# Introduction to GPU and CUDA

*Shaohao Chen*

*ORCD at MIT*

# Outline

❑ GPU basics

❑ CUDA programming

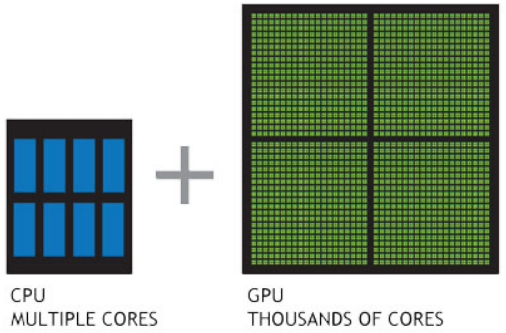
- CUDA libraries
- CUDA parallel hierarchy: grids, blocks, and threads
- Example: vector addition
- Exercise: SAXPY
- Example: matrix multiplication, GPU shared memory

❑ GPU communication

- NCCL

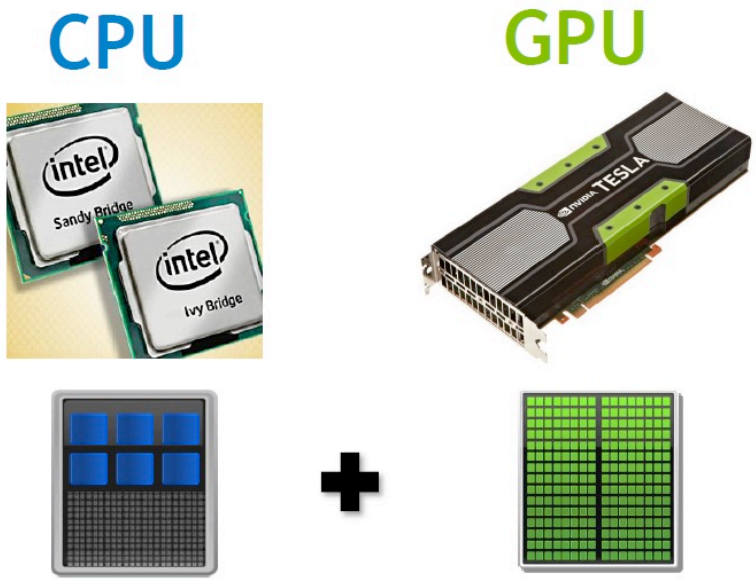
# GPU and GPGPU

- Originally, graphics processing unit (GPU) is dedicated for manipulating computer graphics and image processing. Traditionally GPU is known as “video card”.
- GPU’s highly parallel structure makes it efficient for parallel programs. Nowadays GPUs are used for tasks that were formerly the domain of CPUs, such as scientific computation. This kind of GPU is called general-purpose GPU (GPGPU) .
- In many cases, a parallel program runs faster on GPU than on CPU. Note that a serial program runs slower on GPU than on CPU.
- The most popular type of GPU in the high-performance computing world is NVIDIA GPU.



# GPU is an accelerator

- GPU is a device on a CPU-based system (called host). GPU is connected to CPU via PCIe.
- Computer programs can be parallelized and thus accelerated on GPU.
- CPU and GPU have separate memories. Data transfer between CPU and GPU is required.



# GPU types

❑ NVIDIA GPU family

- Data Center GPU

Tesla series: K20, K80, P100, V100, A100, H100 (For all precisions, FP64, FP32 and lower)

A40, L40 (For FP32 and lower)

- Workstation GPU

Quadro series: Quadro6000, A6000 (For FP32 and lower)

- Desktop GPU

GeForce series: RTX1080, RTX2080, Titan X, RTX3090 (For FP32 and lower)

❑ AMD GPU: MI250, MI300.

❑ Intel Data Center GPU

# GPU compute units

- CUDA cores

  - Number of CUDA cores: thousands – 16,000+.

  - Each CUDA core can process multiple threads simultaneously and execute a floating point and an integer operation concurrently.

- Streaming Multiprocessor (SM)

  - Number of SMs: 10s – 100+

  - CUDA cores are grouped into larger units called streaming multiprocessors (SMs), and each SM can execute up to thousands of threads concurrently.


Kepler GK110 Full chip block diagram

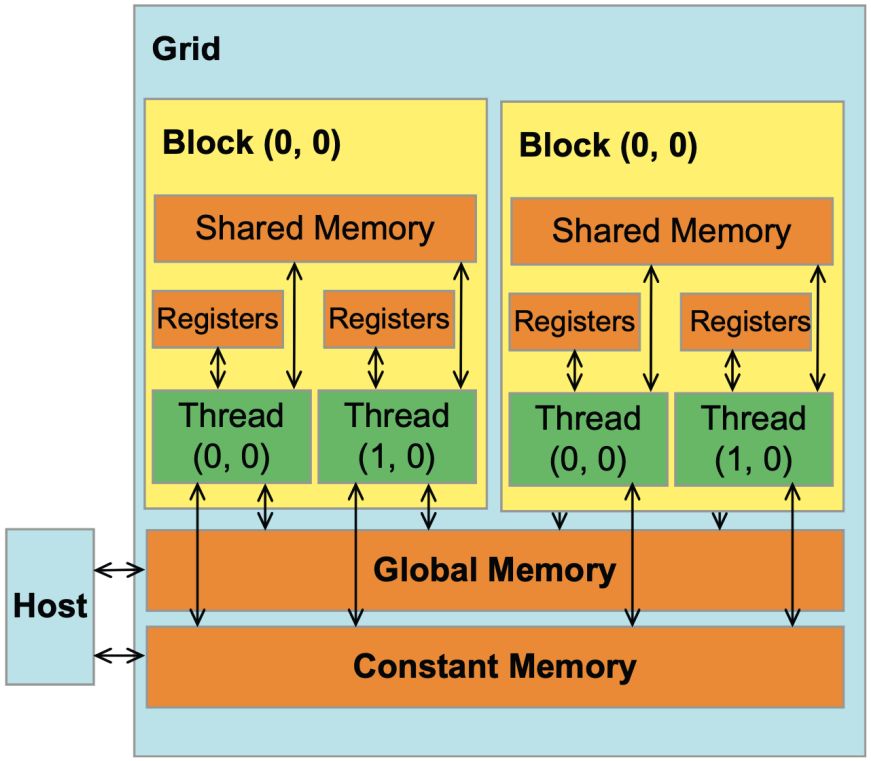
# GPU Memory

Device code can:

- R/W per-thread registers
- R/W per-thread local memory
- R/W per-block shared memory
- R/W per-grid global memory
- Read only per-grid constant memory

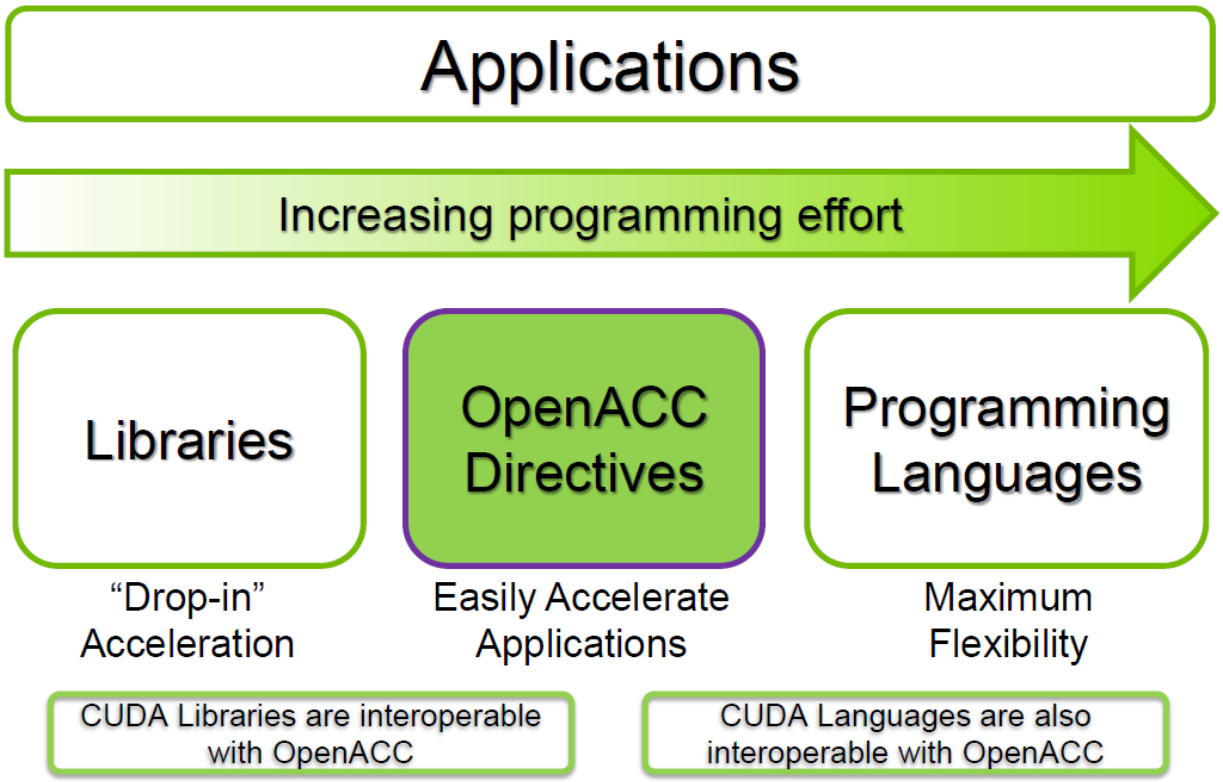
Host code can

- Transfer data to/from per grid global and constant memories




- High memory bandwidth: quickly and easily access large amounts of data.

# GPU Applications

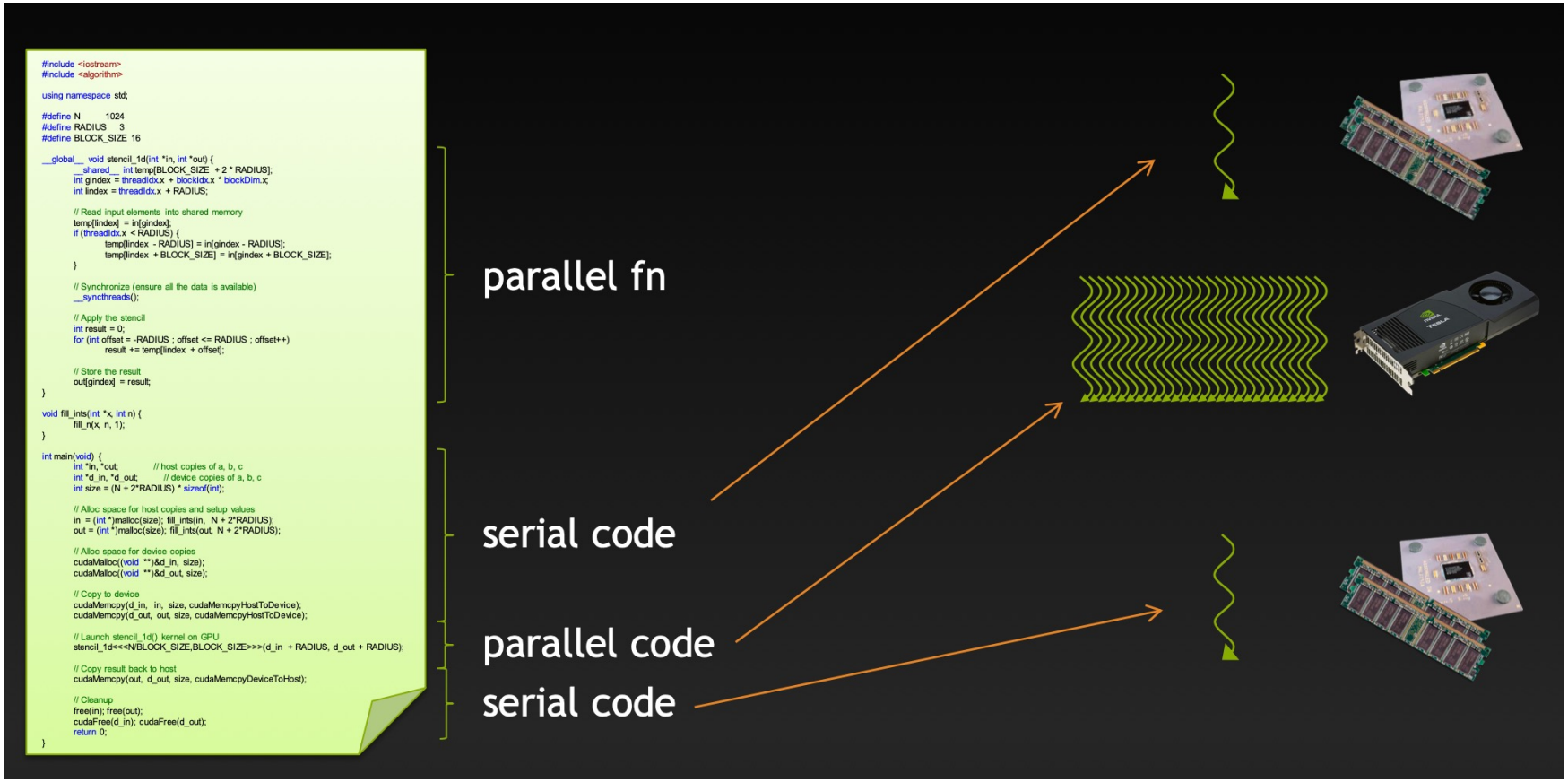


# CUDA libraries

- CUDA Math Libraries: cuBLAS, cuFFT, cuSolver, cuSPARSE
- Parallel Algorithm Libraries: Thrust
- Communication Libraries: NCCL, NVSHMEM
- Deep Learning Core: cuDNN, TensorRT
- Partner Libraries: MAGMA, OpenCV, FFmpeg

- Ref: https://developer.nvidia.com/gpu-accelerated-libraries#data-processing-libs

# CUDA program flow



# CUDA parallel hierarchy

- Compute Unified Device Architecture = CUDA

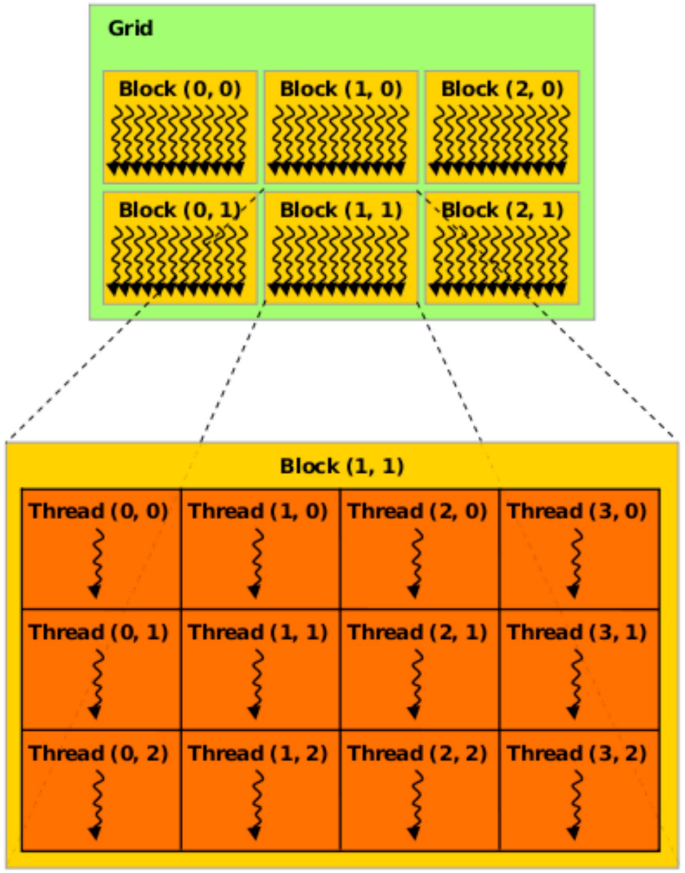
- Grid:

  contains many thread blocks

  (depending on the problem size)
- Thread Block:

  contains many threads (e.g. 256, 512, 1024)
- Thread

  the smallest processing unit



# GPU–CUDA Granularity

- GPU device -- grids

  Each GPU device can process multiple kennels/grids

- Streaming Multiprocessors (SMs) -- blocks:

  Each SM can process multiple thread blocks simultaneously.

  A block scheduler queues a large number of blocks to obtain efficiency.

  When an SM is available, blocks are deposited to the SM.

- CUDA cores – warps:

  Each CUDA core can process 32 threads (a warp) simultaneously.

  -- Warp: a group of 32 threads.

  A warp scheduler handles when instructions are executed on warps.

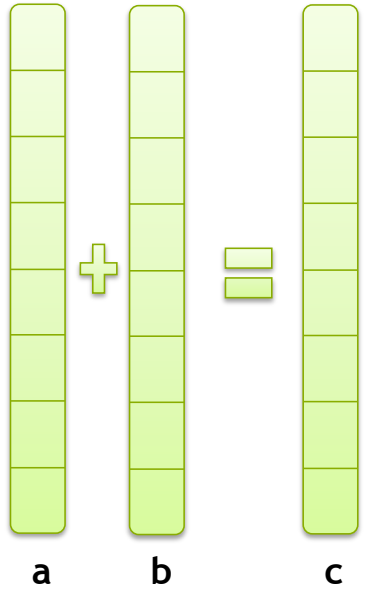
# Info of H100 (SXM card)

- Maximum Threads per Block: 1024
- Maximum Block Dimensions: 1024, 1024, 64
- Maximum Grid Dimensions: 2147483647 x 65535 x 65535
- Number of SMs: 132
- Max Threads Per SM: 2048
- CUDA Cores (FP32) per SM: 128
- CUDA Cores (FP32): 16,896
- Warp Size: 32
- Global Memory Size: ~ 80 GB
- Memory bandwidth: ~ 3.35 TB/s for the SXM5 variant, ~ 2 TB for the PCIe variant.
- Shared memory per SM: ~ 49 KB
- Max clock rate: 1980 MHz
- Peak computing power (FP32): CUDA cores * Clock Rate * 2 = 66.9 TFLOPS
- Peak computing power (FP64): 33.5 TFLOPS

# Serial vector addition on CPU

- Serial code: add elements sequentially.

```
void add_vectors(int a[], int b[], int c[], int size) {
    for (int i = 0; i < size; i++) {
        c[i] = a[i] + b[i];
    }
}
```



# Parallel vector addition in CUDA

- Parallel code: each thread adds elements simultaneously.

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

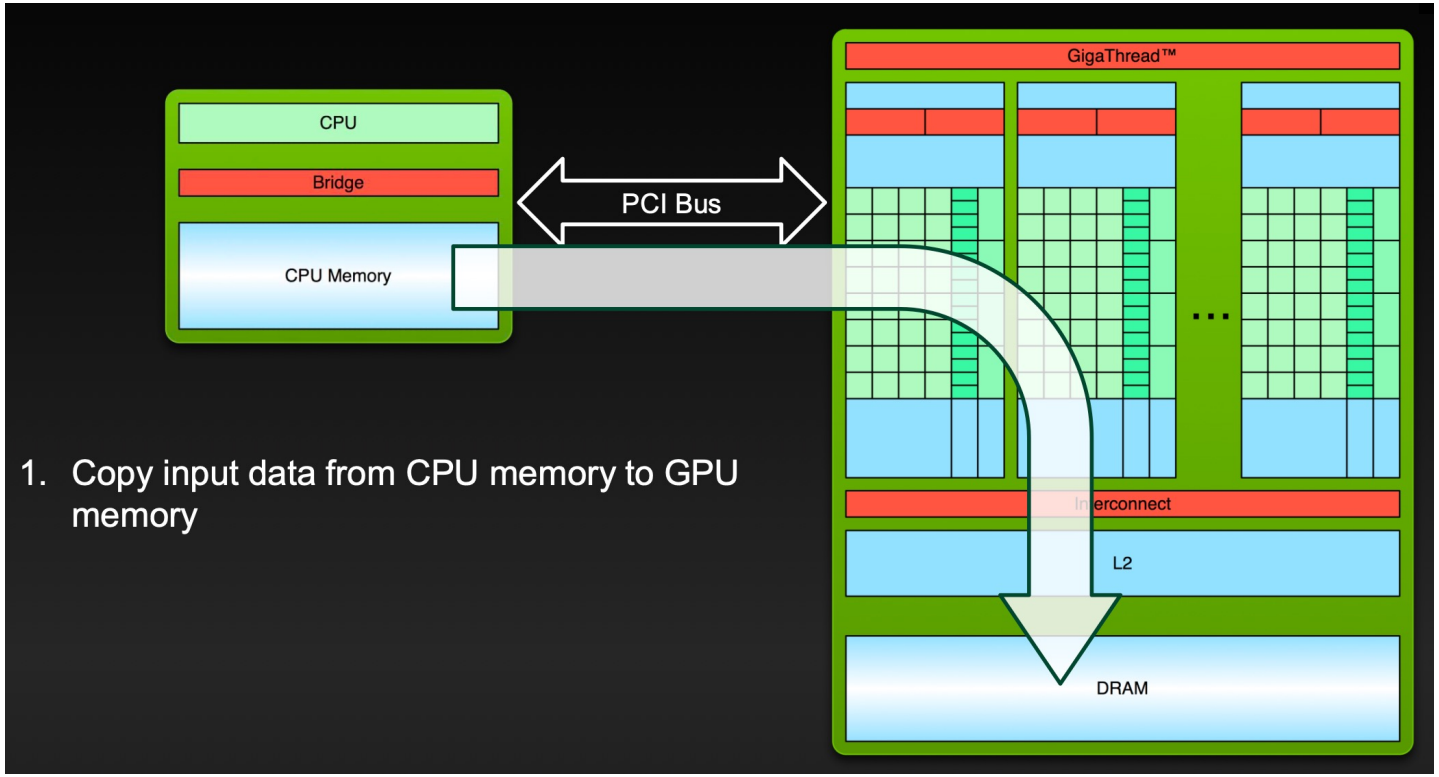
Function Qualifiers

- __global__ : called from the CPU, executed on the GPU.
- __device__ : called from the GPU, executed on the GPU.
- __host__ : called from the CPU, executed on the CPU.

Built-in variables

- threadIdx, blockIdx: Block and thread indices, 3-dimensional.
- blockDim: block dimension, 3-dimensional.

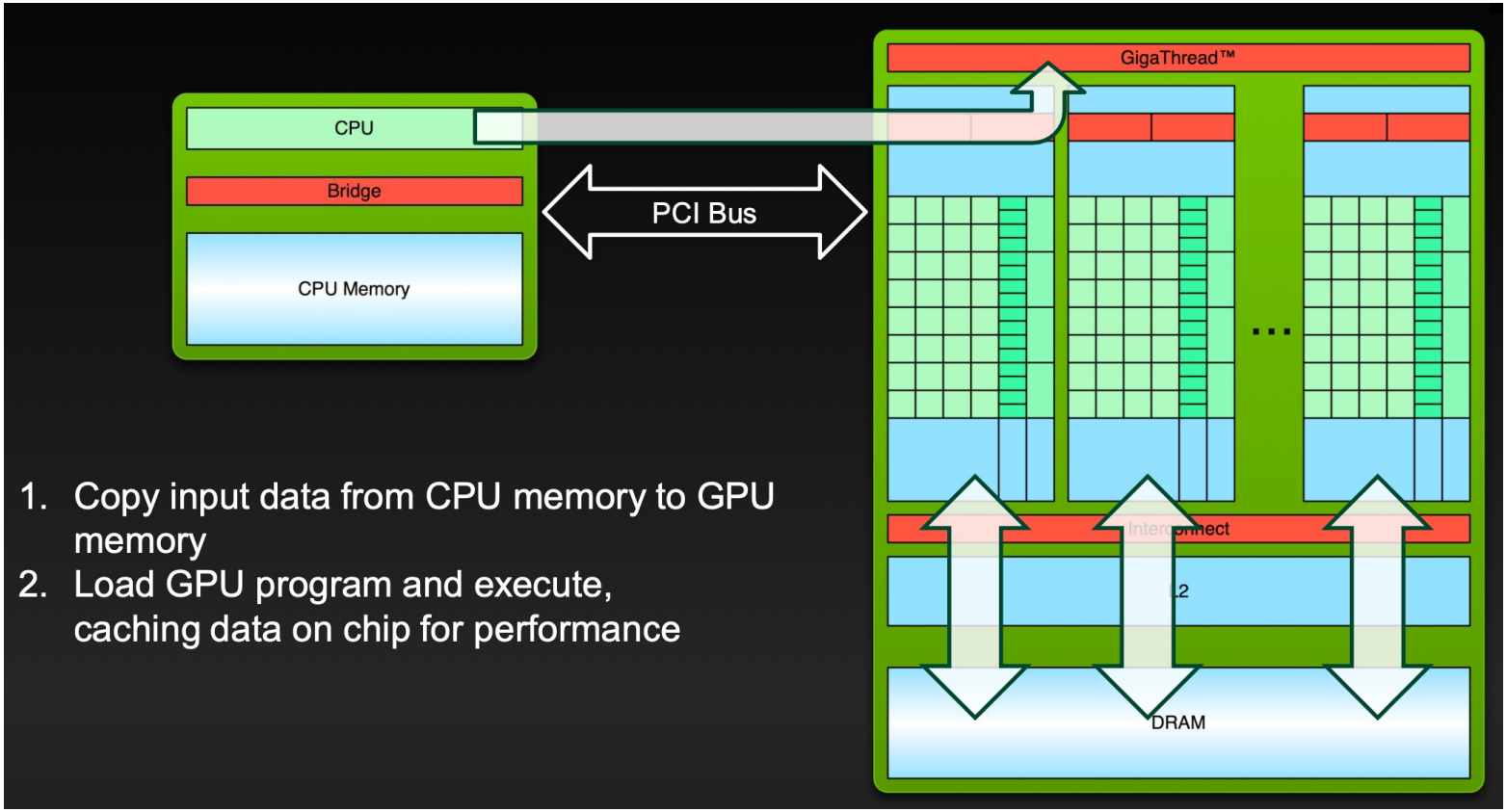
# Data flow (1)




1. Copy input data from CPU memory to GPU memory

```
cudaMemcpy( d_a, a, size, cudaMemcpyHostToDevice );
cudaMemcpy( d_b, b, size, cudaMemcpyHostToDevice );
```

# Data flow (2)




1. Copy input data from CPU memory to GPU memory
2. Load GPU program and execute, caching data on chip for performance

# Launch a CUDA Kernel

- Define the size of the problem, the number of blocks, and threads per block

```
#define N (4096*4096)
#define THREADS_PER_BLOCK 512
```

```
BLOCKS = N / THREADS_PER_BLOCK
```

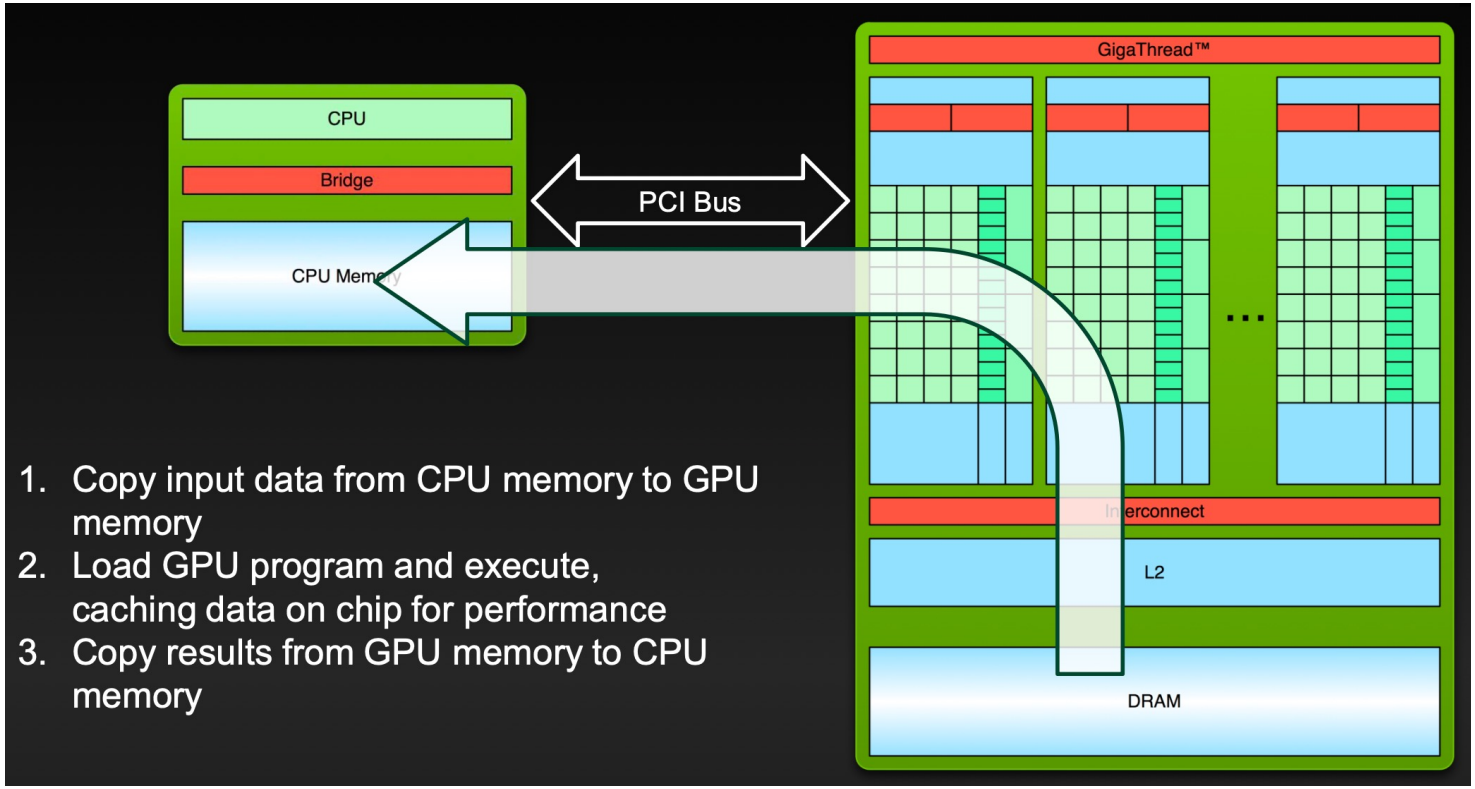
  - ➢ Threads per block must be a multiple of 32. Typical values are 256, 512, or 1024.
  - ➢ The number of blocks should be large enough to keep all SMs busy.

- Launch the kernel on a GPU

```
add<<< N / THREADS_PER_BLOCK, THREADS_PER_BLOCK >>>( d_a, d_b, d_c );
```

  - ➢ The blocks and warps are issued parallelly on SMs and CUDA cores.

# Data flow (3)




1. Copy input data from CPU memory to GPU memory
2. Load GPU program and execute, caching data on chip for performance
3. Copy results from GPU memory to CPU memory

```
cudaMemcpy( c, d_c, size, cudaMemcpyDeviceToHost );
```

# Run a CUDA program on a GPU

- Log in Engaging

```
ssh <user>@eofe10.mit.edu
```

- Work on GPUs

```
srun -t 120 -p mit_normal_gpu -N 1 -n 2 --mem=10GB --gres=gpu:1 --pty bash
module load cuda/12.4.0
```

- Compile and run CUDA programs

```
nvcc vec_add.cu -o vec_add
./vec_add
```

- Check GPU utilization

```
nvidia-smi
nvtop
```

# Exercise: SAXPY

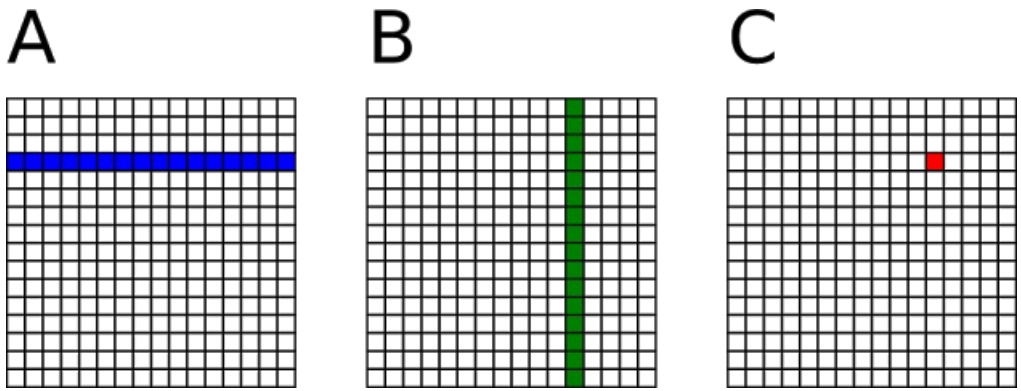
- Given a serial C code computing SAXPY, convert it to a CUDA C code.
- Compile the code and run the program on a GPU.
- Measure the run time and observe the speed up.

**S[i] = a * X[i] + Y[i]**

```
void saxpy(int n, float alpha, float *X, float *Y)
{
    for (int i = 0; i < n; i++) {
        Y[i] = alpha * X[i] + Y[i];
    }
}
```

# Serial matrix multiplication on CPU

- An element of C equals the inner product of one row of A and one column of B.
- The computation is sequential.
- Time complexity is O(N^3).



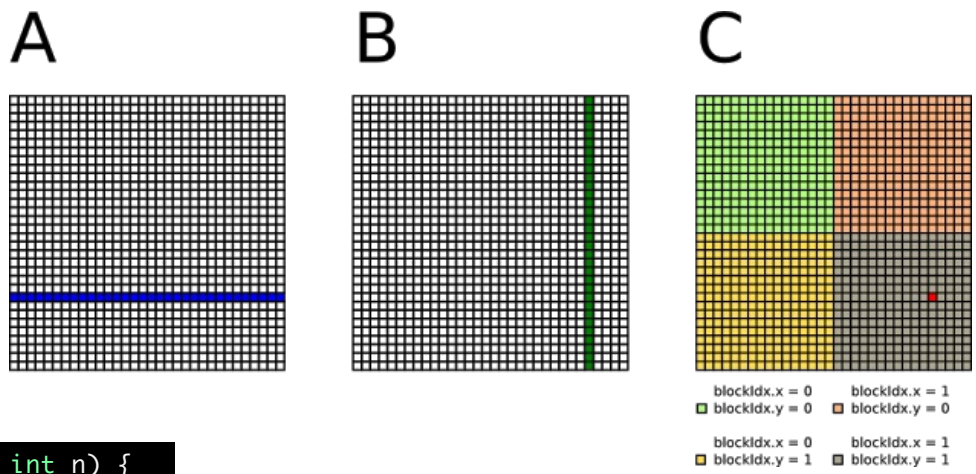

```
void multiply_matrices(float** a, float** b, float** c, int size) {

    for (int i = 0; i < size; i++) {
        for (int j = 0; j < size; j++) {
            for (int k = 0; k < size; k++) {
                c[i][j] += a[i][k] * b[k][j];
            }
        }
    }
}
```

https://ecatue.gitlab.io/gpu2018/pages/Cookbook/matrix_multiplication_cuda.html

# Parallel matrix multiplication in CUDA

- Store matrices in the global memory
- Use 2D indices for blocks and threads
- Each thread loads one row of matrix A and one column of matrix B from global memory, do the inner product, and store the result back to matrix C in the global memory.



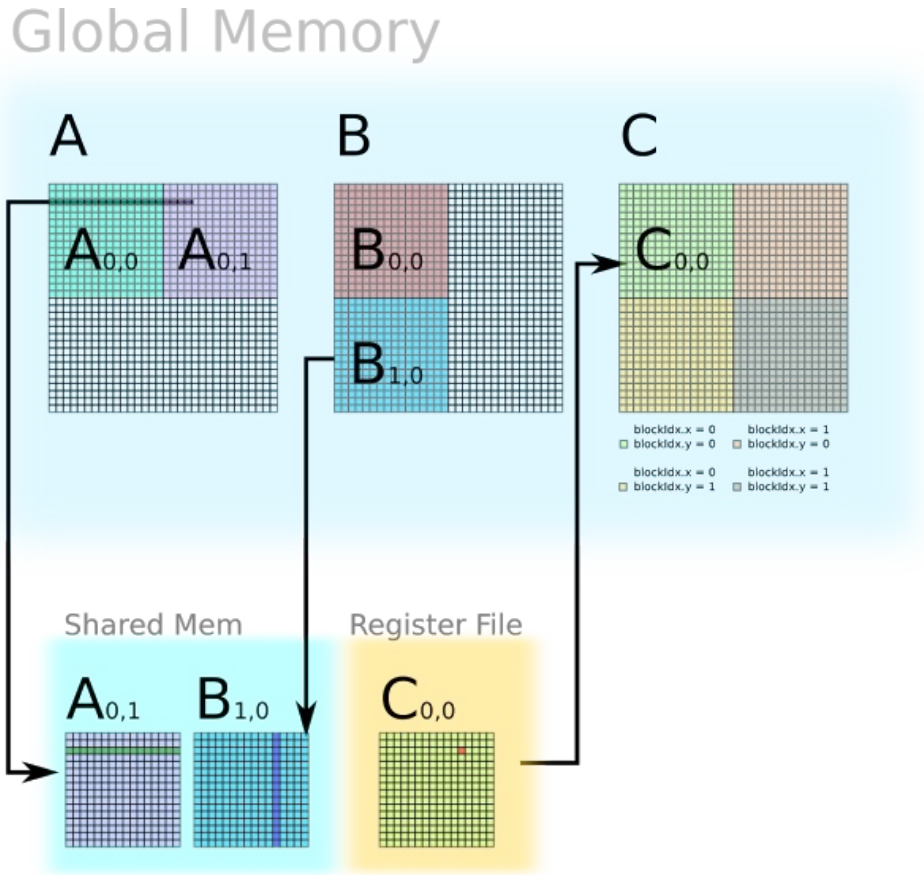

```
__global__ void matrixMulKernel(float *A, float *B, float *C, int n) {
    int row = blockIdx.y * blockDim.y + threadIdx.y;
    int col = blockIdx.x * blockDim.x + threadIdx.x;


    if (row < n && col < n) {
        float value = 0.0;
        for (int k = 0; k < n; ++k) {
            value += A[row * n + k] * B[k * n + col];
        }
        C[row * n + col] = value;
    }
}
```

https://ecatue.gitlab.io/gpu2018/pages/Cookbook/matrix_multiplication_cuda.html

# Parallel matrix multiplication with shared memory (1)

- Store matrices on global memory.
- Divide matrices A and B into smaller tiles.
- Load tiles from global memory into shared memory.
- Tile multiplication: each thread performs the inner product to produce one element of C. The result is stored in the register and will be accumulated across tiles.
- Threads access shared memory with low latency and high bandwidth.
- Synchronize threads after tiles are loaded and after tile multiplications are completed.
- The result in the register file will be stored back into global memory at the end.




https://ecatue.gitlab.io/gpu2018/pages/Cookbook/matrix_multiplication_cuda.html

# Parallel matrix multiplication with shared memory (2)

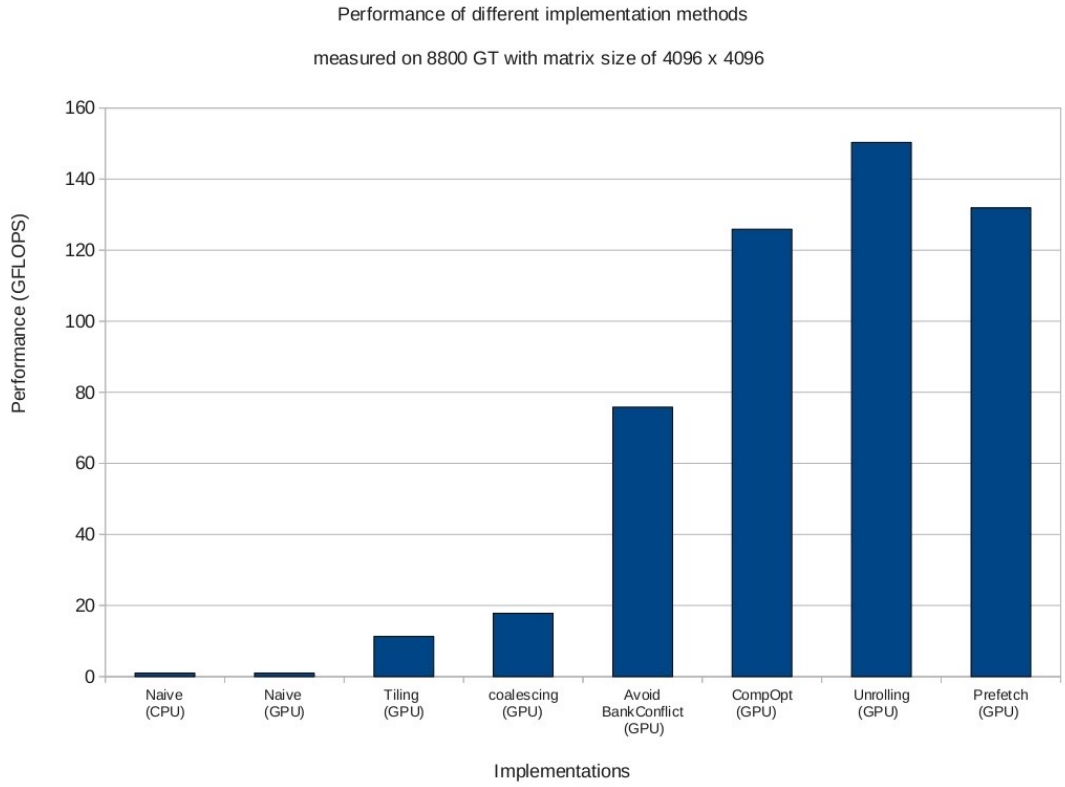
```
__global__ void matrixMulKernel(float *A, float *B, float *C, int n) {
    __shared__ float As[TILE_SIZE][TILE_SIZE];
    __shared__ float Bs[TILE_SIZE][TILE_SIZE];
    int row = blockIdx.y * blockDim.y + threadIdx.y;
    int col = blockIdx.x * blockDim.x + threadIdx.x;
    float value = 0.0;

    for (int t = 0; t < (n + TILE_SIZE - 1) / TILE_SIZE; t++) {
        if (row < n && (t * TILE_SIZE + threadIdx.x) < n) {
            As[threadIdx.y][threadIdx.x] = A[row * n + (t * TILE_SIZE + threadIdx.x)];
        } else {
            As[threadIdx.y][threadIdx.x] = 0.0;
        }
        if (col < n && (t * TILE_SIZE + threadIdx.y) < n) {
            Bs[threadIdx.y][threadIdx.x] = B[(t * TILE_SIZE + threadIdx.y) * n + col];
        } else {
            Bs[threadIdx.y][threadIdx.x] = 0.0;
        }
        __syncthreads();
        for (int k = 0; k < TILE_SIZE; k++) {
            value += As[threadIdx.y][k] * Bs[k][threadIdx.x];
        }
        __syncthreads();
    }
    if (row < n && col < n) {
        C[row * n + col] = value;
    }
}
```

# Optimized parallel matrix multiplication with cuBLAS

- Call a cuBLAS function
- cuBLAS employs several advanced optimizations to enhance performance

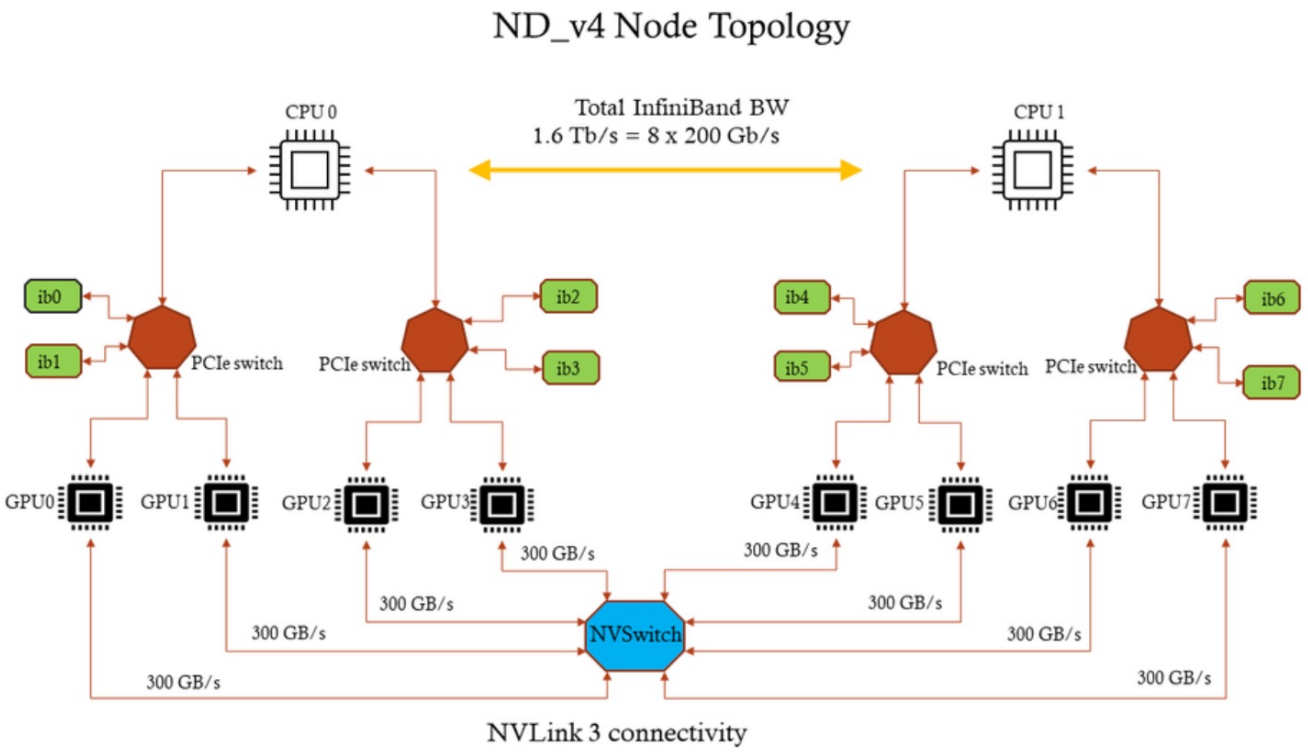
```
cublasSgemm(handle,
            CUBLAS_OP_N,
            CUBLAS_OP_N,
            n, n, n,
            &alpha,
            d_A, n,
            d_B, n,
            &beta,
            d_C, n);
```




https://ecatue.gitlab.io/gpu2018/pages/Cookbook/matrix_multiplication_cuda.html

# GPU Server

- Multiple GPUs on a node
- PCIe: communication between CPU and GPU
- NVlink: fast communication between GPUs within a node.
- Infiniband network: communication across nodes.



# NVIDIA Collective Communications Library (NCCL)

- Automatic topology detection: graph search for the optimal set of rings and trees with the highest bandwidth and lowest latency over PCIe and NVLink high-speed interconnects within a node and over Infiniband network across nodes.
- Provide routines for point-to-point and collective communications.
- Integrated within several deep learning frameworks such as PyTorch and Deepspeed.

PCI

NVLink

NVLink + Infiniband

https://developer.nvidia.com/nccl

# Communication operations across hardware types

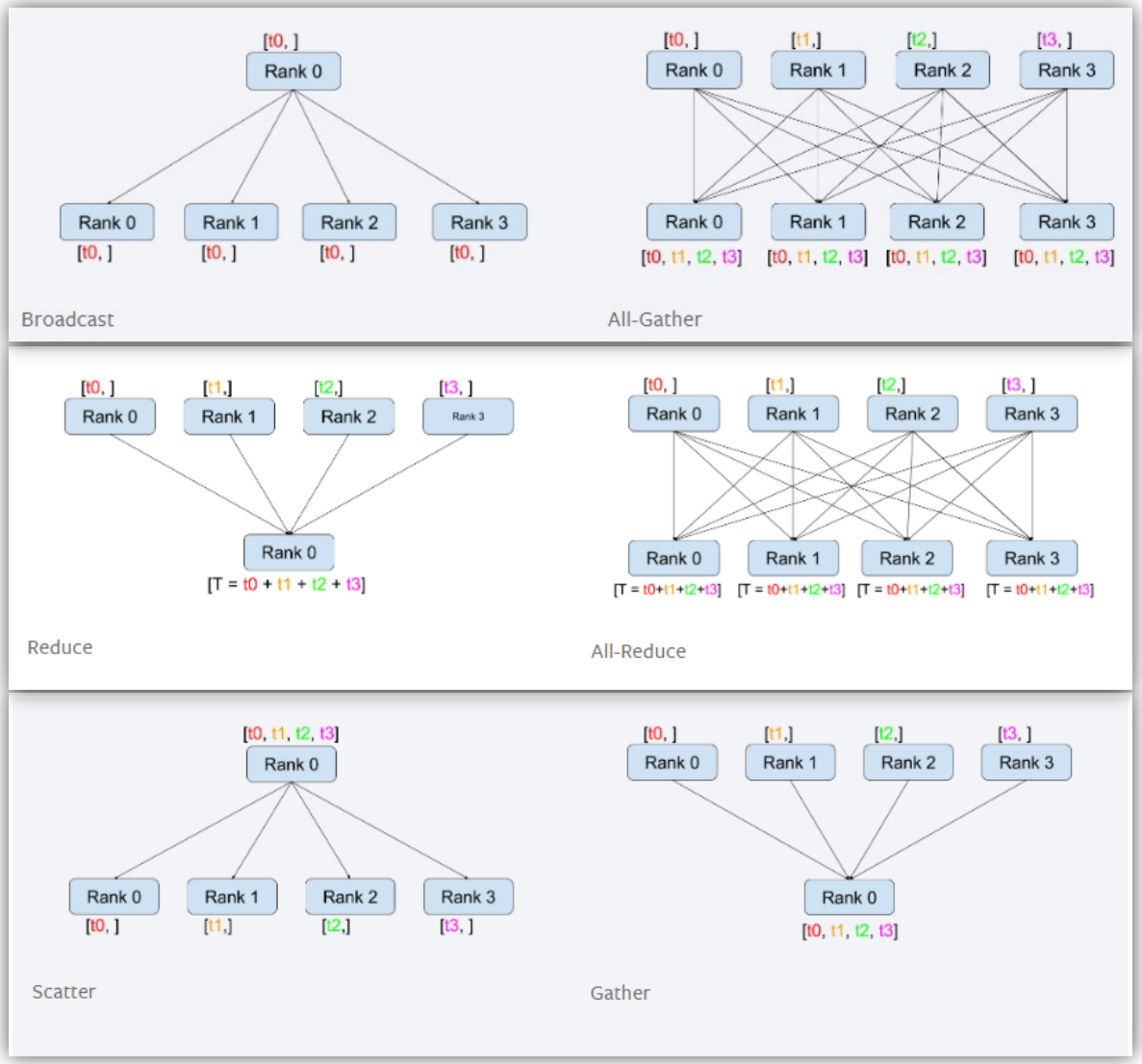
- MPI: originally for CPU, CUDA-aware MPI for NVIDIA GPU
- NCCL: the best communication performance across NVIDIA GPUs
- Gloo: a lightweight alternative for MPI and NCCL
- RCCL: for AMD GPUs

| Backend | gloo | | mpi | | nccl | |
|---|---|---|---|---|---|---|
| Device | CPU | GPU | CPU | GPU | CPU | GPU |
| send | ✓ | ✗ | ✓ | ? | ✗ | ✓ |
| recv | ✓ | ✗ | ✓ | ? | ✗ | ✓ |
| broadcast | ✓ | ✓ | ✓ | ? | ✗ | ✓ |
| all_reduce | ✓ | ✓ | ✓ | ? | ✗ | ✓ |
| reduce | ✓ | ✗ | ✓ | ? | ✗ | ✓ |
| all_gather | ✓ | ✗ | ✓ | ? | ✗ | ✓ |
| gather | ✓ | ✗ | ✓ | ? | ✗ | ✓ |
| scatter | ✓ | ✗ | ✓ | ? | ✗ | ✗ |
| reduce_scatter | ✗ | ✗ | ✗ | ✗ | ✗ | ✓ |
| all_to_all | ✗ | ✗ | ✓ | ? | ✗ | ✓ |
| barrier | ✓ | ✗ | ✓ | ? | ✗ | ✓ |

https://pytorch.org/docs/stable/distributed.html

# Collective Communications

Communications between GPUs are based on NCCL.




https://pytorch.org/tutorials/intermediate/dist_tuto.html#

# What is not covered...

- Data race: atomic, reduce
- Constant memory
- Multiple GPUs without communications: OpenMP + CUDA
- Multiple GPUs with communications: MPI + CUDA, NCCL + CUDA
- CUDA in Python: cupy
- AMD GPUs: ROCm, RCCL