11868 LLM Systems
GPU Programming

Lei Li

Carnegie Mellon University
Language Technologies Institute

Assignment 1

- https://llmsystem.github.io/llmsystem2024spring/assets/files/11868_LLM_Systems_Assignment_1-ee1244bd2b2f8f2de8e5e9e857f6791f.pdf

- Due 2/5 midnight.

Language Model

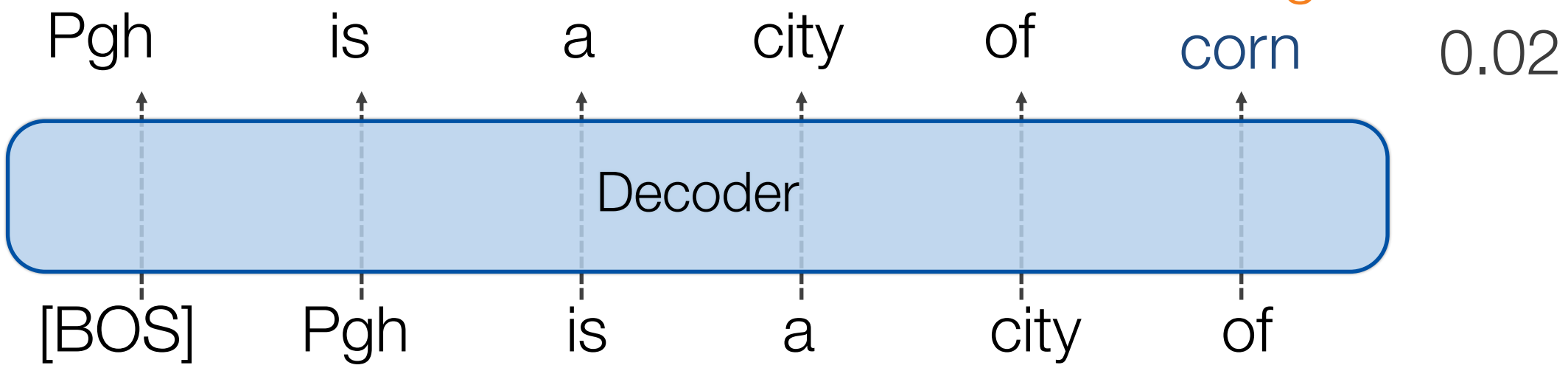
Language Model

$$P(x_{1..T}) = \prod_{t=1}^{T} P(x_{t+1}|x_{1..t})$$

$P(x_t|x_{<t})$

bridges 0.6

corn 0.02



Today's Topic

- Operators needed for Neural network
- GPU Architecture overview
- Basic CUDA operations
- Matrix/Tensor Computation on GPU

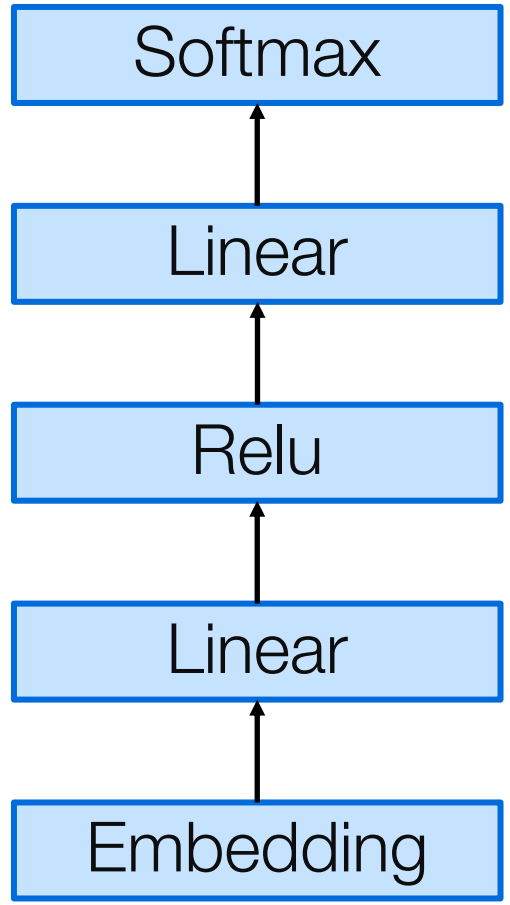
Text Classification

Classifying the sentiment of online movie reviews. (Positive, negative, neutral)

Spider-Man is an almost-perfect extension of the experience of reading comic-book adventures.	👍
The acting is decent, casting is good.	👍
It was a boring! It was a waste of a movie to even be made. It should have been called a family reunion.	👎

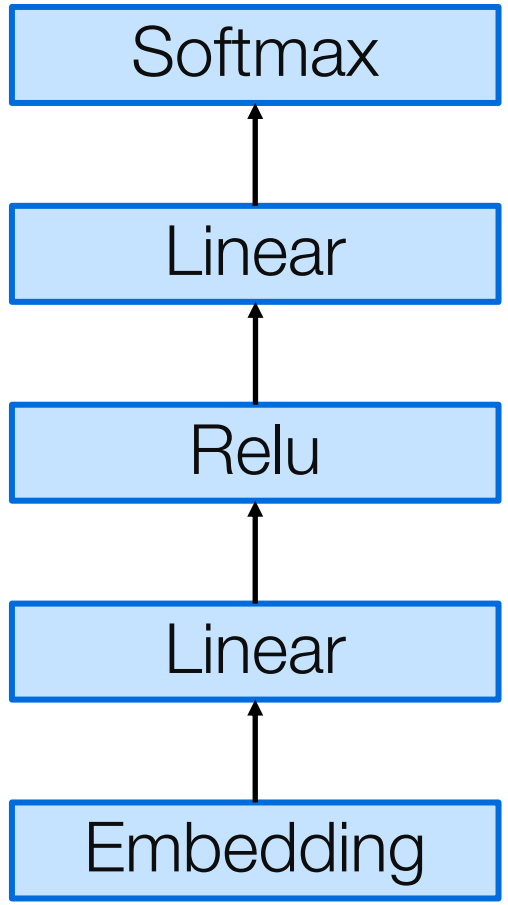
A Simple Feedforward Neural Network

- Layers in FFN
  - Embedding (lookup table)
  - Linear
  - Relu
  - Softmax



Low-level Computation Operators

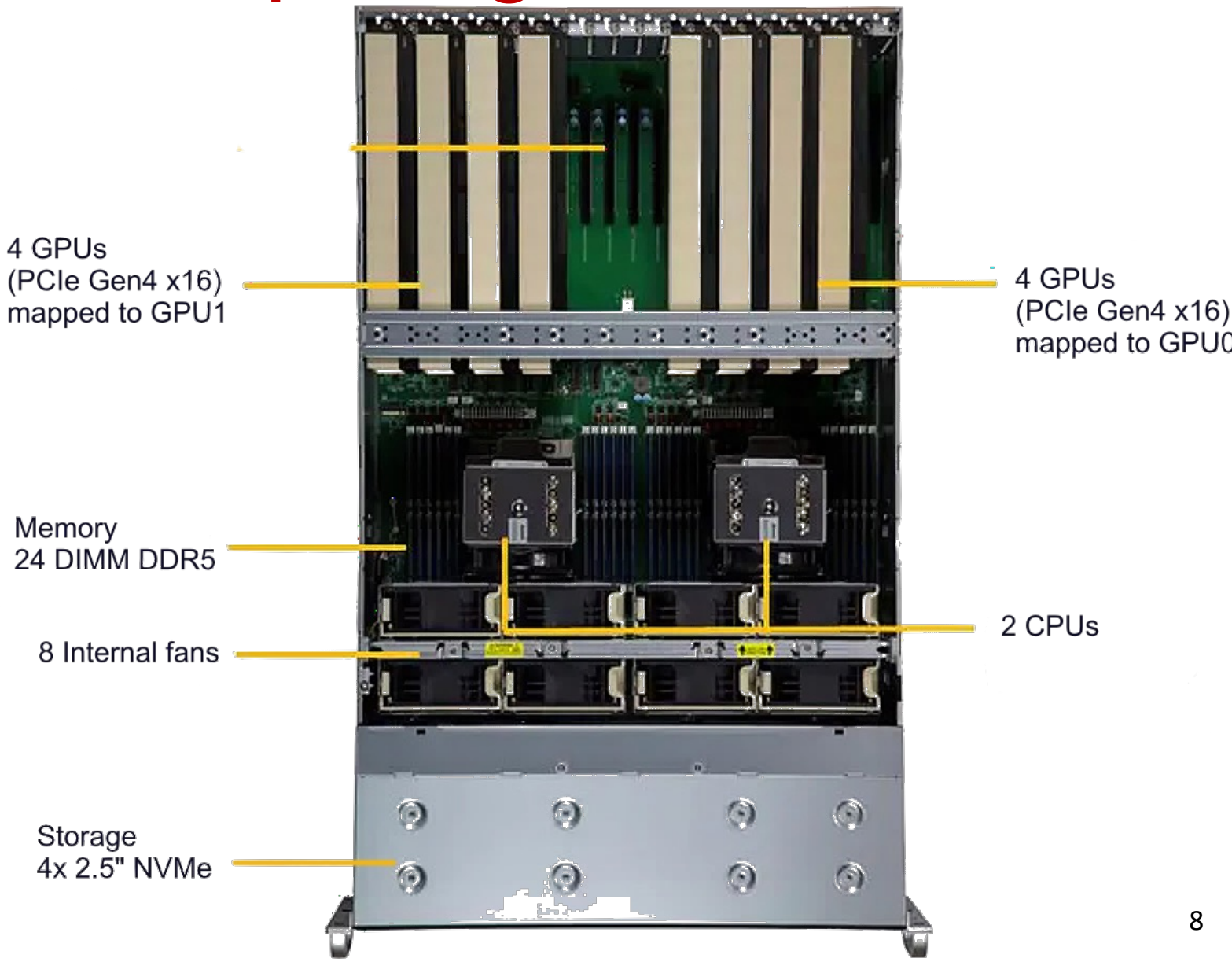
- Matrix multiplication
- Element-wise ops (add, scale, relu)
- Reduce ops (sum, avg)
- Efficient computation requires GPU



A Modern Computing Server

A Sample Config (my lab)

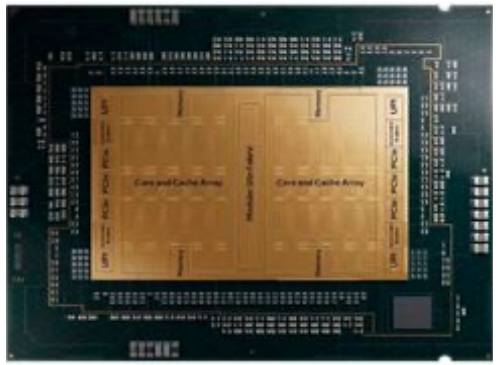
- CX4860s-EK9 4U server
- 2x AMD EPYC 9354 CPU
- 16x 64GB DDR5 mem
- 4x Intel D7 P5520 15.36TB Gen4 NVMe SSD
- 8x Nvidia A6000 48GB
- 4x 2slot NVLink



Computing Devices

CPU

AMD EPYC 9754
128cores
256threads
2.25GHz
256MB L3



Intel Xeon 8593Q
64cores
128threads
2.2GHz
320MB L3

GPU

Nvidia A6000
10,752 cores
48GB
38.7 TFLOPS

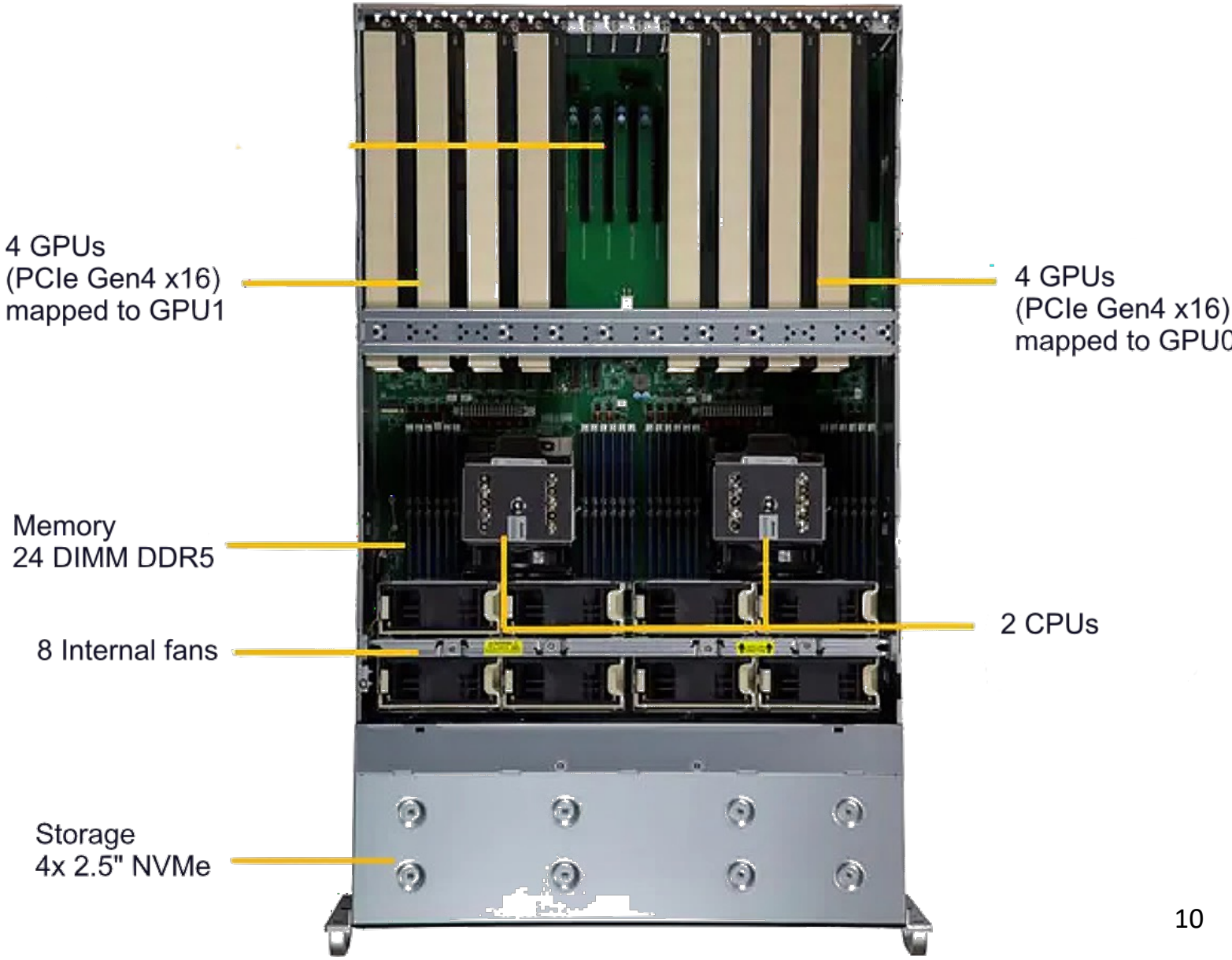
FPGA

More powerful than the #1 supercomputer in 2001

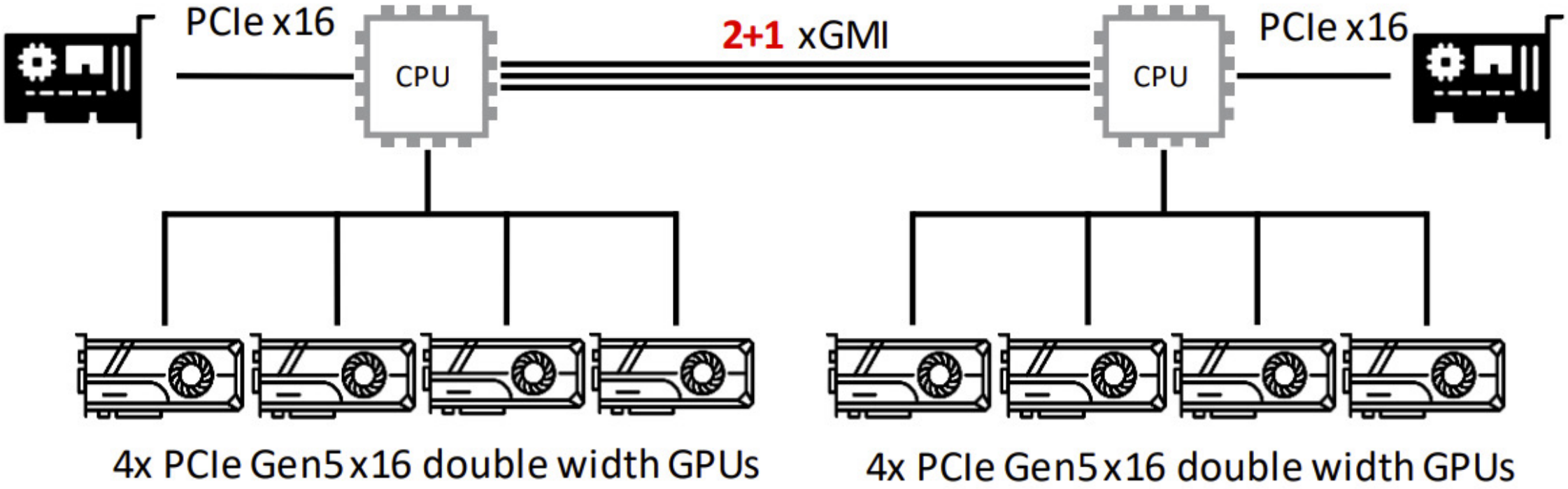
Communication

GPU-to-GPU

- NVLink 112.5 GB/s
- PCIe Gen4 32 GB/s (16x2)



Modern Computing Server Architecture



GPU Architecture

84 SMs (Streaming Multiprocessors)
12 memory controller (32bit ea.), total 384bit
6MB L2 cache

Streaming Multiprocessor

4 partitions per SM, each with 32 cores ➔ 32 threads each
128 cores per SM
64KB register per partition
128KB shared L1 cache per SM

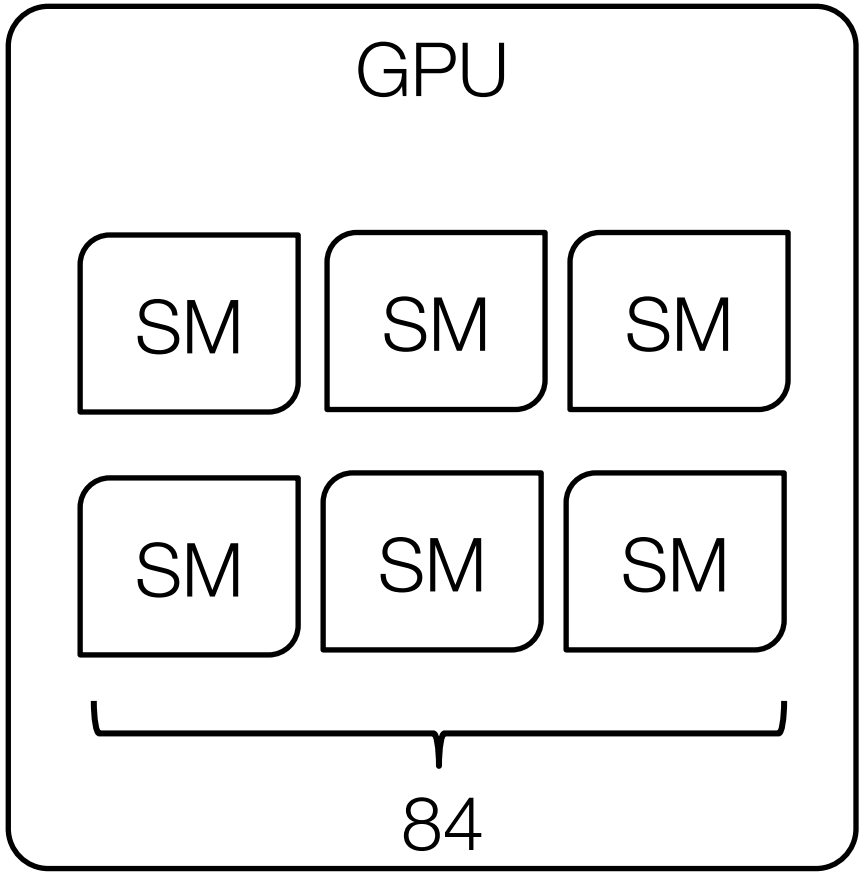
128 FP32 operations in one cycle

CPU vs. GPU

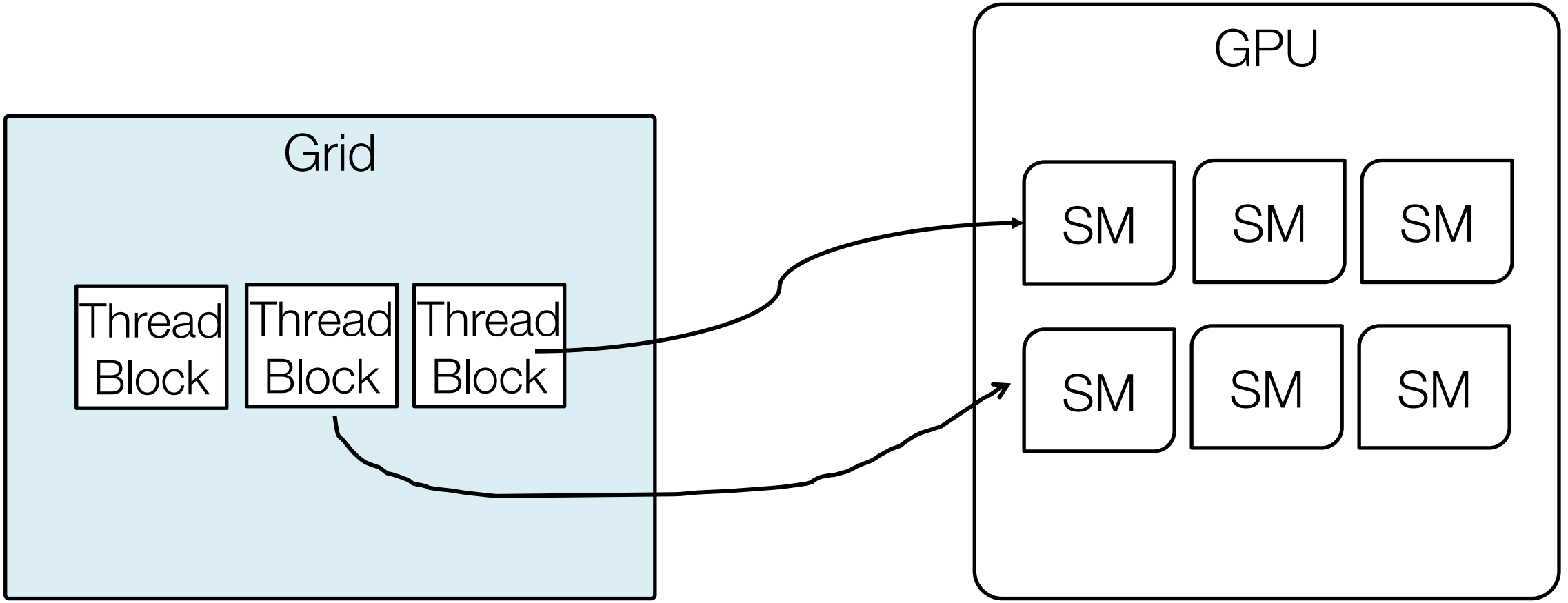
	CPU	GPU
num. threads	256	10752
clock	2.25 GHz	1.8 GHz
compute	576 GFlops	38.7 TFlops

SIMT Execution on GPU

- Threads are grouped into Thread Blocks
- Thread Blocks are grouped into Grid
- Kernel executed as Grid of Blocks of Threads



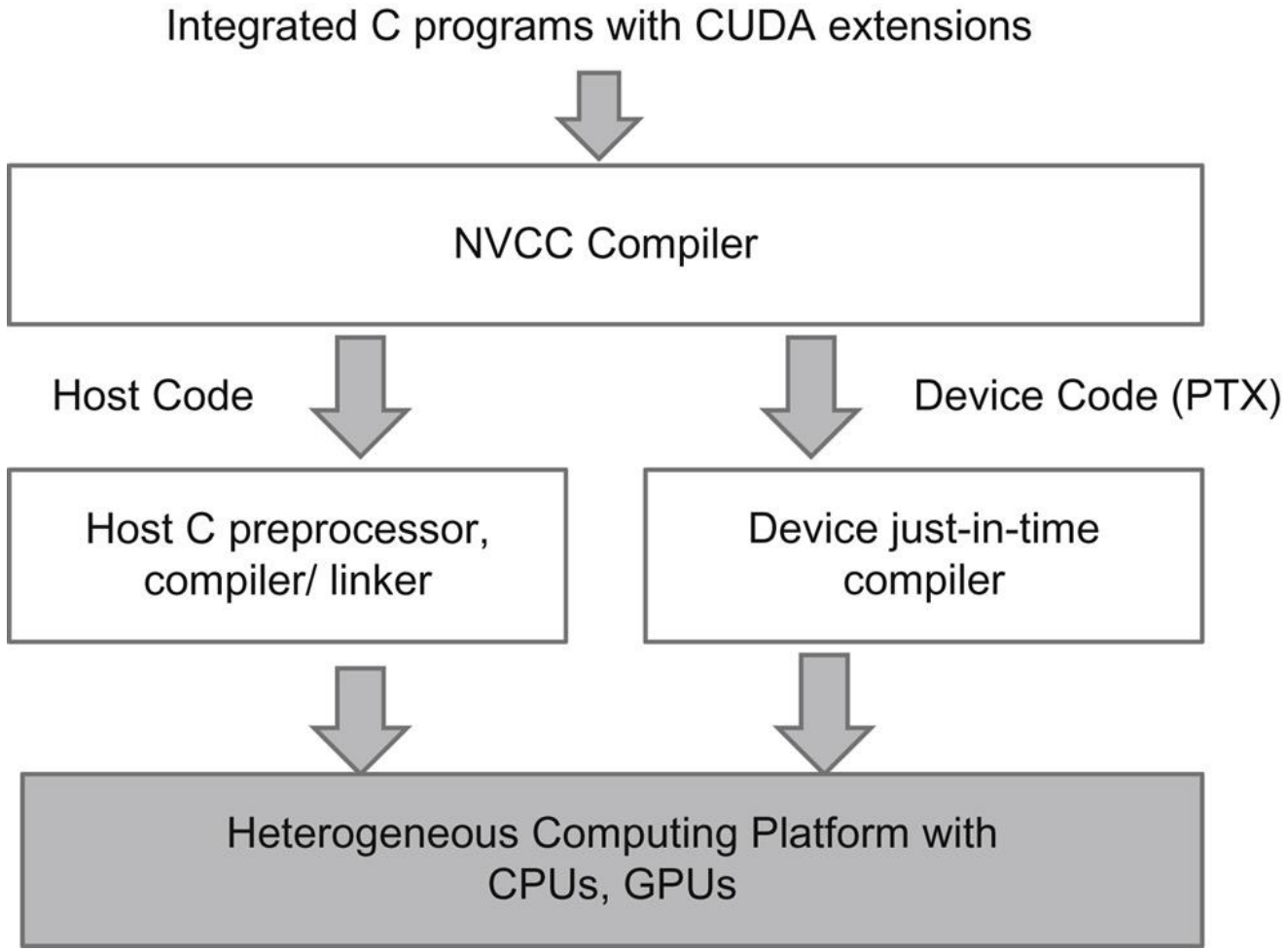
SIMT Execution on GPU



GPU Programming Model

- CPU – host
  - Run normal program (C++)
- GPU – device
  - Run cuda kernel code
- CUDA: one part runs on CPU, one part runs on GPU
- Needs to move data between system memory and GPU memory

Compiling CUDA Code

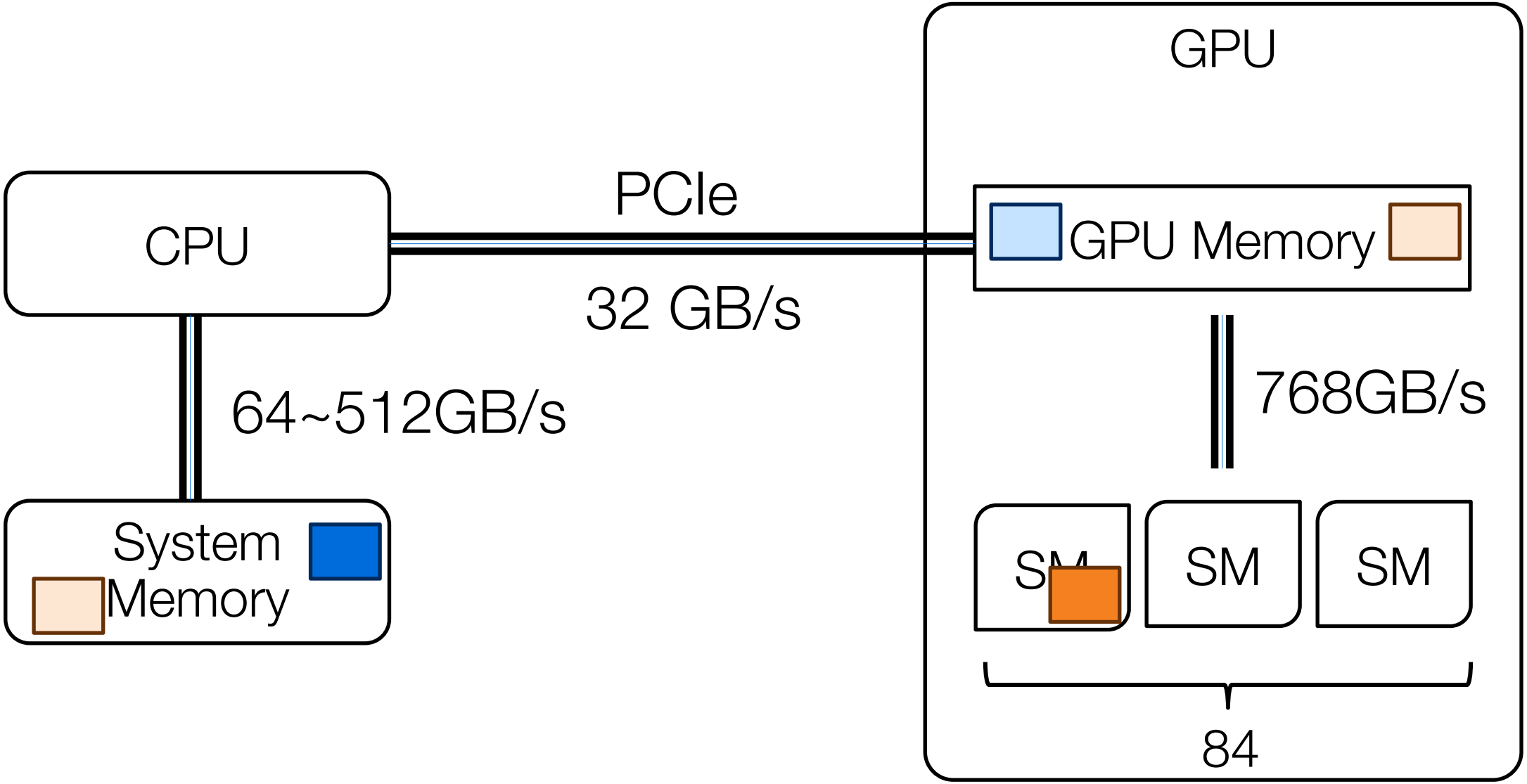


nvcc -o output.so --shared src.cu -Xcompiler -fPIC

CUDA Operations

- CPU allocates GPU memory: `cudaMalloc`
- CPU copies data to GPU memory (host to device): cudaMemcpy
- CPU launches GPU kernels
- CPU copies results from GPU (device to host): cudaMemcpy

CPU-GPU Data Movement



CUDA Kernel

- Each kernel is a function that runs on GPU
- Program itself is serial
- Can simultaneously run many (10k) threads at the same time
- Using thread index to compute on right portion of data

Running GPU kernel

- CPU invokes kernel grid
- Thread blocks in grid distributed to SMs
- Execute concurrently
  - Each SM runs multiple thread blocks
  - Each core runs one thread from one thread block

Code Example

Define kernel function, `__global__`

```
__global__ void VecAddKernel(int* A, int* B, int* C,
int n) {
    int i = blockDim.x * blockIdx.x + threadIdx.x;
    if (i < n) {
        C[i] = A[i] + B[i];
    }
}

int main() {
    VecAddKernel<<<1, N>>>(A, B, C, N);
}
```

Declaration of Host/Device function

- Both host and device code in same `.cu` file
- Indicate where the code will run

keyword	call on	execute on
`__global__`	host (cpu)	device (gpu)
`__device__`	device (gpu)	device (gpu)
`__host__`	host	host

Code Example

Allocating memory in GPU

```
int *dA;
cudaMalloc(&dA, n * sizeof(int));

float *dB;
cudaMalloc(&dA, n * sizeof(float));
```

Code Example

Copy data from cpu to gpu, gpu to cpu

```
cudaMemcpy(dA, Acpu, n * sizeof(int),
   cudaMemcpyHostToDevice);

cudaMemcpy(Ccpu, dC, n * sizeof(int),
   cudaMemcpyDeviceToHost);
```

Code Example

Running kernels on GPU

```
// n: the size of the vector
int threads_per_block = 256;
int num_blocks = (n + threads_per_block - 1) /
                        threads_per_block;
VecAddKernel<<<num_blocks, threads_per_block>>>(dA,
dB, dC, n);
```

Calling Kernel at Runtime

- Host program specifies grid-block-threads configurations for kernel at run time

  `<<<Dg, Db>>>`

- `Dg` and `Db` are either `dim3` or `int`

- `Dg`: size of grid (num. of blocks)
  - `Dg.x * Dg.y * Dg.z` is num. of blocks

- `Db`: size of block
  - `Db.x * Db.y * Db.z` is num. of threads per block, <=1024)

Device Runtime Variables

- Host launches kernels on a gpu device
- Each kernel thread needs to know which thread it is running, i.e. threadId
- Compiler generates build-in variables, with x, y, z fields

`gridDim`	`dim3`	dimensions of grid
`blockIdx`	`uint3`	index of block within grid
`blockDim`	`dim3`	dimensions of block
`threadIdx`	`uint3`	index of thread within block

```
__global__ void MatAddKernel(float* A, float* B,
float* C, int N) {
    int i = blockIdx.x * blockDim.x + threadIdx.x;
    int j = blockIdx.y * blockDim.y + threadIdx.y;
    C[i * N + j] = A[i * N + j] + B[i * N + j];
}

int main() {
    int N = 32;
    dim3 threads_per_block(N, N);
    int num_blocks = 1;
    MatAddKernel<<<num_blocks, threads_per_block>>>(dA,
    dB, dC, N);
}
```

```
int main() {
   dim3 threads_per_block(2, 4, 8);
   dim3 blocks_per_grid(2, 3, 4);
   fullKernel<<<blocks_per_grid,
   threads_per_block>>>(some_input, some_output);
}
```

24 blocks per grid

64 threads per block

1536 threads in total

can you run this simultaneously on A6000?

```
__global__ void fullKernel(float* din, float* dout) {
   int block_id = blockIdx.x + blockIdx.y * gridDim.x
      + blockIdx.z * gridDim.x * gridDim.y;
   int block_offset = block_id * blockDim.x *
      blockDim.y * blockDim.z;
   int thread_offset = threadIdx.x
      + threadIdx.y *  blockDim.x
      + threadIdx.z * blockDim.x * blockDim.y;
   int tid = block_offset + thread_offset;
   dout[tid] = func(din[tid]);
}
```

Vector Addition

```
void VecAddCUDA(int* Acpu, int* Bcpu, int* Ccpu, int n) {
    int *dA, *dB, *dC;
    cudaMalloc(&dA, n * sizeof(int));
    cudaMalloc(&dB, n * sizeof(int));
    cudaMalloc(&dC, n * sizeof(int));
    cudaMemcpy(dA, Acpu, n * sizeof(int), cudaMemcpyHostToDevice);
    cudaMemcpy(dB, Bcpu, n * sizeof(int), cudaMemcpyHostToDevice);
    int threads_per_block = 256;
    int num_blocks = (n + threads_per_block - 1) / threads_per_block;
    VecAddKernel<<<num_blocks, threads_per_block>>>(dA, dB, dC, n);
    cudaMemcpy(Ccpu, dC, n * sizeof(int), cudaMemcpyDeviceToHost);
    cudaFree(dA);
    cudaFree(dB);
    cudaFree(dC);
}
```

Matrix Addition

```
void MatAddCUDA(int* Acpu, int* Bcpu, int* Ccpu, int n) {
    int *dA, *dB, *dC;
    cudaMalloc(&dA, n * n * sizeof(int));
    cudaMalloc(&dB, n * n * sizeof(int));
    cudaMalloc(&dC, n * n * sizeof(int));
    cudaMemcpy(dA, Acpu, n * n * sizeof(int), cudaMemcpyHostToDevice);
    cudaMemcpy(dB, Bcpu, n * n * sizeof(int), cudaMemcpyHostToDevice);
    int THREADS = 32;
    int BLOCKS = (n + THREADS - 1) / THREADS;
    dim3 threads(THREADS, THREADS); // should be <= 1024
    dim3 blocks(BLOCKS, BLOCKS);
    MatAddKernel<<<blocks, threads>>>(dA, dB, dC, n);
    cudaMemcpy(Ccpu, dC, n * n * sizeof(int), cudaMemcpyDeviceToHost);
    cudaFree(dA);
    cudaFree(dB);
    cudaFree(dC);
}
```

GPU memory

- Each thread has private registers (fastest to access)
- Each thread block has shared memory
  - Visible to all threads in a block
  - `__shared__`
- All threads can access global gpu memory
  - Persistent across kernel launches in the same app

How Kernel Threads are Executed

- SM partition a thread block into warps
- Warp is the unit of GPU creating, managing, scheduling and executing threads
- Each warp contains 32 threads
  - Start at same program address
  - Have own program counter and registers
  - Execute one common instruction at a cycle
  - Can branch and execute independently

Warp Execution on GPU

- Execution context stays on SM for lifetime of warp (Program counter, Registers, Shared memory)
- Warp-to-warp context switch is instant
- At running time, warp scheduler
  - Chooses warp with active threads
  - Issues instruction to warp's threads
- Number of warps on SM depends on mem requested and available

Matrix Multiplication with CUDA

- See notebook example.

Summary

- Operators needed for Neural network
- GPU Architecture overview
- Basic CUDA operations
- Matrix/Tensor Computation on GPU