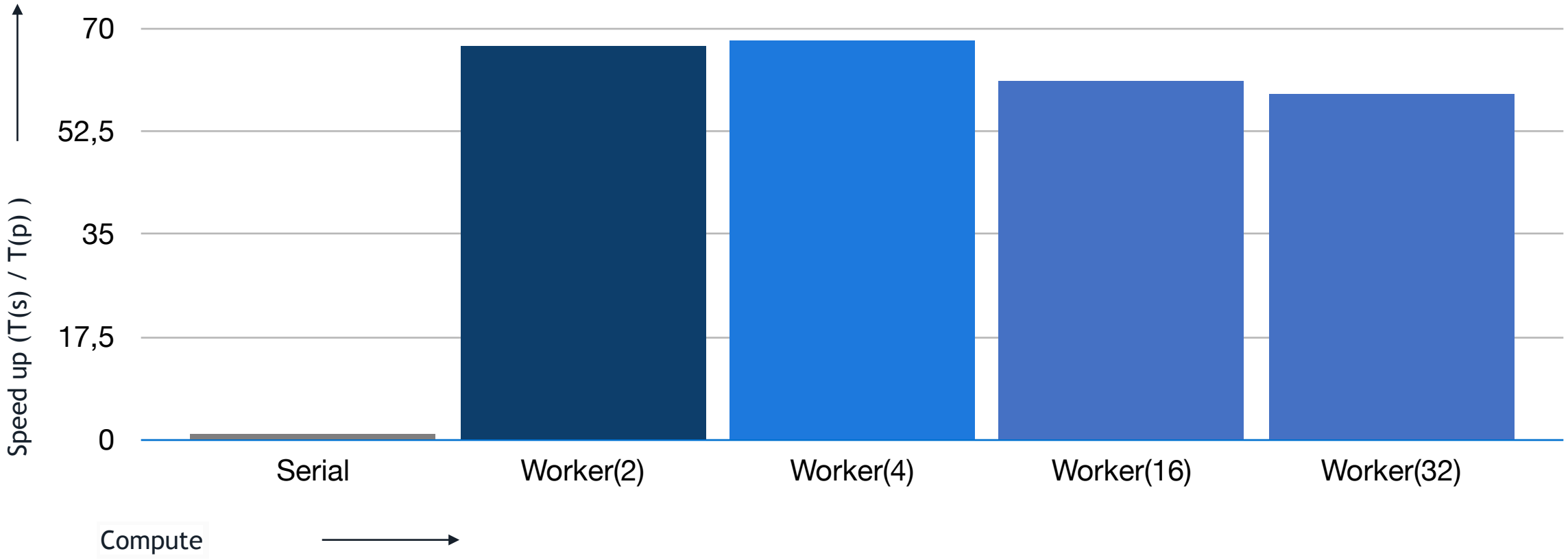
Based on the knowledge that you had optimise the Laplace code

Beat this time = 0.60 s

Try to understand the compiler report to be sure about what the compiler is doing

- nsys profile -t nvtx,openacc --stats=true --force-overwrite true -o laplace ./laplace

# Performance speed up (higher is better)



# Advanced OpenACC: Async and Multi-GPU

# Everything is Synchronous (Blocking)

By default, every OpenACC directive is a **synchronization point**.

```
// 1. CPU sits idle while Data Transfer occurs
#pragma acc data copyin(A[0:N])

// 2. CPU sits idle while GPU computes
#pragma acc parallel loop present(A)

// 3. CPU sits idle while Data Transfer occurs
```

This means the CPU must wait for the GPU to finish its command before it can proceed to the next line of code.

# Asynchronous Execution

The goal is to **hide latency** by overlapping independent tasks.

The `async(id)` clause tells the Host (CPU): "Launch this operation on the GPU, but don't wait for it to finish. Continue executing the next lines of CPU code immediately."

```
// Launches transfer on stream 1, CPU continues...
#pragma acc data copyin(A[0:N]) async(1)

// Launches compute on stream 1, CPU continues...
```

- The `id` must be an integer or integer expression (e.g., 1, 2, 3).
- Tasks launched with the same `id` are executed in order (a stream).

# Synchronization: The 'wait' Directive

Eventually, the CPU needs the GPU's result to proceed (e.g., reading the final array or moving to the next time step).

- `#pragma acc wait(id)`: Blocks the CPU until all pending operations on the specified stream (`id`) are complete.
- `#pragma acc wait` (no id): Blocks the CPU until all operations on ALL streams are complete.

```
// GPU is busy computing asynchronously on stream 1
// ... CPU can run serial work here ...

// CPU must wait for stream 1 before proceeding:
```

# Overlapping Compute and Transfer

This is the core concept of pipelining. If Task A and Task B are independent, we can run them concurrently.

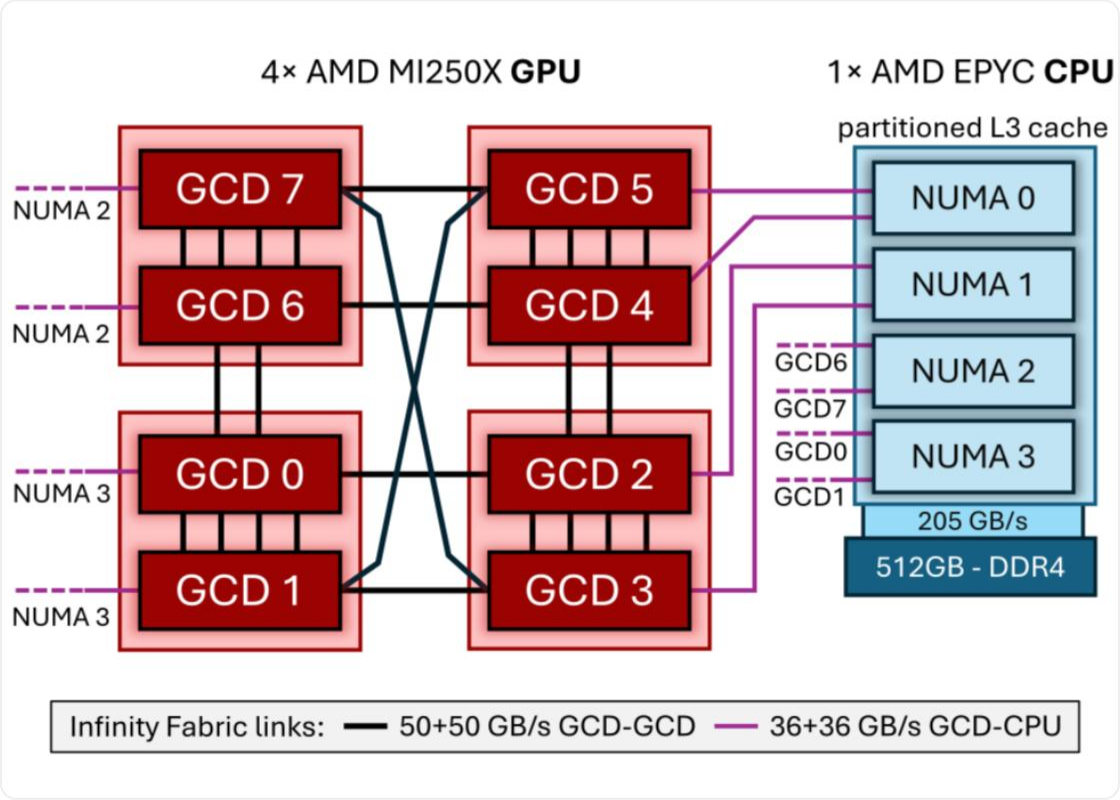
```
// Stream 1: Launches Compute (Long Task)
#pragma acc parallel loop async(1)

// Stream 2: Launches Data Transfer (Short Task)
// The GPU can often run compute and data transfer engines simultaneously!
#pragma acc update device(B[0:M]) async(2)

// CPU does not need either result yet, so it proceeds immediately.
// ... CPU does useful serial work ...

// Wait for stream 1 only when the CPU absolutely needs the result.
#pragma acc wait(1)
```

# Multi-GPU Systems

HPC Nodes often have 4 or 8 GPUs connected via NVLink.

**OpenACC:** Use `acc_set_device_num(id)` to switch focus between GPUs.

**MPI:** Typically 1 MPI Rank per GPU.



# Scaling: Multi-GPU Execution (MPI)

For applications that require multiple accelerators (e.g., on a supercomputer node), we combine OpenACC with MPI (Message Passing Interface).

1. **Initialization:** Use MPI to divide the global problem space among processes.
2. **Device Selection:** Each MPI process must select its own unique GPU.
3. **Computation:** Each process runs an OpenACC kernel on its assigned chunk of the problem.
4. **Communication:** Use MPI for inter-process communication (halo swaps, reductions).

# Scaling: Multi-GPU Execution (MPI)

Every MPI rank (process) needs to know which GPU to use.

```
int rank;
MPI_Comm_rank(MPI_COMM_WORLD, &rank);

// Use the rank ID to select the device
// This assumes one process per GPU and devices are 0, 1, 2, ...
#pragma acc set device_num(rank)
```

- **Key Concept:** Each process now operates independently on its own GPU, managing its own local data region.
- **Crucial:** Data transfers must only involve the memory managed by that local process.

# Transition towards to OpenMP offloading?

# OpenX (X = OMP, ACC) directives that are similar in nature

- OpenMP and OpenACC have similar coding paradigms under the hood
- The developer can instruct the compiler which levels of parallelism to use on given loops by adding clauses
- Applicable to a variety of hardware, but we fill focus a little bit on a GPU specific implementation

| OpenACC | CUDA | Mapping to NVIDIA GPU | OpenMP |
|---|---|---|---|
| Parallel/Kernel | Kernel | GPU | Parallel |
| Gang | Thread block | SMs | Team |
| Worker | Thread | SP or Compute unit | Thread |
| Vector | Warp (32 threads) | 32-wide thread | SIMD |

# OpenX (X = OMP, ACC) directives that are similar in nature

- In theory (according to the standards) the implementation of the levels adapt to the hardware, but in reality some compilers struggle with certain parallelisation levels

| OpenACC | OpenMP |
|---|---|
| acc parallel | omp target teams |
| acc loop gang | omp distribute |
| acc loop worker | omp parallel loop |
| acc loop vector | omp simd |
| acc declare | omp declare target |
| acc data | omp target data |
| acc update | omp target update |
| copy/copy_in/copy_out | map(tofrom/to/from:...) |

# Parallel: Similar but different

## OMP Parallel

- Creates a team of threads
- Very well-defined how the number of threads is chosen
- May synchronize within the team
- Data races are the the user's responsibility

## ACC Parallel

- Creates 1 or more gangs to workers
- Compiler free to choose number of gangs, workers, vector length
- May not synchronize between gangs
- Data races not allowed

# Loop: Similar but different

## OMP Loop (For/Do/

- Splits ("Workshares") the iterations of the next loop to threads in the team, guarantees the user has managed any data races
- Loop will be run over threads and scheduling of loop iterations may restrict the compiler

## ACC Loop

- Declares the loop iterations as independent & race free (parallel) or interesting & should be analyzed (kernels)
- User able to declare independence w/o declaring scheduling
- Compiler free to schedule with gangs/workers/vector, unless overridden by user

# Distribute vs Loop

## OMP Loop (For/Do/

- Must live in a TEAMS region
- Distributes loop iterations over 1 or more thread teams
- Only master thread of each team runs iterations, until PARALLEL is encountered
- Loop iterations are implicitly independent, but some compiler optimizations still restricted

## ACC Loop

- Declares the loop iterations as independent & race free (parallel) or interesting & should be analyzed (kernels)
- Compiler free to schedule with gangs/workers/vector, unless overridden by user

# Distribute example

```
#pragma omp target teams
{
#pragma omp distribute
  for(i=0; i<n; i++)
    for(j=0;j<m;j++)
      for(k=0;k<p;k++)
}
```

Generate a 1 or more thread teams

Distribute “i” over teams.

No information about “j” or “k” loops

```
#pragma acc parallel

#pragma acc loop
=0; i<n; i++)
#pragma acc loop
(j=0;j<m;j++)
#pragma acc loop
        for(k=0;k<p;k++)
}
```

# Distribute example

```
#pragma omp target teams
{
#pragma omp distribute
  for(i=0; i<n; i
    for(j=0;j<m;j
      for(k=0;k<p;k++)
}
```

Generate a 1 or more gangs

These loops are independent, do the *right thing*

```
#pragma acc parallel
{
#pragma acc loop
  for(i=0; i<n; i++)
#pragma acc loop
    for(j=0;j<m;j++)
#pragma acc loop
      for(k=0;k<p;k++)
}
```

# Synchronisation

## OpenMP

- Users may use barriers, critical regions, and/or locks to protect data races
- It's possible to parallelize non-parallel code

## OpenACC

- Users expected to refactor code to remove data races.
- Code should be made truly parallel and scalable

# Synchronisation Example

```
#pragma omp parallel for
for (i=0; i<N; i++)
{
#pragma omp critical
  A[i] = rand();
  A[i] *= 2;
}
```

```
parallelRand(A);
#pragma acc parallel loop
for (i=0; i<N; i++)
{
  A[i] *= 2;
}
```

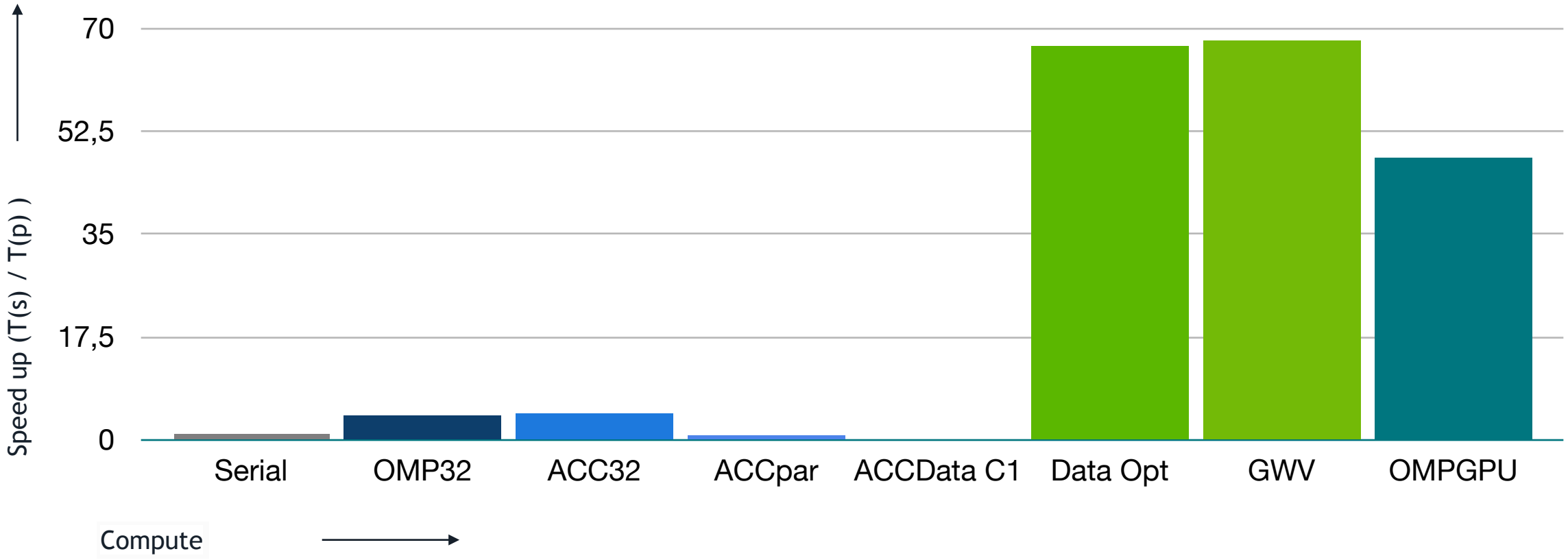
# Laplace Heat: OpenMP offloadings

## Simulation was performed 1000 Iterations on Leonardo

```
#pragma omp target data map(alloc:Anew) map(A)
  while (error > tol && niter < niter_max)
  {
  error = 0.0;
   #pragma omp parallel for reduction(max:error)
    for (int j = 1; j < n-1; ++j) {
      for (int i = 1; i < m-1; ++i) {
        Anew[idx] = 0.25 * ( A[idx+1] + A[idx-1] + A[idx-m] + A[idx+m]);
        error = fmax(error, fabs(Anew[idx] - A[idx])); }
        }

    #pragma omp target teams distributed parallel for collapse(2) schedule(static,1)
    for (int j = 1; j < n-1; ++j)
      for (int i = 1; i < m-1; ++i)
        A[j][i] = Anew[j][i]
  }
```

# Checkpoint-4:  Offload with OpenMP directives

## Performance Speed Up (Higher Is Better)



# Closing thoughts

### Accelerating C/C++/Fortran code with OpenX

Simple, Powerful, Portable

### Profile driven approach

Data dependencies and Data movement must be understood

### Optimization

Memory movement: Data regions can be used to make data movement coarse grain

Loop optimisation: use collapse, seq, independent etc

E4

COMPUTER ENGINEERING

# CURIOUS ABOUT US?