LLM Sys

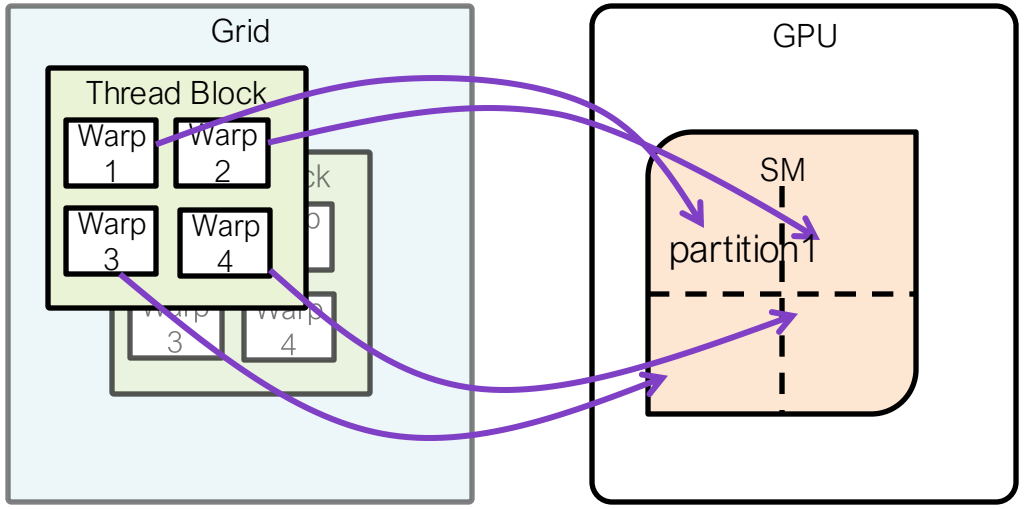
GPU Programming

Lei Li

Carnegie Mellon University
Language Technologies Institute

Recap

- 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 on one SM.



Assignment 1

https://llmsystem.github.io/llmsystem2025springhw/assignment_1/

Starter code: https://github.com/llmsystem/llmsys_s25_hw1.git

Due Feb 3

Reminder: start form your project team (2-3 students)

project proposal due Feb. 26/2025

Outline

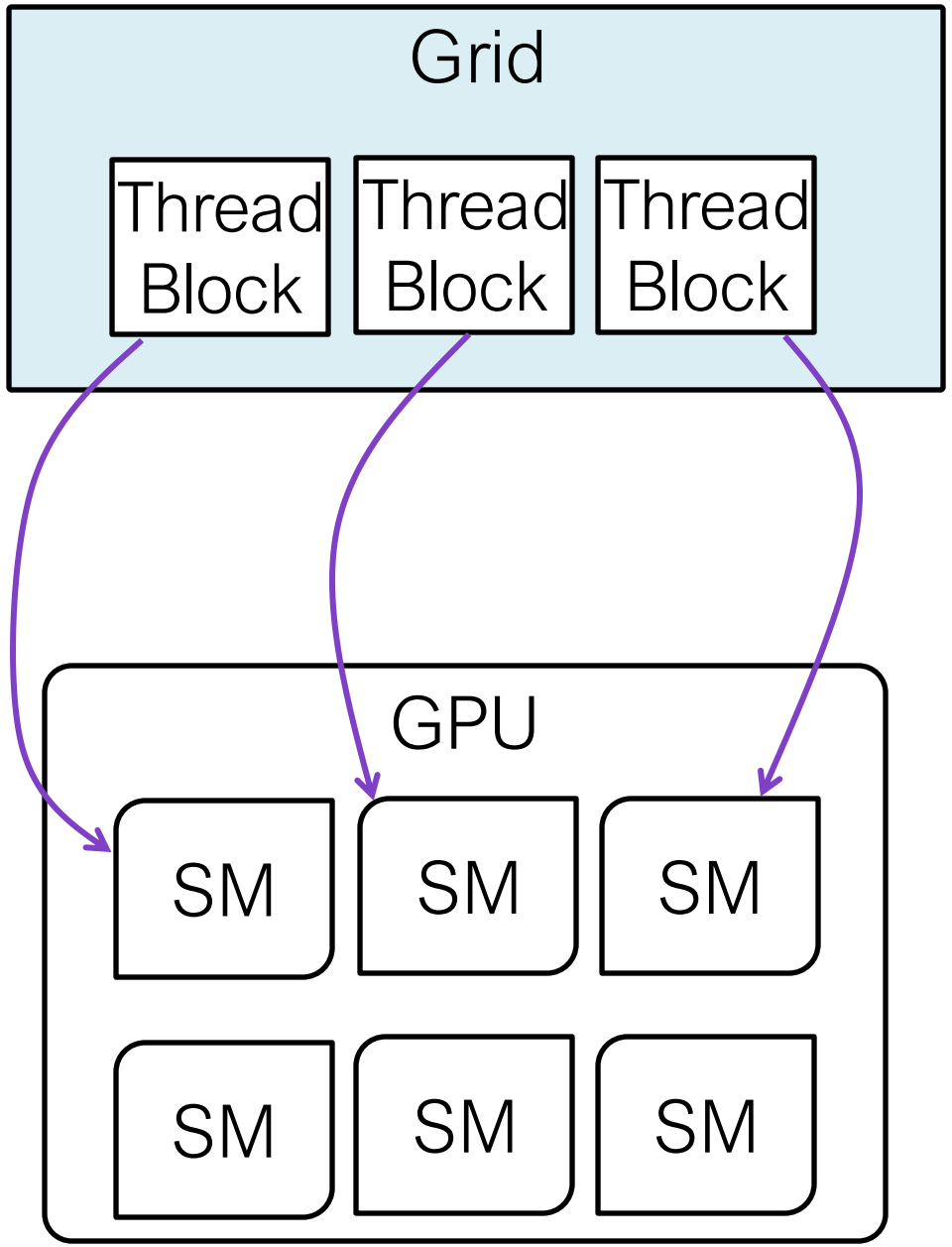
- Basic GPU CUDA operations
  - memory management
  - creating threads
  - defining kernel functions for arithmetics
- Matrix/Tensor Computation on GPU

CUDA Kernel

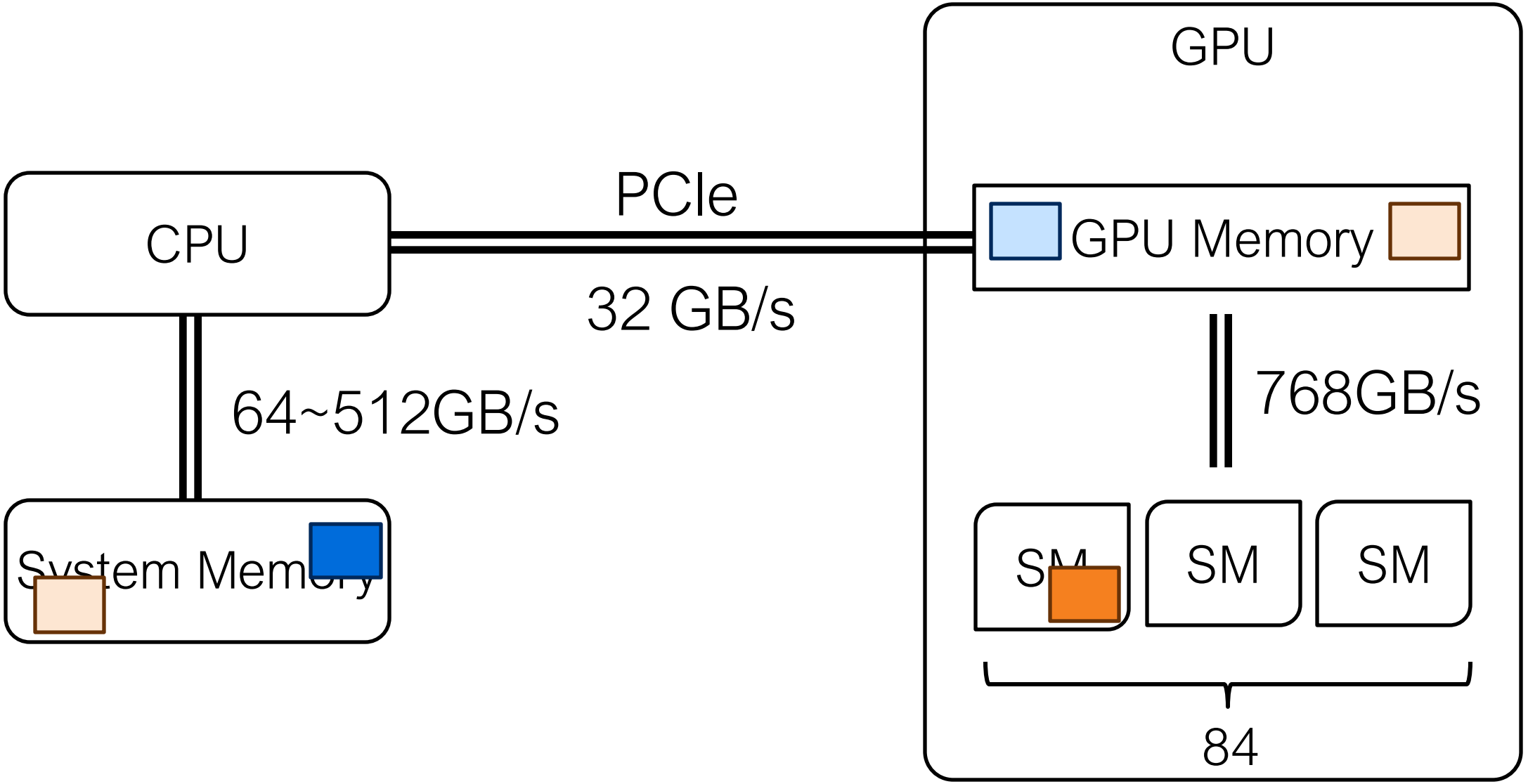
- Each kernel is a function (program) that runs on GPU
- Program itself is serial
- Can simultaneously run many (10k) threads at the same time
- Using thread index to compute on right portion of data

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



CPU-GPU Data Movement



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`
- Freeing GPU memory `cudaFree`

Allocate GPU Memory

cudaError_t cudaMalloc(void** devPtr, size_t size)

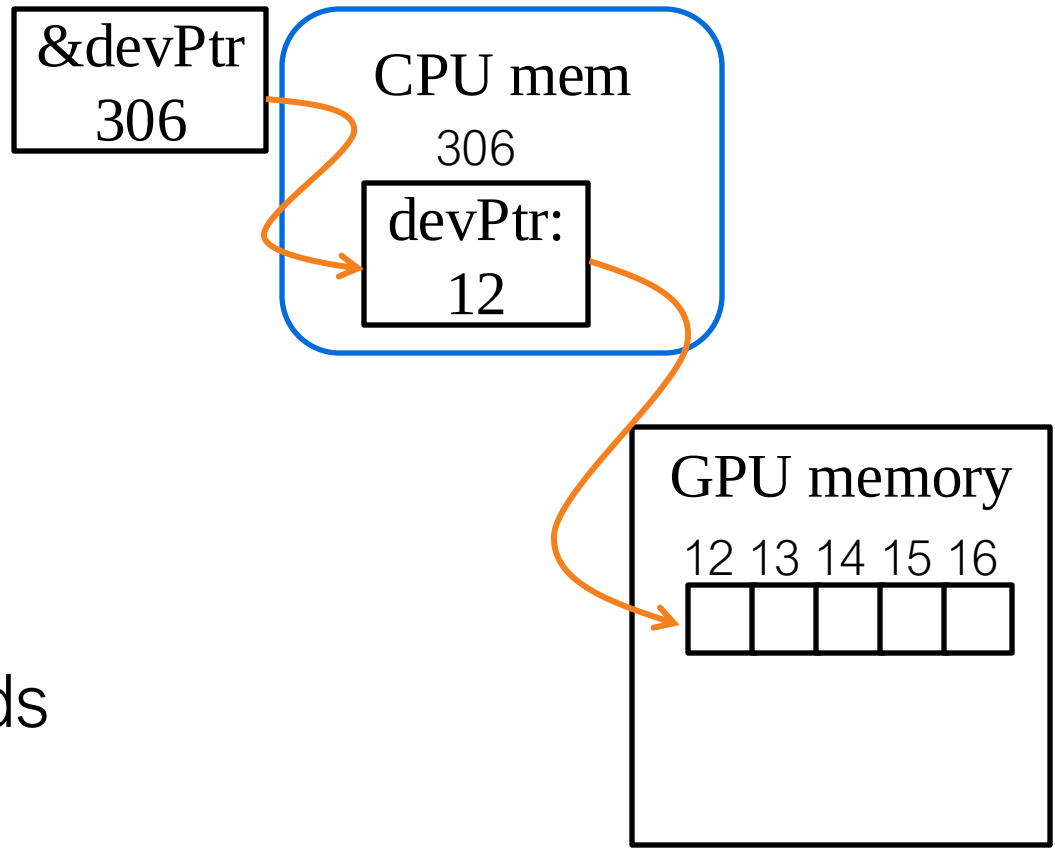
devPtr- Pointer to allocated device memory

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.

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);
```

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

- 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

Recommended 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

Next

- Auto Differentiation (automatically calculate gradients for any composite functions)