Introduction to GPU and CUDA

Shaohao Chen

ORCD at MIT

Outline

❑ GPU basics

❑ CUDA programming

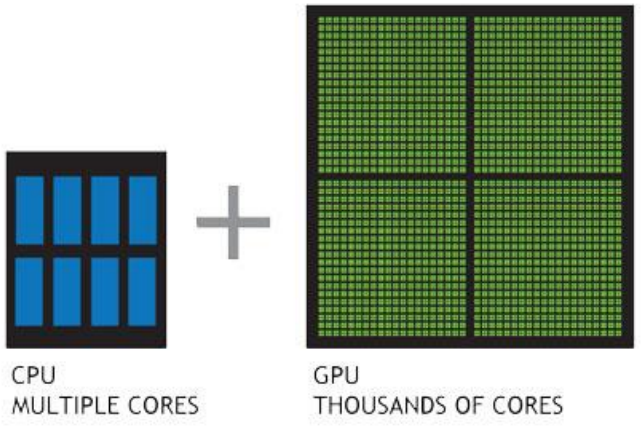
- CUDA libraries
- CUDA parallel hierarchy: grids, blocks, and threads
- Example: vector addition
- Exercise: SAXPY
- Example: matrix multiplication, GPU shared memory

❑ GPU-GPU communication

- NCCL

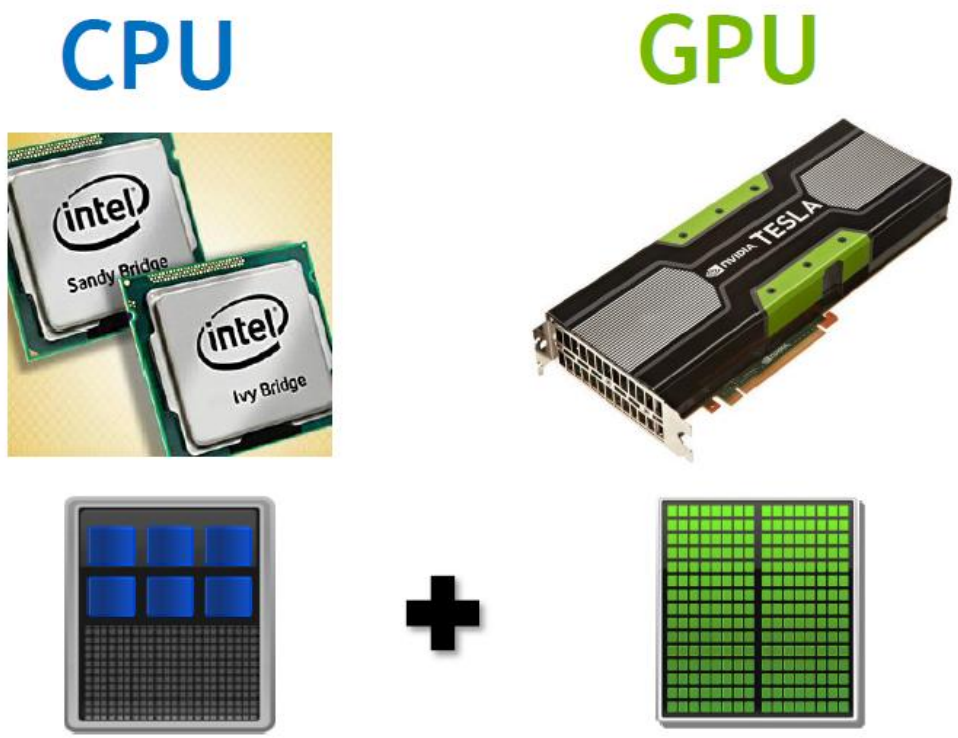
GPU and GPGPU

- 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) .
- 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 (host).
- Many computer programs can be parallelized and thus accelerated on GPU.
- Host memory *vs* GPU memory. CPU-GPU data communication.



GPU types

❑ NVIDIA GPU family

- Data Center GPU

Tesla series: K80, V100, A100, H100, H200, B200 (For all precisions)

A40, L40S (For FP32 and lower)

- Workstation GPU

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

- Desktop GPU

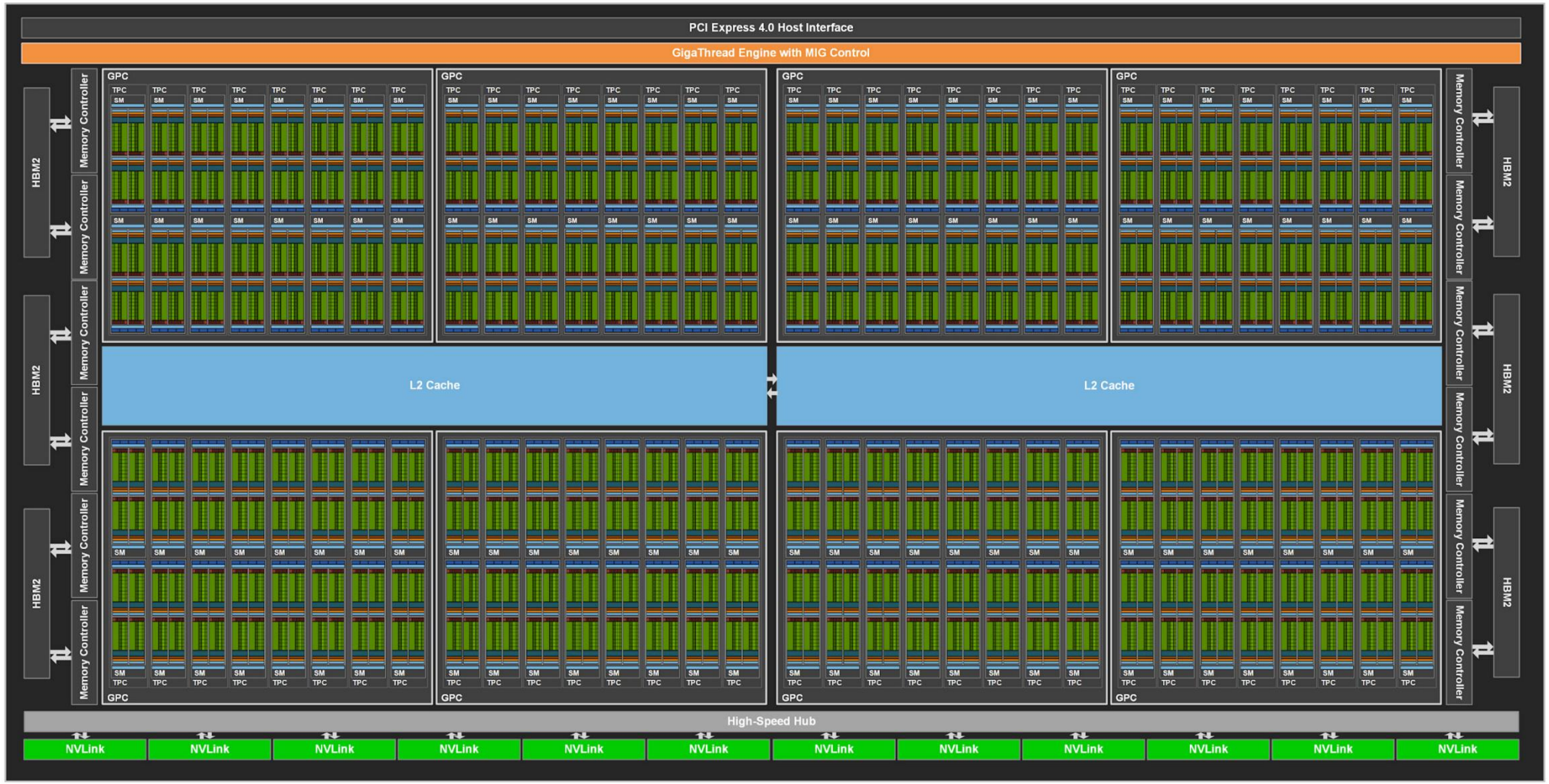
GeForce series: RTX1080, RTX2080, Titan X, RTX3090 (For FP32 and lower)

❑ AMD GPU: MI250, MI300.

❑ Intel Data Center GPU

GPU compute units (1)

- Streaming Multiprocessor (SM): 10s – 100+ SMs. Each SM can execute up to thousands of threads concurrently.



GPU compute units (2)

❑ CUDA core

- From 1000s to 16,000+ CUDA cores
- Each CUDA core can process multiple threads simultaneously and execute a floating-point and an integer operation concurrently.

❑ Tensor core

- Hundreds of tensor cores
- Specialized cores for mixed-precision matrix multiplication/addition.

Inside an SM

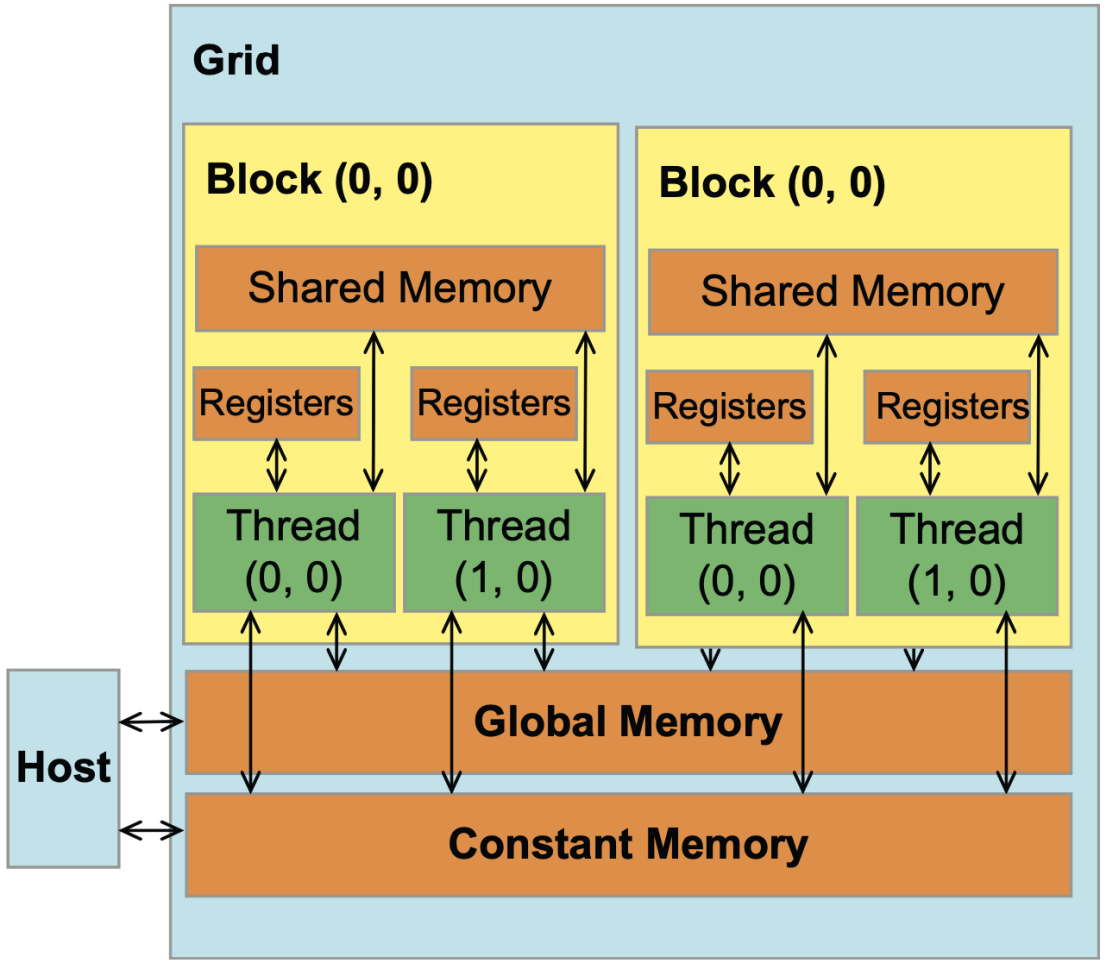
GPU Memory

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

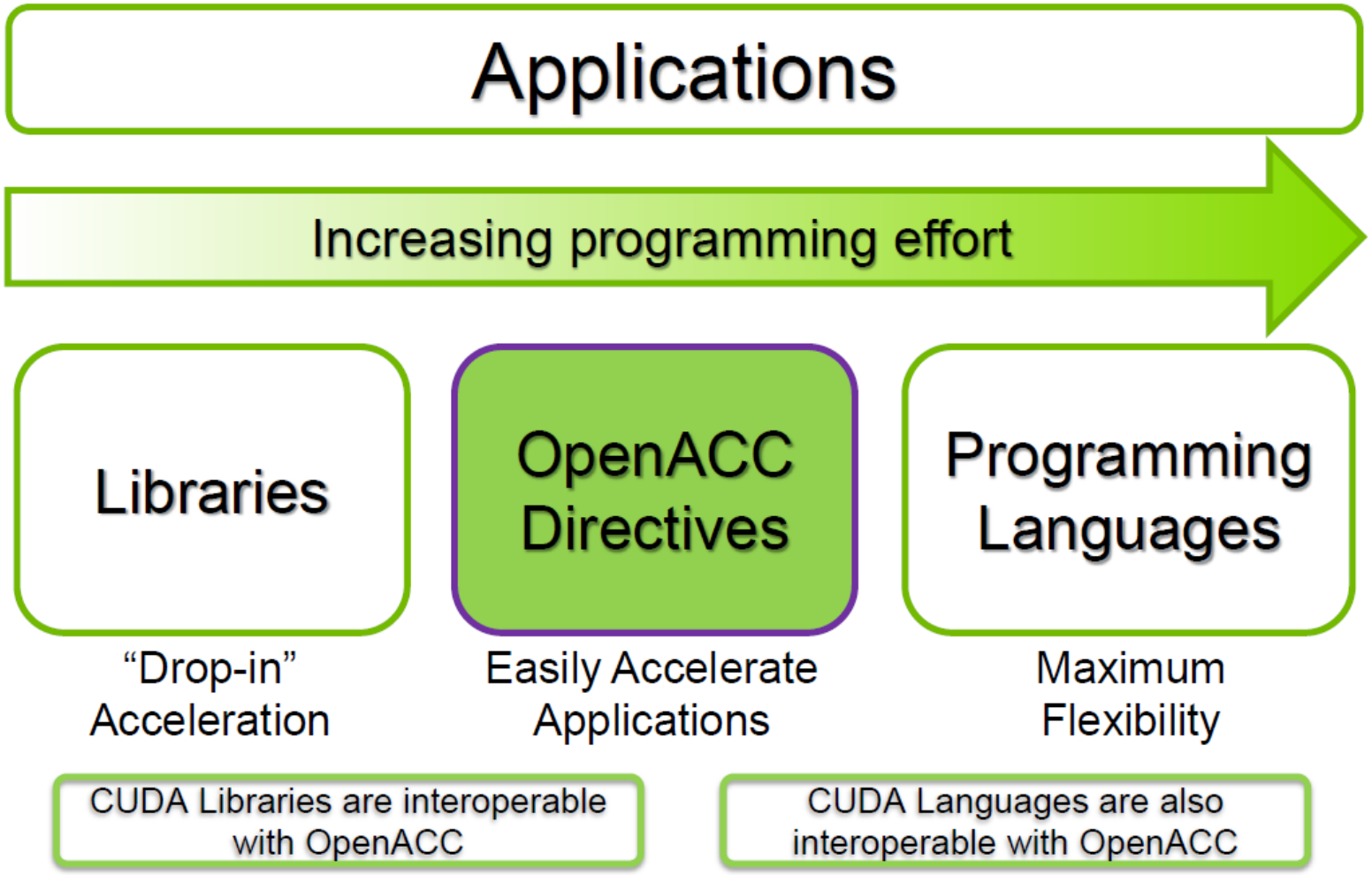
Host code can

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



- GPU has high memory bandwidth, quickly and easily accessing 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

https://developer.nvidia.com/cuda/cuda-x-libraries

CUDA parallel hierarchy

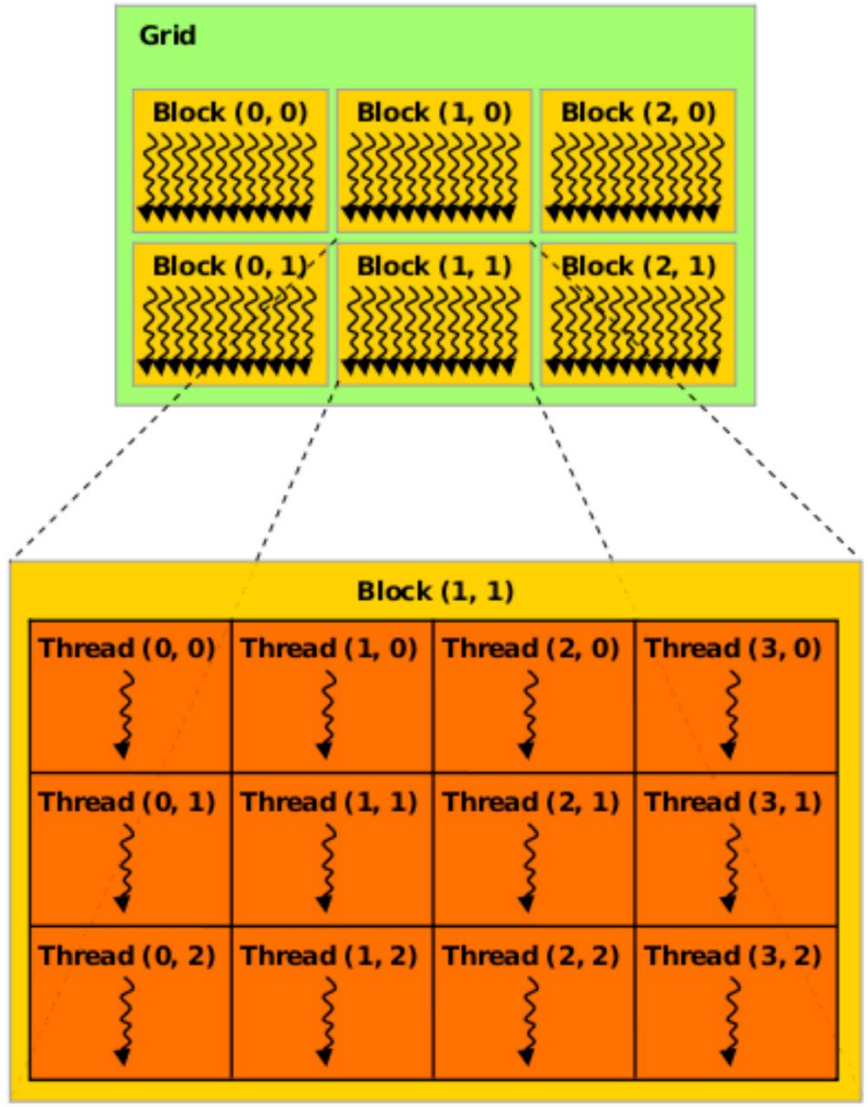
- Compute Unified Device Architecture = CUDA

- Grid

  contains many thread blocks
- 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
- Warp Size: 32
- CUDA Cores (FP32) per SM: 128
- CUDA Cores (FP32): 16,896
- Tensor Cores: 640
- Max clock rate: 1980 MHz

- Global Memory Size: 80 GB
- Shared memory per SM: 228 KB
- Global memory bandwidth:

  ~ 3.35 TB/s for the SXM5 variant

  ~ 2 TB for the PCIe variant

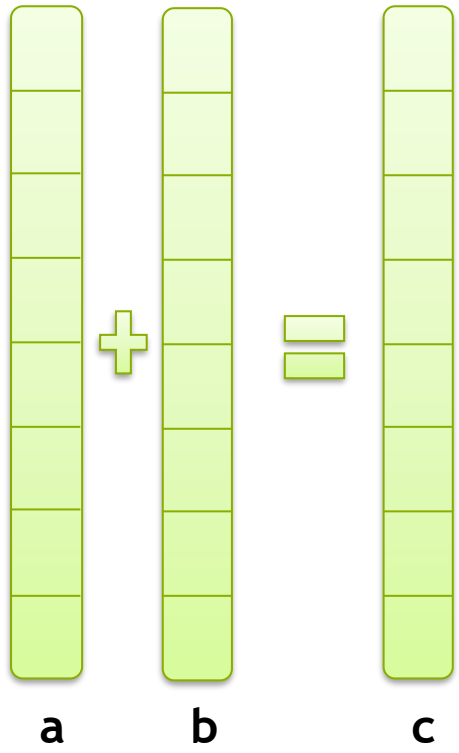
❑ Peak computing power = CUDA cores * Clock Rate * 2

- 66.9 TFLOPS for FP32
- 33.5 TFLOPS for FP64
- 989.43 TFLOPS for mixed precision (FP16-FP32) with tensor cores

Serial vector addition on CPU

- Serial code: adds 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 with 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.

CUDA program flow

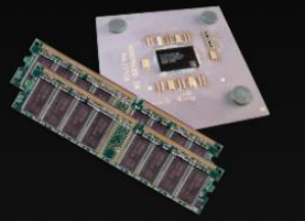
```
#include <iostream>
#include <algorithm>

using namespace std;

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

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

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

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

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

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

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

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

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

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

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

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

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

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

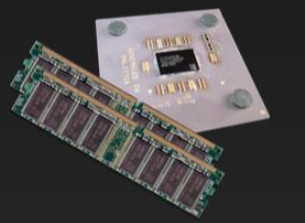
parallel fn

serial code

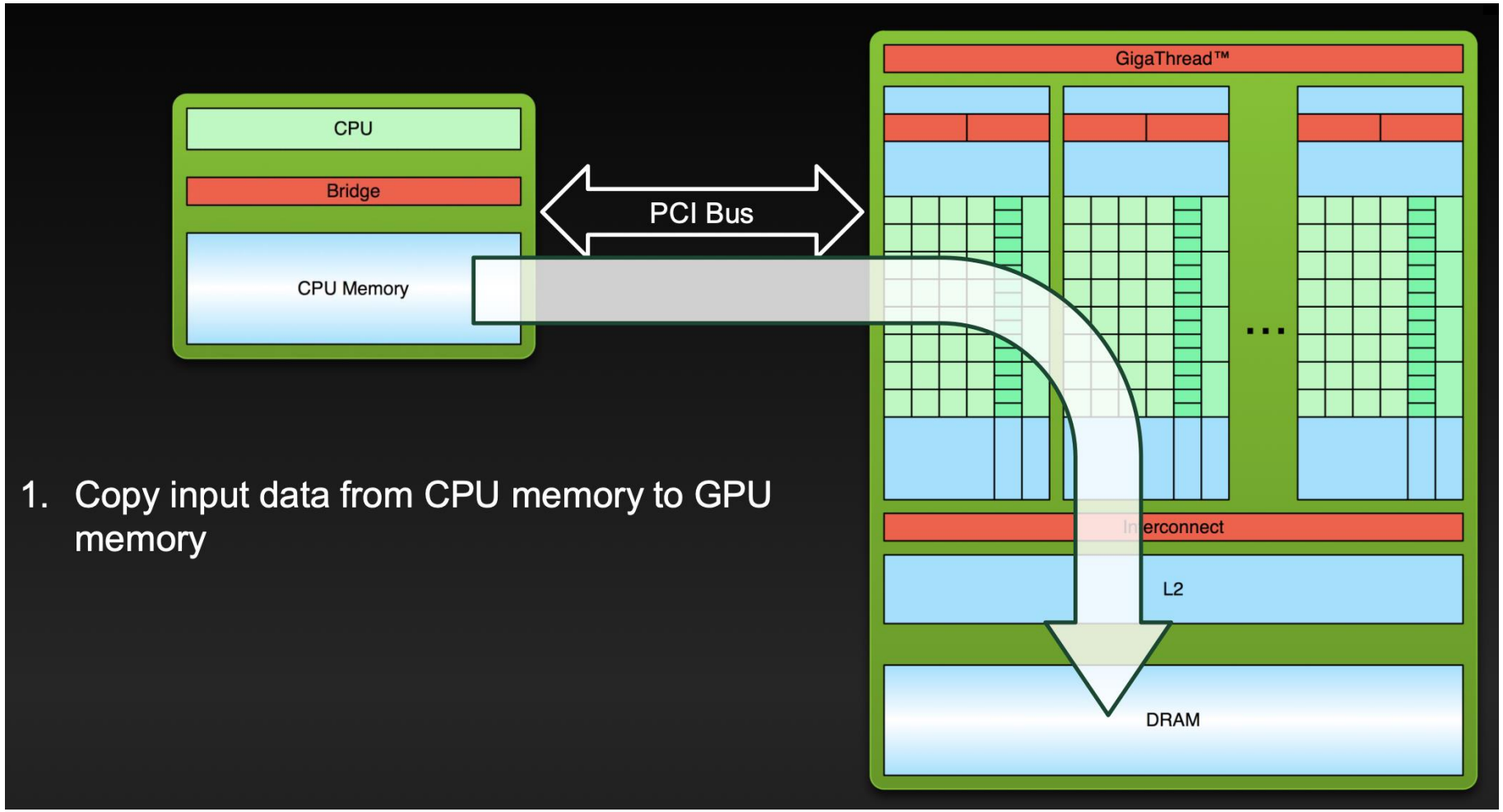
parallel code

serial code





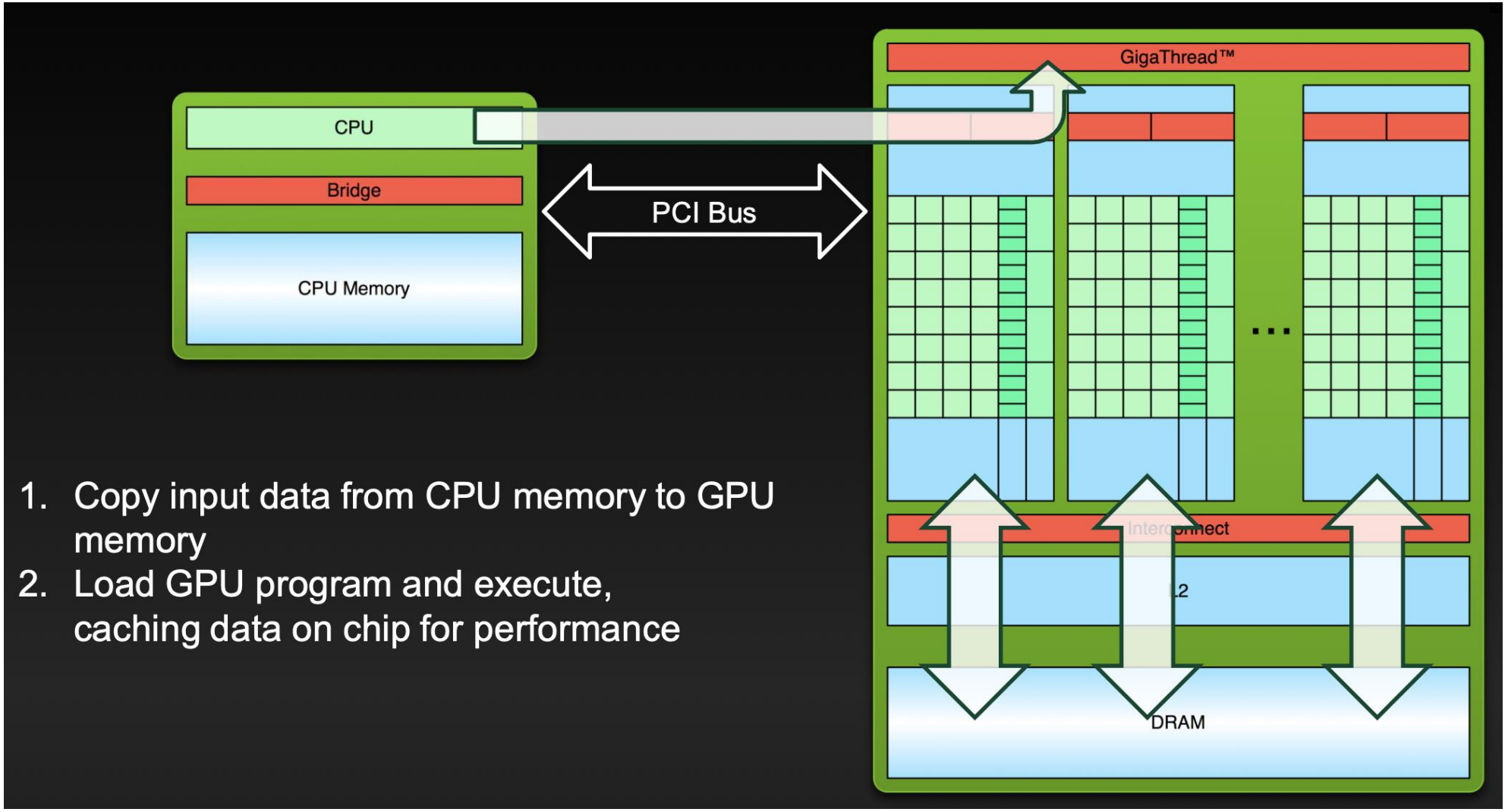
Data flow (1)



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
```

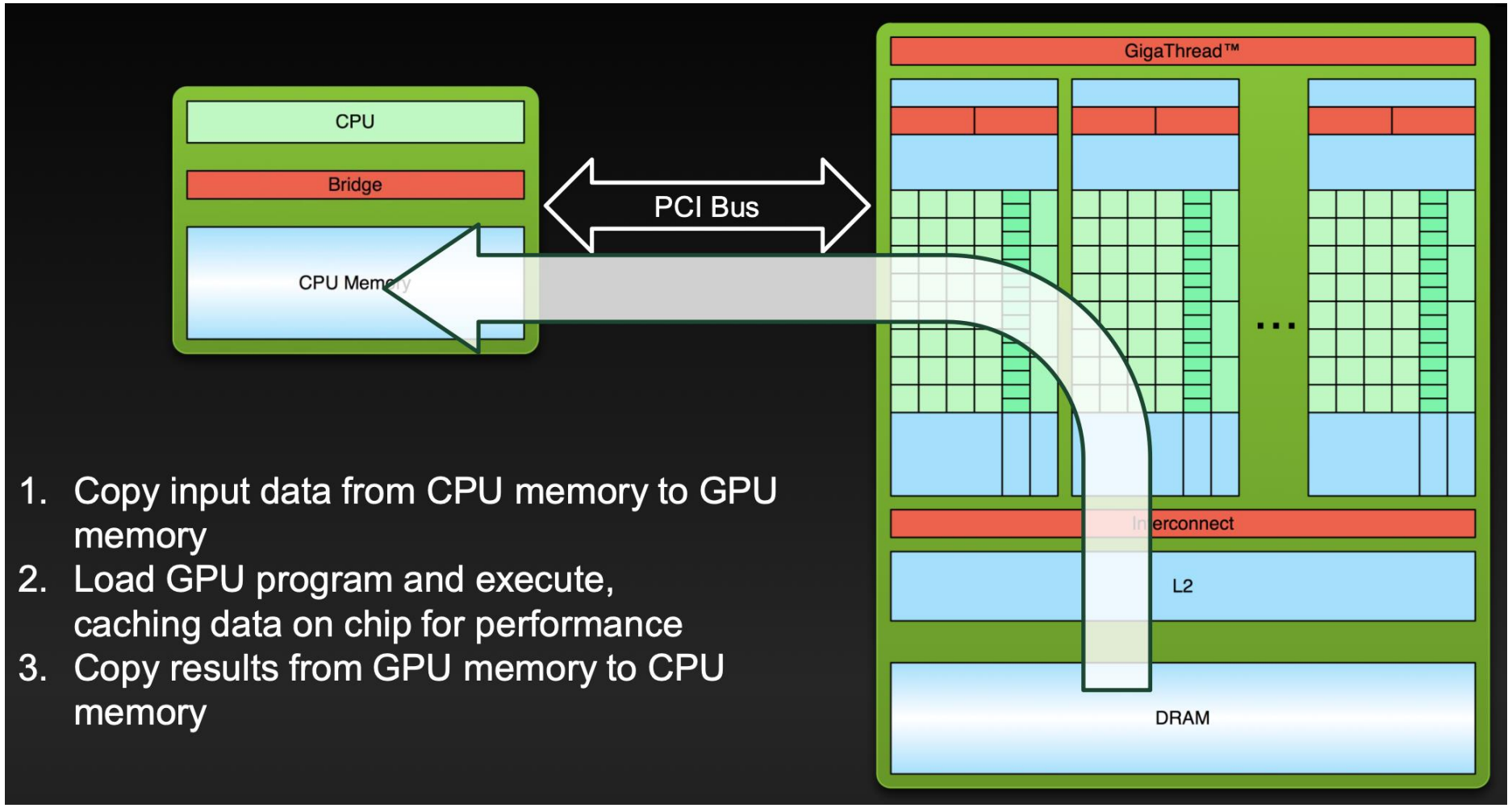
  - ➢ Threads per block must be a multiple of 32. Typical values are 256 or 512.
  - ➢ The number of blocks should be large enough to keep all SMs busy.

- 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)




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

Run a CUDA program on a GPU

- Log in Engaging

```
ssh <user>@orcd-login001.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

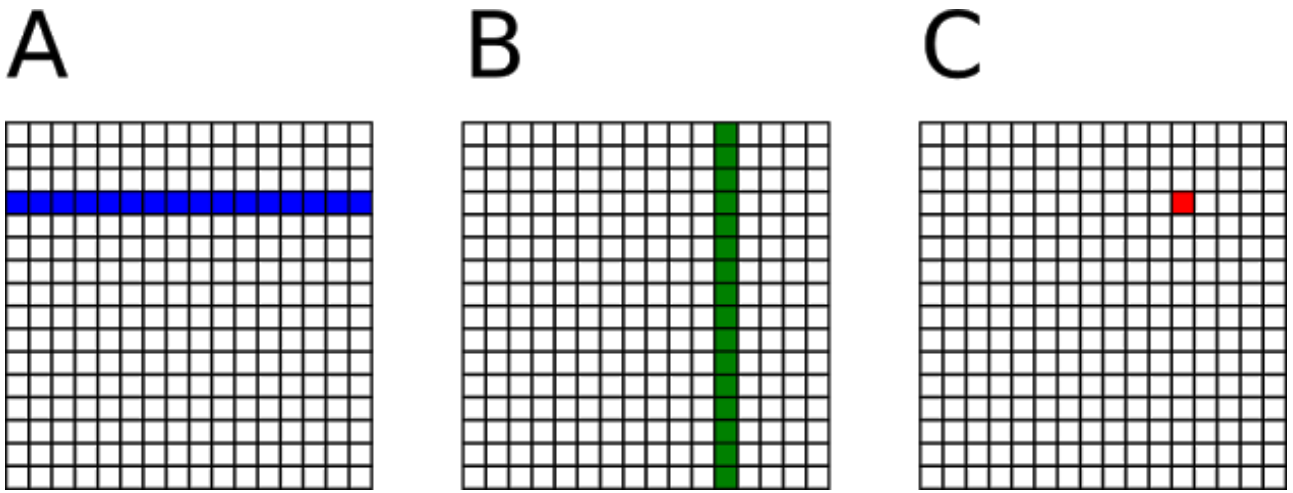
- Given a serial C code computing SAXPY, convert it to a CUDA C code.
- Compile the code and run the program on a GPU.
- Measure the run time and observe the speed up.

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.
- The time complexity is O(N^3).

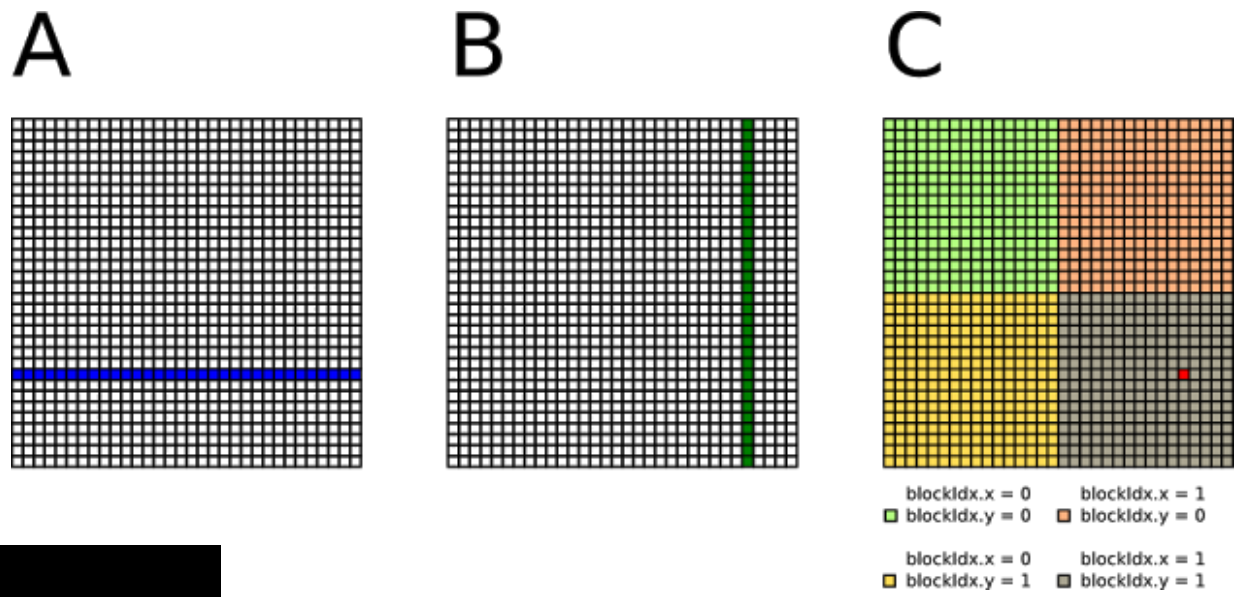



```
void multiply_matrices(float** a, float** b, float** c, int size) {

   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.

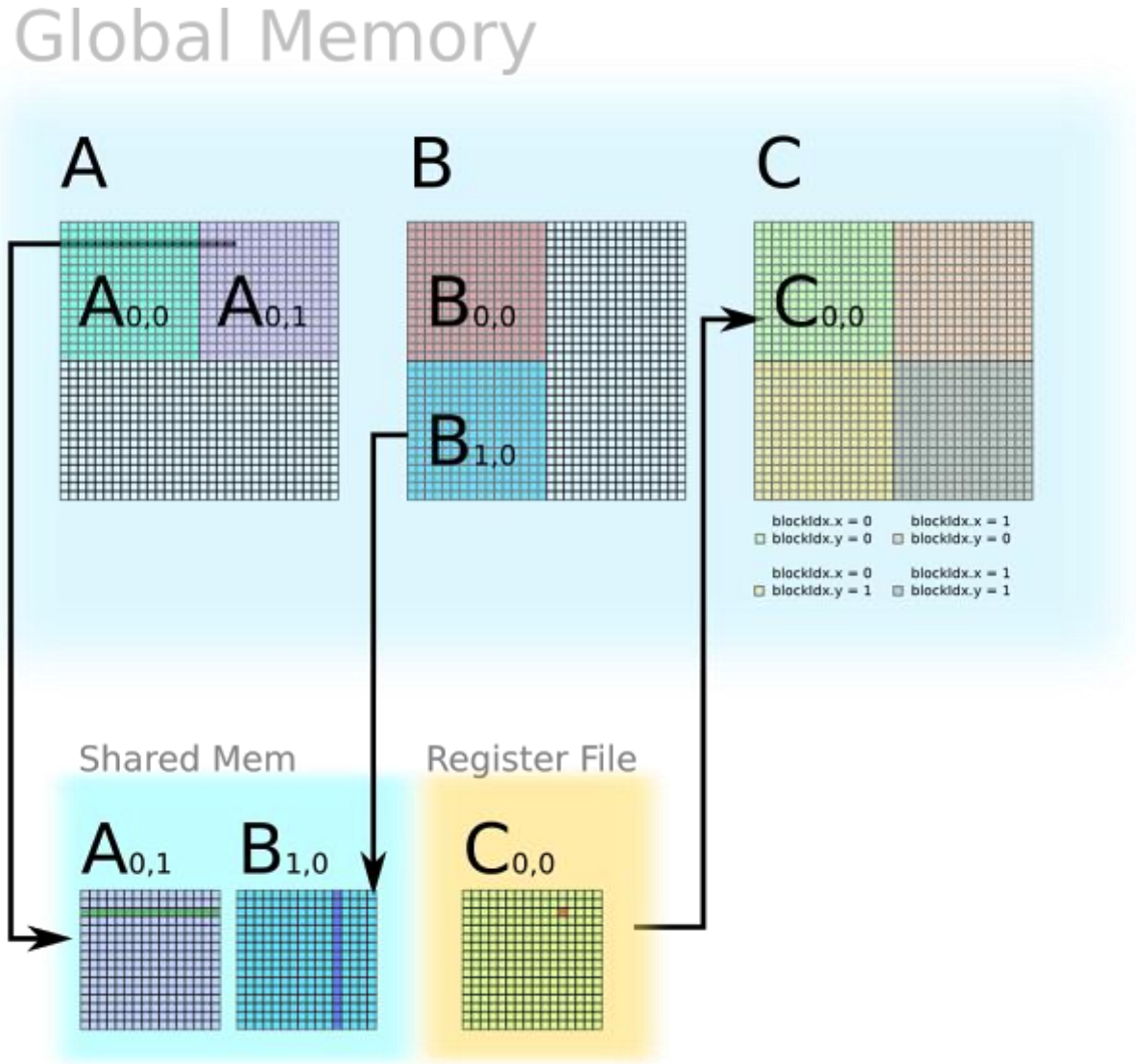



```
__global__ void matrixMulKernel(float *A, float *B, float *C, int n) {
  int row = blockIdx.y * blockDim.y + threadIdx.y;
  int col = blockIdx.x * blockDim.x + threadIdx.x;


  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.
- Threads access shared memory with low latency and high bandwidth.
- 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.
- 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

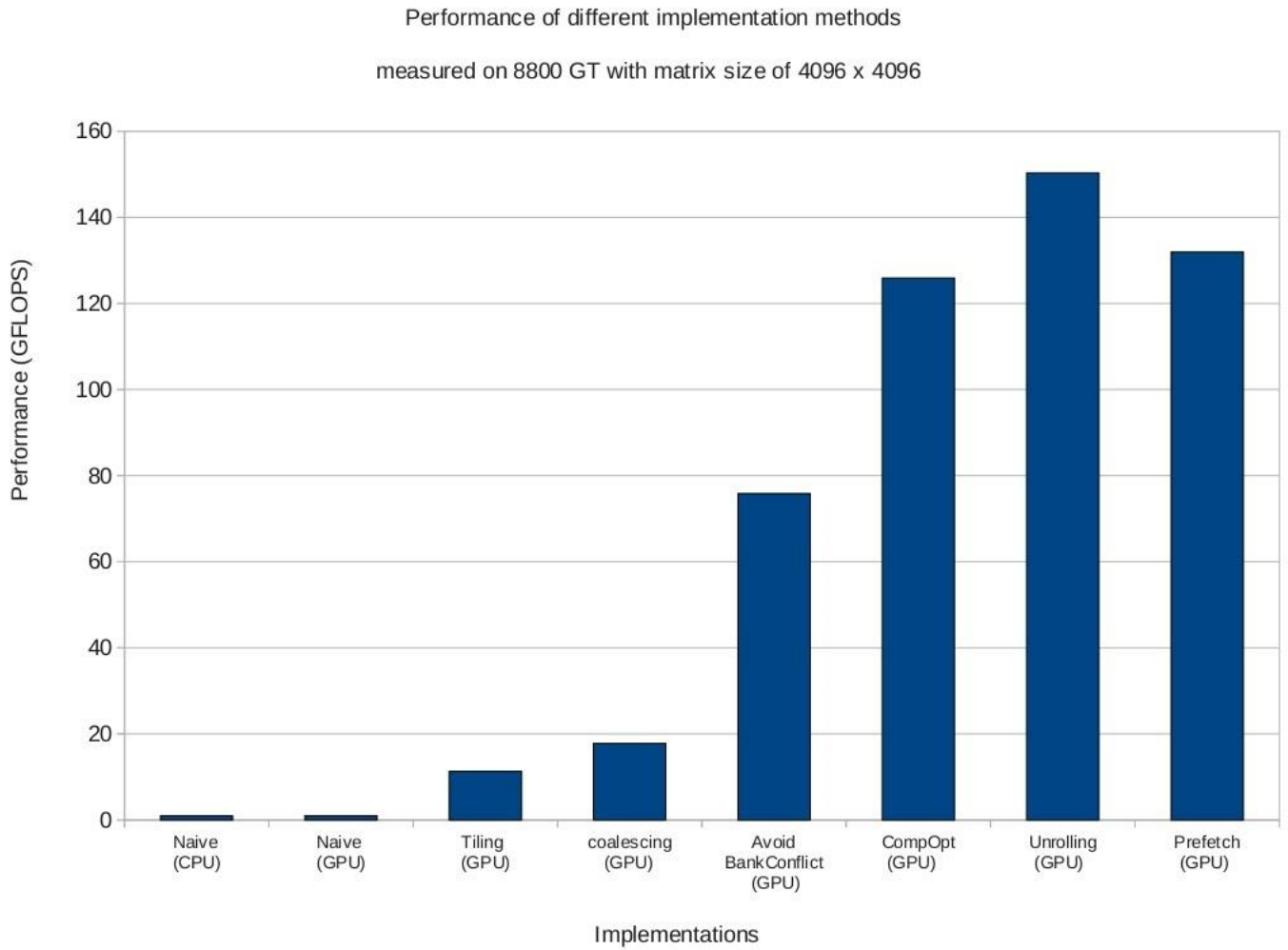
Parallel matrix multiplication with shared memory (2)

```
__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

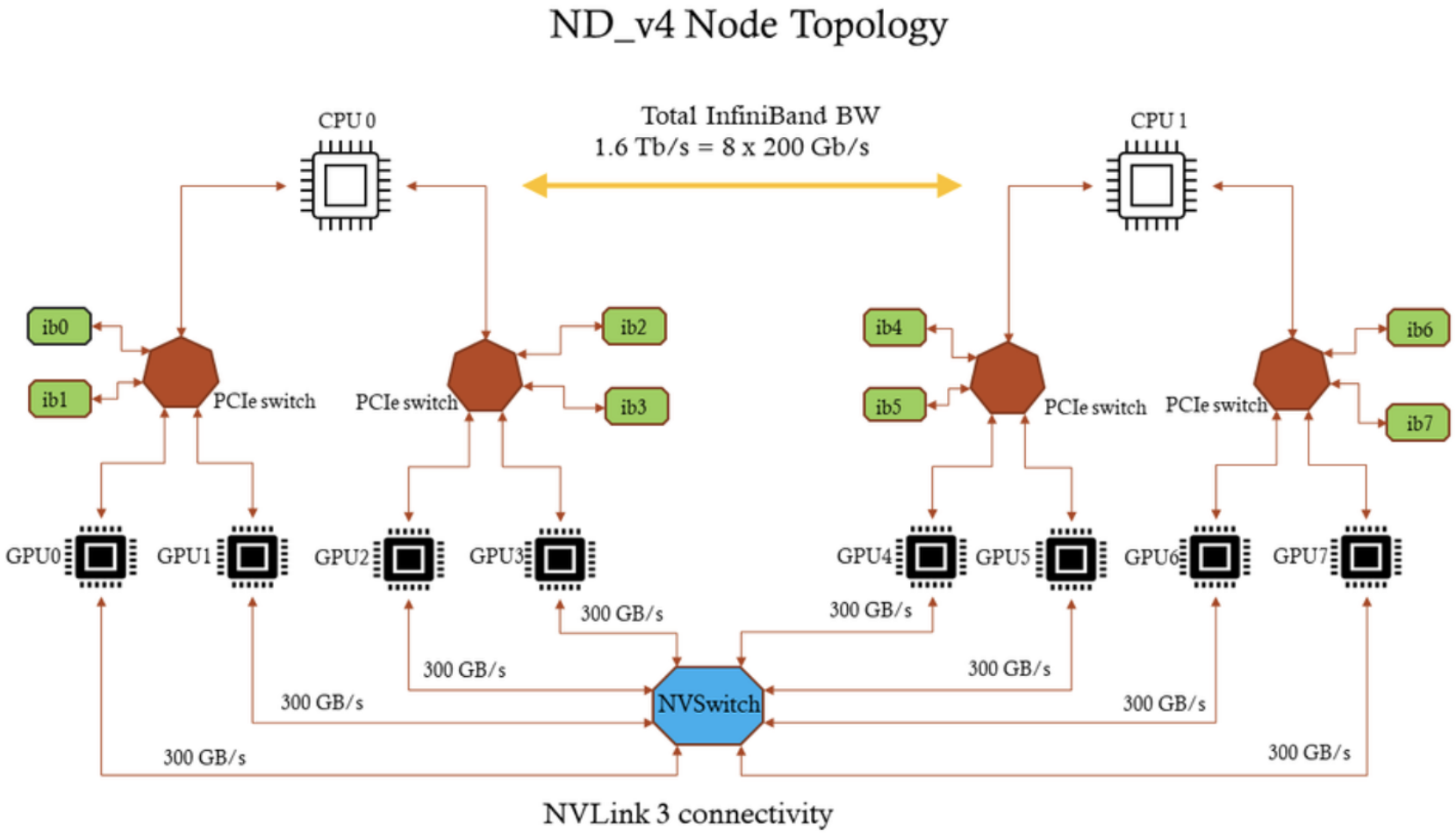
```
cublasSgemm(handle,
        CUBLAS_OP_N,
        CUBLAS_OP_N,
        n, n, n,
        &alpha,
        d_A, n,
        d_B, n,
        &beta,
        d_C, n);
```



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

GPU Server

- Multiple GPUs on a sever (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

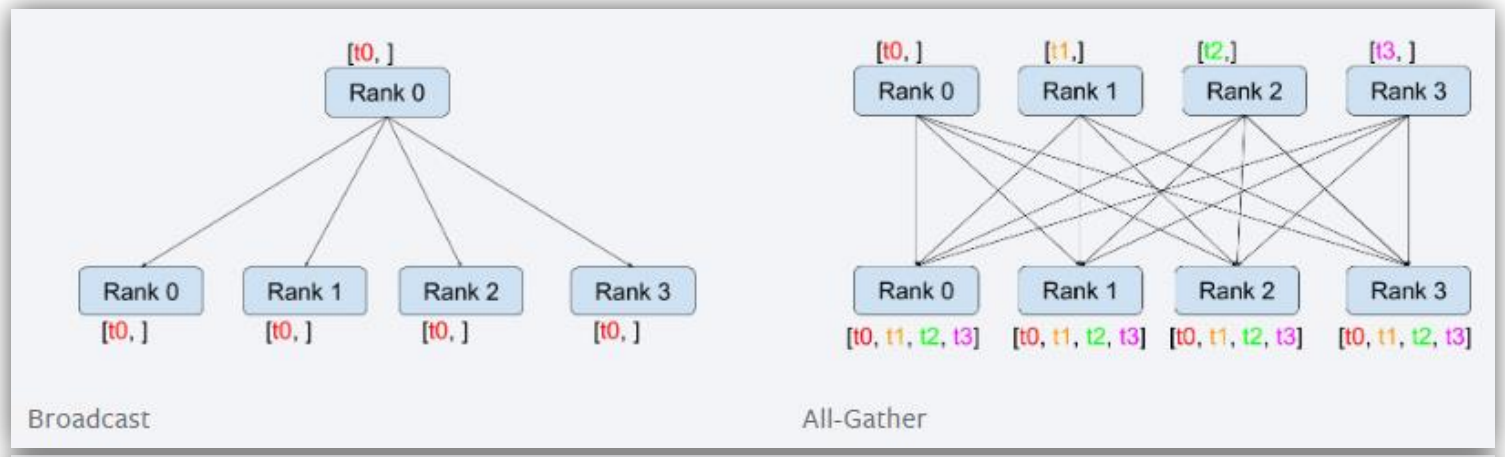
- MPI: originally for CPU, CUDA-aware MPI for NVIDIA GPU
- NCCL: the best communication performance across NVIDIA GPUs
- Gloo: a lightweight alternative for MPI and NCCL
- RCCL: for AMD GPUs

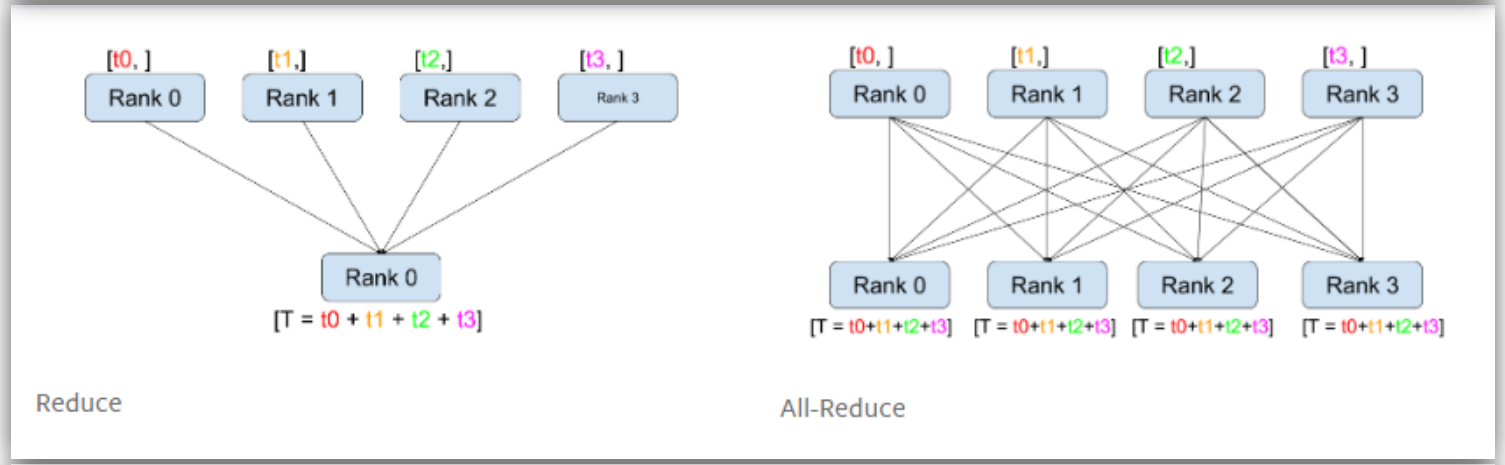
Backend	gloo		mpi		nccl	
Device	CPU	GPU	CPU	GPU	CPU	GPU
send	✓	✗	✓	?	✗	✓
recv	✓	✗	✓	?	✗	✓
broadcast	✓	✓	✓	?	✗	✓
all_reduce	✓	✓	✓	?	✗	✓
reduce	✓	✗	✓	?	✗	✓
all_gather	✓	✗	✓	?	✗	✓
gather	✓	✗	✓	?	✗	✓
scatter	✓	✗	✓	?	✗	✗
reduce_scatter	✗	✗	✗	✗	✗	✓
all_to_all	✗	✗	✓	?	✗	✓
barrier	✓	✗	✓	?	✗	✓

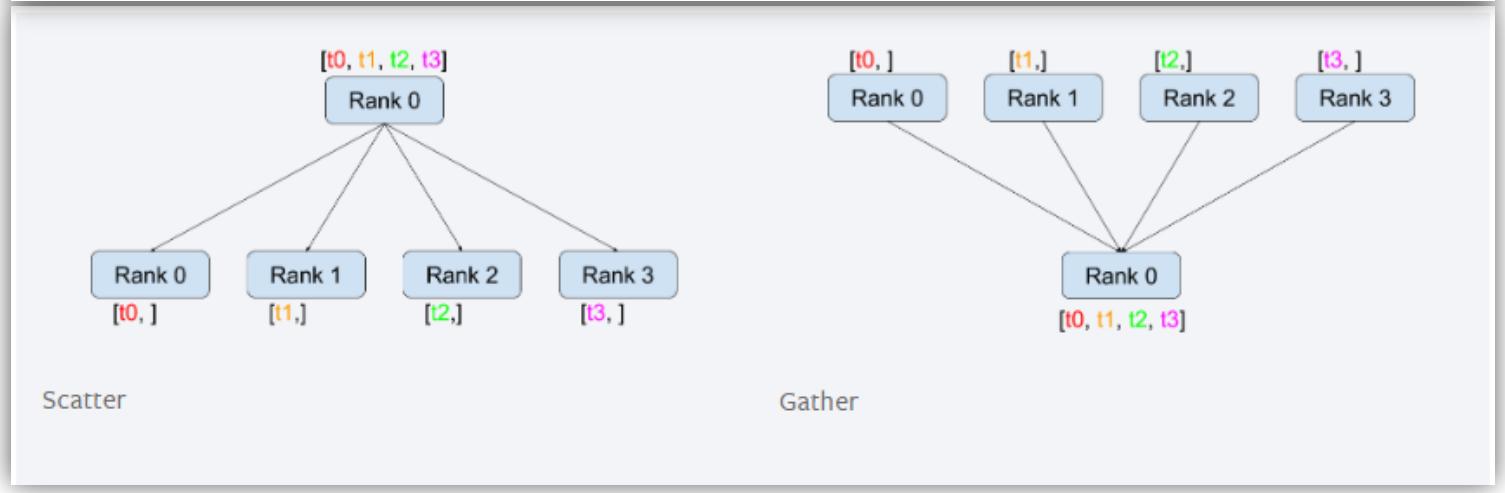
https://pytorch.org/docs/stable/distributed.html

Collective Communications

Communications between GPUs are based on NCCL.







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

What is not covered…

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