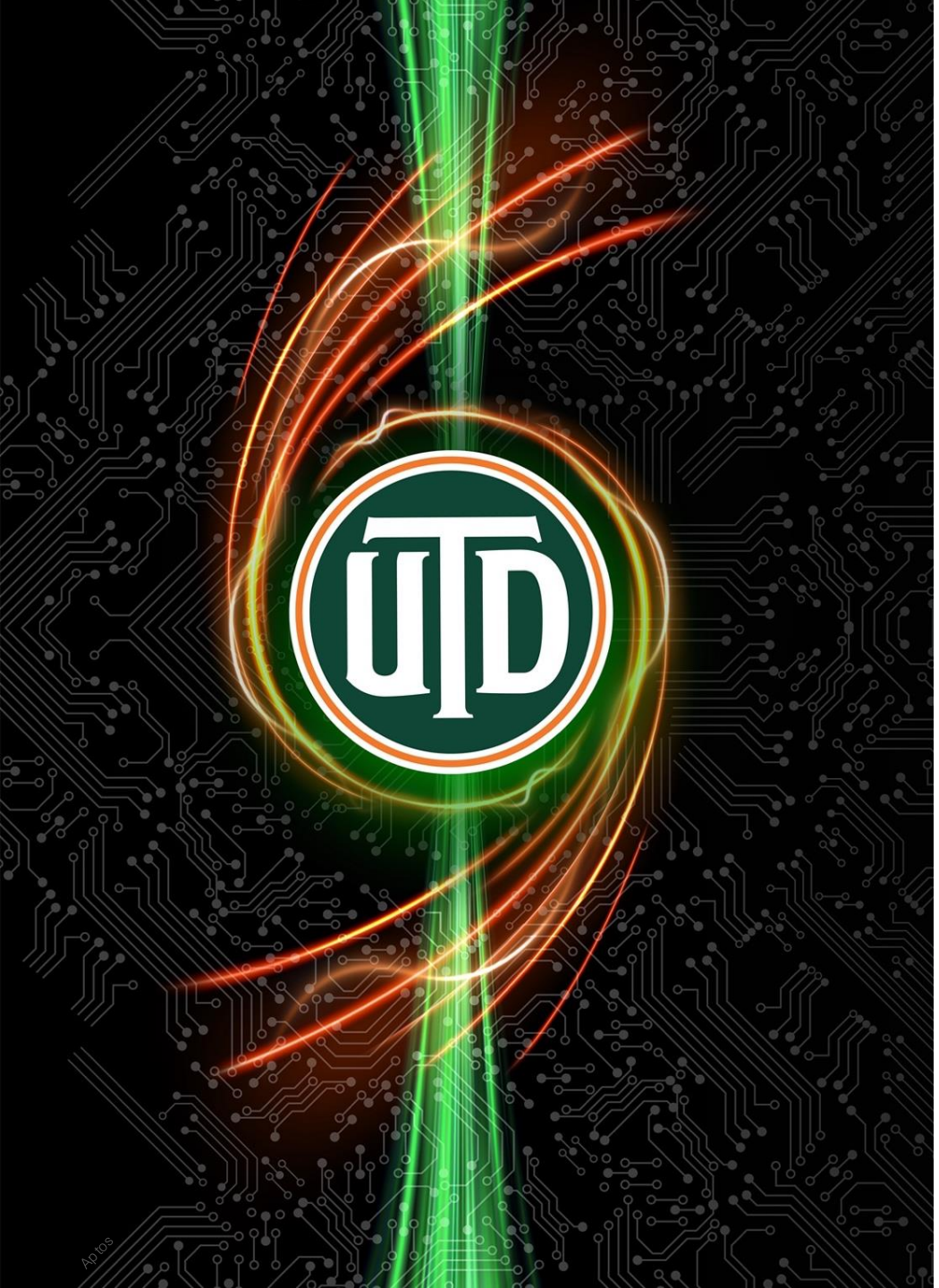
Don’t have a Juno account?

- Juno is currently our flagship high performance computing cluster.
- All materials in this workshop were tested and ready-to-run on Juno.
- Scan the QR code below or access the hyperlink to open an account if you don’t have one:

https://atlas.utdallas.edu/TDClient/30/Portal/Requests/ServiceDet?ID=493



Accelerating Python with GPUs

August 2025

HPC@UTD

Outline

- Introduction – Python and its limitations
- Overview of Python acceleration on HPC clusters
- Accelerating Python with GPUs
  - CuPy
  - Numba
  - Pytorch/TensorFlow
- Hands-on: matrix multiplication, 2D heat diffusion with Jacobi solver

- What we don’t cover today: GPU programming in C/C++/Fortran

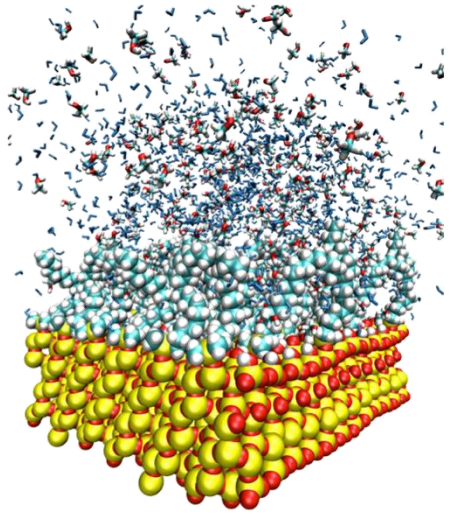
Introduction – Python and its limitations

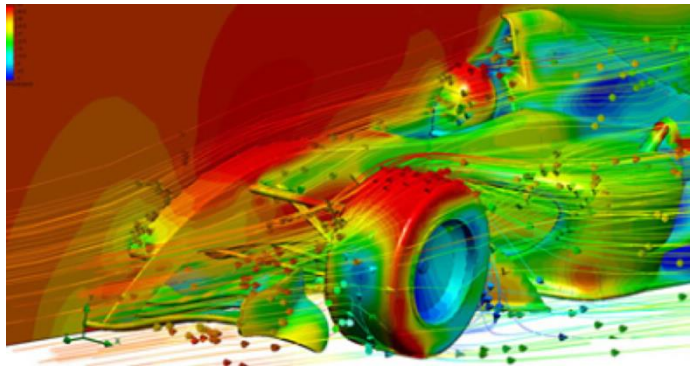
Python is omnipresent

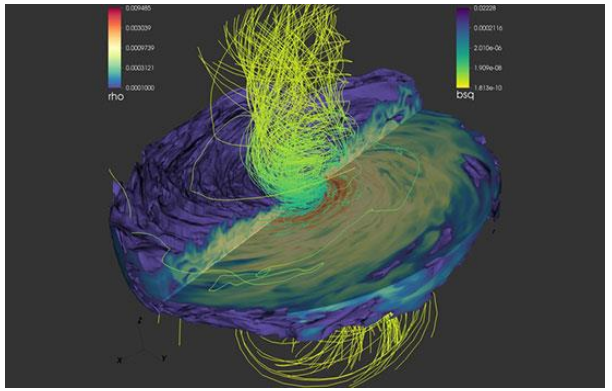
- Real world needs:
  - Scientific simulations
  - Molecular dynamics, fluid dynamics, astrophysics

    Require **millions of floating point ops per timestep**

    Real-time simulation impractical on CPUs alone

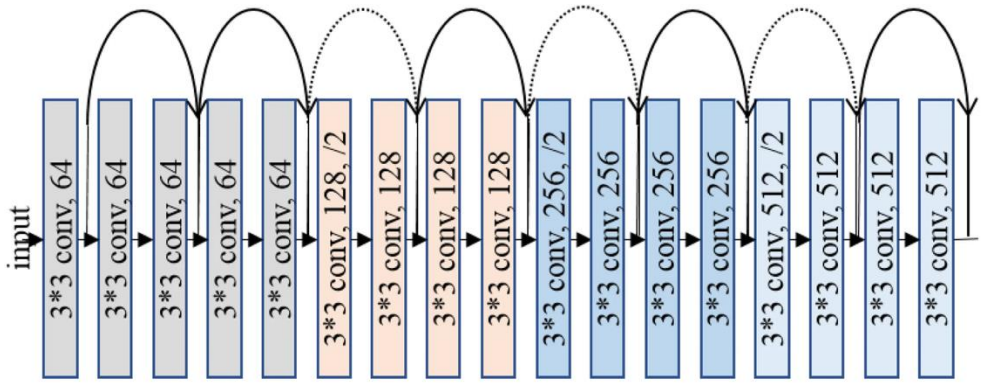






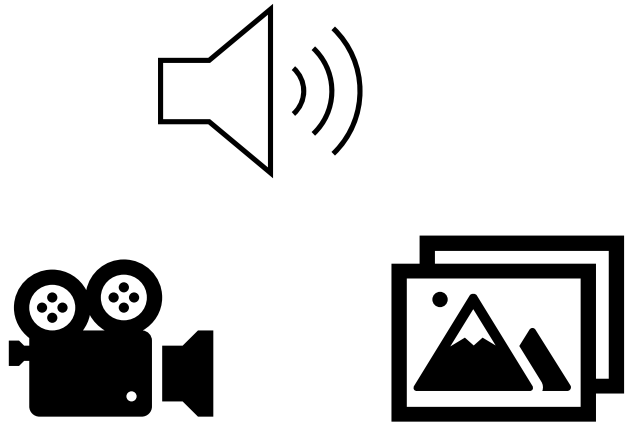
Python is omnipresent

- Real world needs:
  - Machine learning, deep learning
  - Training large models (ResNet, LLMs, Transformers)
    - Billions of weights, gradients, tensor ops → GPUs dominate
    - Even small models take hours on CPU



Python is omnipresent

- Real world needs:
  - Big data analytics
  - ETL pipelines with GB-TB datasets
    - Tools like **RAPIDS (cuDF)** mimic pandas on GPU to handle large tables
    - Image/video/audio processing at scale

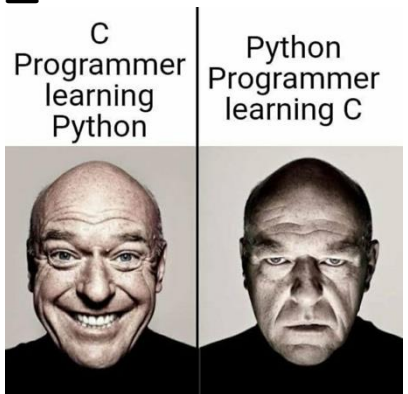


Python vs. C++

- Numpy (Python): easy to write

```python
import numpy as np
import time

# Matrix size
N = 2048

# Create random matrices
A = np.random.rand(N, N).astype(np.float32)
B = np.random.rand(N, N).astype(np.float32)

# Time the multiplication
start = time.time()
C = A @ B
end = time.time()

print(f"NumPy matrix multiplication took {end - start:.4f} seconds")
```

matrix_multiplication_numpy.py



- CUDA (C++): verbose

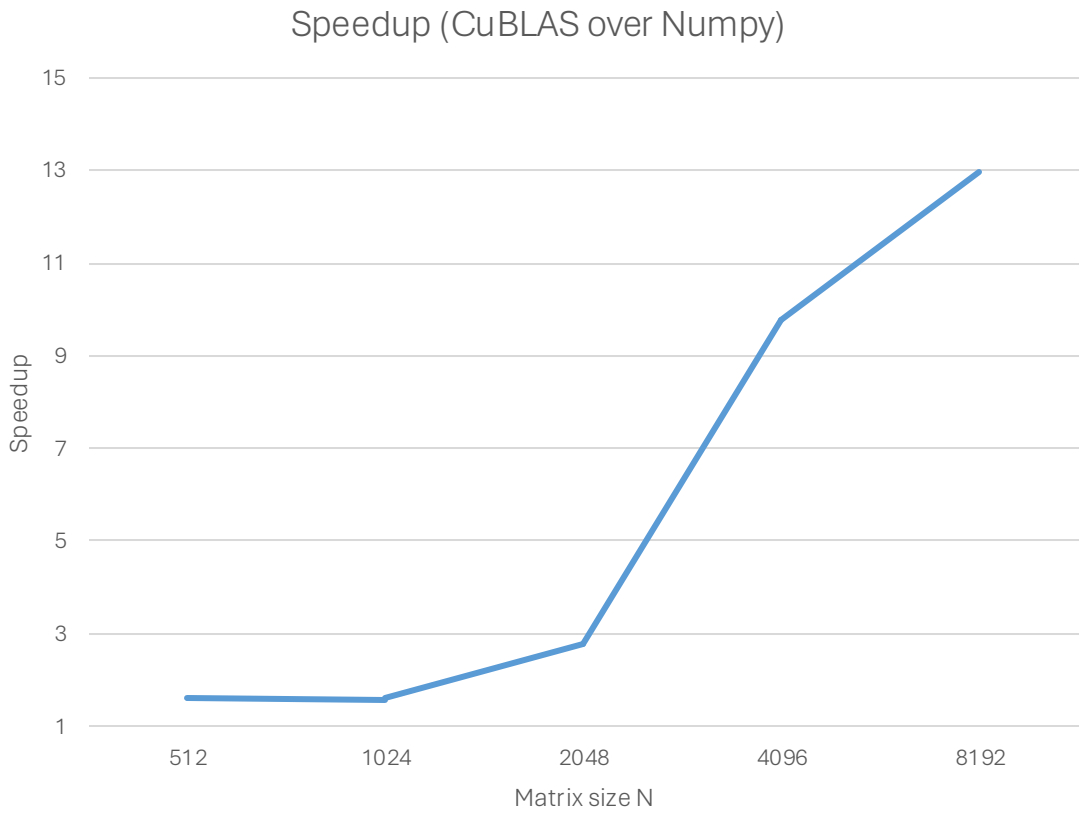
```cpp
#include <iostream>
#include <cstdlib>
#include <cuda_runtime.h>
#include <cublas_v2.h>
#include <chrono>

#define CHECK_CUDA(call) \
    if ((call) != cudaSuccess) { \
        std::cerr << "CUDA error at " << __FILE__ << ":" << __LINE__ << std::endl; \
        exit(EXIT_FAILURE); \
    }

int main() {
    const int N = 2048;
    const int SIZE = N * N;
    const float alpha = 1.0f;
    const float beta = 0.0f;

    float *h_A = (float*)malloc(SIZE * sizeof(float));
    float *h_B = (float*)malloc(SIZE * sizeof(float));
    float *h_C = (float*)malloc(SIZE * sizeof(float));

    for (int i = 0; i < SIZE; ++i) {
        h_A[i] = static_cast<float>(rand()) / RAND_MAX;
        h_B[i] = static_cast<float>(rand()) / RAND_MAX;
    }

    float *d_A, *d_B, *d_C;
    CHECK_CUDA(cudaMalloc(&d_A, SIZE * sizeof(float)));
    CHECK_CUDA(cudaMalloc(&d_B, SIZE * sizeof(float)));
    CHECK_CUDA(cudaMalloc(&d_C, SIZE * sizeof(float)));

    CHECK_CUDA(cudaMemcpy(d_A, h_A, SIZE * sizeof(float), cudaMemcpyHostToDevice));
    CHECK_CUDA(cudaMemcpy(d_B, h_B, SIZE * sizeof(float), cudaMemcpyHostToDevice));

    cublasHandle_t handle;
    cublasCreate(&handle);

    auto start = std::chrono::high_resolution_clock::now();

    cublasSgemm(
        handle, CUBLAS_OP_N, CUBLAS_OP_N,
        N, N, N,
        &alpha,
        d_B, N,
        d_A, N,
        &beta,
        d_C, N
    );

    cudaDeviceSynchronize();
    auto end = std::chrono::high_resolution_clock::now();
    double elapsed_sec = std::chrono::duration<double>(end - start).count();

    std::cout << "CUDA cuBLAS matrix multiplication took " << elapsed_sec << " seconds\n";

    CHECK_CUDA(cudaMemcpy(h_C, d_C, SIZE * sizeof(float), cudaMemcpyDeviceToHost));

    cudaFree(d_A);
    cudaFree(d_B);
    cudaFree(d_C);
    cublasDestroy(handle);
    free(h_A);
    free(h_B);
    free(h_C);

    return 0;
}
```

matrix_multiplication_cuda.cu

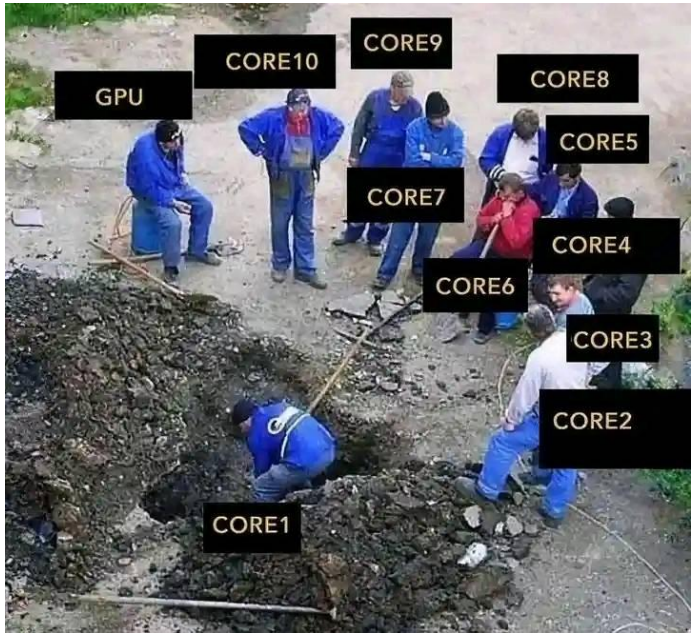
Python vs. C++

- Matrix multiplication N × N by Numpy vs. CuBLAS (on Juno A30 partition)

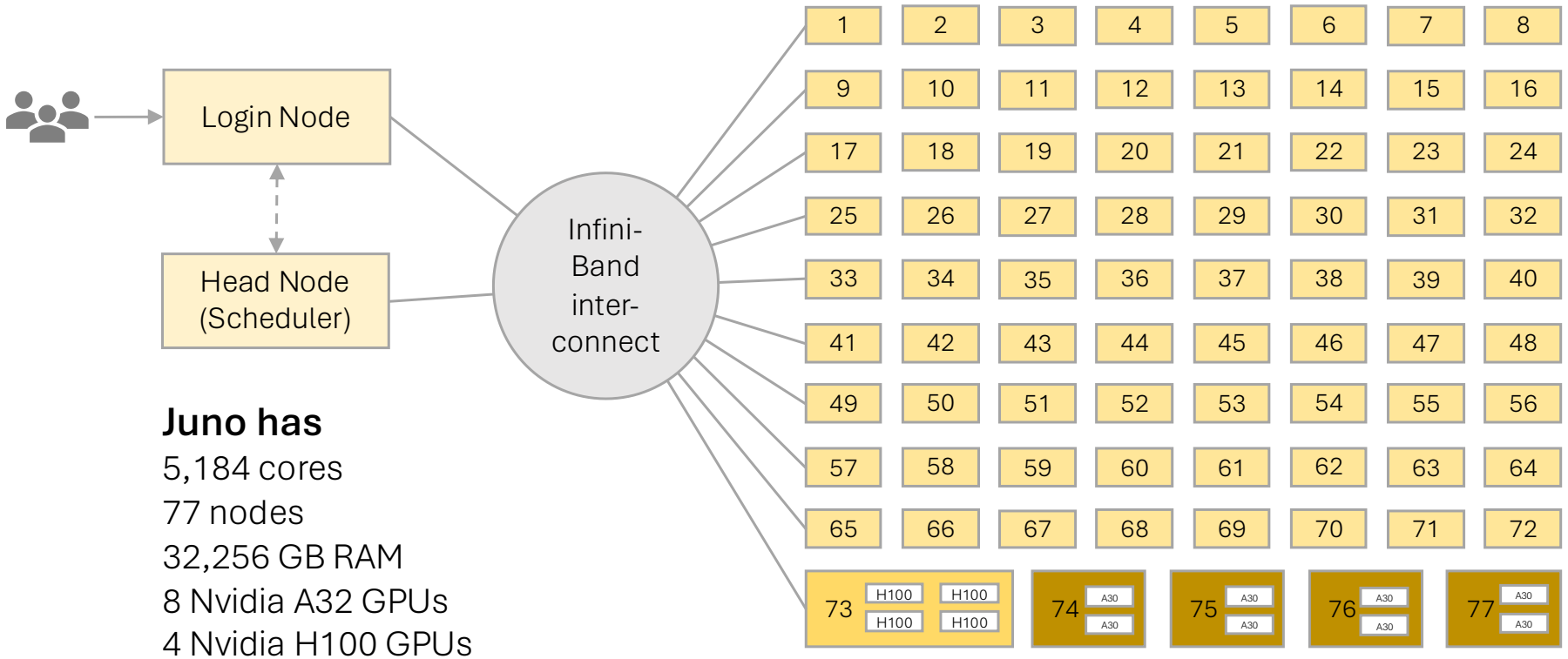
N	Numpy (s)	CuBLAS (s)	Speedup
512	0.04856	0.03024	1.6
1024	0.0484	0.03038	1.6
2048	0.09082	0.0327	2.8
4096	0.47834	0.04902	9.8
8192	2.446	0.1885	13



Overview of Python acceleration on HPC clusters

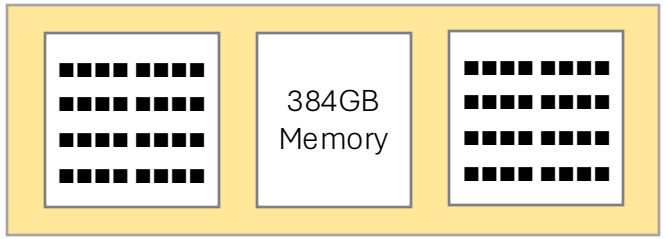


Juno’s configuration

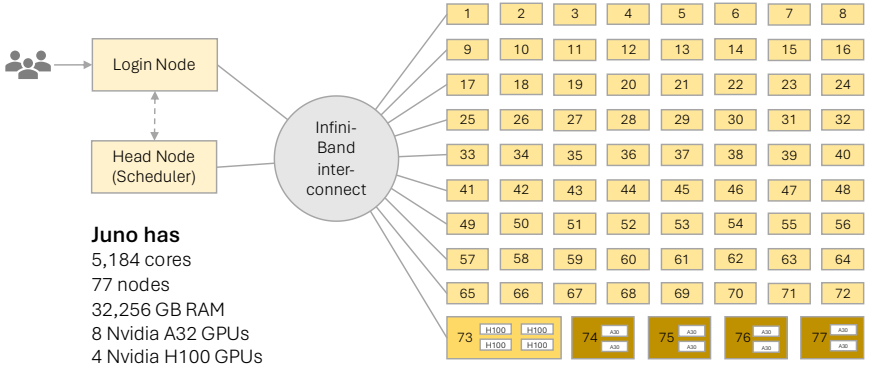


Juno has
5,184 cores
77 nodes
32,256 GB RAM
8 Nvidia A32 GPUs
4 Nvidia H100 GPUs

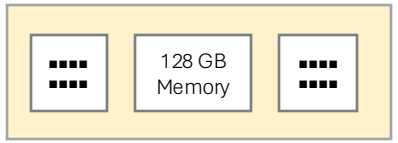
Juno’s node configuration



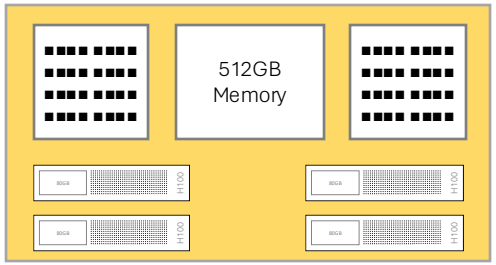
Compute Nodes (72)
2x AMD EPYC 9334 2.7GHz 32C/64T 128M cache
384GB RAM (12x32GB), 480GB SSD
HDR100 InfiniBand, 10/25Gbps Ethernet



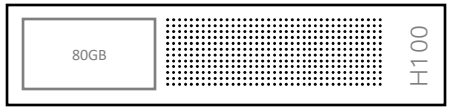
Juno has
5,184 cores
77 nodes
32,256 GB RAM
8 Nvidia A32 GPUs
4 Nvidia H100 GPUs

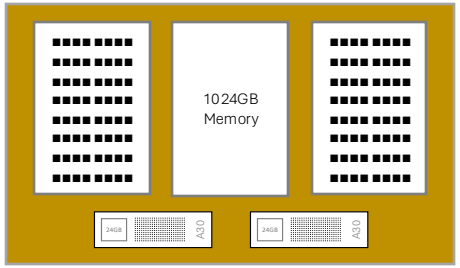


Login and Head Nodes (2)
2x Intel Xeon Sliver 4309Y 2.8GHz 8C/16T 12M cache
128 GB RAM (8x16GB), 3.84TB (2x1.92TB SSD)
HDR100 InfiniBand, 2x1Gbps Ethernet

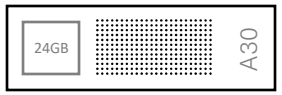


GPU Compute Node (1x4H100)
2x Intel Xeon Platinum 8462Y 2.8GHz 32C/64T 60M cache
4x Nvidia HGX H100 80GB HBM3 GPUs
512 GB RAM (32x16GB), 480GB
HDR100 InfiniBand, 2x10/25Gbps Ethernet, 2x1Gbps Ethernet



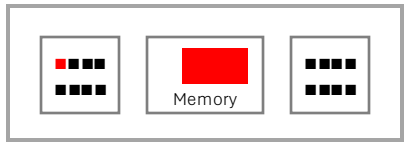


GPU Compute Nodes (4x2A30)
2x AMD EPYC 9534 2.45GHz 64C/128T 256M cache
2x Nvidia A30 24GB GPUs
1024GB RAM (16x64GB), 480GB SSD
HDR100 InfiniBand, 2x10/25Gbps Ethernet, 2x1Gbps Ethernet

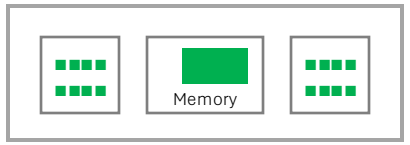


Achieving high performance on HPC clusters

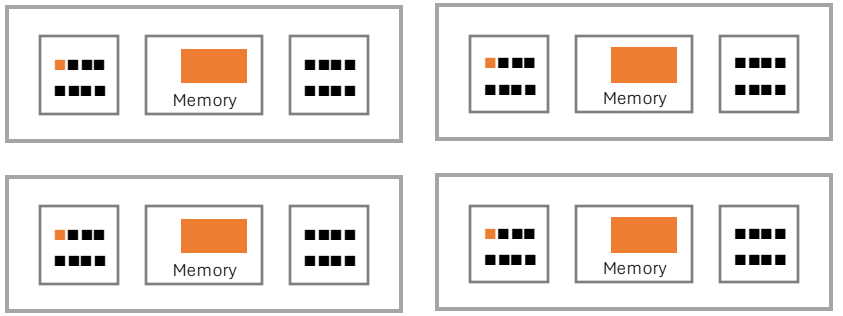
- Programs normally perform their work on just one core
  - This is called serial execution
- Achieving high performance requires the program to split their work and perform it simultaneously on multiple cores
  - This is called parallel execution
- On an HPC cluster, parallel execution can happen in 3 ways
  - Use multiple cores in single node (shared memory parallel, or multithreaded parallel execution)
  - Use cores in multiple nodes (distributed memory parallel, or message passing parallel execution)
  - Use the tiny "processors" in GPUs (GPU accelerated execution)

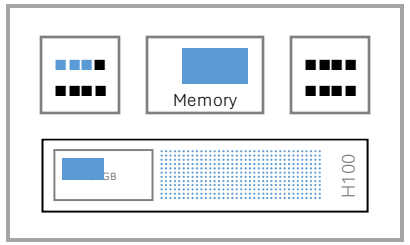


A typical Python program



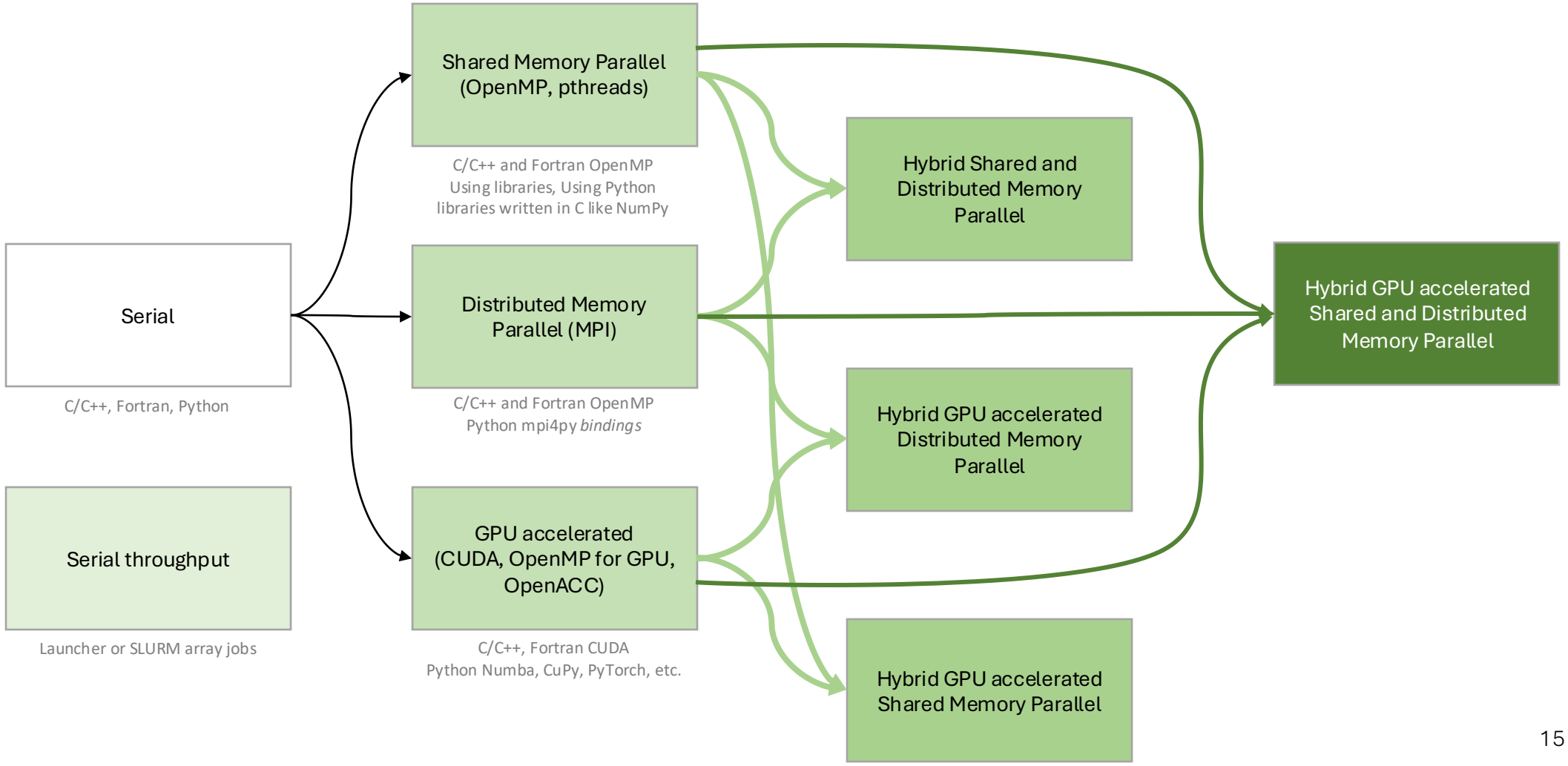
NumPy library



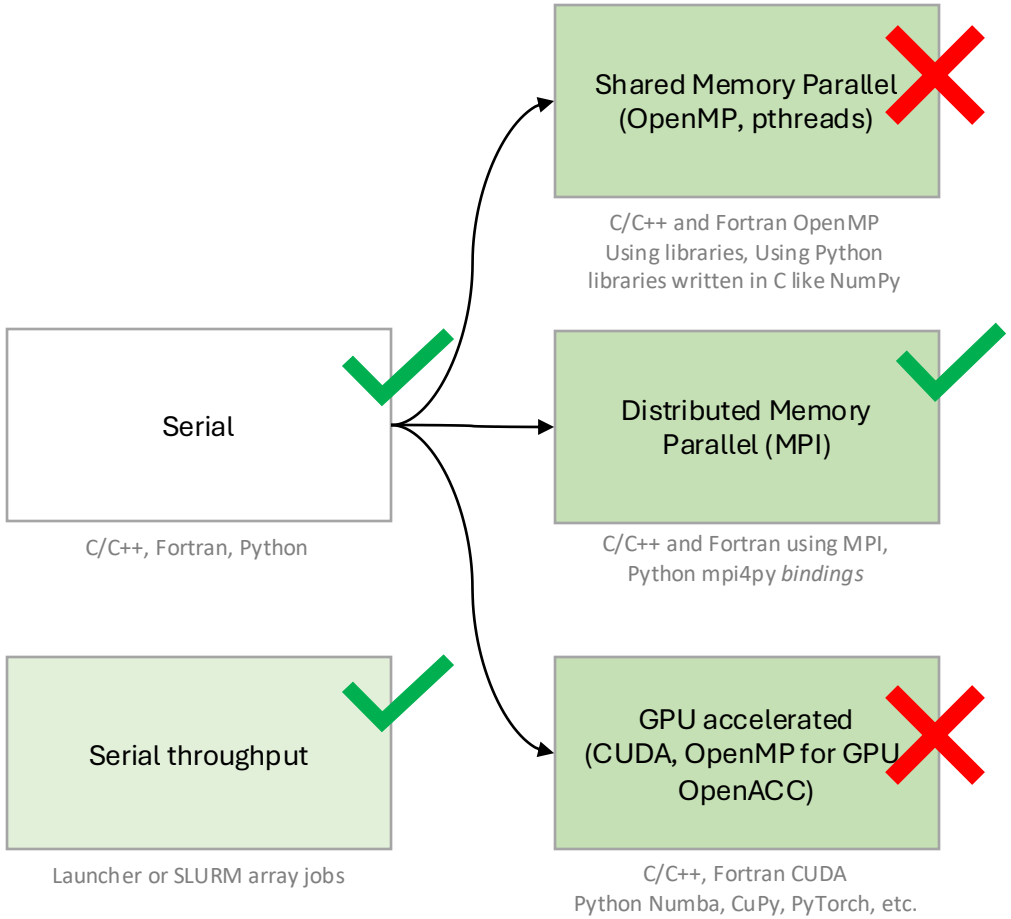


A Fortran/C/C++ program with CUDA

Models of parallel processing on HPC clusters



Can pure Python programs run in these serial and parallel modes?



Python's Global Interpreter Lock (GIL) prevents Python threads from running on multiple cores, only one thread runs on once core at a time.

Python programs can call modules utilizing library functions written in C. Those C functions can run in shared memory or multiheaded parallel mode. NumPy is an example.

The mpi4py module provides Python bindings for the MPI library.

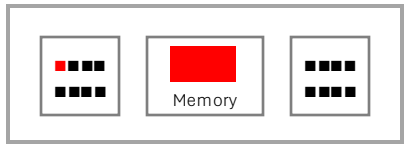
GPU programs are written using C and CUDA (C extensions and library).

The two most common ways Python programs use GPUs are:

- Use Python modules with functions written in C and CUDA (e.g., CuPy, PyTorch)
- Use Numba which translates Python code into CUDA code

Python performance bottlenecks

- Python limitations: global interpreter lock
  - Only one thread executes Python bytecode at a time—even on multi-core CPUs.
  - Limits true multithreading.
  - Even with **`threading.Thread`**, Python is often I/O-bound, not CPU-parallel.



Only one active CPU core

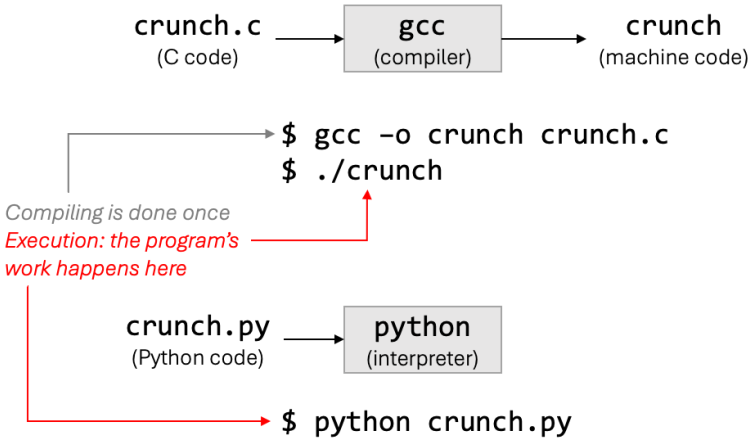
```
import threading
def work():
    for i in range(10**7): pass

threads = [threading.Thread(target=work) for _ in range(4)]
for t in threads: t.start()
```

One thread executed one after another
If in C/C++/Fortran: this spawns 4 threads simultaneously

Running Python (interpreted) programs

- C and Fortran are compiled languages
- Python programs are not compiled, but are interpreted
- An interpreter is a program that directly executes Python code line by line without the need for prior compilation into machine code
- Run a Python program

```
$ python crunch.py
```



Python performance bottlenecks

- Python limitations: memory bound operations
  - Python/Numpy are often **memory-bound**, not compute-bound.
  - CPU caches vs memory bandwidth becomes bottleneck.
  - Even multithreaded NumPy via MKL/OpenBLAS hits limits on RAM access speed.
- Python is high-level and interpreted — each instruction costs overhead.
- Loops and custom Python functions are **slow** unless vectorized or Just-in-time (JIT) compiled.

Accelerating Python with GPUs

Introduction – Python and its limitations
Overview of Python acceleration on HPC clusters
Accelerating Python with GPUs
Hands-on: 2D heat diffusion with Jacobi solver

Why GPUs?

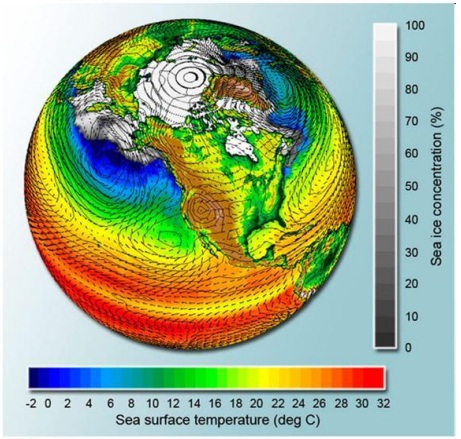
- GPU in scientific computing
  - Widely used in:

    Molecular dynamics (GROMACS, AMBER with CUDA)

    Quantum chemistry (Gaussian, TeraChem)

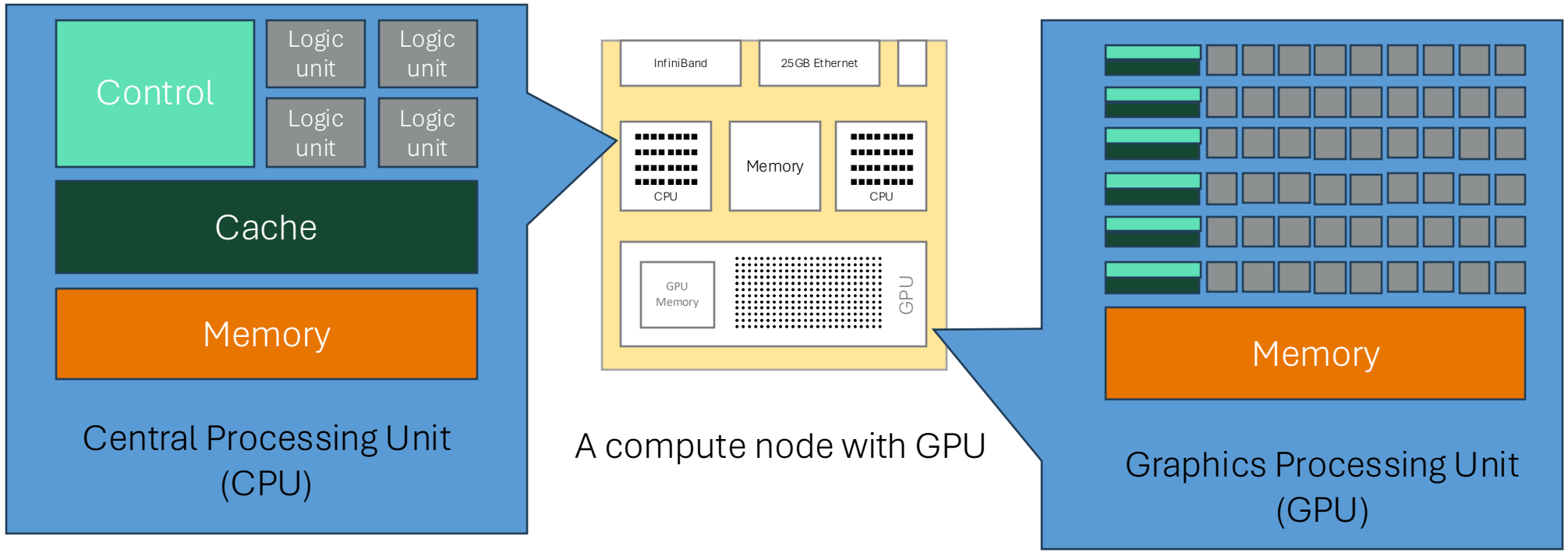
    Weather simulations, CFD, climate modeling
- GPU use cuts simulation time from **days to hours**.

GROMACS
FAST. FLEXIBLE. FREE.



Why GPUs?

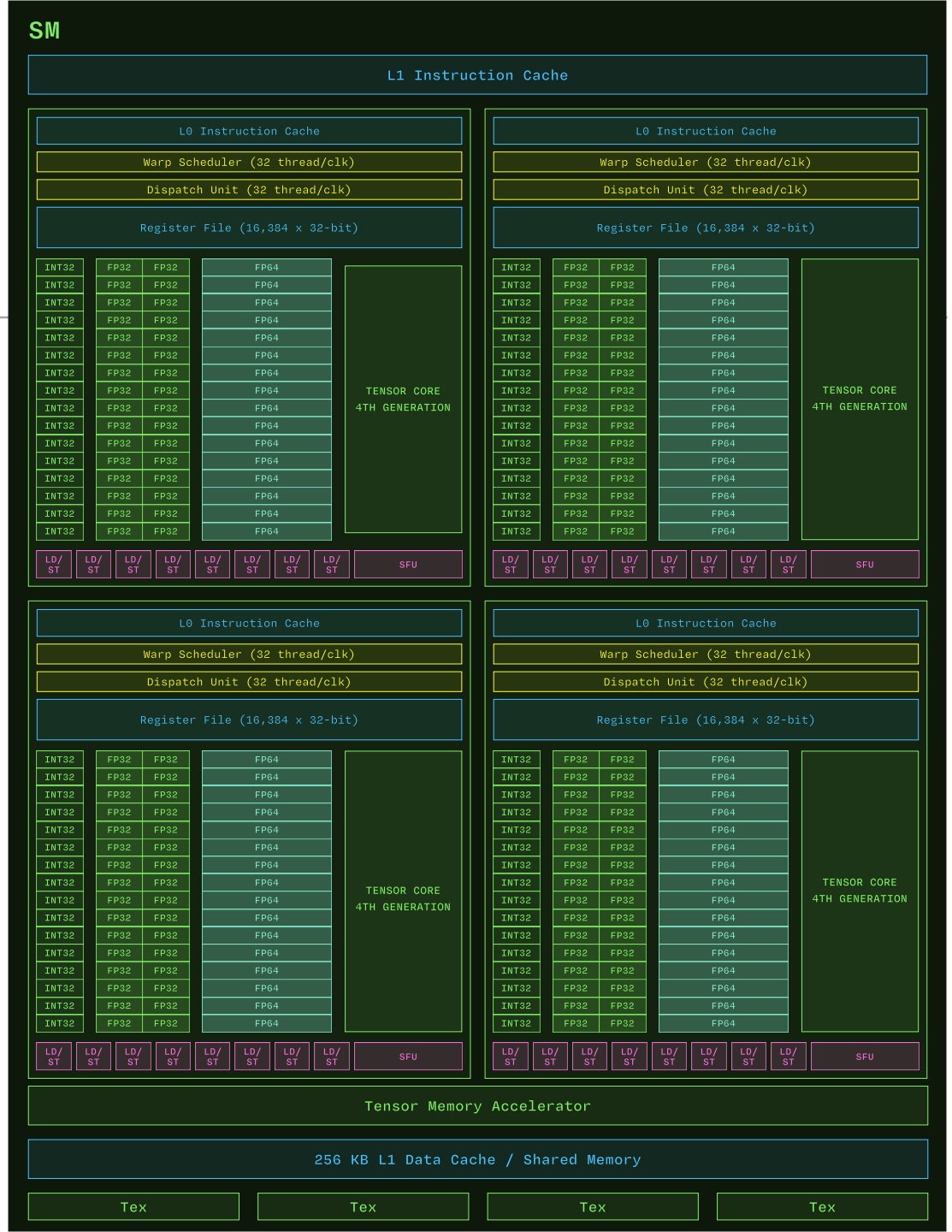
- Graphics processing unit (GPU) is originally designed for rendering digital images.
- It has potential to speed up scientific computations owing to its parallel architectures.
- GPU memory: high bandwidth, high latency -> focus on data parallelism



Why GPUs?

- CUDA cores → general-purpose math (adds, multiplies, loads, stores)
- Tensor cores → specialized matrix math (e.g., matmul, conv2d)

- If **you use Tensor Cores** (e.g., through PyTorch, TensorFlow):
  - The warp will **issue tensor math instructions**
  - Tensor cores do the heavy lifting
  - CUDA cores are **available for other warps or mixed instructions**
- If you're **not using Tensor Cores**:
  - CUDA cores perform scalar ops (slower)
  - Tensor cores sit idle



On a NVIDIA H100 GPU

https://modal.com/gpu-glossary/device-hardware/cuda-device-architecture

CPU vs. GPU: What's the real difference?

Feature	CPU	GPU
Core count	Few (4–64)	Thousands (e.g., 10,000+)
Clock speed	Higher (3–5 GHz)	Lower (~1–2 GHz)
Memory cache	Large & fast	Smaller, specialized
Parallelism model	Optimized for latency	Optimized for throughput
Target workload	Serial, branching logic	Massive parallelism, math-heavy

Why Not Just Use More CPUs?

- CPUs hit limits due to:
  - Power consumption
  - Heat dissipation
  - Memory bandwidth
- GPUs scale better **per watt** and **per dollar** for math-heavy workloads.
  - For example, a GPU may perform **10 teraflops (trillion floating-point operations per second)** with 300 watts of power consumption, costing $1500, while a CPU may perform **1 teraflop** with 100 watts of power consumption, costing $500.
- **Conclusion: Python + GPU = high-level control + low-level power**

https://www.icdrex.com/cpu-vs-gpu-which-one-is-right-for-your-workload/
https://www.pugetsystems.com/solutions/ai-and-hpc-workstations/scientific-computing/hardware-recommendations/?srsltid=AfmBOooaFMs20QZYxEf-lTswVtCRpIUOwrxjf1NVzEBpw7-PwlgI_vMl&utm

Why GPUs?

- Every major ML framework runs on GPU: PyTorch, TensorFlow, JAX
- Training deep neural networks = tons of matrix/tensor math (dot products, convolutions, activations)
- GPUs accelerate **forward & backward passes** via optimized tensor cores (e.g., in RTX/Quadro cards)
- For example, **`conv2d`**, **`matmul`**, **`softmax`** run in parallel and have GPU options

GPU programming – a typical C program

```
#include <iostream>
#include <cuda_runtime.h>
#define N 4  // rows
#define M 4  // columns
__global__ void scale_matrix(float* matrix, float scale, int rows, int cols) {
    int i = blockIdx.y * blockDim.y + threadIdx.y;  // row
    int j = blockIdx.x * blockDim.x + threadIdx.x;  // column


    if (i < rows && j < cols) {
        int idx = i * cols + j;
        matrix[idx] *= scale;
    }
}

int main() {
    float scale = 2.0f;
    float h_matrix[N * M];


    // Initialize host matrix
    for (int i = 0; i < N * M; ++i) {
        h_matrix[i] = static_cast<float>(i);
    }


    float* d_matrix;
    size_t size = N * M * sizeof(float);
    cudaMalloc(&d_matrix, size);
    cudaMemcpy(d_matrix, h_matrix, size, cudaMemcpyHostToDevice);


    dim3 blockSize(16, 16);
    dim3 gridSize((M + blockSize.x - 1) / blockSize.x,
                  (N + blockSize.y - 1) / blockSize.y);


    scale_matrix<<<gridSize, blockSize>>>(d_matrix, scale, N, M);
    cudaMemcpy(h_matrix, d_matrix, size, cudaMemcpyDeviceToHost);


    // Print result
    std::cout << "Scaled matrix:\n";
    for (int i = 0; i < N; ++i) {
        for (int j = 0; j < M; ++j) {
            std::cout << h_matrix[i * M + j] << "\t";
        }
        std::cout << "\n";
    }

    cudaFree(d_matrix);
    return 0;
}
```

Define kernel (GPU) function

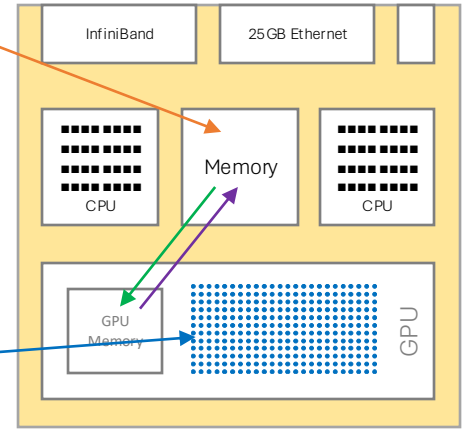
1. Allocate array(s) and initialize on CPU

2. Allocate array(s) on GPU and copy data to GPU

3. Compute on GPU

4. Copy data from GPU back to CPU

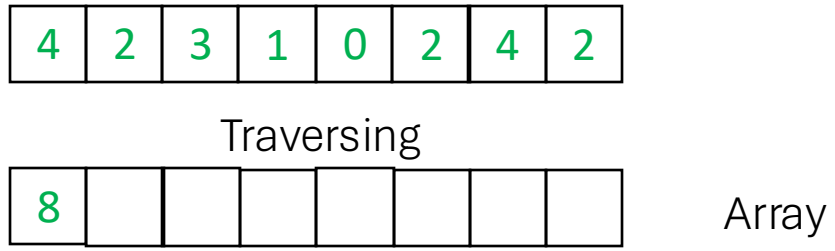
5. Free up memory



Kernels

- A *kernel function* is a GPU function that is meant to be launched from CPU code:
  - kernels cannot explicitly return a value → all data must be written in an array.
  - kernels explicitly declare their thread hierarchy when called: i.e. the number of thread blocks and the number of threads per block

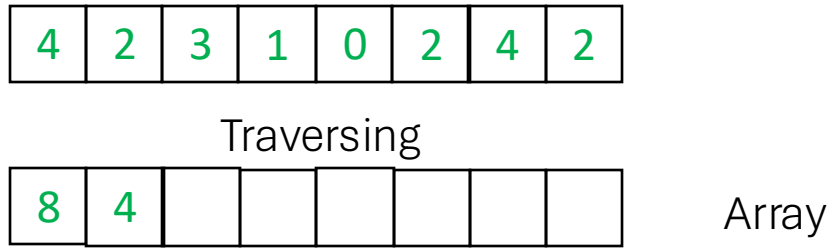
```
CPU code
void scale_array(float* arr, float scale, int size) {
    for (int i = 0; i < size; ++i) {
        arr[i] *= scale;
    }
}
```



Kernels

- A *kernel function* is a GPU function that is meant to be launched from CPU code:
  - kernels cannot explicitly return a value → all data must be written in an array.
  - kernels explicitly declare their thread hierarchy when called: i.e. the number of thread blocks and the number of threads per block

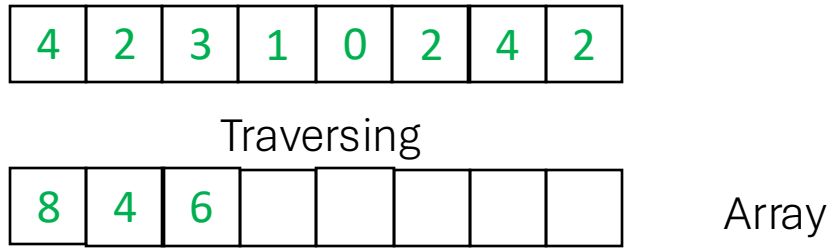
```
CPU code
void scale_array(float* arr, float scale, int size) {
    for (int i = 0; i < size; ++i) {
        arr[i] *= scale;
    }
}
```



Kernels

- A *kernel function* is a GPU function that is meant to be launched from CPU code:
  - kernels cannot explicitly return a value → all data must be written in an array.
  - kernels explicitly declare their thread hierarchy when called: i.e. the number of thread blocks and the number of threads per block

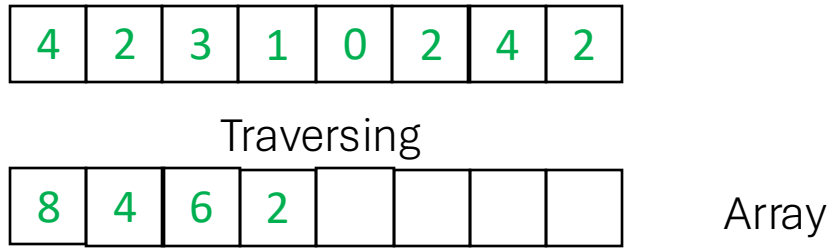
```
CPU code
void scale_array(float* arr, float scale, int size) {
    for (int i = 0; i < size; ++i) {
        arr[i] *= scale;
    }
}
```



Kernels

- A *kernel function* is a GPU function that is meant to be launched from CPU code:
  - kernels cannot explicitly return a value → all data must be written in an array.
  - kernels explicitly declare their thread hierarchy when called: i.e. the number of thread blocks and the number of threads per block

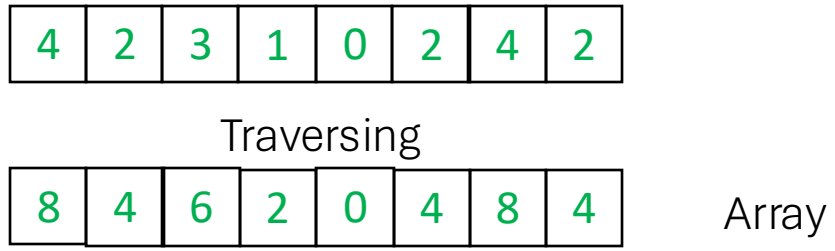
```
CPU code
void scale_array(float* arr, float scale, int size) {
    for (int i = 0; i < size; ++i) {
        arr[i] *= scale;
    }
}
```



Kernels

- A *kernel function* is a GPU function that is meant to be launched from CPU code:
  - kernels cannot explicitly return a value → all data must be written in an array.
  - kernels explicitly declare their thread hierarchy when called: i.e. the number of thread blocks and the number of threads per block

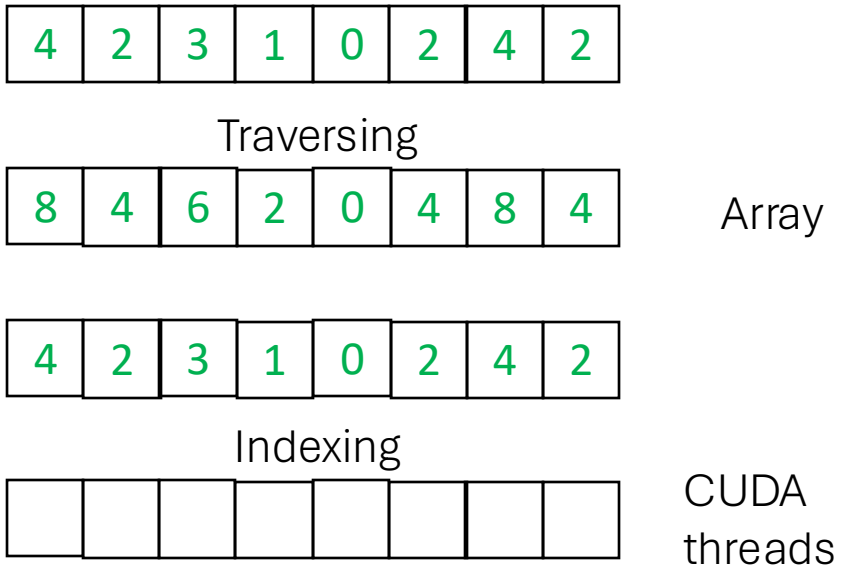
```
CPU code
void scale_array(float* arr, float scale, int size) {
    for (int i = 0; i < size; ++i) {
        arr[i] *= scale;
    }
}
```



Kernels

- A *kernel function* is a GPU function that is meant to be launched from CPU code:
  - kernels cannot explicitly return a value → all data must be written in an array.
  - kernels explicitly declare their thread hierarchy when called: i.e. the number of thread blocks and the number of threads per block

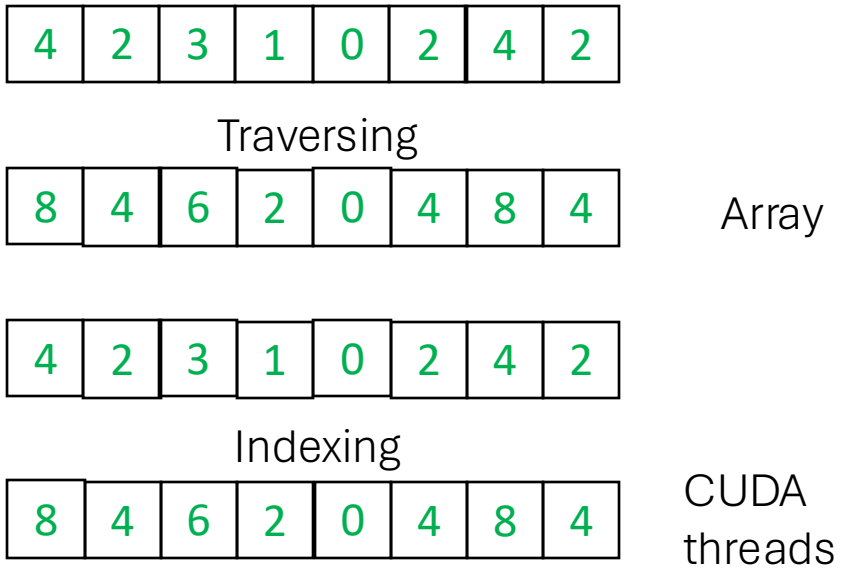
```
CPU code
void scale_array(float* arr, float scale, int size) {
    for (int i = 0; i < size; ++i) {
        arr[i] *= scale;
    }
}

GPU kernel
__global__ void scale_array(float* arr, float scale, int size) {
    int idx = blockIdx.x * blockDim.x + threadIdx.x;

    if (idx < size) {
        arr[idx] *= scale;
    }
}
```



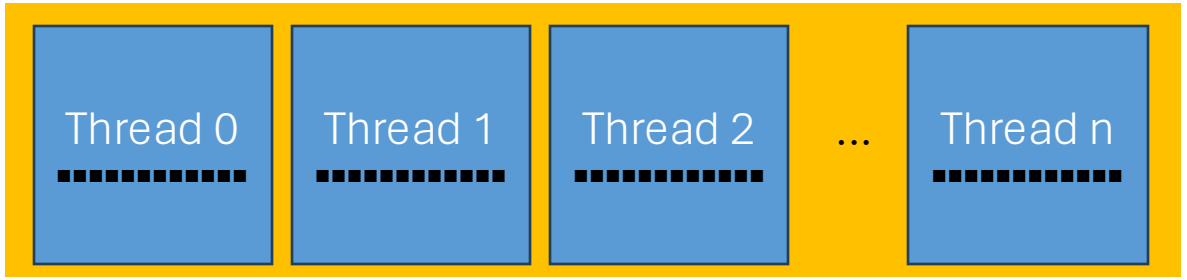
Kernels

- A *kernel function* is a GPU function that is meant to be launched from CPU code:
  - kernels cannot explicitly return a value → all data must be written in an array.
  - kernels explicitly declare their thread hierarchy when called: i.e. the number of thread blocks and the number of threads per block

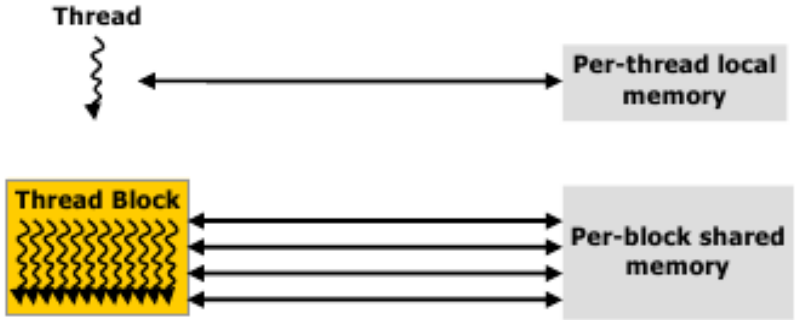
```
CPU code
void scale_array(float* arr, float scale, int size) {
    for (int i = 0; i < size; ++i) {
        arr[i] *= scale;
    }
}

GPU kernel
__global__ void scale_array(float* arr, float scale, int size) {
    int idx = blockIdx.x * blockDim.x + threadIdx.x;

    if (idx < size) {
        arr[idx] *= scale;
    }
}
```

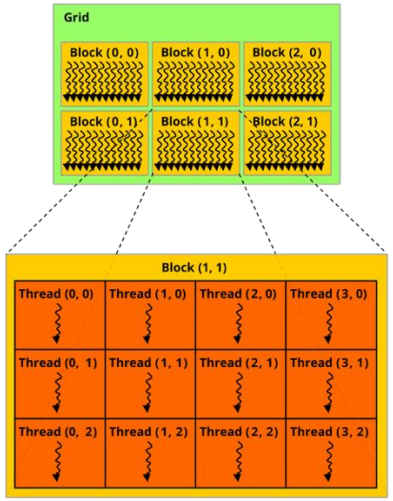


Thread hierarchy




```
int i = blockIdx.x * blockDim.x + threadIdx.x;
```



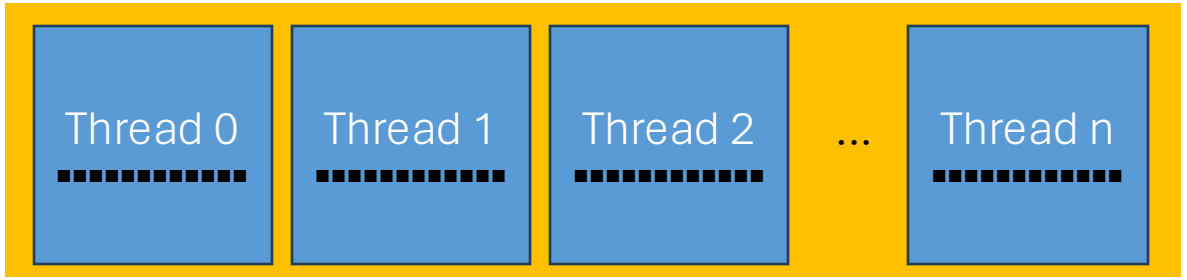


GPU kernel

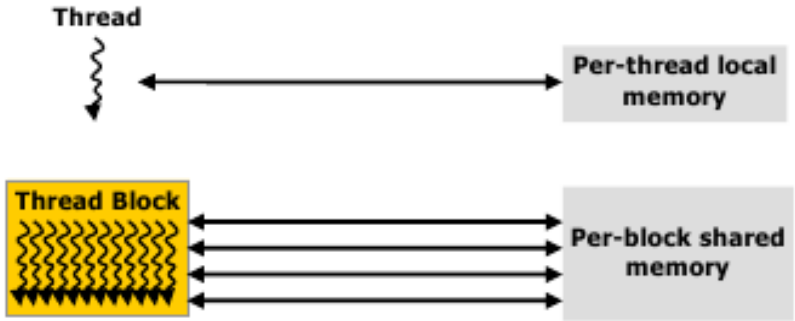
```
__global__ void scale_array(float* arr, float scale,
int size) {
    int idx = blockIdx.x * blockDim.x + threadIdx.x;

    if (idx < size) {
        arr[idx] *= scale;
    }
}
```

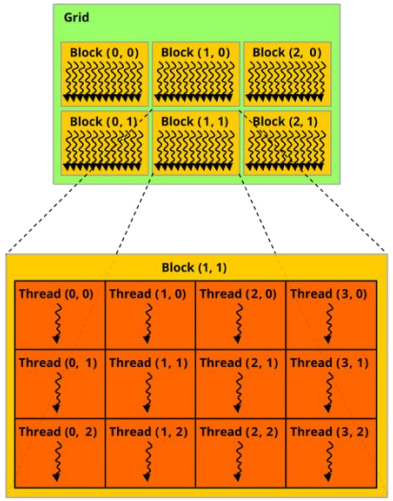
https://docs.nvidia.com/cuda/archive//11.7.1/cuda-c-programming-guide/index.html

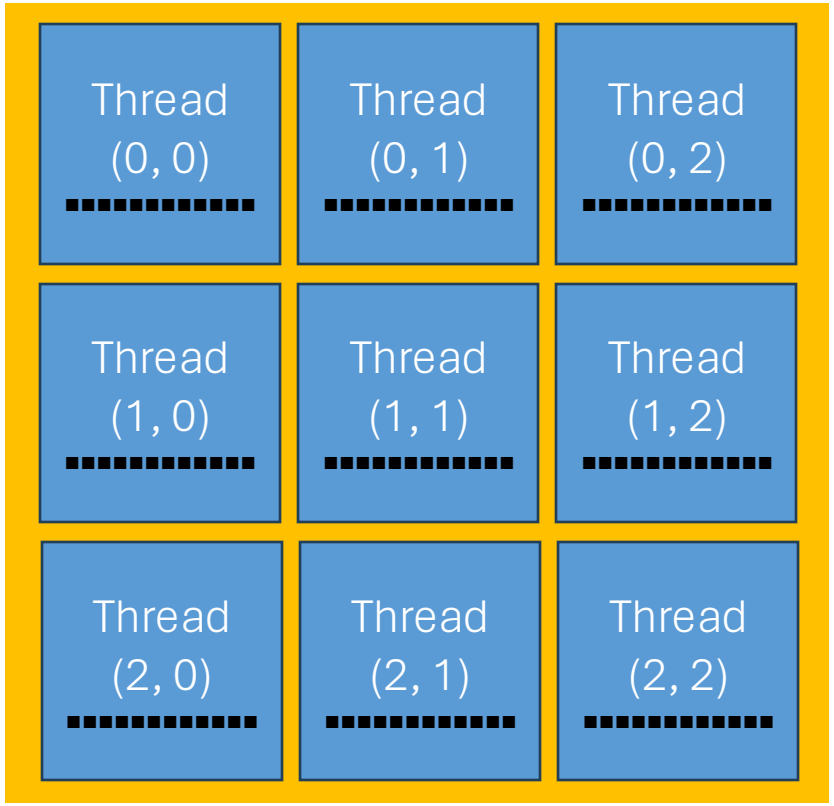
Thread hierarchy




```
int i = blockIdx.x * blockDim.x + threadIdx.x;
```

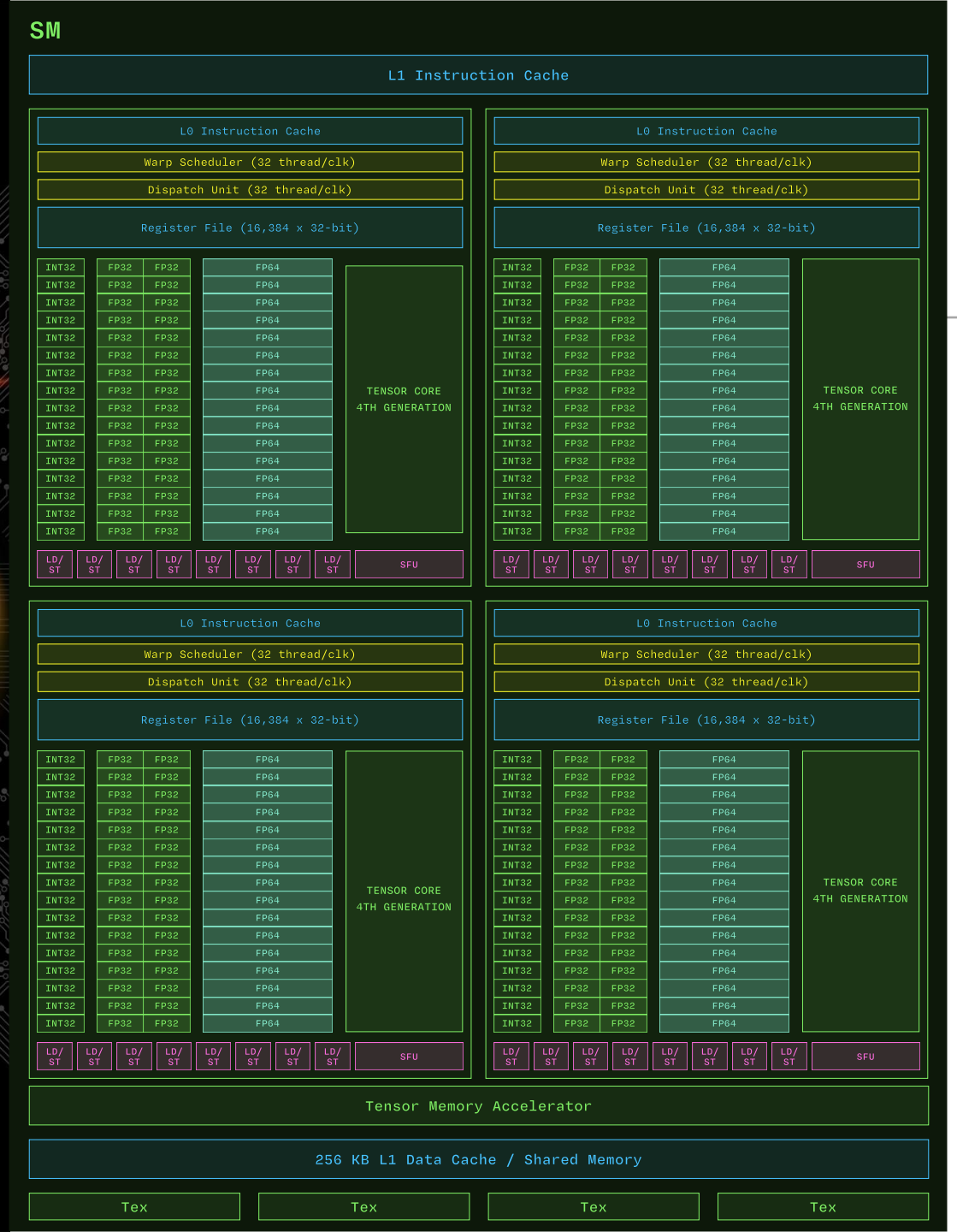







```
int i = blockIdx.y * blockDim.y +
threadIdx.y;  // row
int j = blockIdx.x * blockDim.x +
threadIdx.x;  // column
```

https://docs.nvidia.com/cuda/archive//11.7.1/cuda-c-programming-guide/index.html

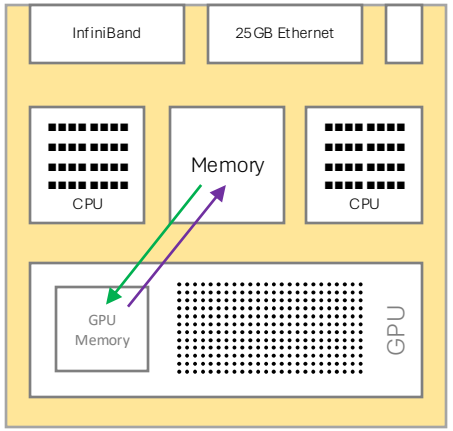


Thread hierarchy

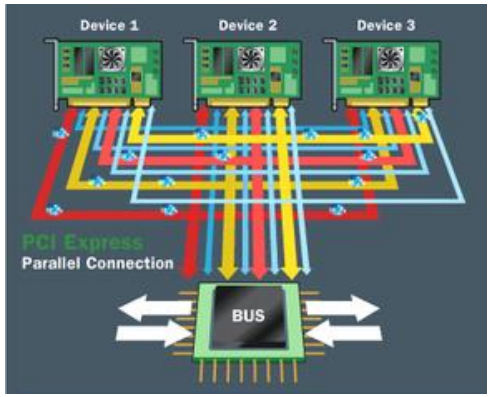
- **Thread block:** a group of threads defined when launching a kernel. A block can have **any number of threads (up to 1024)**.
- Warps:
  - group of **32 threads** that execute the **same instruction simultaneously** on a CUDA-enabled GPU
  - NVIDIA GPUs execute instructions at the warp level, not individual threads (the GPU scheduler dispatches one **instruction** to an entire warp)
  - GPU **cannot share a warp between blocks, even if the blocks are small (threads per block < 32)**

Data transfer

- **Host (CPU) - Device (GPU) transfers:**
  - From CPU to GPU
  - From GPU to CPU
  - From GPU to other GPU

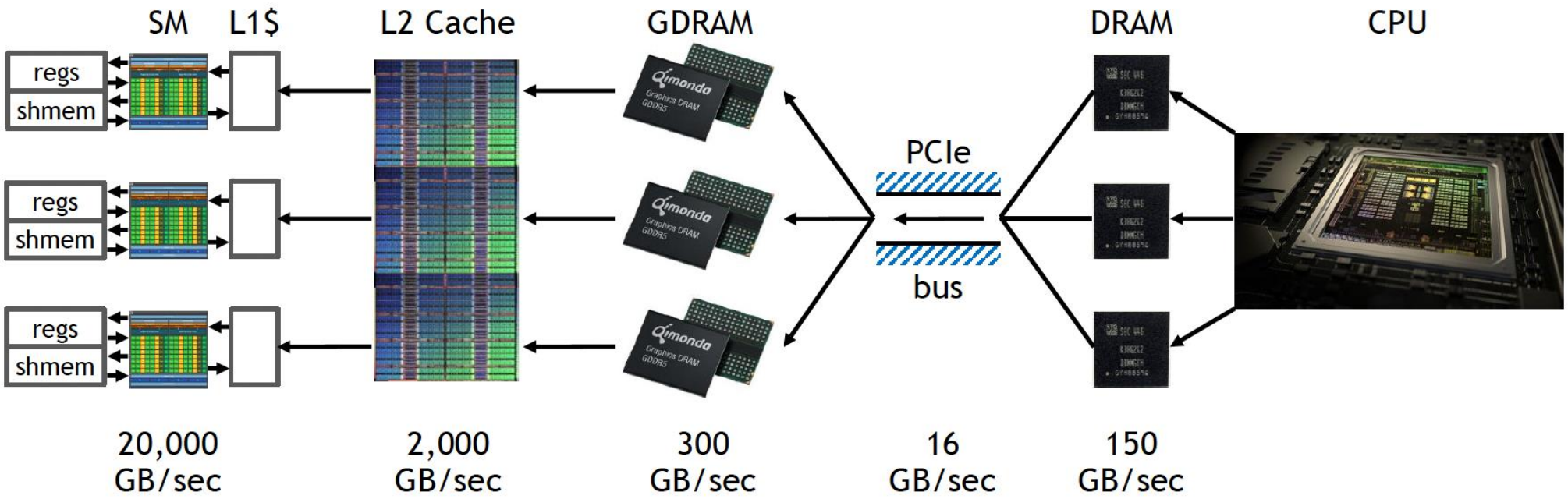


- Data transfers by Peripheral Component Interconnect Express (PCIe) buses.



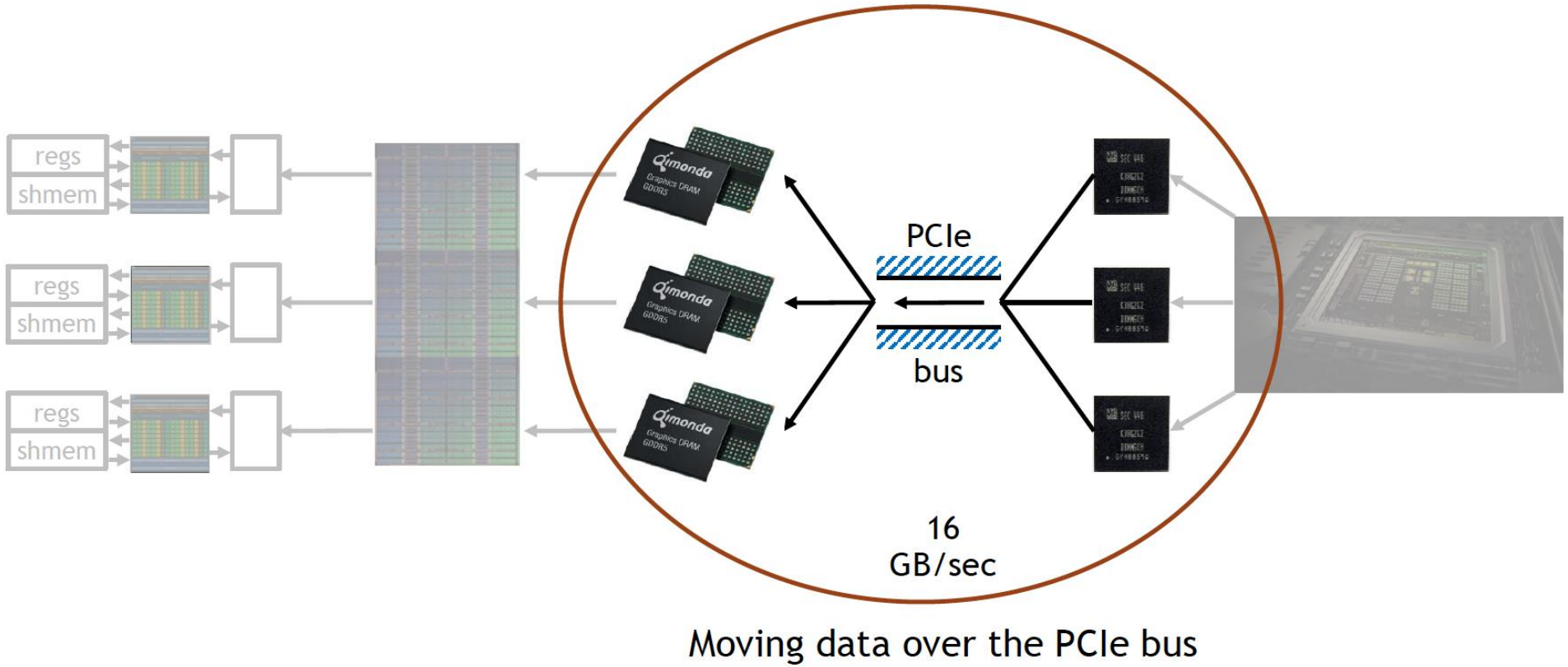
https://computer.howstuffworks.com/pci.htm

Data transfer



From NVIDIA

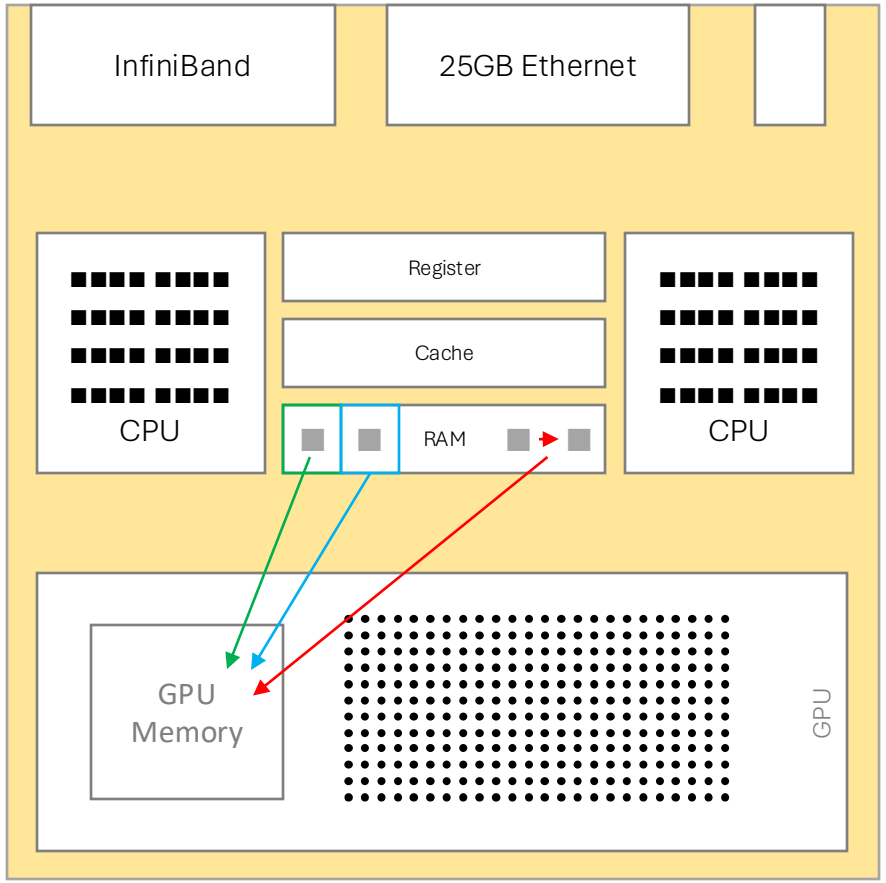
Data transfer



Moving data over the PCIe bus

From NVIDIA

Types of data transfer



Default transfer (pageable memory):

1. The data is first copied to a temporary pinned buffer
2. The data is transferred to GPU

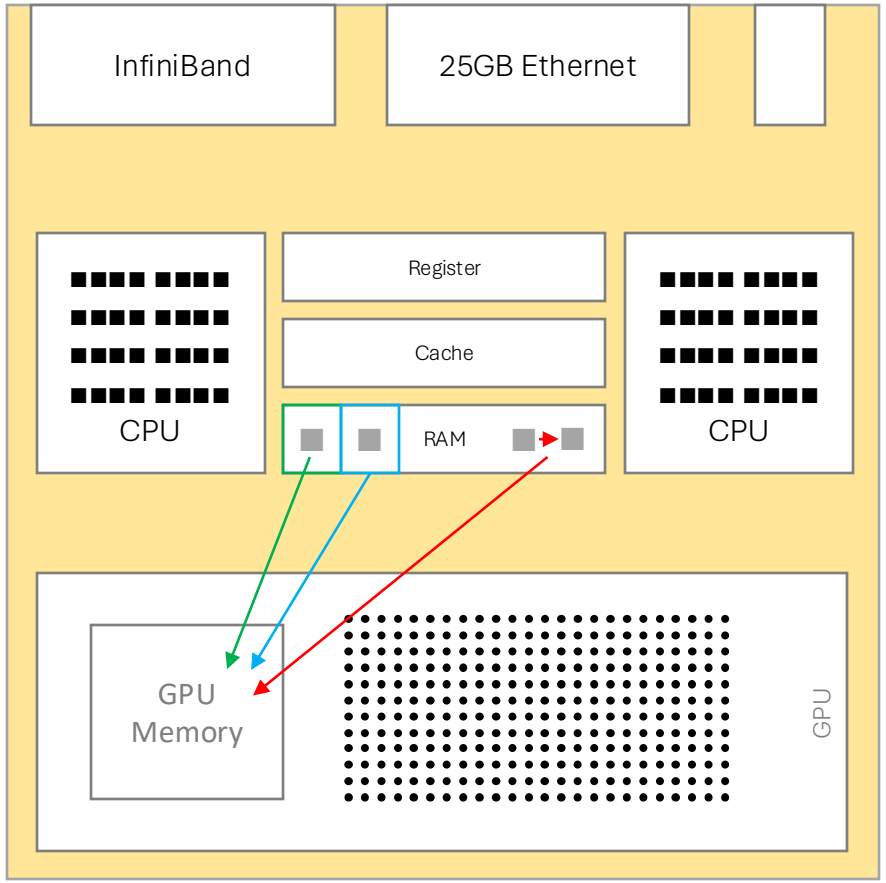
Pinned (page-locked) memory:

1. Prior to allocation, RAM makes a compartment
2. The data is directly transferred to GPU via direct memory access (DMA)

Mapped memory:

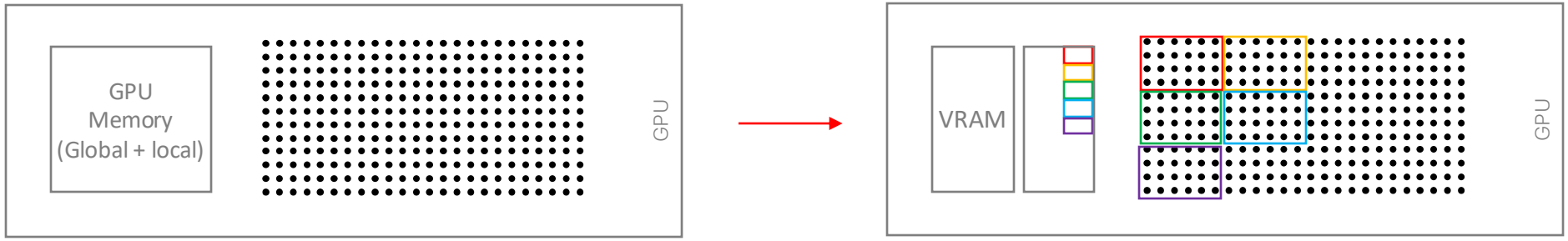
1. RAM exposes the address to GPU
2. The address can be seen and accessed by GPU

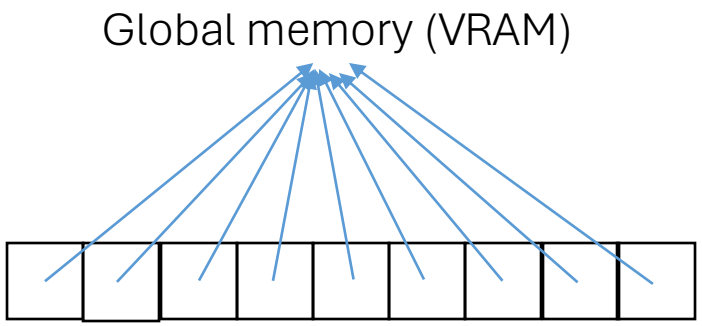
Types of data transfer

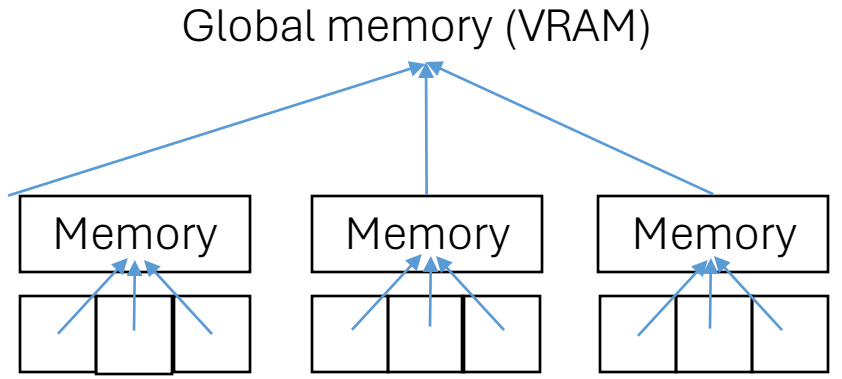


Feature	Pinned Memory	Mapped Memory
GPU can directly access?	No	Yes
Uses DMA?	Yes	No
Typical performance	High for big transfers	Slow for random access
Use case	Fast, repeated copies	Small or sparse access

Shared memory on GPU



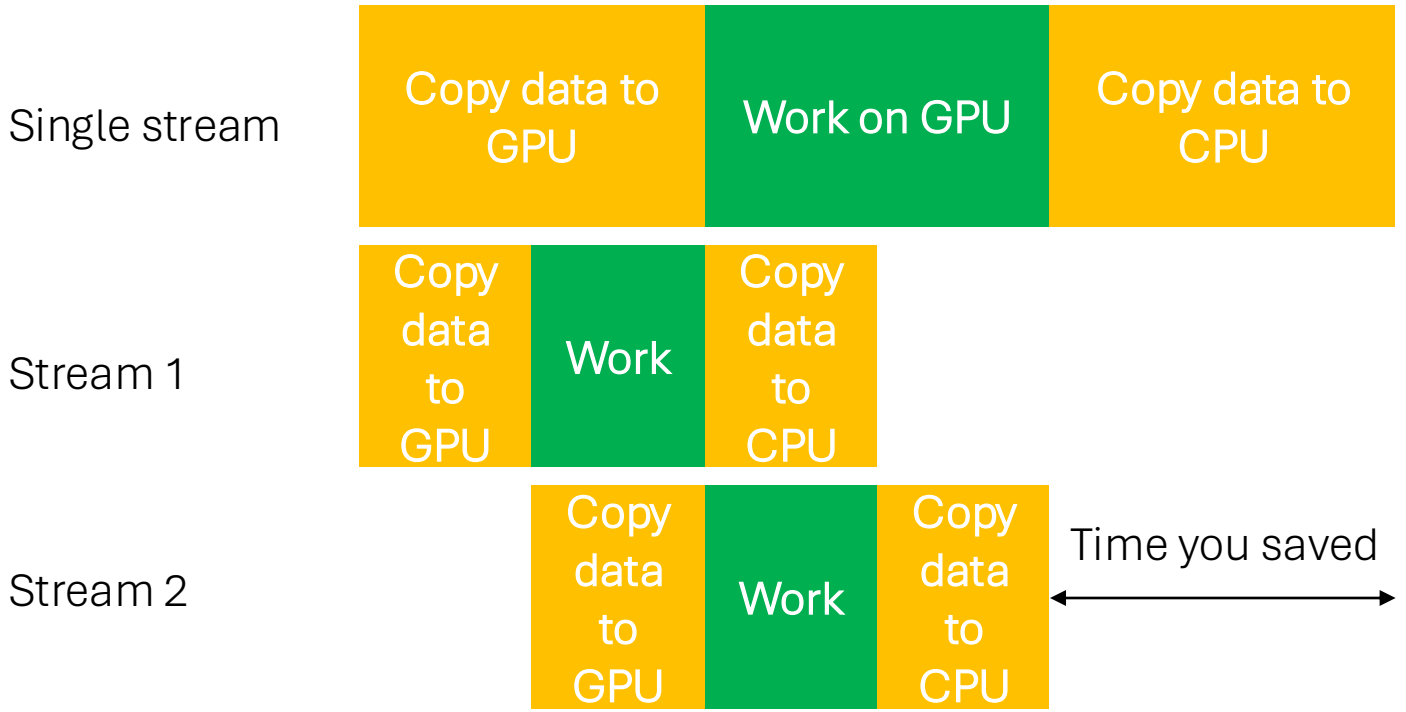




Shared memory is allocated per block (not from VRAM but from fast on-chip memory).
Each block gets its own private shared memory space.

Streams & asynchronous execution

- A **stream** is a queue of operations (kernels, memcopies) that execute in order **on the GPU**.
- **Multiple streams can run concurrently**, letting you **overlap** operations like computation and memory transfer.



Accelerating using GPUs - principles

```cpp
#include <iostream>
#include <cuda_runtime.h>
#define N 4  // rows
#define M 4  // columns
__global__ void scale_matrix(float* matrix, float scale, int rows, int cols) {
    int i = blockIdx.y * blockDim.y + threadIdx.y;  // row
    int j = blockIdx.x * blockDim.x + threadIdx.x;  // column


    if (i < rows && j < cols) {
        int idx = i * cols + j;
        matrix[idx] *= scale;
    }
}

int main() {
    float scale = 2.0f;
    float h_matrix[N * M];


    // Initialize host matrix
    for (int i = 0; i < N * M; ++i) {
        h_matrix[i] = static_cast<float>(i);
    }


    float* d_matrix;
    size_t size = N * M * sizeof(float);
    cudaMalloc(&d_matrix, size);
    cudaMemcpy(d_matrix, h_matrix, size, cudaMemcpyHostToDevice);


    dim3 blockSize(16, 16);
    dim3 gridSize((M + blockSize.x - 1) / blockSize.x,
                  (N + blockSize.y - 1) / blockSize.y);


    scale_matrix<<<gridSize, blockSize>>>(d_matrix, scale, N, M);
    cudaMemcpy(h_matrix, d_matrix, size, cudaMemcpyDeviceToHost);


    // Print result
    std::cout << "Scaled matrix:\n";
    for (int i = 0; i < N; ++i) {
        for (int j = 0; j < M; ++j) {
            std::cout << h_matrix[i * M + j] << "\t";
        }
        std::cout << "\n";
    }

    cudaFree(d_matrix);
    return 0;
}
```

Define kernel (GPU) function

1. Allocate array(s) and initialize on CPU

2. Allocate array(s) on GPU and copy data to GPU

3. Compute on GPU

4. Copy data from GPU back to CPU

5. Free up memory

Maximize this:
Exploit all threads, use shared memory

Minimize these:
Keep data if reused,
use pinned memory, streams

Overview of Python acceleration with GPUs

Cupy

GPU-based Numpy replacement

Numba

Making Python a complied language temporariry

Deep Learning

Accelerating DL traning in PyTorch and TensorFlow

CuPy: drop-in Numpy replacement

CuPy

- CuPy is a GPU-accelerated array library that provides a subset of the NumPy interface.
- CuPy offers: **`ndarray`**, sparse matrices, NumPy/SciPy routines.
- Most of NumPy syntaxes have their counterparts in CuPy.
- Routines are backed by CUDA libraries (cuBLAS, cuFFT, cuSPARSE, cuSOLVER, cuRAND), Thrust, CUB, and cuTENSOR to provide the best performance.

CuPy

CuPy – basics

- Installation:

`$ pip install cupy-cuda12x` (depending on CUDA version)

- Check GPU availability:

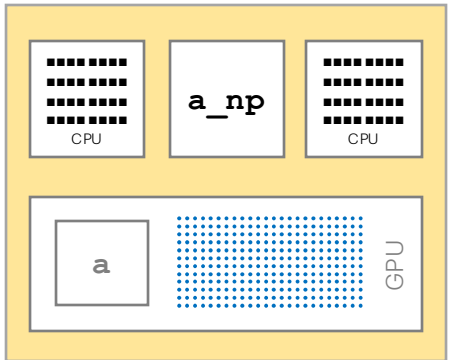
```
>>> import cupy as cp
>>> cp.cuda.runtime.getDeviceCount()
```

CuPy SLURM script

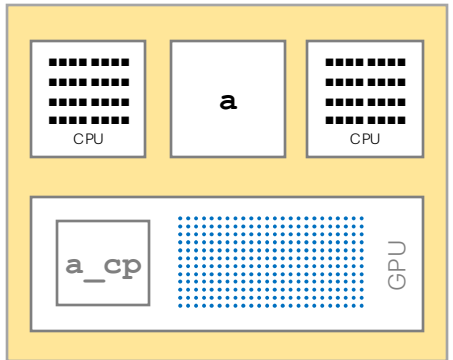
```
#!/bin/bash
#SBATCH --partition=a30                 # Partition
#SBATCH --mem=256000                    # Memory (in MB)
#SBATCH --job-name=job_name             # Job Name
#SBATCH -o LOG                          # Log file
#SBATCH --time=00:10:00                 # WallTime
#SBATCH --nodes=1                       # Number of Nodes
#SBATCH --gres=gpu:1                    # Number of GPUs

module purge
module load gnu12
module load cuda/12.6
module load python/3.11.11              # Make sure you load the correct version of Python, CuPy is not compatible with Python 3.12 or above (as of June 2025)
python3 -m venv your_env
source your_env/bin/activate
pip3 install cupy-cuda12x
python3 your_program.py
```

CuPy arrays vs. NumPy arrays

```
$ python3
>>> import cupy as cp
>>> a = cp.array([1, 2, 3])
>>> type(a)
<class 'cupy.ndarray'>

>>> a_np = cp.asnumpy(a)
>>> type(a_np)
<class 'numpy.ndarray'>
```




```
$ python3
>>> import numpy as np
>>> a = np.array([1, 2, 3])
>>> type(a)
<class 'numpy.ndarray'>

>>> a_cp = cp.asarray(a)
>>> type(a_cp)
<class 'cupy.ndarray'>
```

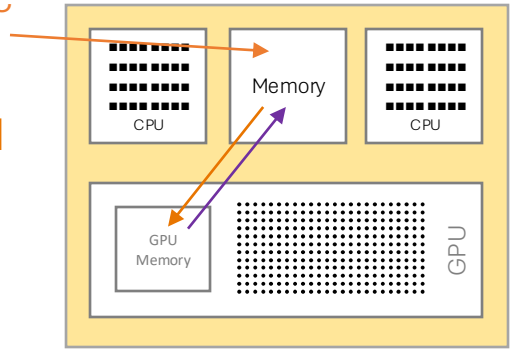


CuPy arrays vs. NumPy arrays

```
$ python3
>>> import cupy as cp
>>> a = cp.array([1, 2, 3])
>>> type(a)
<class 'cupy.ndarray'>

>>> a_np = cp.asnumpy(a)
>>> type(a_np)
<class 'numpy.ndarray'>
```

Allocate array(s) and initialize on CPU

Allocate array(s) on GPU and copy data to GPU



Copy data from GPU back to CPU

Ufuncs and elementwise operations

```
>>> import cupy as cp
>>> a = cp.array([1, 2, 3])
>>> b = cp.arange(9).reshape(3, 3)

>>> cp.sin(a)
array([0.84147098, 0.90929743,
0.14112001])
>>> cp.exp(b)
array([[1.00000000e+00,
2.71828183e+00, 7.38905610e+00],
       [2.00855369e+01,
5.45981500e+01, 1.48413159e+02],
       [4.03428793e+02,
1.09663316e+03, 2.98095799e+03]])
>>> cp.dot(a, b)
array([24, 30, 36])
```

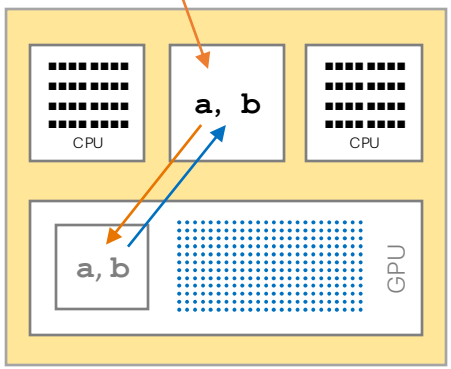
```
>>> import numpy as np
>>> a = np.array([1, 2, 3])
>>> b = np.arange(9).reshape(3, 3)

>>> np.sin(a)
array([0.84147098, 0.90929743,
0.14112001])
>>> np.exp(b)
array([[1.00000000e+00,
2.71828183e+00, 7.38905610e+00],
       [2.00855369e+01,
5.45981500e+01, 1.48413159e+02],
       [4.03428793e+02,
1.09663316e+03, 2.98095799e+03]])
>>> np.dot(a, b)
array([24, 30, 36])
```

Ufuncs and elementwise operations

```
>>> import cupy as cp
>>> a = cp.array([1, 2, 3])
>>> b = cp.arange(9).reshape(3, 3)

>>> cp.sin(a)
array([0.84147098, 0.90929743,
0.14112001])
>>> cp.exp(b)
array([[1.00000000e+00,
2.71828183e+00, 7.38905610e+00],
       [2.00855369e+01,
5.45981500e+01, 1.48413159e+02],
       [4.03428793e+02,
1.09663316e+03, 2.98095799e+03]])
>>> cp.dot(a, b)
array([24, 30, 36])
```

Define kernel (GPU) function *(a priori)*

1. Allocate array(s) and initialize on CPU
2. Allocate array(s) on GPU and copy data to GPU
3. Compute on GPU
4. Copy data from GPU back to CPU



Ufuncs benchmarking

```
# sin.py
import numpy as np
import cupy as cp
import time

# NumPy benchmark
x_np = np.random.rand(1_000_000)
start = time.time()
np.sin(x_np)
end = time.time()
print(f"NumPy sin: {end - start:.6f} seconds")

# CuPy benchmark
x_cp = cp.asarray(x_np)
cp.sin(x_cp)  # warm-up to reduce overhead
cp.cuda.Device(0).synchronize()

start = time.time()
cp.sin(x_cp)
cp.cuda.Device(0).synchronize()  # wait for syncing
end = time.time()
print(f"CuPy sin: {end - start:.6f} seconds")
```

```
NumPy sin: 0.009736 seconds
CuPy sin: 0.000070 seconds
```

CuPy is 139 times faster than NumPy!!!

At N = 10 million, CuPy is 357 times faster
At N = 100 million, CuPy is 439 times faster
At N = 1 billion, CuPy is 466 times faster

Ufuncs and elementwise operations

Ufunc/operations	NumPy	CuPy
sin, **cos**, **exp**, etc.	✓	✓
dot, **matmul**	✓	✓
fft, **ifft**	✓	✓
sqrt, **log**	✓	✓
nanmean, **nanstd**	✓	✓
polyfit, **histogram**	✓	✗
Object dtype ops	✓	✗
String functions	✓	✗
Advanced broadcasting	✓	✓

Check availability of a function in CuPy : **`print(hasattr(cp, 'your_func'))`**

CuPy vs NumPy: what works, what doesn’t

What works:

- Basic math, reductions, broadcasting
- Matrix ops: **`matmul`**, **`dot`**, **`linalg.inv`**, **`svd`**, etc.
- Random number generation
- Fourier Tranform: **`cp.fft`**
- Memory management: **`cp.cuda.memory`**

CuPy vs NumPy: what works, what doesn't

What doesn't:

- Keep in mind: GPUs love numbers and arrays of numbers.

- Scenario: you want to train a language model
  - Tokenization: "Should I use CuPy?" → "Should", "I", "use", "CuPy", "?" ✗
  - Vectorization: "Should", "I", "use", "CuPy", "?" → vectors ✗
  - Model training (neural networks) ✓

CuPy vs NumPy: what works, what doesn't

What doesn't:

- Keep in mind: GPUs love numbers and arrays of numbers.

```
#test.py
import cupy as cp
try:
        cp.array(["hello"])   # fails
except Exception as e:
        print(e)
$ python3 test.py
Unsupported dtype <U5
```

Data transfer

- Basic tranfers:
  - Host to device
  - Device to host

```
>>> import numpy as np
>>> import cupy as cp
>>> x_cpu = np.arange(5)
>>> x_gpu = cp.asarray(x_cpu)          #Transfer from host to device
>>> x = cp.asnumpy(x_gpu)              #Transfer from device to host
>>> x == x_cpu
array([ True,  True,  True,  True,  True])
>>> x == x_gpu
TypeError: Unsupported type <class 'numpy.ndarray'>
```

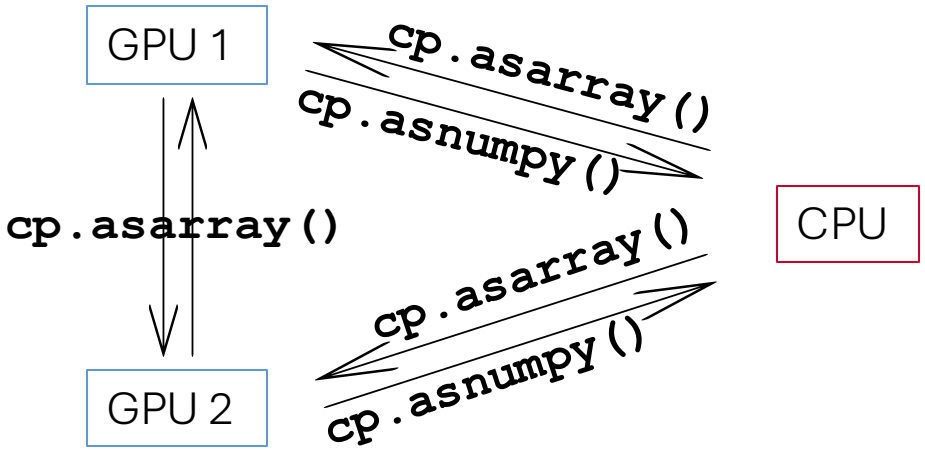
Alternatively,

```
>>> x_cpu = x_gpu.get()                #Transfer from device to host
```

Data transfer

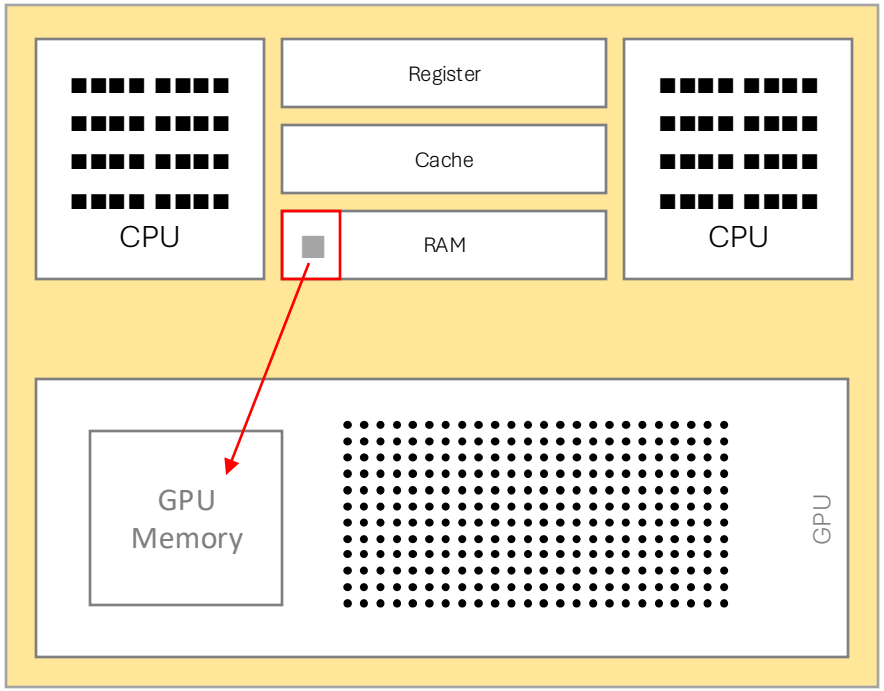
- From device to device

```
with cp.cuda.Device(0):
    x_gpu_0 = cp.ndarray([1, 2, 3])        # create an array in GPU 0
with cp.cuda.Device(1):
    x_gpu_1 = cp.asarray(x_gpu_0)          # move the array to GPU 1
```



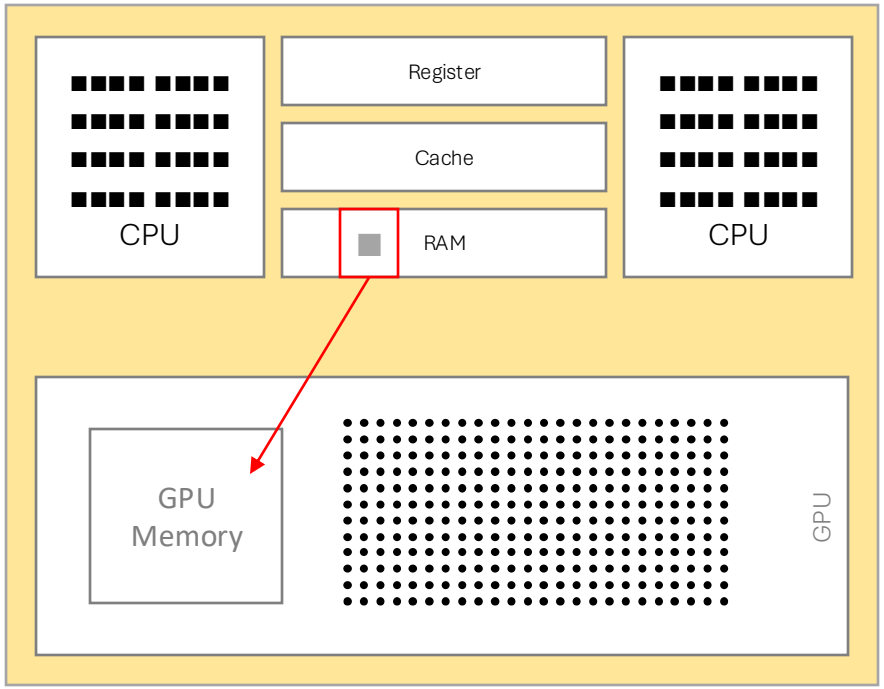
Pinned memory

```
import cupy as cp
import numpy as np
from cupy.cuda import alloc_pinned_memory,
PinnedMemoryPointer, MemoryPointer

# ---------------------------
# 1. Allocate pinned memory
# ---------------------------
n_bytes = 1024  # size in bytes
pinned = alloc_pinned_memory(n_bytes)

# Wrap it with PinnedMemoryPointer to make it
accessible like an array
pinned_arr = cp.ndarray((256,), dtype=np.uint8,
memptr=PinnedMemoryPointer(pinned))

# Fill the array from CPU side
pinned_arr[:] = np.arange(256, dtype=np.uint8)

# Copy to GPU for processing
gpu_arr = cp.asarray(pinned_arr)
print("Pinned to GPU array:", gpu_arr[:10])
```



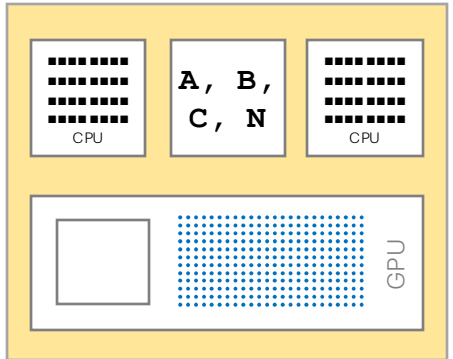
Mapped memory

```
# --------------------------
# 2. Allocate mapped memory
# --------------------------
# Allocate mapped memory using CuPy's backend
mapped_mem = cp.cuda.mapped_memory()  # returns
MappedMemory object
mapped_ptr = MemoryPointer(mapped_mem, 0)

# Create an array-like object on mapped memory
mapped_arr = cp.ndarray((256,), dtype=np.uint8,
memptr=mapped_ptr)

# Fill the array from CPU side
mapped_arr[:] = np.arange(255, -1, -1,
dtype=np.uint8)

# GPU can access it directly
mapped_gpu = cp.array(mapped_arr)
squared = mapped_gpu ** 2
print("Mapped memory GPU computation:",
squared[:10])
```

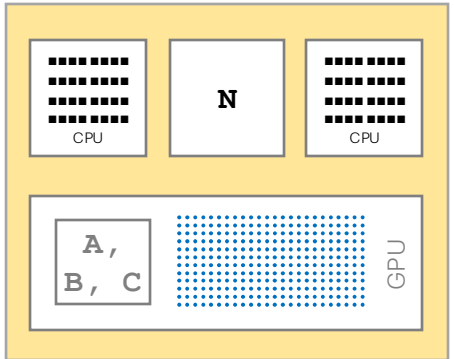


Example – matrix multiplication

```python
# matrix_multiplication_numpy.py
import numpy as np
import time
import sys

if len(sys.argv) < 2:       # Parse N from command line
    print("Usage: python matrix_multiplication_numpy.py <N>")
    sys.exit(1)
N = int(sys.argv[1])

A = np.random.rand(N, N).astype(np.float32)# Create matrix A
B = np.random.rand(N, N).astype(np.float32)# Create matrix B



start = time.time()
C = A @ B

end = time.time()
elapsed = end - start
print(f"N={N}, NumPy matrix multiplication took {elapsed:.4f}
seconds")
```

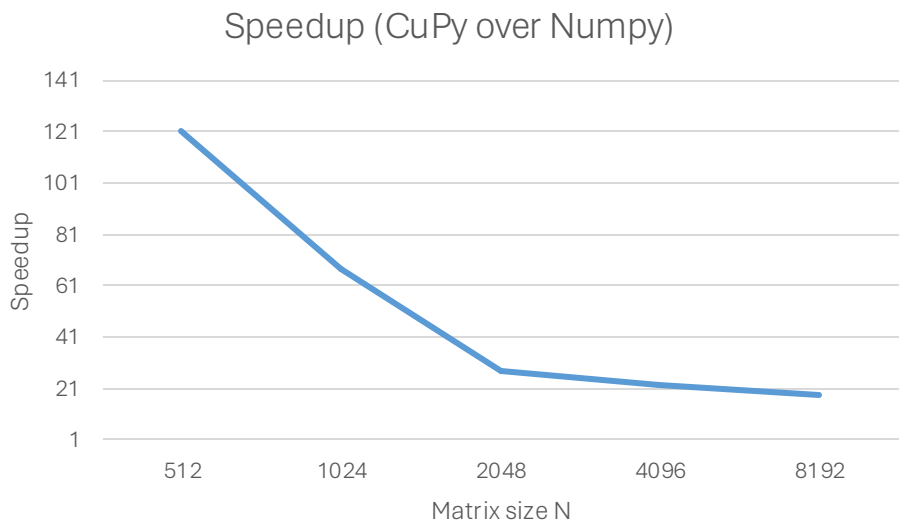



```python
# matrix_multiplication_cupy.py
import cupy as cp
import time
import sys

if len(sys.argv) < 2:      # Parse N from command line
    print("Usage: python matrix_multiplication_cupy.py <N>")
    sys.exit(1)
N = int(sys.argv[1])

A = cp.random.rand(N, N, dtype=cp.float32)# Create matrix A
B = cp.random.rand(N, N, dtype=cp.float32)# Create matrix A

_ = A @ B                                  # Warm-up
cp.cuda.Device().synchronize()

start = time.time()
C = A @ B
cp.cuda.Device().synchronize() # Ensure GPU computation is done
end = time.time()
elapsed = end - start
print(f"N={N}, CuPy matrix multiplication took {elapsed:.4f}
seconds")
```



Example – matrix multiplication – benchmark

N	Numpy(s)	CuPy(s)	Speedup
512	0.04856	0.0004	121.4
1024	0.0484	0.00072	67.2
2048	0.09082	0.00326	27.9
4096	0.47834	0.02154	22.2
8192	2.446	0.13206	18.5



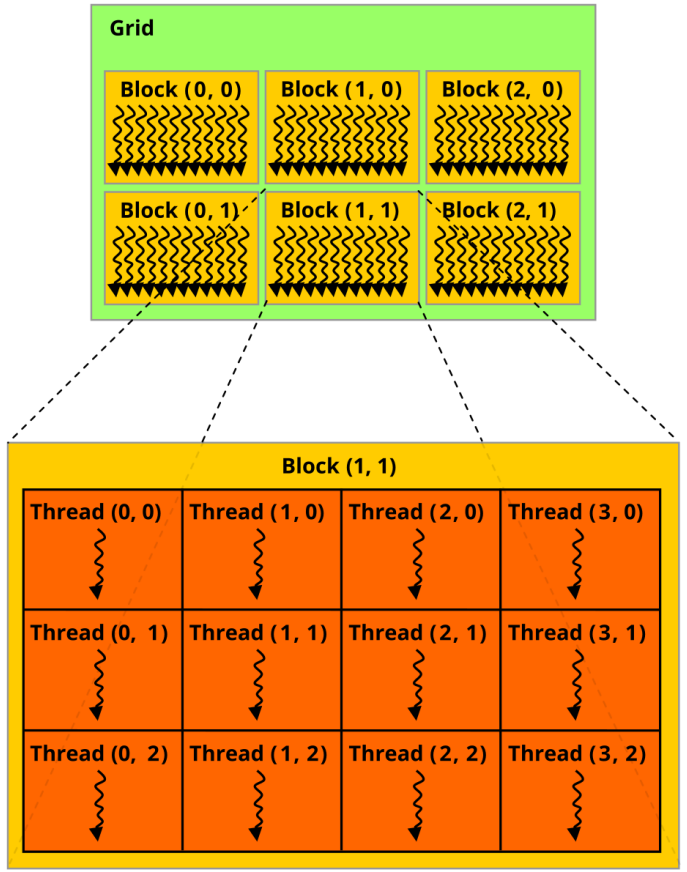
Even though CuPy still faster than NumPy, the relative speedup drops:

- Small N (N = 512, ~268 million operations) → massive speedup due to **low CPU efficiency**.
- For large N (like 8192×8192, ~1.1 trillion operations), the multiplication becomes **memory-bound** rather than compute-bound.
- GPU cores may become fully **saturated**, so adding more work does not linearly improve throughput.
- NumPy also begins to scale better at large sizes due to **cache locality** and **parallel basic linear algebra subprograms** (e.g., OpenBLAS or MKL).

Advanced: user-defined kernels

- Total lower-level control
- Expose CUDA programming to Python
- Faster for element-wise or custom ops

```
mod = cp.RawModule(code=r'''
extern "C" __global__
void scale(float* x, int N) {
    int i = blockDim.x * blockIdx.x +
threadIdx.x;
    if (i < N) x[i] *= 2.0;
}
''')
kernel = mod.get_function("scale")
x_gpu = cp.arange(10, dtype=cp.float32)
kernel((1,), (10,), (x_gpu, x_gpu.size))
```



Advanced: user-defined kernels

- Always check statistics about memory allocation: **`cp.get_default_memory_pool().used_bytes()`**
- Wrap kernel in **`try`**/**`except`** with **`sync`** for debugging

- When to use?
  - Complex indexing
  - Better performance needs
  - Atomic operations
  - Memory coalescing

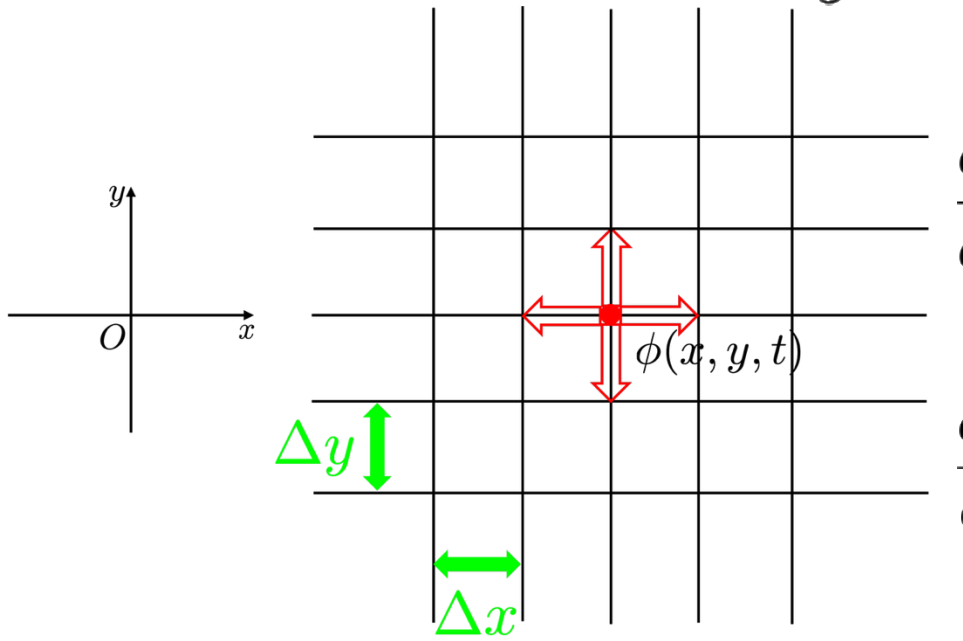
Task type	CuPy array ops	User-defined kernel
Basic math, stencil grids	Yes	No
Loop-heavy logic	Sometimes	Yes
Memory reuse, shared mem	No	Yes
Debuggability	Easier	Harder

Questions on CuPy?

Hands-on tutorials

- 2D heat diffusion by Jacobi iterations:

$$\frac{\partial^2 \phi}{\partial x^2} + \frac{\partial^2 \phi}{\partial y^2} + 1 = 0$$



$$\frac{\partial^2 \phi}{\partial x^2} \approx \frac{\phi_{i+1,j} + \phi_{i-1,j} - 2\phi_{i,j}}{\Delta x^2}$$

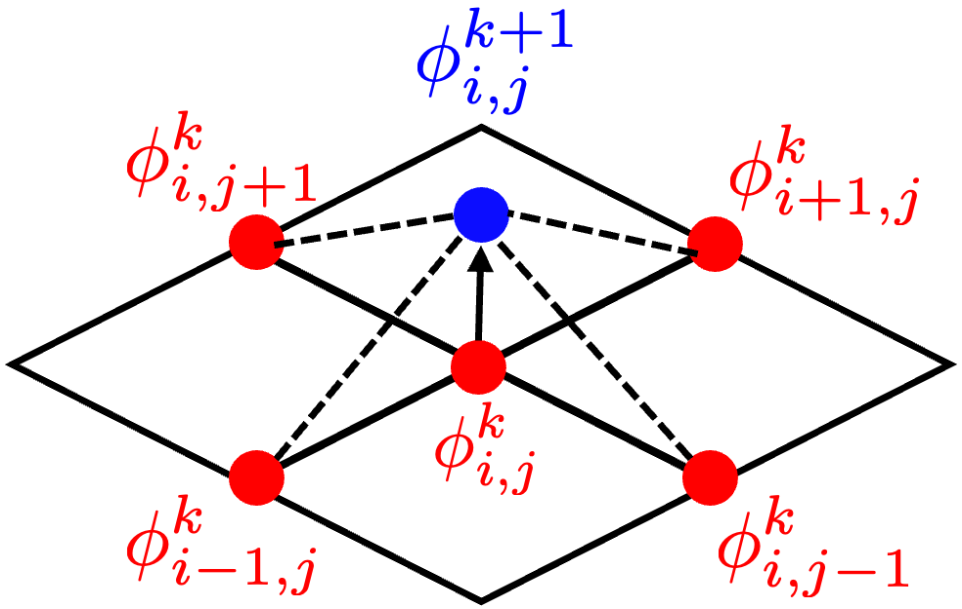
$$\frac{\partial^2 \phi}{\partial y^2} \approx \frac{\phi_{i,j+1} + \phi_{i,j-1} - 2\phi_{i,j}}{\Delta y^2}$$

Hands-on tutorials

- 2D heat diffusion by Jacobi iterations:

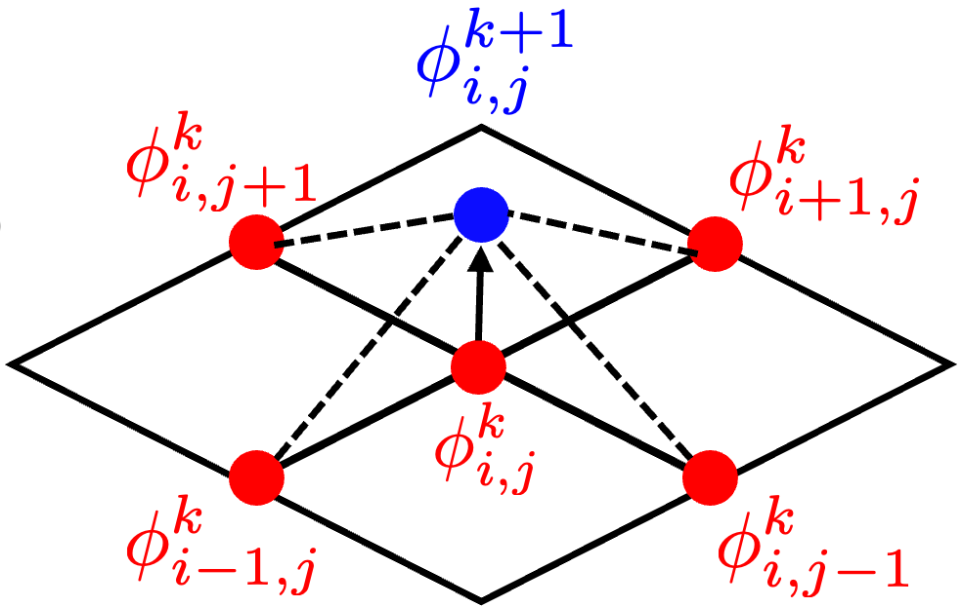
$$\frac{\phi_{i+1,j}+\phi_{i-1,j}-2\phi_{i,j}}{\Delta x^2}+\frac{\phi_{i,j+1}+\phi_{i,j-1}-2\phi_{i,j}}{\Delta y^2}+1=0$$

$$\phi_{i,j}^{k+1}=\frac{1}{4}\left(\phi_{i+1,j}^{k}+\phi_{i-1,j}^{k}+\phi_{i,j+1}^{k}+\phi_{i,j+1}^{k}+\Delta x^2\right)$$



Hands-on tutorials

```
import numpy as np
import matplotlib.pyplot as plt
# Parameters
N = 256
tolerance = 1e-4
max_iters = 100000
dx = 1.0 / N
# Initialize grid
p = np.zeros((N, N))
f = np.zeros((N, N))
f[N//2, N//2] = 10000  # Heat source
# Boundary conditions: p = 0 on the edges (already zero)
# Jacobi iteration
for iteration in range(max_iters):
    p_new = p.copy()
    p_new[1:-1, 1:-1] = 0.25 * (
        p[:-2, 1:-1] + p[2:, 1:-1] +
        p[1:-1, :-2] + p[1:-1, 2:] +
        dx**2 * f[1:-1, 1:-1]
    )
    diff = np.linalg.norm(p_new - p)
    p = p_new
    if diff < tolerance:
        print(f"Converged after {iteration} iterations")
        break
print(p[N//2, N//2])

#plt.imshow(p, cmap='hot')
#plt.colorbar()
#plt.title("NumPy Heat Diffusion")
#plt.show()
```



Hands-on tutorials

- Setting things up on Juno:
  - Create a Python environment:

```
python3 -m venv your_venv
source your_venv/bin/activate
pip3 install cupy-cuda12x
```

```
$ git clone https://gitlab.circ.utdallas.edu/hpc-utd/workshop/accelerating-python-with-gpus.git
```

- We provided a numpy Jacobi solver under **tutorials/Heat_diffusion_Numpy.py**
- Alternatively, you can copy that directory to your home directory: **cp -r /scratch/juno/hpcre/workshop/python-gpu/ .**
- Allocate a GPU node: **salloc -p a30-4.6gb --mem=2GB --time=00:30:00 --reservation=hpcre-aug8**
- Login to the GPU node: **ssh g-04-01**
- Exercise: Rewrite the code using CuPy.

```
module purge
module load gnu12 python/3.11.11 cuda/12.6
```

Numba: Just-in-time (JIT) compiler

Numba

- JIT for CPU & GPU
- Basics
- Defining kernels and launching them
- Memory management

What is JIT compilation?

- JIT = Just-In-Time Compilation
  - It compiles Python functions **on-the-fly** to make them **run faster**.
  - Python is an interpreted language, not compiled.

Python (normal)	Python + JIT
Interprets each line every time	Translates to fast machine code once, reuses it
Easier to use, but slow	Needs setup, but runs faster

How JIT works

- First call → Compile Python to fast native code (takes a moment)
- Subsequent calls → Run compiled code (much faster)

[Python code] —(JIT Compile on 1st run)—► [Machine code]

▲

Fast reuse from here

- Similar to Fortran and C/C++: you compile, and you run the executables.

Why JIT?

- Greatly faster for numeric loops
- Easy to use, just add **`@jit`** decorator.
- Minimal changes on the existing Python code.

- Installation: **`$ pip3 install numba`**

JIT example

```python
#jit_benchmark.py
import math
import numpy as np
import time
from numba import jit

# 1. Pure Python loop (slow)
def slow_sin(arr):
    result = np.empty_like(arr)
    for i in range(len(arr)):
        result[i] = math.sin(arr[i])
    return result

# 2. Numba JIT-compiled version
@jit(nopython=True)
def fast_sin(arr):
    result = np.empty_like(arr)
    for i in range(len(arr)):
        result[i] = math.sin(arr[i])
    return result


# 3. NumPy vectorized version
def numpy_sin(arr):
    return np.sin(arr)
```

N = 100.000.000

```
>>> python3 jit_benchmark.py
Pure Python loop: 17.0030 seconds
Numba JIT version: 1.0890 seconds
NumPy vectorized: 2.0746 seconds
All results equal: True
```

Method	Pattern	Speed
Python loop	`for i in range(len(arr))`	Slow
Numba JIT	`@jit(nopython=True)`	Fast
NumPy	`numpy.sin(arr)`	Fast

JIT + GPU acceleration

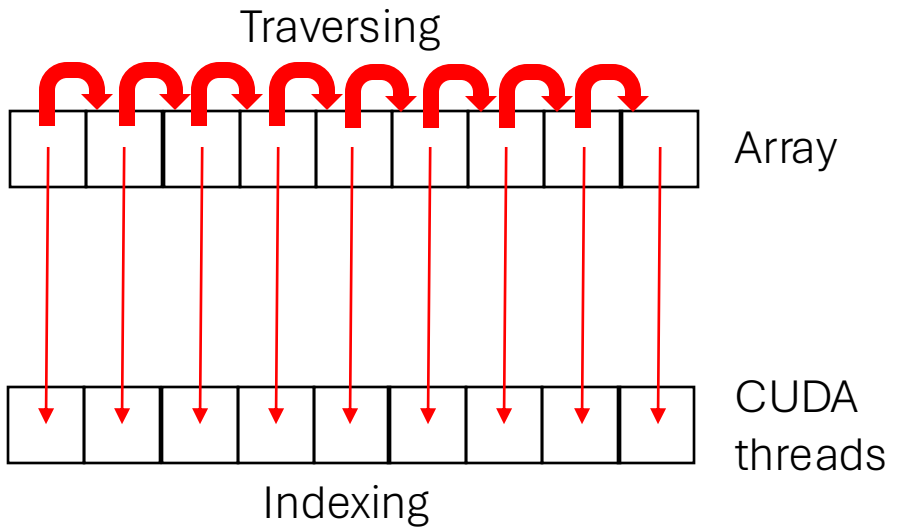
- Numba supports CUDA GPU programming by directly compiling a restricted subset of Python code into CUDA kernels.
- Kernels written in Numba appear to have direct access to NumPy arrays.

```
#!/bin/bash
#SBATCH --partition=a30                          # Partition
#SBATCH --mem=256000                             # Memory (in MB)
#SBATCH --job-name=job_name                      # Job Name
#SBATCH -o LOG                                   # Log file
#SBATCH --time=00:10:00                          # WallTime
#SBATCH --nodes=1                                # Number of Nodes
#SBATCH --gres=gpu:1                             # Number of GPUs
#SBATCH --ntasks-per-node=1                      # Number of tasks (MPI processes)
#SBATCH --cpus-per-task=4                        # Number of processors per task OpenMP threads()
#SBATCH --mail-type=ALL                          # Add all jobs to the the mailing list
#SBATCH --mail-user=<your-netID>@utdallas.edu      # Send notification to the address when job begins and ends

module purge
module load gnu12
module load python/3.11.11
module load cuda/12.6
source your_venv/bin/activate
pip3 install numba
for N in 512 1024 2048 4096 8192; do
    echo "Running GPU Numba with shared memory matrix multiplication with N=$N"
    python3 matrix_multiplication_shared_numba.py $N
done
```

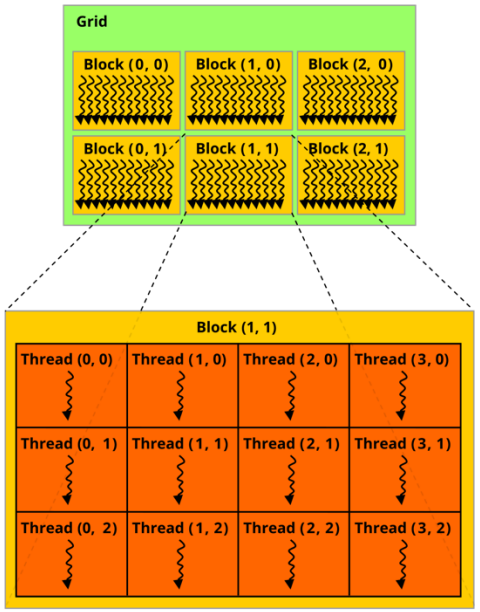
Writing CUDA kernels

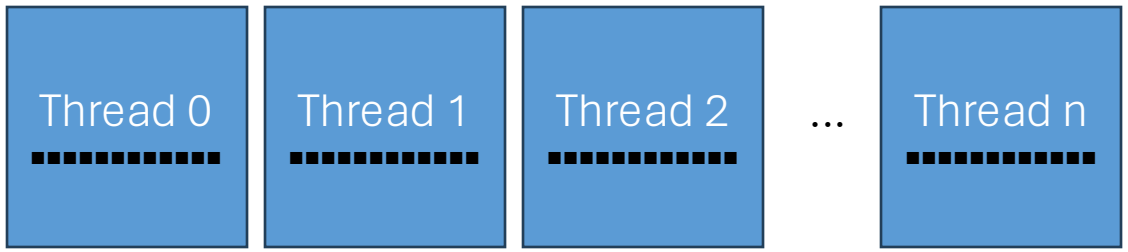
- A *kernel function* is a GPU function that is meant to be called from CPU code:
  - kernels cannot explicitly return a value → all data must be written in an array.
  - decorator in Numba: **`@cuda.jit`**
  - kernels explicitly declare their thread hierarchy when called: i.e. the number of thread blocks and the number of threads per block

```
@cuda.jit
def add_kernel(a, b, out):
    i = cuda.grid(1)
    if i < a.size:
        out[i] = a[i] + b[i]

add_kernel[blocks, threads](a, b, out)
```

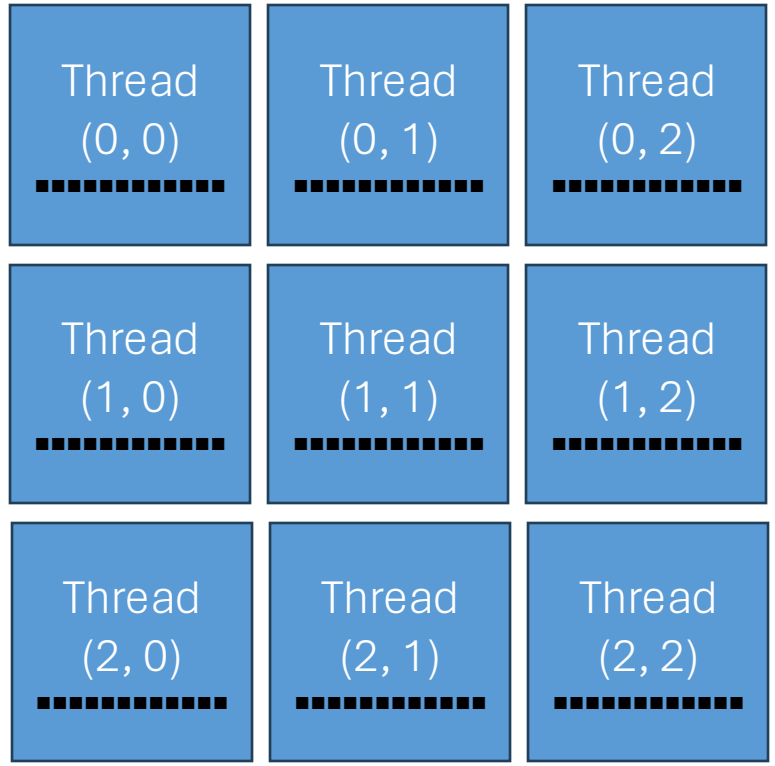


Thread hierarchy





`cuda.grid(1)`

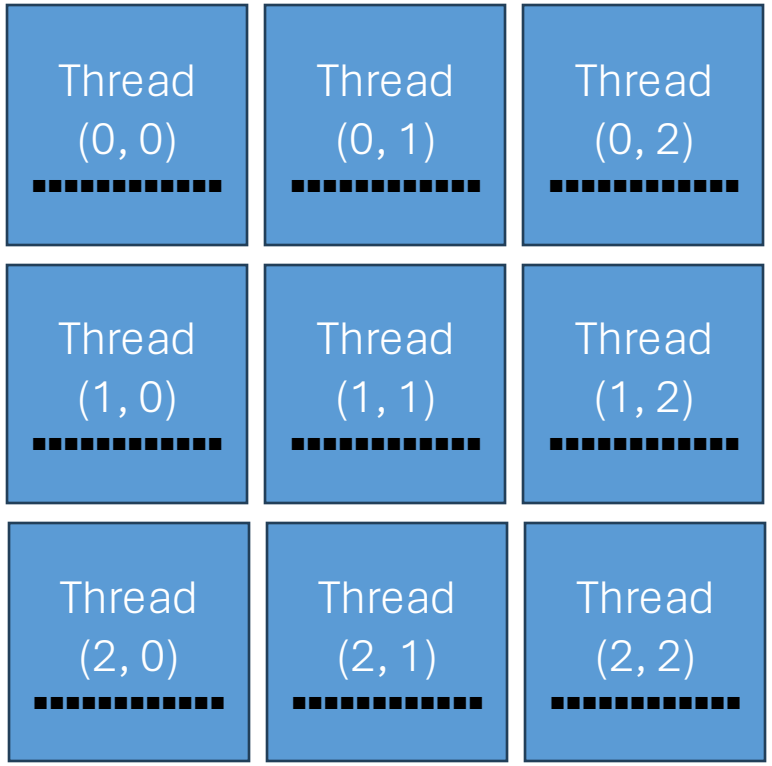


`cuda.grid(2)`

Thread hierarchy

```
@cuda.jit
def scale_matrix(matrix):
    x, y = cuda.grid(2)
    if x < matrix.shape[0] and y <
matrix.shape[1]:
        matrix[x, y] *= 2
```

```
#launching
threadsperblock = (16, 16)
blockspergrid_x = math.ceil(matrix.shape[0] /
threadsperblock[0])
blockspergrid_y = math.ceil(matrix.shape[1] /
threadsperblock[1])
blockspergrid = (blockspergrid_x, blockspergrid_y)
scale_matrix[blockspergrid, threadsperblock](an_array)
```



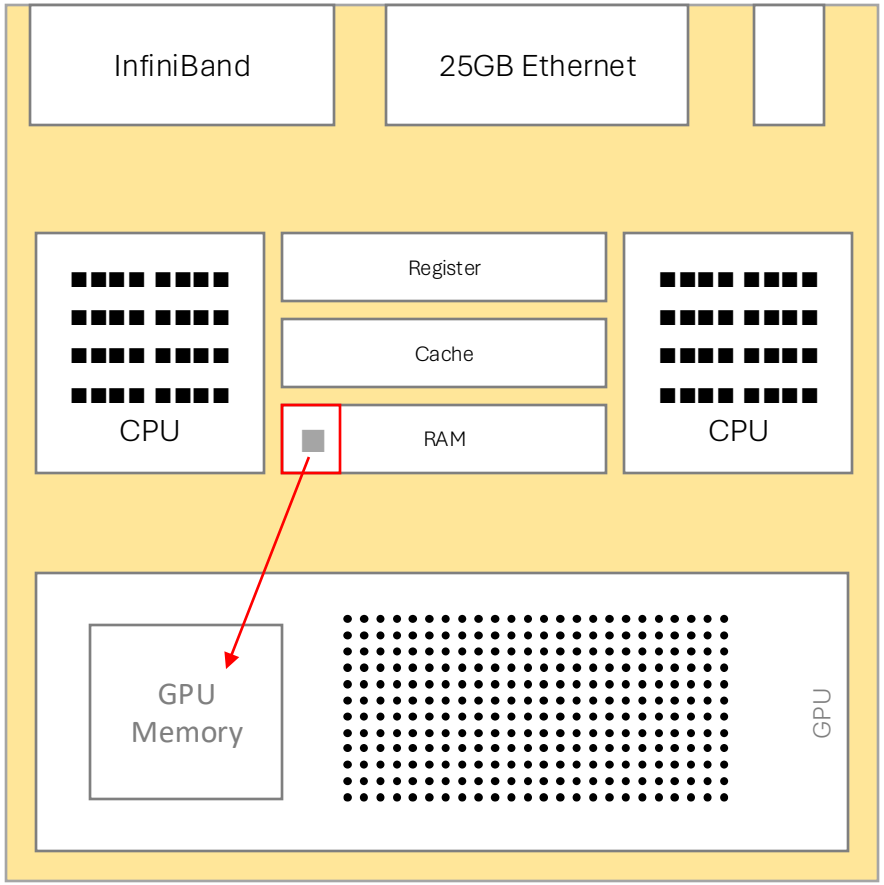
Data transfer

- Numba automatically transfers NumPy array to device and copies back when a kernel finishes, even when not strictly needed.

- Host - Device Transfers:
  - To GPU: **`cuda.to_device(np_array)`**
  - From GPU: **`device_array.copy_to_host()`**
  - From GPU to a known array on host: **`device_array.copy_to_host(host_array)`**
  - Allocate on GPU: **`cuda.device_array(shape, dtype)`**
- Best practices:
  - Minimize transfers
  - Keep intermediate data on device

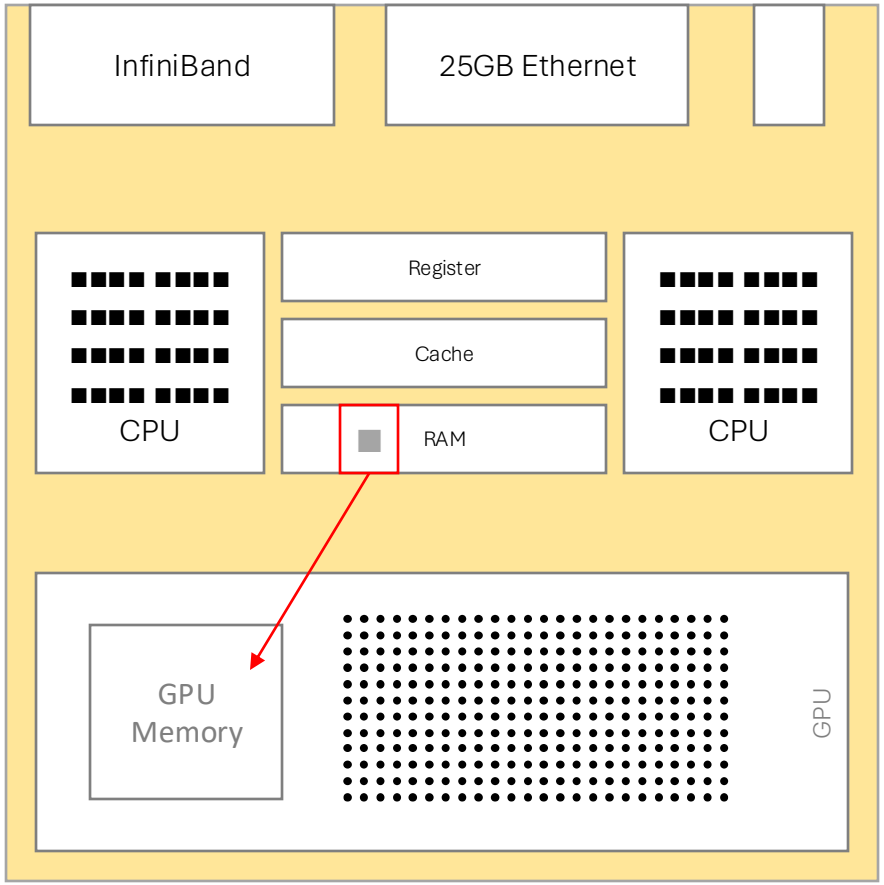
Pinned memory

```
from numba import cuda
import numpy as np


# Allocate pinned (page-locked) host memory
pinned_array = cuda.pinned_array(10, dtype=np.float32)


# Fill and transfer to device
pinned_array[:] = np.arange(10)
device_array = cuda.to_device(pinned_array)


print(device_array.copy_to_host())
```



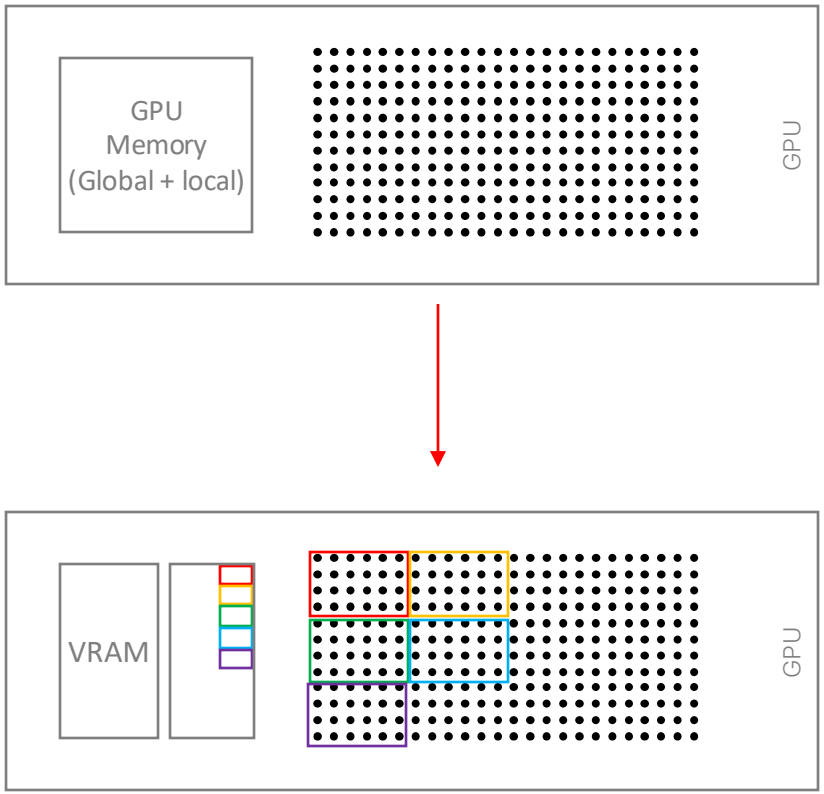
Mapped memory

```
from numba import cuda
import numpy as np

# Allocate mapped memory (accessible by GPU and CPU)
mapped_array = cuda.mapped_array(10,
dtype=np.float32)

# Fill from host
mapped_array[:] = np.arange(10)

@cuda.jit
def kernel(arr):
    i = cuda.grid(1)
    if i < arr.size:
        arr[i] += 10

kernel[1, 10](mapped_array)
print(mapped_array)
```



Shared memory

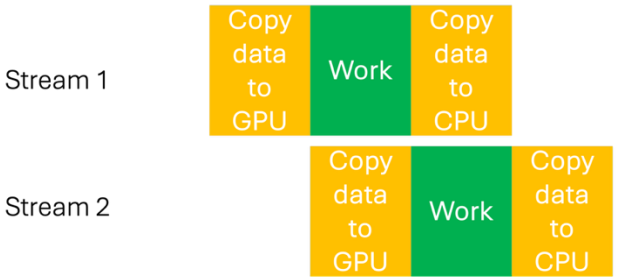
```
from numba import cuda
import numpy as np

@cuda.jit
def add_shared(x, y):
    shared = cuda.shared.array(256, dtype=cuda.float32)
    i = cuda.threadIdx.x
    if i < x.size:
        shared[i] = x[i] + y[i]
        cuda.syncthreads()
        x[i] = shared[i]

x = np.ones(256, dtype=np.float32)
y = np.ones(256, dtype=np.float32)
dx = cuda.to_device(x)
dy = cuda.to_device(y)
add_shared[1, 256](dx, dy)
print(dx.copy_to_host())
```



Streams & asynchronous execution

```
import numpy as np
from numba import cuda
import time

# Simple GPU kernel
@cuda.jit
def add_kernel(a, b, out):
    i = cuda.grid(1)
    if i < a.size:
        out[i] = a[i] + b[i]

N = 10**6
a_host = np.random.rand(N).astype(np.float32)
b_host = np.random.rand(N).astype(np.float32)

# Allocate output on host
out_host = np.empty_like(a_host)

# Create a custom stream
stream = cuda.stream()
```

```
# Allocate device memory and copy asynchronously using the stream
a_dev = cuda.to_device(a_host, stream=stream)
b_dev = cuda.to_device(b_host, stream=stream)
out_dev = cuda.device_array_like(a_dev, stream=stream)

# Launch kernel in the same stream
threads_per_block = 256
blocks_per_grid = (N + threads_per_block - 1) // threads_per_block
add_kernel[blocks_per_grid, threads_per_block, stream](a_dev, b_dev, out_dev)

# Copy result back to host (asynchronously)
out_dev.copy_to_host(out_host, stream=stream)

# Wait for all operations in the stream to finish
stream.synchronize()
```



Streams & asynchronous execution

Feature	Explanation
`cuda.stream()`	Creates a new CUDA stream
`to_device(..., stream=...)`	Copies data asynchronously
`kernel[grid, block, stream]`	Launches kernel into the stream
`copy_to_host(..., stream=...)`	Async result transfer to CPU
`stream.synchronize()`	Waits for all tasks in stream to finish

- Streams allow overlapping host-to-device transfers, kernel execution, and device-to-host transfers when possible.
- Use multiple streams for batch operations or pipelining tasks.
- All operations in the same stream are serialized; cross-stream operations may overlap.

Fastmath in CUDA

- Enable with: **`@cuda.jit(fastmath=True)`**
- Uses faster, less precise math (e.g., **`expf`**, **`sinf`**)
- Faster execution, small accuracy loss
- **Use Case:** Deep learning, image processing

Atomic operations

An atomic operation is a sequence of actions treated as a single, indivisible unit of work

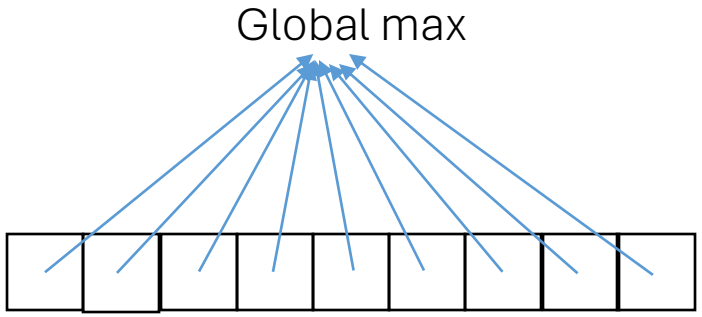
Why Use Atomic Operations?

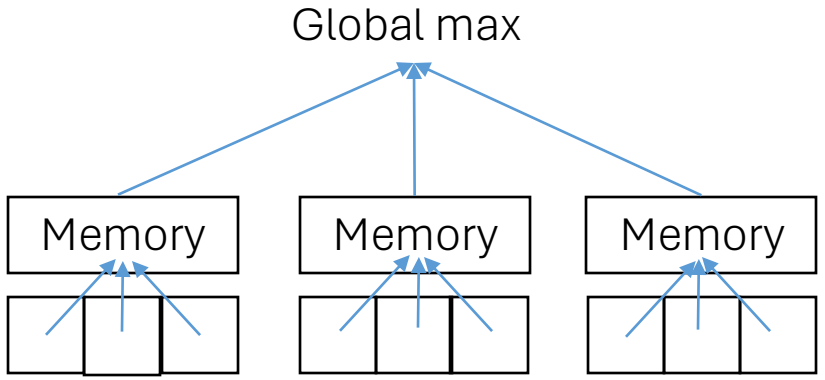
- **Race-condition-free** updates from multiple threads.
- Essential for **parallel reductions**, **histogramming**, **counters**, etc.
- Simpler than manual locking or shared-memory reductions.

```
counter = 0
# Dangerous! Not atomic.
counter = counter + 1
```

Atomic operations

- **Naive Global Atomic:** Slow due to high contention from many threads trying to update the same global variable.
- **Shared Memory + Global Atomic:** Reduces contention by using shared memory within blocks first.
- **NumPy on CPU:** Much slower due to lack of parallel execution.





Shared memory:
Works as a binary tree, reducing execution time

Atomic operations

- **Naive Global Atomic:** Slow due to high contention from many threads trying to update the same global variable.
- **Shared Memory + Global Atomic:** Reduces contention by using shared memory within blocks first.
- **NumPy on CPU:** Much slower due to lack of parallel execution.

Method	Time (ms)	Notes
Naive Global Atomic	~35–50	Many threads contend for same address
Shared Memory + Atomic	~8–15	Efficient: local max per block first
NumPy (CPU)	~60–100	Slowest due to lack of parallelism

Atomic operations

- Example: **Finding the maximum value in an array** using parallel threads:

```
import numpy as np
from numba import cuda

@cuda.jit
def max_kernel(result, values):
    tid = cuda.grid(1)
    if tid < values.size:
        cuda.atomic.max(result, 0, values[tid])
# Host code:
arr = np.random.rand(16384).astype(np.float32)
res = np.zeros(1, dtype=np.float32)
d_arr = cuda.to_device(arr)
d_res = cuda.to_device(res)

max_kernel[256, 64](d_res, d_arr)
cuda.synchronize()
print("GPU max:", d_res.copy_to_host()[0], " vs NumPy
max:", arr.max())
```

```
# In a 3D reduction:
cuda.atomic.max(result, (i, j, k),
values[i, j, k])
```

Atomic operations

Intrinsic	Description	Supported Types
`add(a, idx, val)`	Atomic `a[idx] += val`	int32, int64, float32, float64
`sub(a, idx, val)`	Atomic `a[idx] -= val`	int32, int64, float32, float64
`max(a, idx, val)`	Atomic `a[idx] = max(a[idx], val)`	All above types
`min(a, idx, val)`	Atomic `a[idx] = min(a[idx], val)`	All above types
`nanmax(a, idx, val)`	Like `max`, respects NaN semantics	Same types
`nanmin(a, idx, val)`	Like `min`, respects NaN semantics	Same types
`and_(a, idx, val)`	Atomic `a[idx] &= val`	int32/64, uint32/64
`or_(a, idx, val)`	Atomic `` `a[idx] ``	Same as `and_`
`xor(a, idx, val)`	Atomic `a[idx] ^= val`	Same as `and_`
`exch(a, idx, val)`	Atomic `a[idx] = val`	Same as `and_`
`inc(a, idx, val)`	Atomic increment: wrap at `val`	uint32/64
`dec(a, idx, val)`	Atomic decrement or reset logic	uint32/64
`cas(a, idx, old, val)` & `compare_and_swap(a, old, val)`	Atomic compare-and-swap	int32/64, uint32/64

Atomic operations

Best Practices

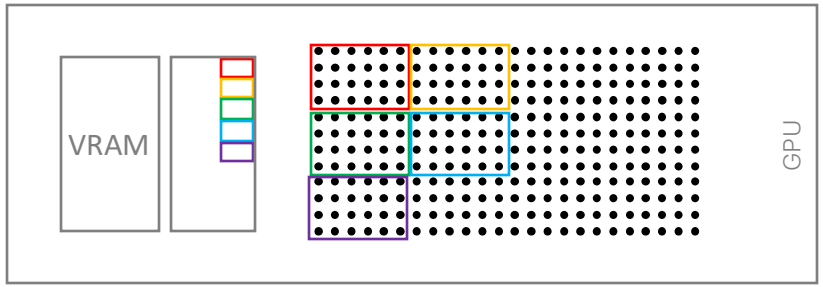
- Use atomic ops **only when needed** — they are slower than per-thread accumulations.

For reductions:

- Prefer block-level reduction with shared memory → then atomic write once per block.
- Use global atomics sparingly.
- Ensure **data and types match** — e.g., **`atomic.inc()`** is only for unsigned types.

Shared memory atomic operations

```
from numba import cuda, float32
import numpy as np
import time

# Kernel 1: Naive global atomic max
@cuda.jit
def global_atomic_max_kernel(arr, result):
    idx = cuda.grid(1)
    if idx < arr.size:
        cuda.atomic.max(result, 0, arr[idx])
```




```
Array size: 1048576
Grid size: 4096, Block size: 256


Global atomic max: 999.999
Time: 0.612753 sec
Shared reduction max: 999.999
Time: 0.091727 sec
```

```
# Kernel 2: Shared memory + final global atomic
@cuda.jit
def shared_reduction_max_kernel(arr, result):
    smem = cuda.shared.array(256, float32)
    tid = cuda.threadIdx.x
    i = cuda.grid(1)

    if i < arr.size:
        smem[tid] = arr[i]
    else:
        smem[tid] = -1e20 #sentinel value

    cuda.syncthreads()

    stride = cuda.blockDim.x // 2
    while stride > 0:
        if tid < stride:
            smem[tid] = max(smem[tid], smem[tid + stride])
        cuda.syncthreads()
        stride //= 2

    if tid == 0:
        cuda.atomic.max(result, 0, smem[0])
```

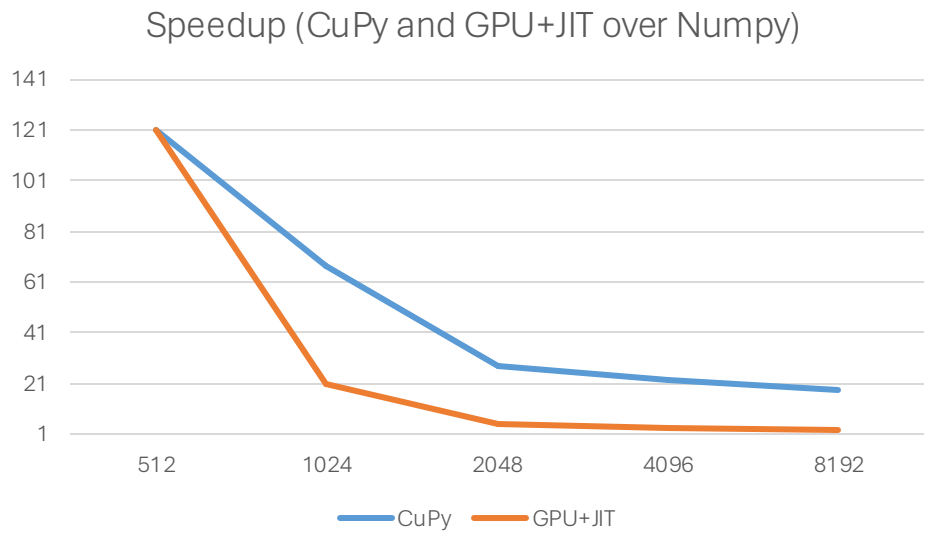
Example – matrix multiplication – benchmark

N	Numpy (s)	JIT (s)	JIT+GPU (s)
512	0.04856	0.3325	0.0004
1024	0.0484	0.4962	0.0023
2048	0.09082	5.0768	0.0176
4096	0.47834	62	0.1394
8192	2.446	484	1.1117

```
# GPU kernel for matrix multiplication
@cuda.jit
def matmul_gpu(A, B, C):
    i, j = cuda.grid(2)
    N = A.shape[0]


    if i < N and j < N:
        tmp = 0.0
        for k in range(N):
            tmp += A[i, k] * B[k, j]
        C[i, j] = tmp
```

Accessing global memory (A[i, k], B[k, j]) in every iteration is **slow**

Example – matrix multiplication



N	Numpy (s)	JIT (s)	JIT+GPU (s)	CuPy (s)
512	0.04856	0.3325	0.0004	0.0004
1024	0.0484	0.4962	0.0023	0.00072
2048	0.09082	5.0768	0.0176	0.00326
4096	0.47834	62	0.1394	0.02154
8192	2.446	484	1.1117	0.13206

Same speedup trends as CuPy -> for large N, NumPy is able to catch up, it is critical to custom GPU kernels for better performance!

Example – matrix multiplication – benchmark

Benefits of Shared Memory:

- Each global memory read happens **once per tile**, not per thread.
- Computation is done in **fast shared memory**.
- Better **coalesced memory access** patterns.

N	JIT (s)	JIT+GPU (s)	CuPy (s)	Shared memory JIT+GPU (s)
512	0.3325	0.0004	0.0004	0.0001
1024	0.4962	0.0023	0.00072	0.0005
2048	5.0768	0.0176	0.00326	0.0031
4096	62	0.1394	0.02154	0.0244
8192	484	1.1117	0.13206	0.1951

```
TPB = 16  # Threads per block (tile width)
@cuda.jit
def matmul_shared(A, B, C):
    sA = cuda.shared.array((TPB, TPB), dtype=float32)
    sB = cuda.shared.array((TPB, TPB), dtype=float32)

    tx = cuda.threadIdx.x
    ty = cuda.threadIdx.y
    bx = cuda.blockIdx.x
    by = cuda.blockIdx.y

    x = bx * TPB + tx
    y = by * TPB + ty

    tmp = 0.0
    for m in range((A.shape[1] + TPB - 1) // TPB):
        if y < A.shape[0] and m * TPB + tx < A.shape[1]:
            sA[ty, tx] = A[y, m * TPB + tx]
        else:
            sA[ty, tx] = 0.0


        if x < B.shape[1] and m * TPB + ty < B.shape[0]:
            sB[ty, tx] = B[m * TPB + ty, x]
        else:
            sB[ty, tx] = 0.0

        cuda.syncthreads()

        for k in range(TPB):
            tmp += sA[ty, k] * sB[k, tx]

        cuda.syncthreads()

    if y < C.shape[0] and x < C.shape[1]:
        C[y, x] = tmp
```

Summary

- When to use CuPy
  - Pre-defined kernels
  - NumPy familiarity
- When to use Numba
  - User-defined kernels
  - Advanced functions

Questions on Numba?

Accelerating PyTorch and TensorFlow with GPUs

Model training in ML/DL

- Massive matrix/tensor operations

Operation type	Example layers	GPU-accelerated?
Matrix/Tensor Multiply	Dense, Attention	Yes
Convolutions	Conv2D, Conv3D	Yes
Element-wise ops	ReLU, sigmoid	Yes
Reductions	BatchNorm, Loss	Yes
Gradients (Autograd)	All layers (train)	Yes
Optimizer updates	Adam, SGD	Yes

- Training time: hours → minutes
- Real-world examples: image recognition, NLP, simulations

Why TensorFlow & PyTorch?

Feature	PyTorch	TensorFlow
Dynamic Graphs	Yes	Yes (Eager mode)
GPU Use	Yes	Yes
JIT Compiler	Yes, **`torch.compile()`**	Yes, **`tf.function()`**
Multi-GPU	Yes	Yes

CUDA libraries under the hood

- CuDNN and cuBLAS are essential to understanding **why frameworks like PyTorch and TensorFlow are so fast on GPUs**

Library	Purpose	Used for
cuBLAS	CUDA Basic Linear Algebra Subprograms	Matrix multiplies (GEMM), dot products
cuDNN	CUDA Deep Neural Network Library	Convolution, batch norm, pooling, activations

- These are **highly optimized GPU libraries** from NVIDIA, written in CUDA, and provide fast low-level implementations of math-heavy operations.

CUDA libraries under the hood

- **PyTorch** and **TensorFlow** automatically call:
  - cuBLAS for operations like **`torch.matmul`**, **`tf.matmul`**
  - cuDNN for **`Conv2d`**, **`BatchNorm`**, **`ReLU`**, etc.
- You don't call them directly — the frameworks abstract it for you.

```
High-level API
 ├— torch.nn.Linear()
 ├— tf.keras.layers.Conv2D()
   ↓
cuDNN / cuBLAS
   ↓
GPU hardware
```

- Frameworks let you write clean Python code, but under the hood they **dispatch to cuDNN/cuBLAS kernels**, which are **highly optimized**, often beating hand-written CUDA.

CUDA libraries under the hood

- **Why this matters**
  - You don’t need to reinvent GPU kernels
- cuDNN/cuBLAS provide:
  - Kernel fusion
  - Precision tuning (e.g., FP16, Tensor cores)
  - Best practices and hardware-specific tuning
- You can customize a kernel as an advanced user

Checking availability

```
# PyTorch
import torch
device = torch.device('cuda' if
torch.cuda.is_available() else 'cpu')

# TensorFlow
import tensorflow as tf
print(tf.config.list_physical_devices('GPU'))
```

MatMul on GPUs

```
# PyTorch
A = torch.randn(1000, 1000).to(device)
B = torch.randn(1000, 1000).to(device)
C = A @ B

# TensorFlow
A = tf.random.normal([1000, 1000])
B = tf.random.normal([1000, 1000])
C = tf.matmul(A, B)
```

AutoGrad

```
# PyTorch
x = torch.tensor(1.0,
requires_grad=True,
device='cuda')
y = x**2 + 2*x + 1
y.backward()
print(x.grad)
```

```
# TensorFlow
tape = tf.GradientTape()
with tape:
    x = tf.Variable(1.0)
    y = x**2 + 2*x + 1
grad = tape.gradient(y, x)
```

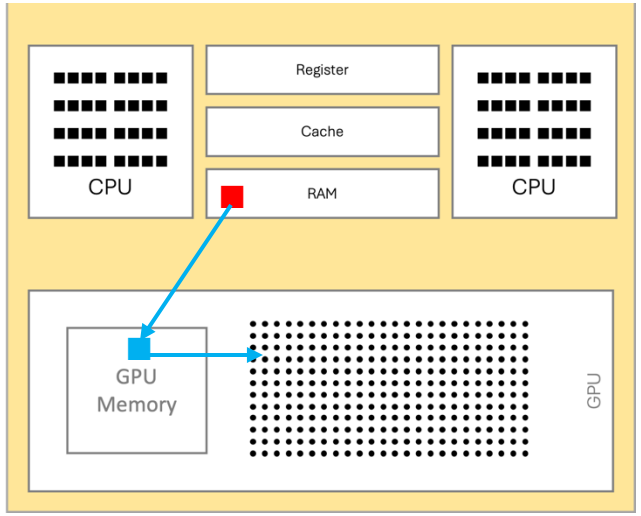
Training on GPU (PyTorch)

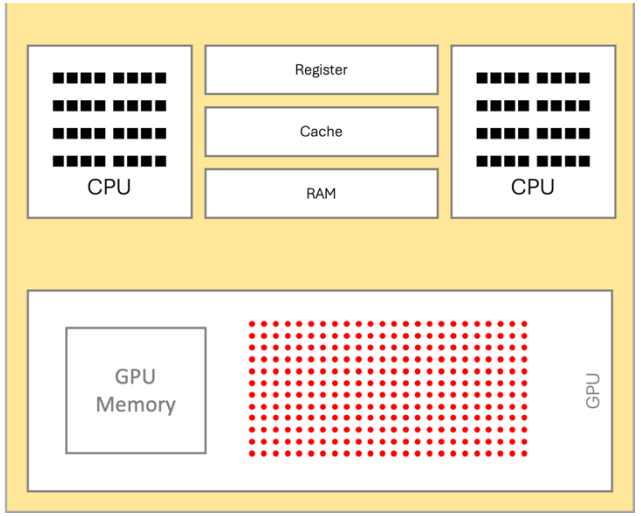
```
model = MyModel().to(device)
for x, y in dataloader:
    x, y = x.to(device), y.to(device)
    optimizer.zero_grad()
    loss = model(x).loss(y)
    loss.backward()
    optimizer.step()
```

Avoid CPU ↔ GPU transfers inside loops

```
for i in range(1000):
    data = torch.tensor([i], device='cpu')  # Created on CPU
    data = data.cuda()  # Transfers to GPU every time!
    result = data * 2
```

```
device = torch.device('cuda')
data = torch.arange(1000, device=device)  # Created directly on GPU
for i in range(1000):
    result = data[i] * 2
```

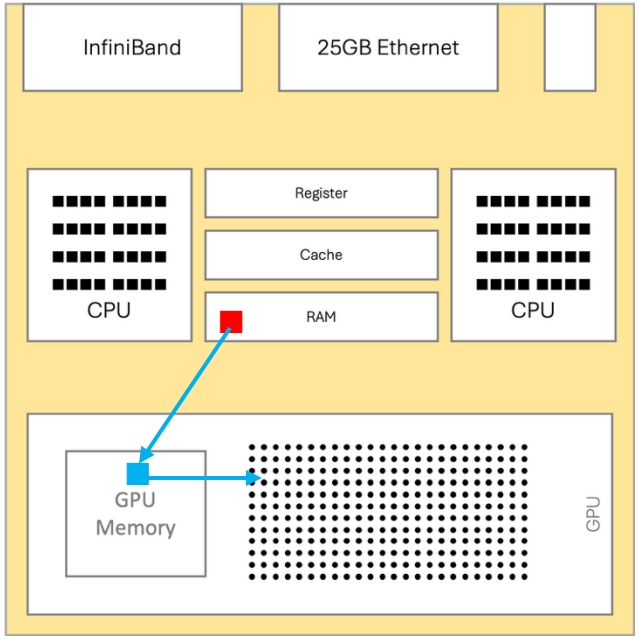


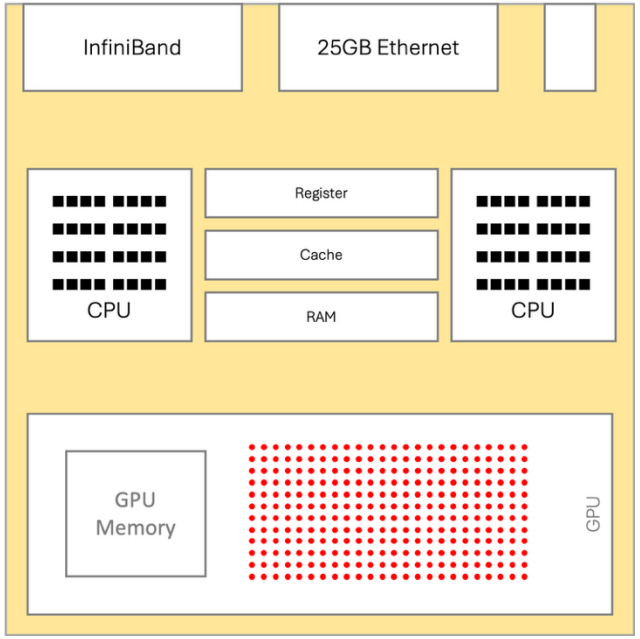


Avoid CPU ↔ GPU transfers inside loops

```
import tensorflow as tf

for i in range(1000):

    data = tf.constant([i], dtype=tf.int32)  # Created on CPU by default

    result = tf.multiply(data, 2)  # TensorFlow will move to GPU if needed
```

```
with tf.device('/GPU:0'):
    data = tf.range(1000)


for i in range(1000):

    result = data[i] * 2  # No transfer overhead
```





Multi-GPU training

- **Goal**: Split training across multiple GPUs to speed things up.
- **Typical use cases**: Large models, large datasets, faster training.

Common Strategies:

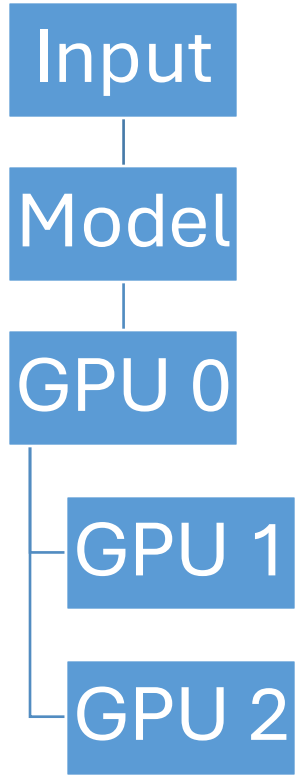
- **Data parallelism**: Copy model to multiple GPUs, feed each a mini-batch.
- **Model parallelism** (less common): Split model layers across GPUs.

Multi-GPU training – data parallelism

DataParallel

```
import torch
import torch.nn as nn

model = MyModel()
if torch.cuda.device_count() > 1:
    print(f"Using {torch.cuda.device_count()} GPUs")
    model = nn.DataParallel(model)  # Automatically
splits data
model = model.cuda()
```

Pytorch **DataParallel**

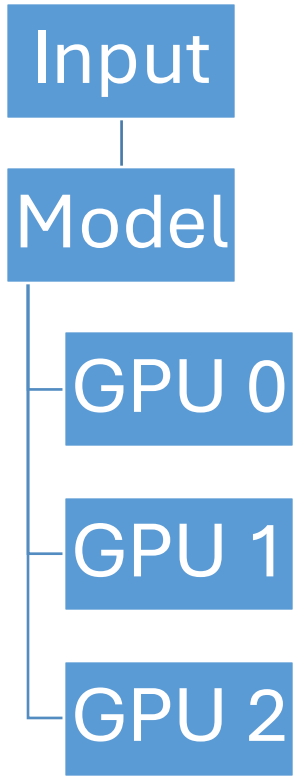


Multi-GPU training – data parallelism

DistributedDataParallel

```
import torch
import torch.distributed as dist
import torch.nn as nn
from torch.nn.parallel import DistributedDataParallel as
DDP

def setup():
    dist.init_process_group("nccl")

def cleanup():
    dist.destroy_process_group()

def main():
    setup()
    model = MyModel().cuda()
    ddp_model = DDP(model)

    # Forward + backward
    outputs = ddp_model(inputs)
    loss = loss_fn(outputs, labels)
    loss.backward()
    optimizer.step()
    cleanup()
```

Pytorch
DistributedDataParallel



Multi-GPU training – data parallelism

DataParallel

```
import torch
import torch.nn as nn

model = MyModel()
if torch.cuda.device_count() > 1:
    print(f"Using {torch.cuda.device_count()} GPUs")
    model = nn.DataParallel(model)  # Automatically
splits data
model = model.cuda()
```

Feature	DataParallel	DistributedDataParallel
Process model	Single-process	Multi-process (1 per GPU)
Communication	Centralized (through main GPU)	Decentralized (peer-to-peer)
Speed	Slower	Faster, scalable
Use case	Quick, small jobs	Large-scale training

DistributedDataParallel (Rec.)

```
import torch
import torch.distributed as dist
import torch.nn as nn
from torch.nn.parallel import DistributedDataParallel as
DDP

def setup():
    dist.init_process_group("nccl")

def cleanup():
    dist.destroy_process_group()

def main():
    setup()
    model = MyModel().cuda()
    ddp_model = DDP(model)

    # Forward + backward
    outputs = ddp_model(inputs)
    loss = loss_fn(outputs, labels)
    loss.backward()
    optimizer.step()
    cleanup()
```

Multi-GPU training – data parallelism

Tensorflow **`MirroredStrategy`**

```
import tensorflow as tf

# Automatically uses all available GPUs
strategy = tf.distribute.MirroredStrategy()
print(f"Number of devices: {strategy.num_replicas_in_sync}")

with strategy.scope():
    model = tf.keras.Sequential([
        tf.keras.layers.Dense(512, activation='relu'),
        tf.keras.layers.Dense(10)
    ])
    model.compile(optimizer='adam',
                  loss='sparse_categorical_crossentropy',
                  metrics=['accuracy'])
# Now this trains in parallel
model.fit(train_dataset, epochs=5)
```

Automatically splits batches and syncs gradients.

Multi-GPU training

Feature	PyTorch **DataParallel**	PyTorch **DDP**	TF **MirroredStrategy**
Setup	Simple	Advanced	Simple
Speed	Slower	Faster	Fast
Gradient Sync	Centralized	NCCL (all-reduce)	All-reduce
Scalability	Low	High	High

Just-in-time compilation

- Neural networks involve lots of repetitive, numeric computation.
- Removing Python interpretation overhead makes training/inference much faster.

- TensorFlow: **`@tf.function`**
- PyTorch: **`torch.compile(model)`** (2.0+)

JIT compilation

```
import torch
import torch.nn as nn


class SimpleModel(nn.Module):
    def __init__(self):
        super().__init__()
        self.linear = nn.Linear(1000, 1000)


    def forward(self, x):
        return torch.relu(self.linear(x))


model = SimpleModel()
compiled_model = torch.compile(model)  # JIT-compiled version


x = torch.randn(64, 1000)
output = compiled_model(x)
```

```
import tensorflow as tf


@tf.function  # Compile this function
def simple_train_step(x, y, model, loss_fn, optimizer):
    with tf.GradientTape() as tape:
        pred = model(x)
        loss = loss_fn(y, pred)
    grads = tape.gradient(loss, model.trainable_variables)
    optimizer.apply_gradients(zip(grads, model.trainable_variables))
    return loss
```

JIT compilation

- **Use JIT compilation to squeeze out more performance** from your models.
- Works best on **static or semi-static** computation patterns (e.g., model layers, training steps).
- Combine with GPU acceleration for **maximum speed gains**.

When GPU won’t help

- Tiny models or batches
  - The GPU needs large workloads to amortize its latency and data transfer overhead.

```
# Small model and batch — CPU is actually faster
import torch
import torch.nn as nn

model = nn.Linear(10, 10).to("cuda")
x = torch.randn(4, 10).to("cuda")  # Tiny batch
```

CPU bottlenecks: data I/O and preprocessing

```
from torchvision import transforms
from PIL import Image
import os

transform = transforms.ToTensor()
images = []

for filename in os.listdir("images"):
    img = Image.open(f"images/{filename}")
    tensor = transform(img)  # On CPU
    images.append(tensor.to("cuda"))  # Then move to GPU
```

✅ **Solution:**

Use **`torch.utils.data.DataLoader`** with multiple workers (num_workers > 0)

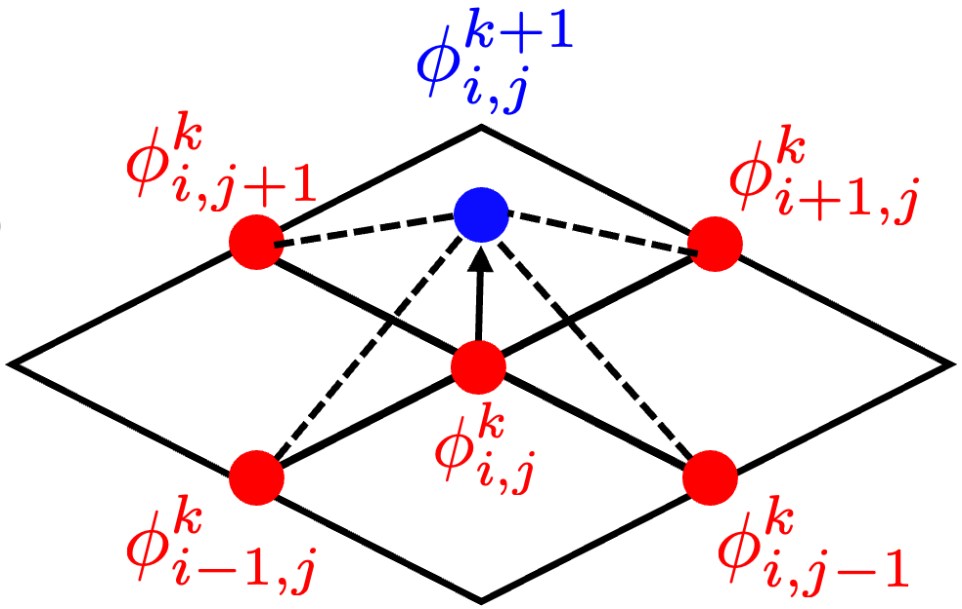
Use **`PrefetchGenerator`**, **`dali`**, or **`preprocess`** on GPU if possible.

```
import torch
from torch.utils.data import Dataset, DataLoader
from torchvision import transforms
from PIL import Image
import os

# Dataset and DataLoader
dataset = ImageFolderDataset("images", transform=transform)
dataloader = DataLoader(dataset, batch_size=32, shuffle=True,
                        num_workers=4, pin_memory=True)

# Move batch to GPU inside training loop
device = torch.device("cuda")

for batch in dataloader:
    batch = batch.to(device, non_blocking=True)
    # Proceed with model(batch)
```

Questions on PyTorch/TensorFlow GPU?

Hands-on tutorials

```
import numpy as np
import matplotlib.pyplot as plt
# Parameters
N = 256
tolerance = 1e-4
max_iters = 100000
dx = 1.0 / N
# Initialize grid
p = np.zeros((N, N))
f = np.zeros((N, N))
f[N//2, N//2] = 10000  # Heat source
# Boundary conditions: p = 0 on the edges (already zero)
# Jacobi iteration
for iteration in range(max_iters):
    p_new = p.copy()
    p_new[1:-1, 1:-1] = 0.25 * (
        p[:-2, 1:-1] + p[2:, 1:-1] +
        p[1:-1, :-2] + p[1:-1, 2:] +
        dx**2 * f[1:-1, 1:-1]
    )
    diff = np.linalg.norm(p_new - p)
    p = p_new
    if diff < tolerance:
        print(f"Converged after {iteration} iterations")
        break
print(p[N//2, N//2])

#plt.imshow(p, cmap='hot')
#plt.colorbar()
#plt.title("NumPy Heat Diffusion")
#plt.show()
```



Hands-on tutorials

- Setting things up on Juno:
  - Activate your Python environment: **`source your_venv/bin/activate`**
    `pip3 install numba`

`$ git clone` https://gitlab.circ.utdallas.edu/hpc-utd/workshop/accelerating-python-with-gpus.git

- We provided a numpy Jacobi solver under **`tutorials/Heat_diffusion_Numpy.py`**
- Alternatively, you can copy that directory to your home directory: **`cp -r /scratch/juno/hpcre/workshop/python-gpu/ .`**
- Allocate a GPU node: **`salloc -p a30-4.6gb --mem=2GB --time=00:30:00 --reservation=hpcre-aug8`**
- Login to the GPU node: **`ssh g-04-01`**
- Exercise: Rewrite the code using Numba and Numba+GPU.

```
module purge
module load gnu12
module load python/3.11.11
module load cuda/12.6
```

Pytorch hands-on tutorial (**tutorials/pytorch_test.py**)

```
import torch
import torch.nn as nn
import torch.optim as optim
from torchvision import datasets, transforms
from torch.utils.data import DataLoader

# Set device
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
print(f"Using device: {device}")

# Define a simple CNN
class SimpleCNN(nn.Module):
    def __init__(self):
        super().__init__()
        self.model = nn.Sequential(
            nn.Conv2d(1, 32, 3, padding=1),  # input: 1x28x28 → 32x28x28
            nn.ReLU(),
            nn.MaxPool2d(2, 2),              # → 32x14x14
            nn.Conv2d(32, 64, 3, padding=1), # → 64x14x14
            nn.ReLU(),
            nn.MaxPool2d(2, 2),              # → 64x7x7
            nn.Flatten(),
            nn.Linear(64 * 7 * 7, 128),
            nn.ReLU(),
            nn.Linear(128, 10)
        )
    def forward(self, x):
        return self.model(x)
```

```
# Load MNIST dataset
transform = transforms.Compose([transforms.ToTensor()])
train_dataset = datasets.MNIST(root="./data", train=True,
transform=transform, download=True)
train_loader = DataLoader(train_dataset, batch_size=64, shuffle=True)


# Initialize model, loss, optimizer
model = SimpleCNN().to(device)
criterion = nn.CrossEntropyLoss()
optimizer = optim.Adam(model.parameters(), lr=0.001)


# Training loop
for epoch in range(1, 6):
    running_loss = 0.0
    for images, labels in train_loader:
        images, labels = images.to(device), labels.to(device)
        optimizer.zero_grad()
        outputs = model(images)
        loss = criterion(outputs, labels)
        loss.backward()
        optimizer.step()
        running_loss += loss.item()
    print(f"Epoch {epoch}, Loss: {running_loss/len(train_loader):.4f}")
```

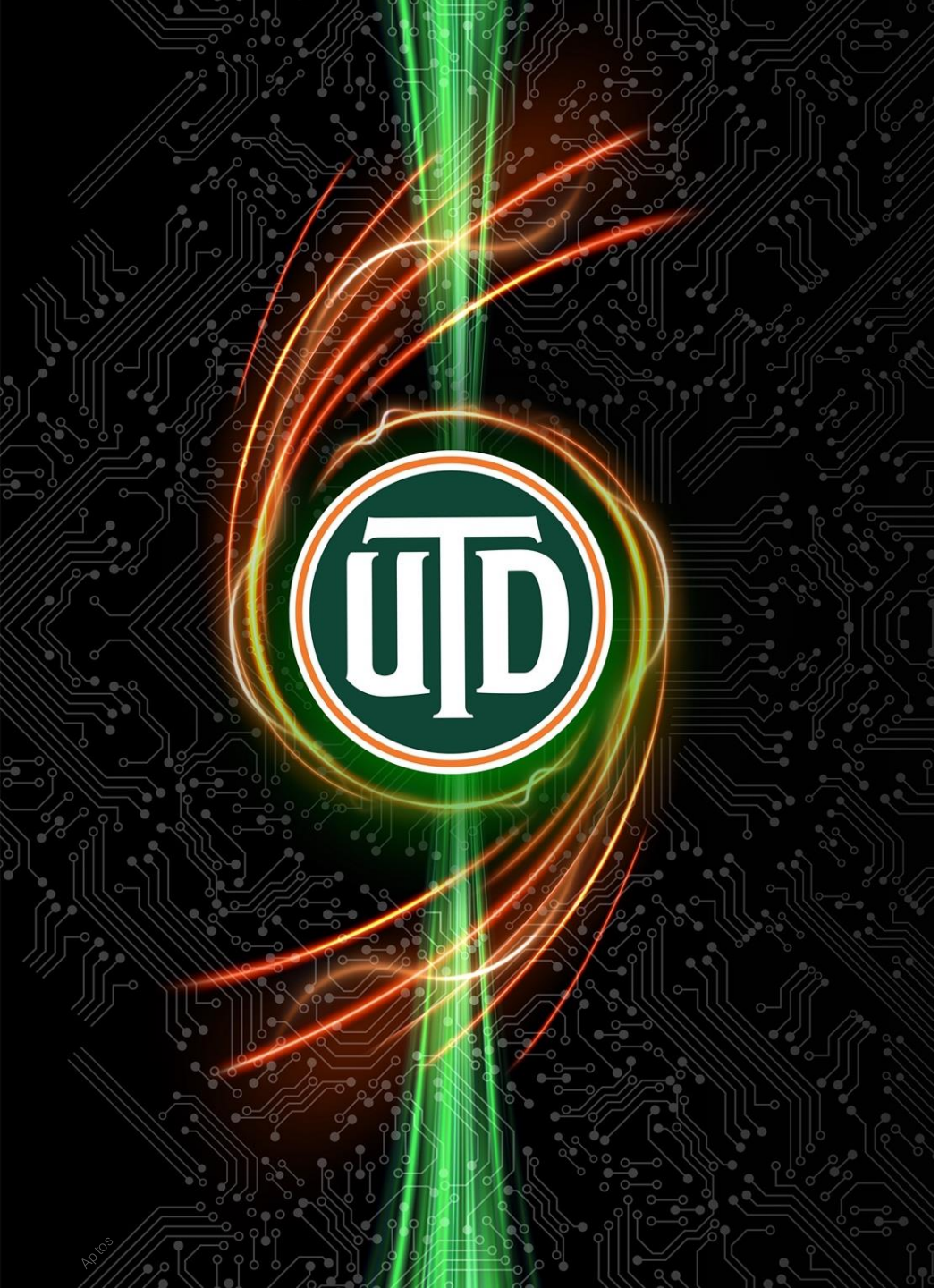
Exercise: Time and compare the training time on CPU (on **normal** partition) and GPU (on **h100** partition. You can also play with different variations of the code.

References

- mpi4py documentation
- CuPy documentation
- Numba documentation
- PyTorch
- TensorFlow

Any questions?

If you have any Python script that needs to be accelerated, do it today!
We’re happy to help!



Thank you

HPC@UTD