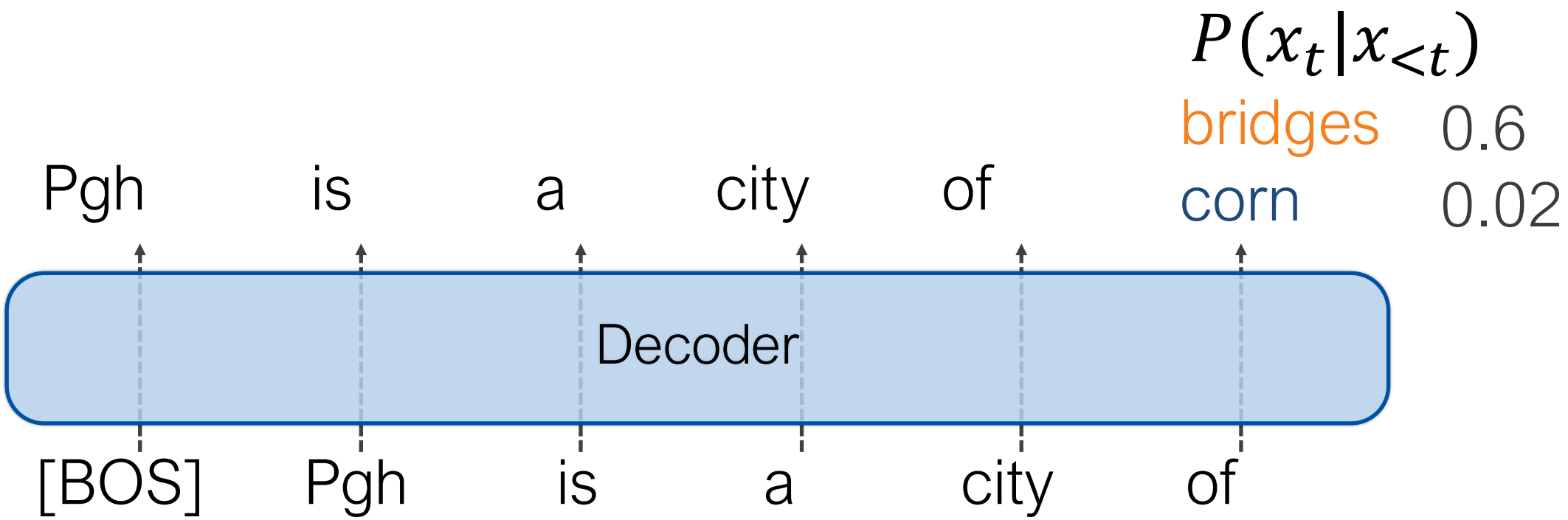
GPU Programming

Lei Li

Language Technologies Institute | Carnegie Mellon University School of Computer Science

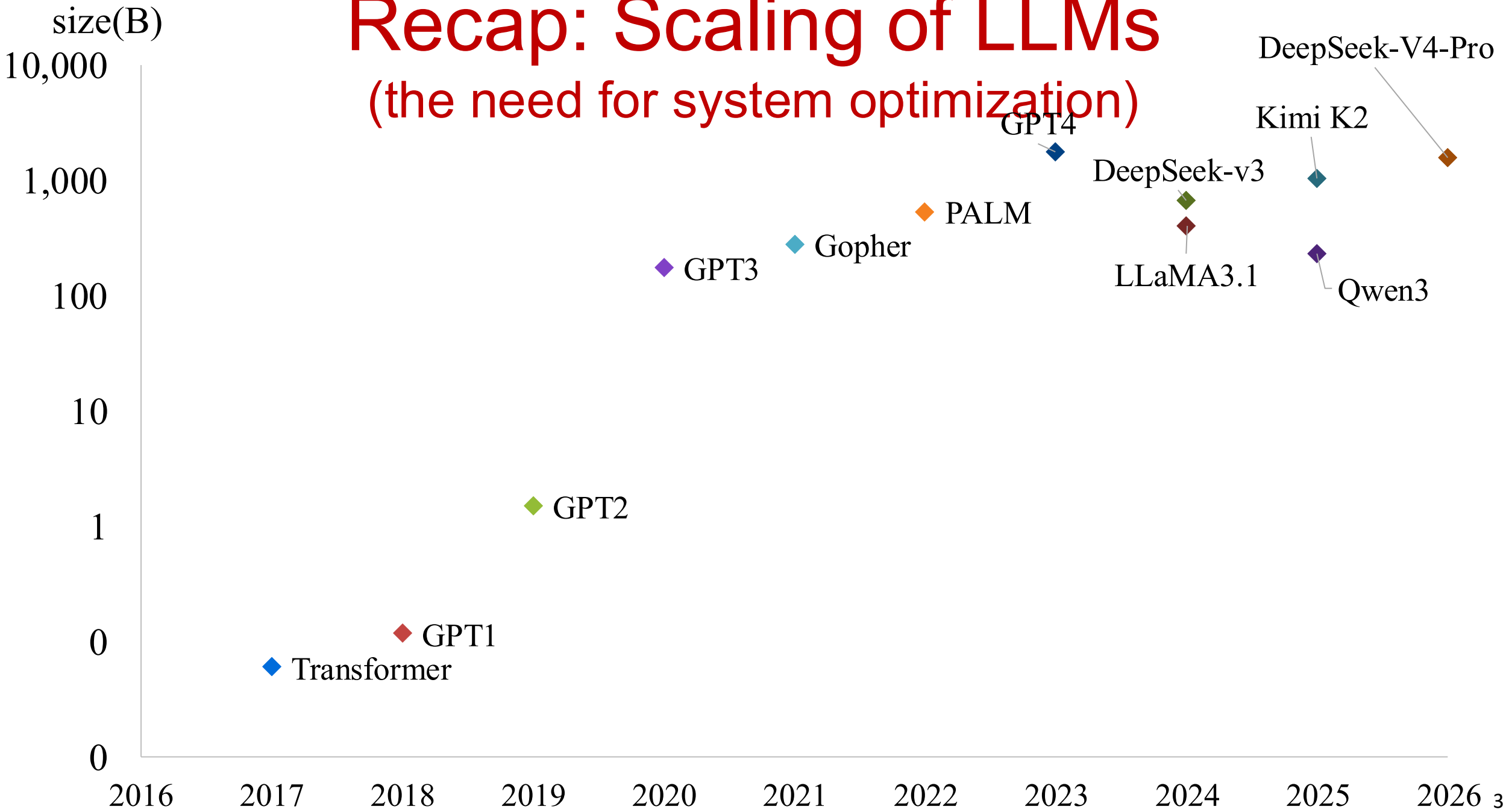
Recap: (Autoregressive) Language Model

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



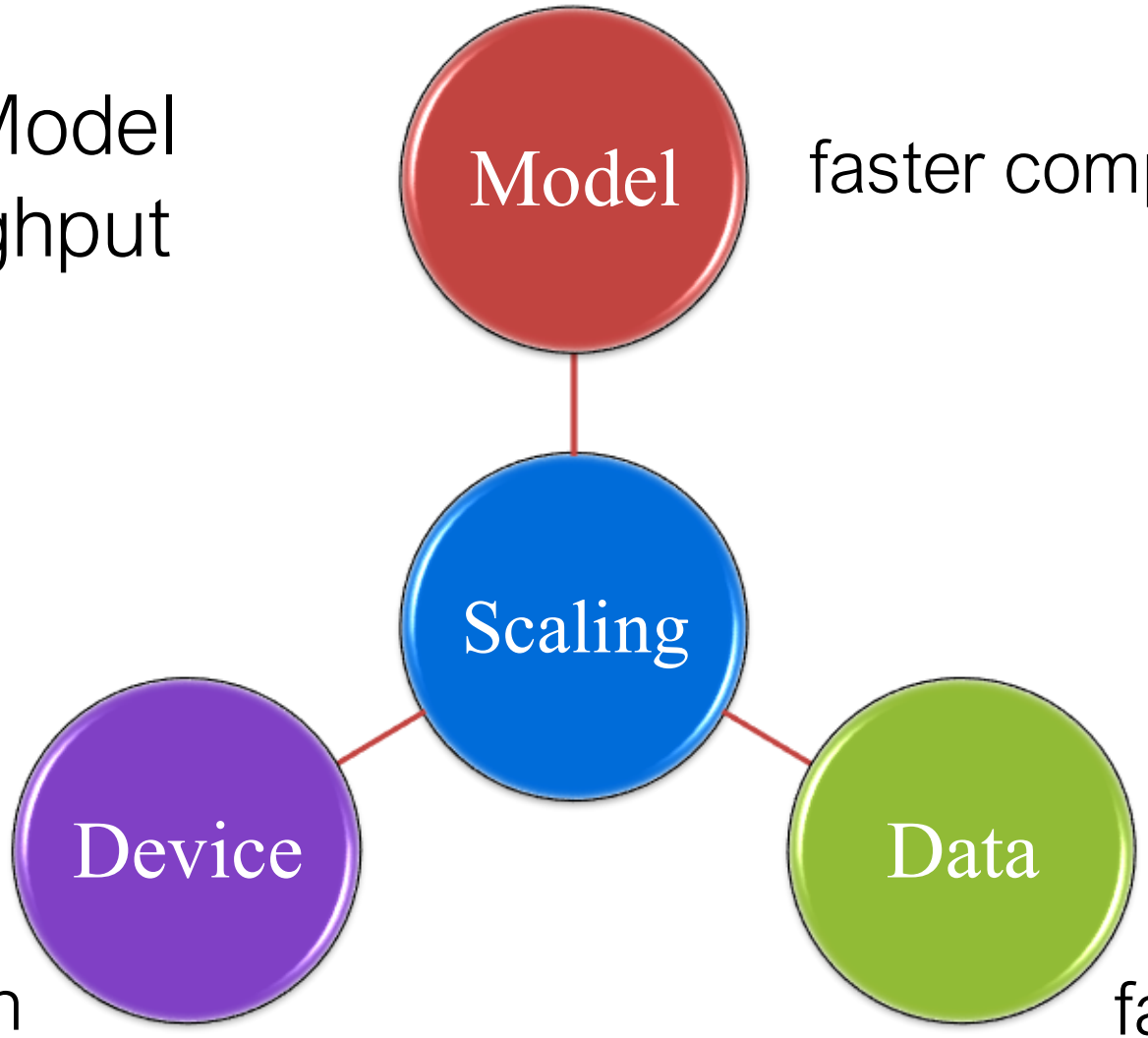
Recap: Scaling of LLMs

(the need for system optimization)



System Challenges to Scale to Larger LLMs

- Programming Model
- Latency/Throughput
- Reliability
- Security



faster computation

faster communication and coordination among devices

faster data movement

Outline

- Neural Network Layer and low-level operators
- GPU Architecture
- Program Execution on GPU
- CUDA Memory Operation
- Defining CUDA Kernel

Text Classification

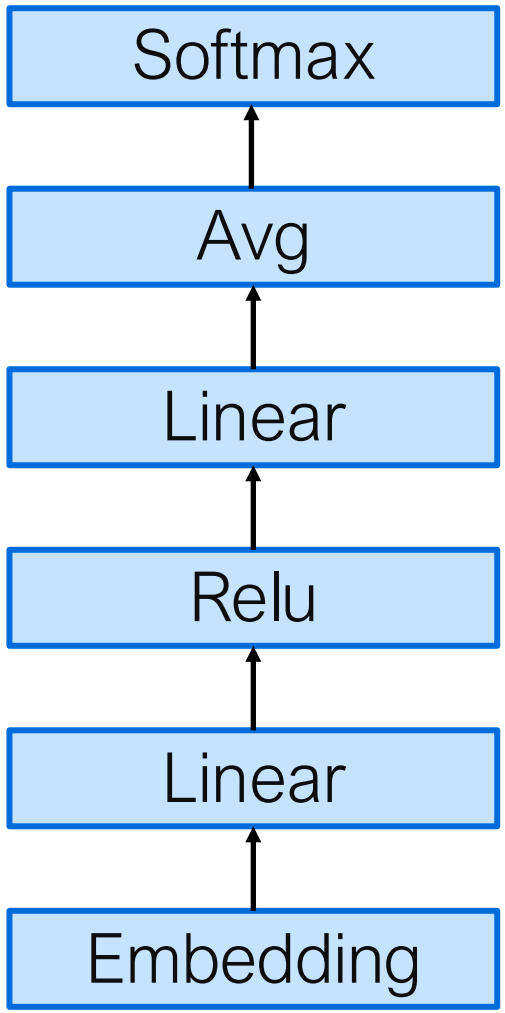
Classifying the sentiment of online movie reviews. (Positive, negative, neutral)

Spider-Man is an almost-perfect extension of the experience of reading comic-book adventures.

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

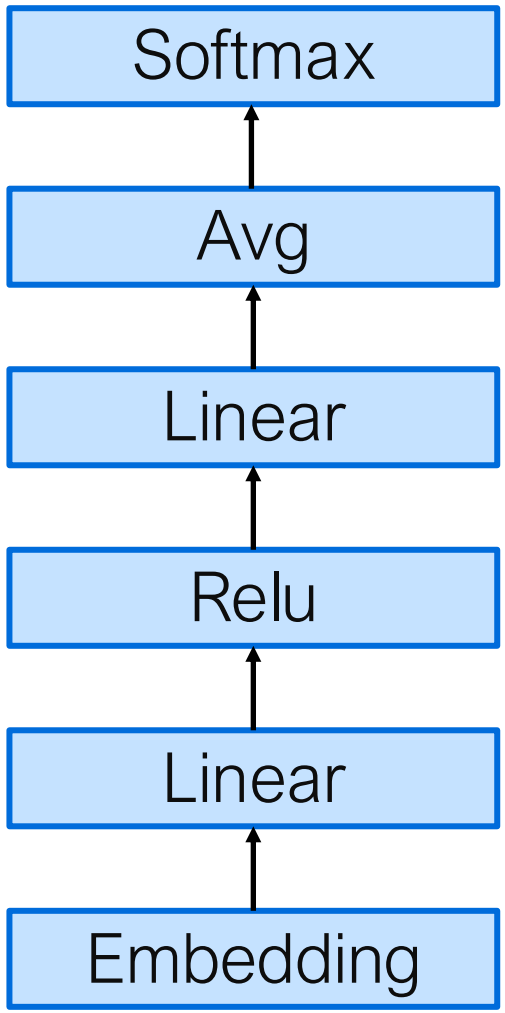
- Neural network layers
  - Embedding (lookup table)
  - Linear
  - Relu
  - Average pooling
  - Softmax



"It is a good movie"

Low-level Computing Operators

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



“It is a good movie”

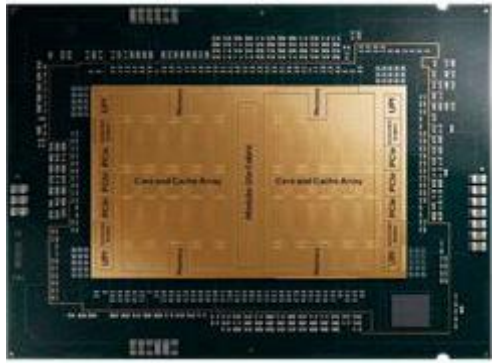
Outline

- Neural Network Layer and low-level operators
- GPU Architecture
- Program Execution on GPU
- CUDA Memory Operation
- Defining CUDA Kernel

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

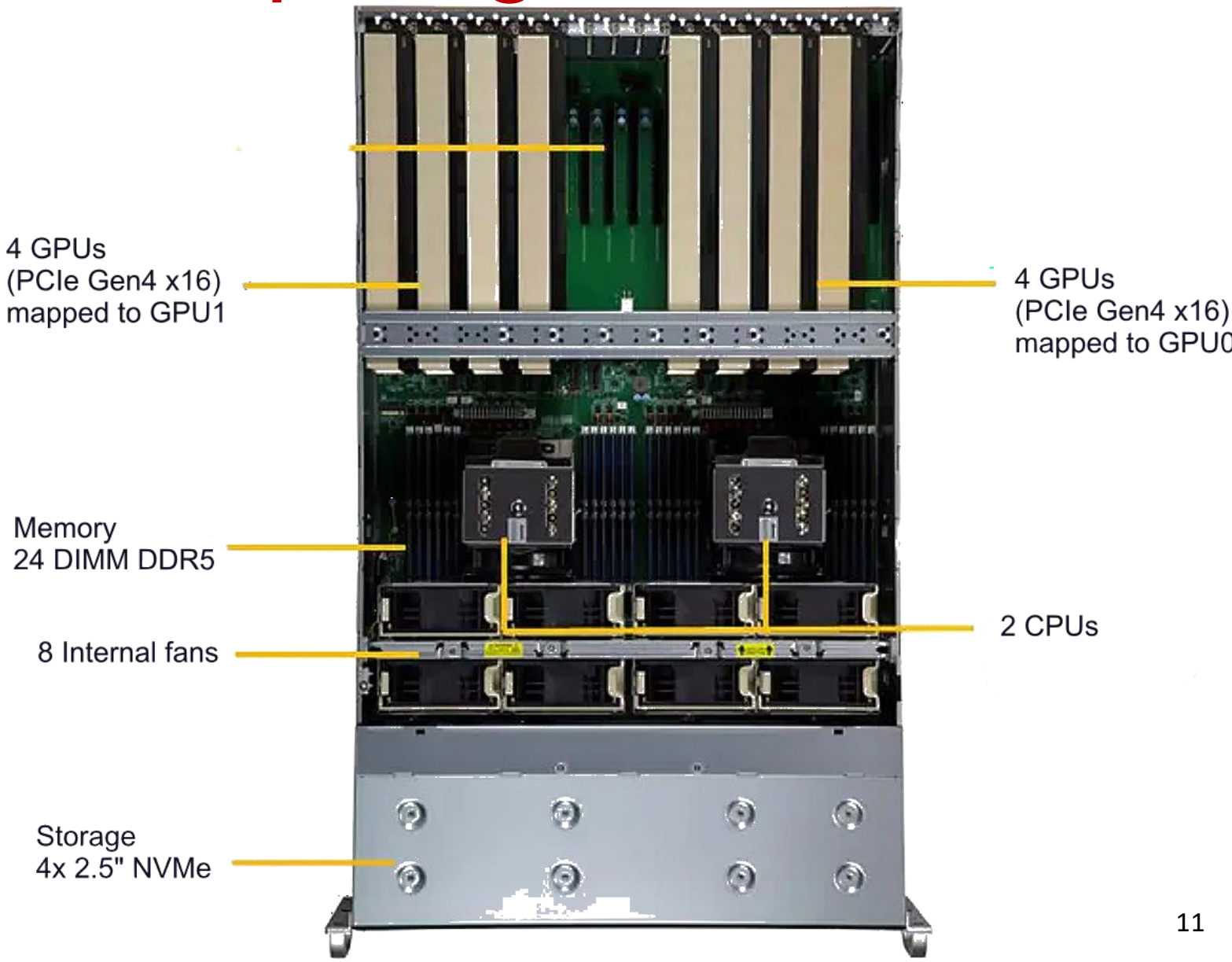
FPGA

More powerful than the #1 supercomputer in 2001

A Modern Computing Server

A Sample Config (my lab)

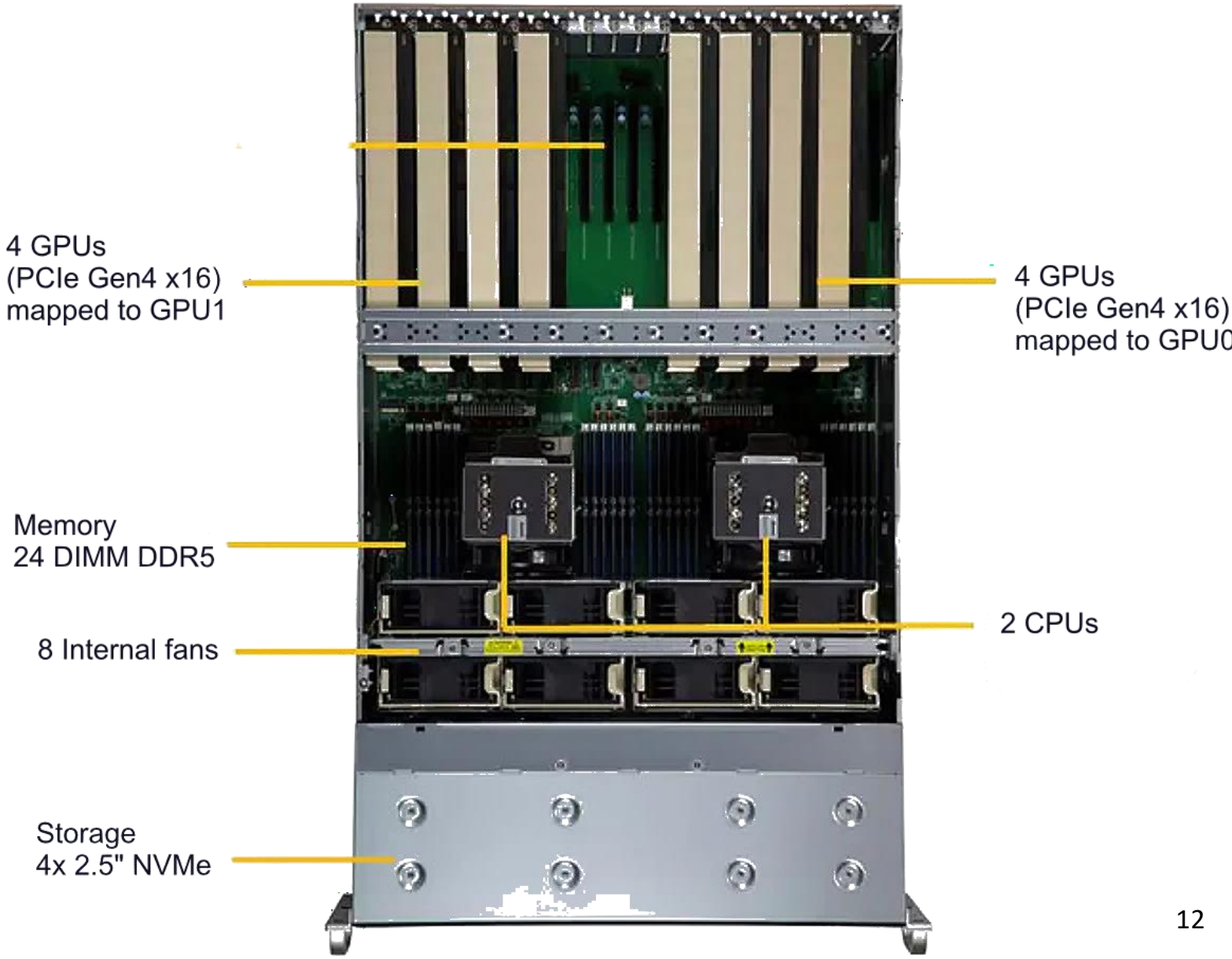
- CX4860s-EK9 4U server
- 2x AMD EPYC 9354 CPU
- 16x 64GB DDR5 mem
- 4x Intel D7 P5520 15.36TB Gen4 NVMe SSD
- 8x Nvidia A6000 48GB
- 4x 2slot NVLink



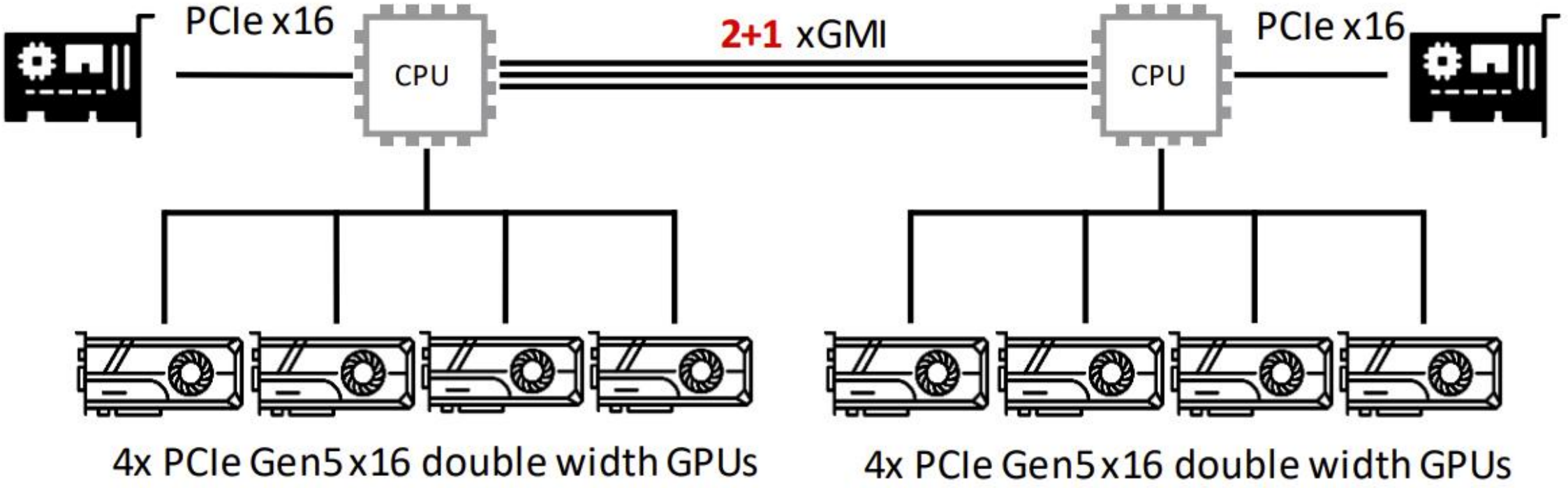
Communication

GPU-to-GPU

- NVLink 112.5 GB/s
- PCIe Gen4 32 GB/s (16 lanes x 2 GB/s per lane)



Modern Computing Server Architecture



Why is it relevant?

Considering moving gradients from one GPU to another (in data parallel)

Inside a GPU server

- where the compute happens

GPU Lineup

Architecture	Blackwell	Hopper	Ampere
GPU Name	NVIDIA B200	NVIDIA H100	NVIDIA A100
FP64	37 teraFLOPS	34 teraFLOPS	9.7 teraFLOPS
FP64 Tensor Core	37 teraFLOPS	67 teraFLOPS	19.5 teraFLOPS
FP32	75 teraFLOPS	67 teraFLOPS	19.5 teraFLOPS
FP32 Tensor Core	2.2 petaFLOPS	989 teraFLOPS	312 teraFLOPS
FP16/BF16 Tensor Core	4.5 petaFLOPS	1979 teraFLOPS	624 teraFLOPS
INT8 Tensor Core	9 petaOPs	3958 teraOPs	1248 teraOPs
FP8 Tensor Core	9 petaFLOPS	3958 teraFLOPS	-
FP4 Tensor Core	18 petaFLOPS	-	-
GPU Memory	192GB HBM3e	80GB HBM3	80GB HBM2e
Memory Bandwidth	Up to 7.7TB/s	3.2TB/s	2TB/s

GPU Architecture

Nvidia RTX A6000 (GA102)
84 SMs (Streaming Multiprocessors)
12 memory controller (32bit ea.) total 384bit.
6MB L2 cache.

H100 Architecture

132/114 SMs, 80G HBM3

Streaming Multiprocessor

4 partitions per SM, each with 32 cores ➔ 32 threads each

128 cores per SM

64KB register per partition (fastest to access)

128KB shared L1 cache per SM

128 FP32 operations in one cycle

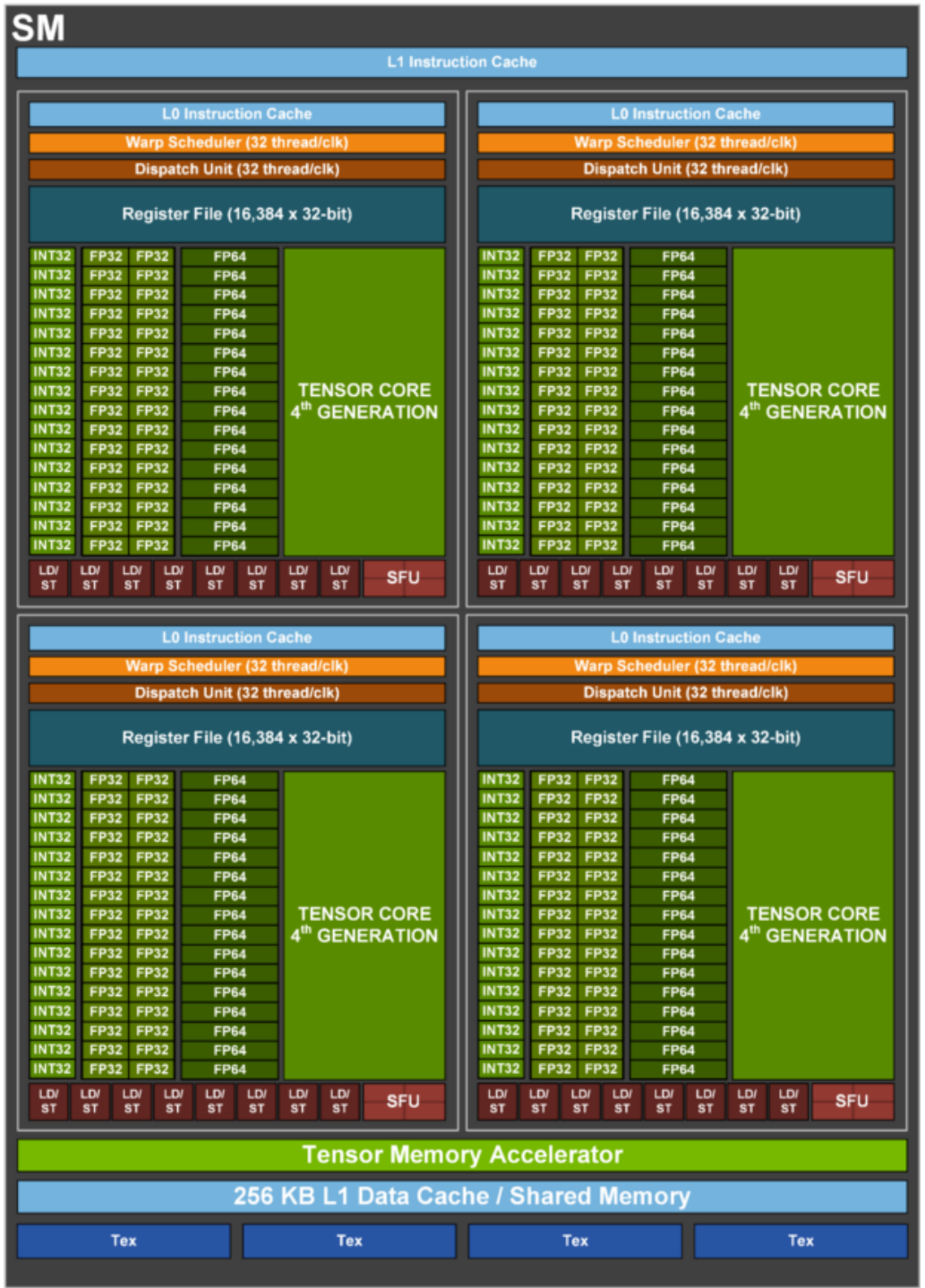
H100 SM

4 partitions per SM, each with 32 cores ➔ 32 threads each

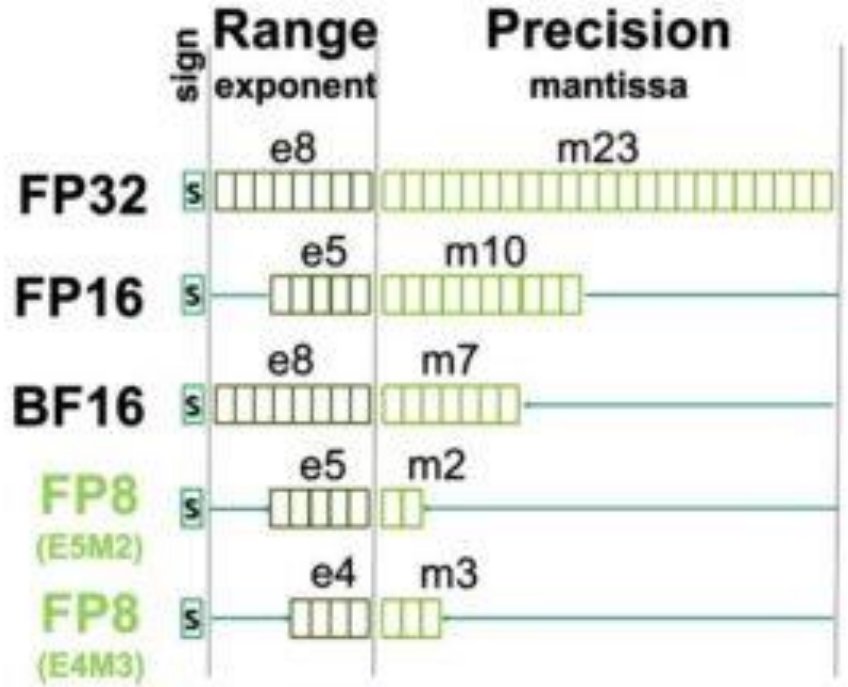
128 cores per SM

64KB register per partition (fastest to access)

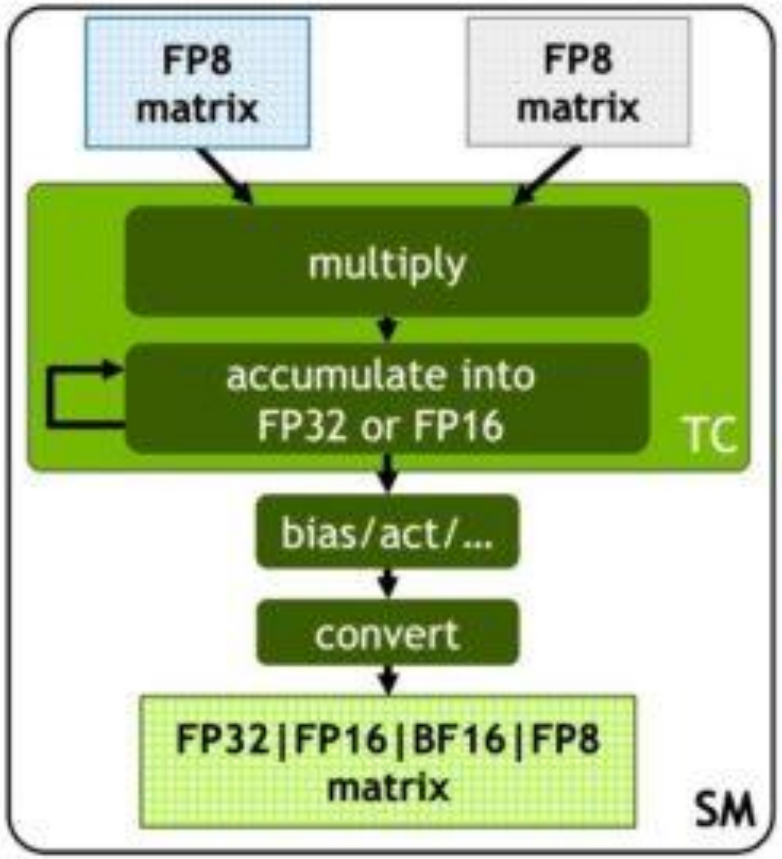
256KB shared L1 cache per SM



New in H100, FP8 operations



Allocate 1 bit to either range or precision



Support for multiple accumulator and output types

CPU vs. GPU

	CPU (AMD EPYC 9754)	GPU (A6000)
num. threads	256	10752
clock	2.25 GHz	1.8 GHz
compute	576 GFlops	38.7 TFlops
Power	360W	300W

Outline

- Neural Network Layer and low-level operators
- GPU Architecture
- Program Execution on GPU
- CUDA Memory Operation
- Defining CUDA Kernel

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

Example: Adding two vectors with CUDA

Define CUDA kernel function

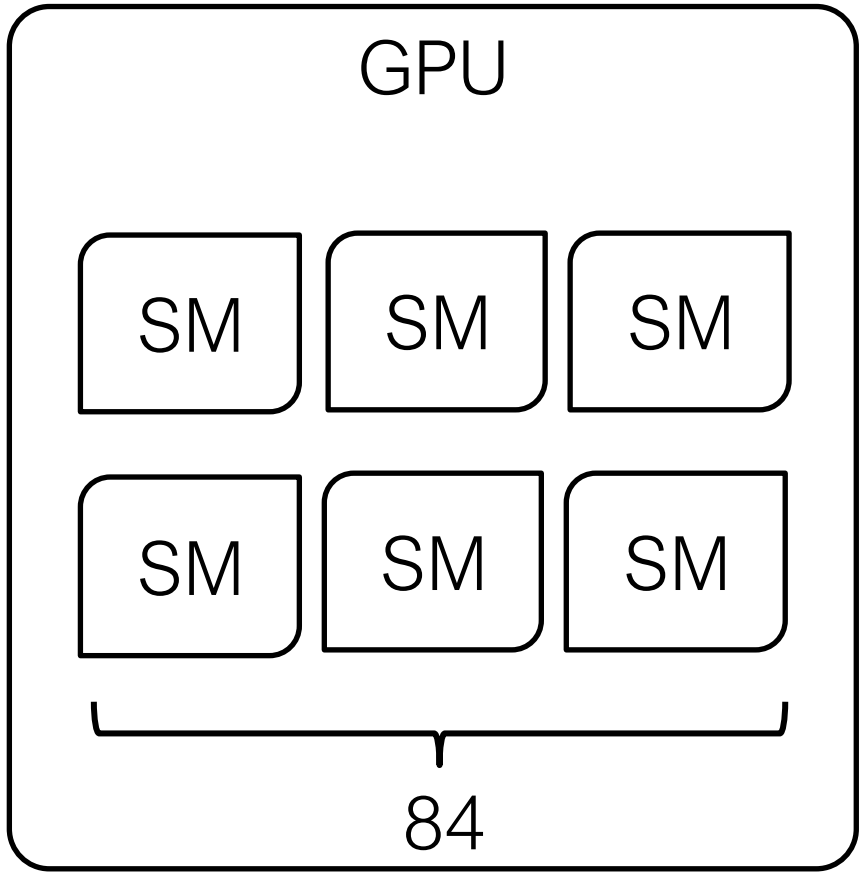
```
__global__ void VecAddKernel(int* A, int* B, int* C, int n) {
    int i = blockDim.x * blockIdx.x + threadIdx.x;
    if (i < n) {
        C[i] = A[i] + B[i];
    }
}

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

CUDA Kernel

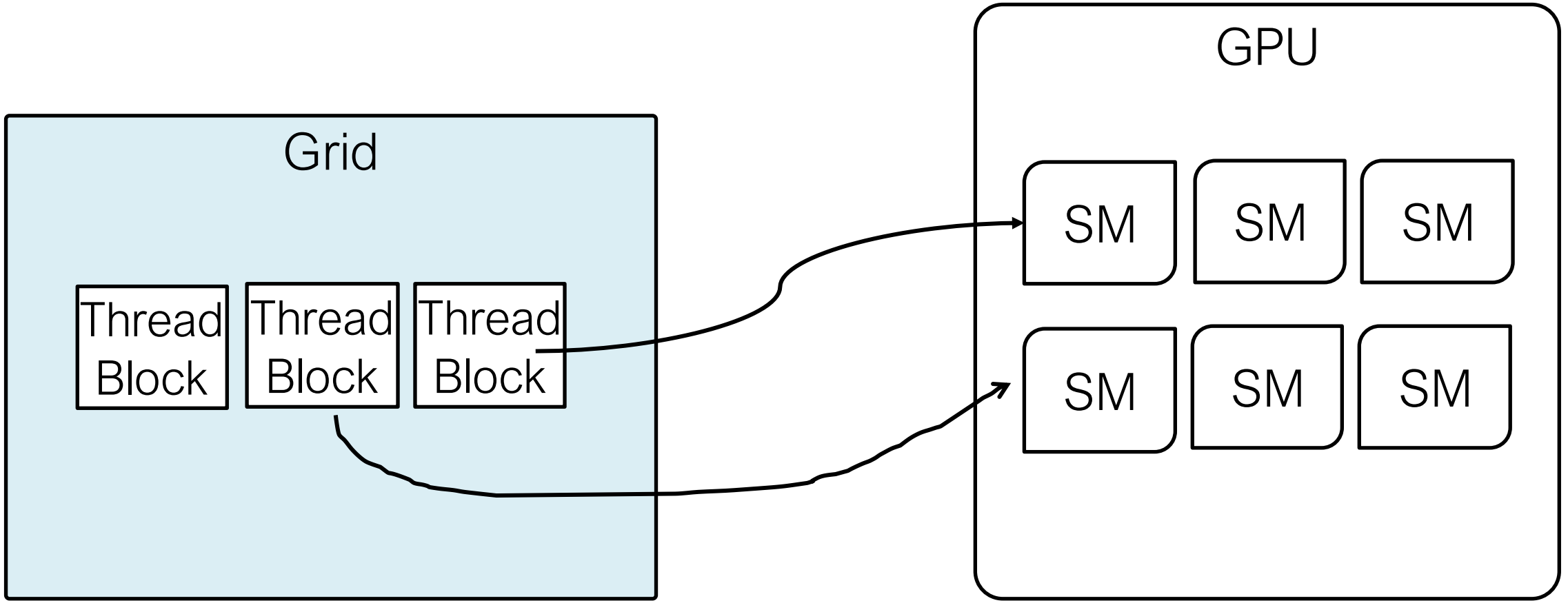
- Each kernel is a function (program) that runs on GPU
- Program itself is serial
- Can simultaneously run many (10k) threads at the same time
- Using thread index to compute on right portion of data

SIMT Execution on GPU

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



How instructions are executed on GPU



How Kernel Threads are Executed

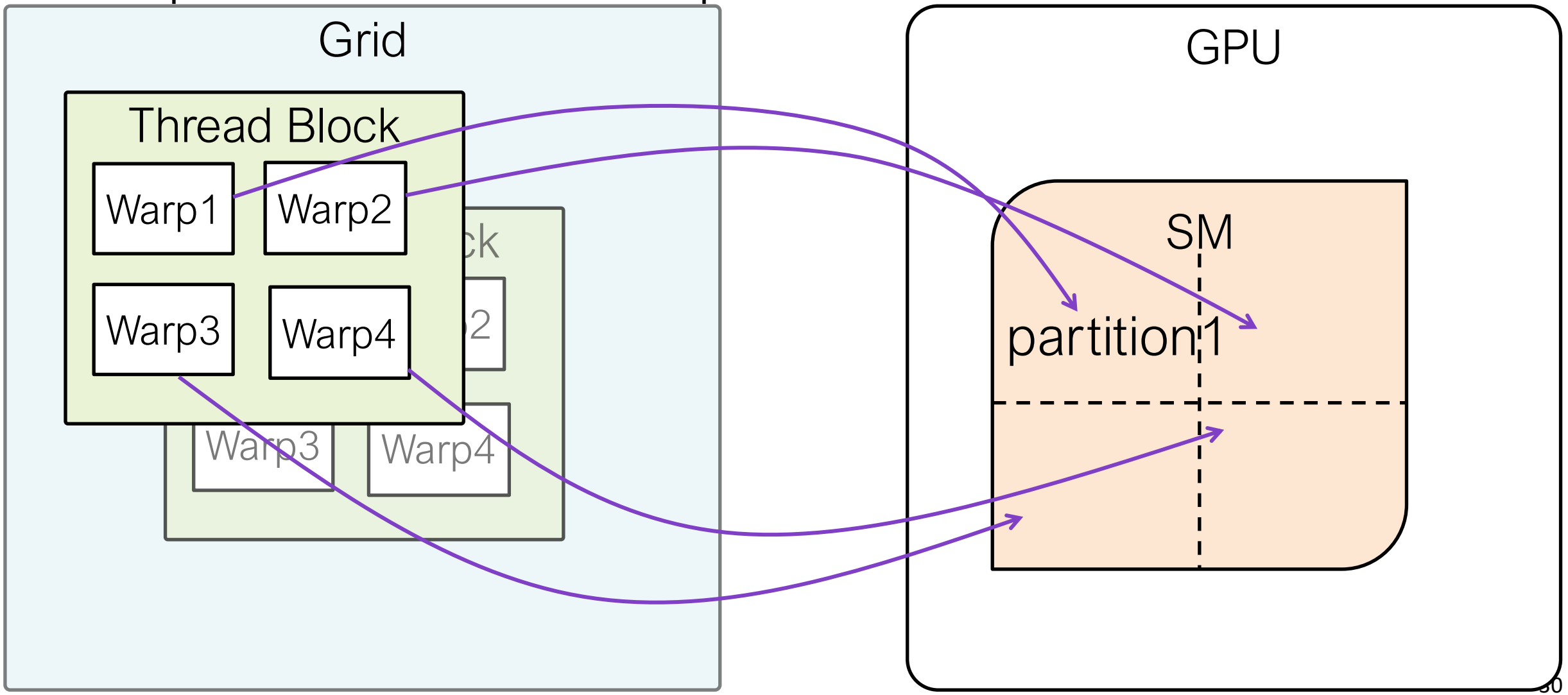
- SM partition a thread block (CUDA program) into warps
- Warp is the unit of GPU creating, managing, scheduling and executing threads
- Each warp contains **32** threads (why 32?)
  - Start at same program address
  - Have own program counter and registers
  - Execute one common instruction at a cycle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

Warp Execution on GPU

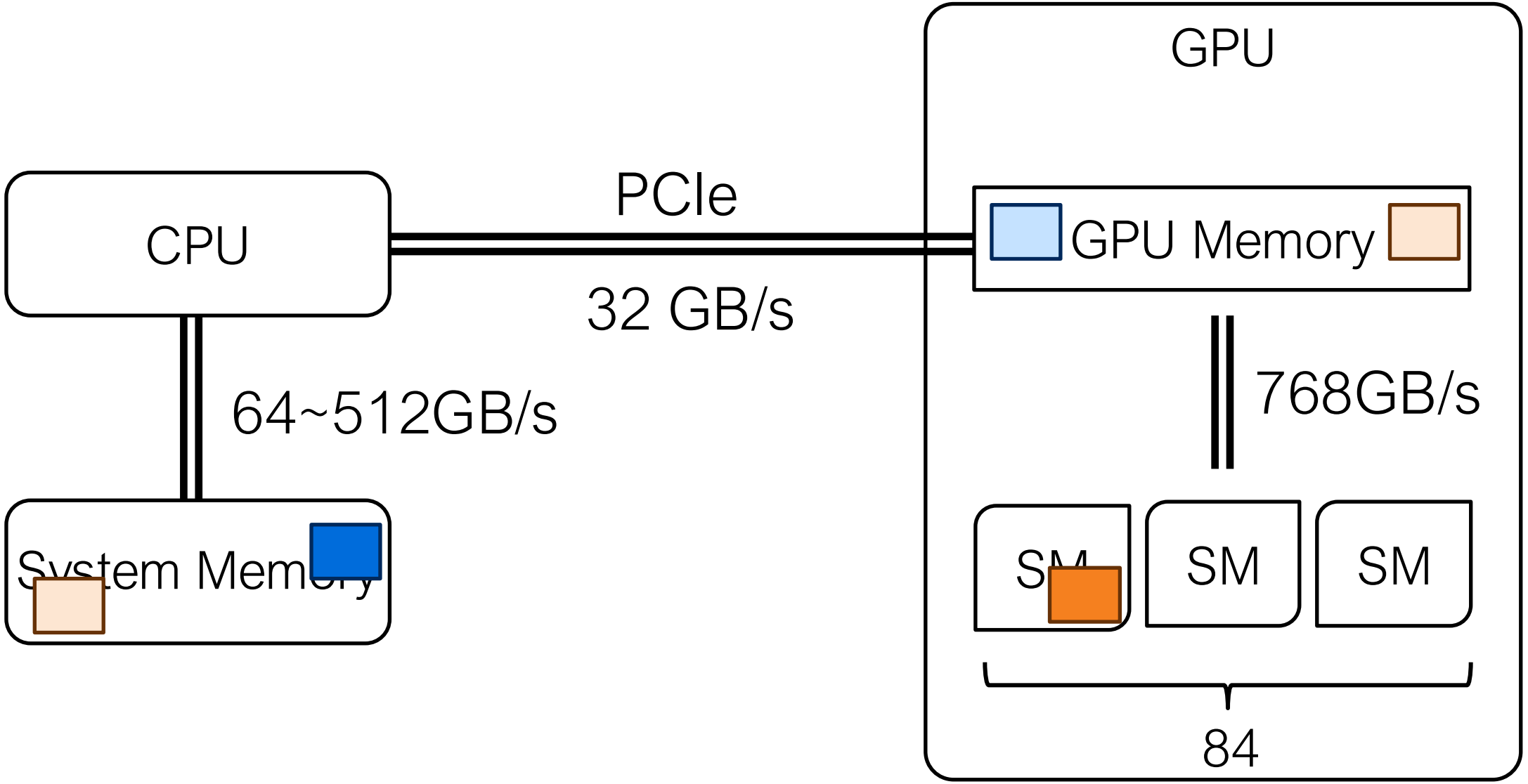
- Execution context stays on SM for lifetime of warp (Program counter, Registers, Shared memory)
- Warp-to-warp context switch is instant
- At running time, warp scheduler
  - Chooses warp with active threads
  - Issues instruction to warp's threads
- Number of warps on SM depends on mem requested and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

Executing one Thread block on one SM

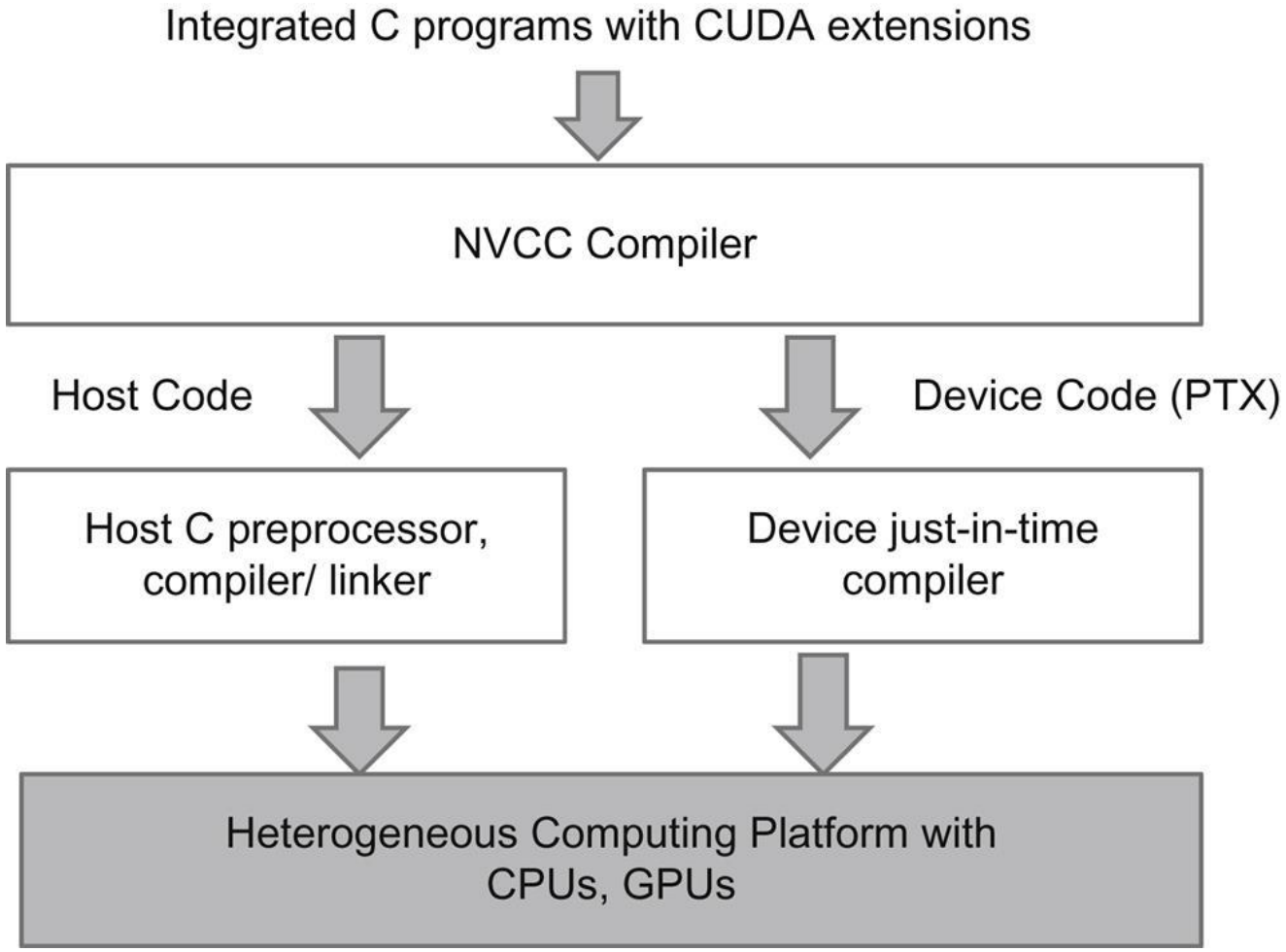
4 warps can be executed in parallel at one time on each SM



CPU-GPU Data Movement



Compiling CUDA Code



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

Outline

- Neural Network Layer and low-level operators
- GPU Architecture
- Program Execution on GPU
- CUDA Memory Operation
- Defining CUDA Kernel

CUDA Program Execution

- 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`
- Freeing GPU memory `cudaFree`

Allocate GPU Memory

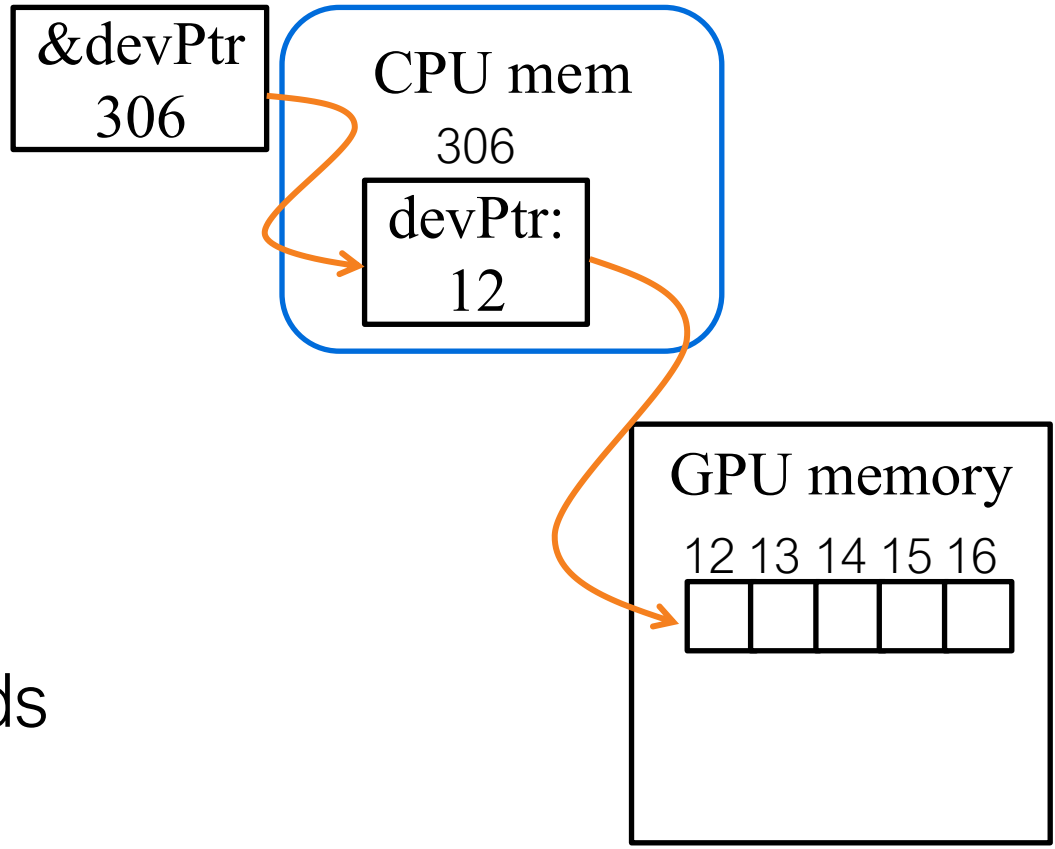
cudaError_t cudaMalloc(void** devPtr, size_t size)

devPtr- Pointer to allocated device memory

size- Requested allocation size in bytes

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

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

The allocated mem is accessible by all threads



Free GPU memory

- donot forgot!

```
cudaError_t cudaFree(void *devPtr);
```

Parameters

devPtr: A device pointer to the memory you want to free.

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

Data Movement

Copy data from devices: cpu to gpu, gpu to cpu

cudaError_t cudaMemcpy(void* dst, const void* src,
size_t count, cudaMemcpyKind kind)

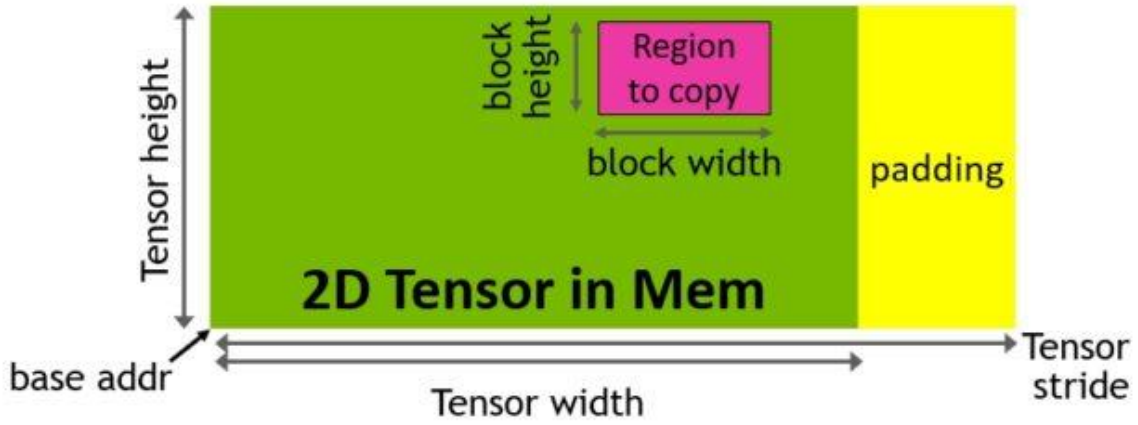
Parameters:

dst: Destination memory address
src: Source memory address
count: Size in bytes to copy
kind: Type of transfer, cudaMemcpyHostToDevice or cudaMemcpyDeviceToHost

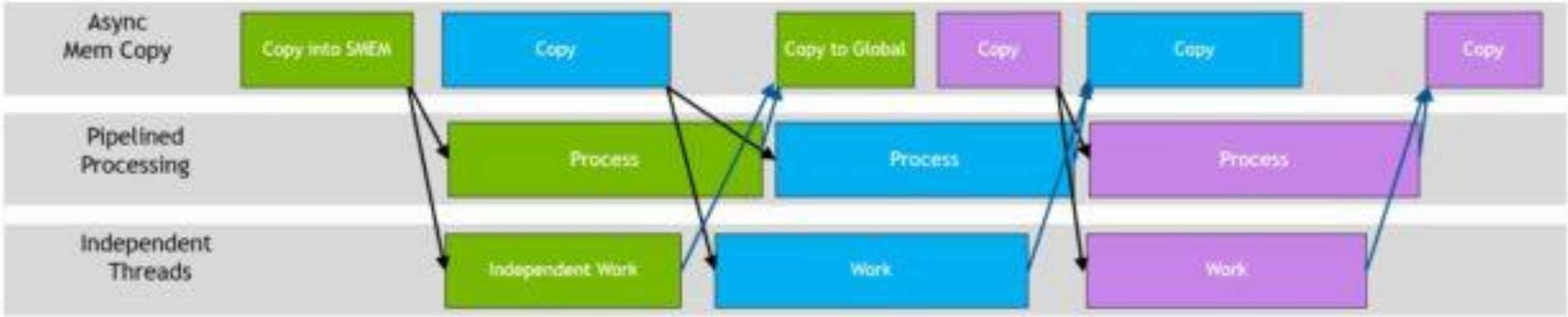
```
cudaMemcpy(dGPU, hCpu, n * sizeof(int), cudaMemcpyHostToDevice);
```

New in H100: Tensor Memory Accelerator (TMA)

```
void cuda::memcpy_async(void* destination, void const* source,
Shape size, cuda::barrier<Scope, CompletionFunction>& barrier);

        barrier.wait()
```



Asynchronous execution in Hopper GPU



CUDA Programming Model Exposure	A100	New for H100
Barrier.arrive(),	Asynchronous Barrier	**Asynchronous Transaction Barrier**
Barrier.wait()	Waiter spins in SMEM	**Waiter sleeps until all threads arrive**
Memcpy_async()	Direct copy to SMEM	**Asynchronous mem copy unit (called TMA)**

Outline

- Neural Network Layer and low-level operators
- GPU Architecture
- Program Execution on GPU
- CUDA Memory Operation
- Defining CUDA Kernel

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

Defining Functions to be executed on GPU

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);
}
```

Calling Kernel at Runtime

- Host program specifies grid-block-threads configurations for kernel at run time
  - Dg and Db are either **dim3** or **int**

```
dim3 Dg(4, 2, 1);
dim3 Db(8, 8, 1);
kernelFuncName<<<Dg, Db>>>(args)
```

- **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
- 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

Calling CUDA Kernel from CPU

Running kernels on GPU

```
// n: the size of the vector
int n = 1024;
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);
```

CUDA Code Examples for Matrix Computation

Matrix Multiplication with CUDA

- See notebook example.
- https://github.com/llmsystem/llmsys_code_examples/blob/main/simple_cuda_demo/CUDA_Code_Examples.ipynb
- You may upload and run it in Google Colab.

```
__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);
}
```

Summary

- Neural network
  - is composed of layers, each layer defines input and output vectors/embeddings
  - Each layer's computation consists of low-level operators, which is executed on a computing device (GPU, CPU, FPGA, etc)

Summary

- GPU is composed of
  - streaming processing units (SMs)
    - each with four partitions of 32 cores
    - shared L1 cache
  - memory
  - L2 cache: share with all SMs
- Threads organized in
  - grid of thread blocks
  - each block is divided into warps running in parallel on one SM.

Summary

- Basic GPU CUDA operations
  - memory allocation
  - data movement
  - creating threads and running on SMs
    - specifying number of threads and number of blocks in a grid
  - referring to data in GPU memory within a thread
    - using building index variables to refer to the data

Reading

- Chap 2,3,4 of ”Programming Massively Parallel Processors, 4th Ed. https://learning.oreilly.com/library/view/programming-massively-parallel/9780323984638/?sso_link=yes&sso_link_from=cmu-edu

- Free for CMU students

Assignment 1

https://llmsystem.github.io/llmsystemhomework/

Quiz 1.2