ECE 587 – Hardware/Software Co-Design
Lecture 17 CUDA I: Introduction

Professor Jia Wang
Department of Electrical and Computer Engineering
Illinois Institute of Technology

March 9, 2026

Outline

Reading Assignment

- This lecture: CUDA Introduction
- Next lecture: CUDA Memory Access

Outline

CUDA

- Exploit GPU for general purpose computing needs.
  - Leverage GPU's massive parallelism and high memory bandwidth.
- CUDA C/C++
  - Standard C/C++ + extensions to enable heterogeneous programming
  - APIs to manage devices and memory.
- Example computations
  - Vector additions
  - General matrix multiplications (GEMM)
  - Parallel reductions
  - Learn to write code for correct functionality first, then measure and optimize.

Heterogeneous Computing

- Host: CPU and its memory
  - Executes sequential code and manages GPU resources.
- Device: GPU and its memory
  - Executes parallel kernels and performs massively parallel computations.
- Typical code structure
  - Define kernels (specialized C/C++ functions) to run on the device.
  - Use host code (usual C/C++ code) to control and interact with device, and to complete overall application logic.
- Typical run
  - Host code allocates memory on the device
  - Host code copies data from host to device.
  - Host code launches kernel on the device.
  - Host code copies results back from device to host.

Outline

CPU Vector Addition

```
#pragma omp parallel for
for (int i = 0; i < n; ++i) {
  c[i] = a[i] + b[i];
}
```

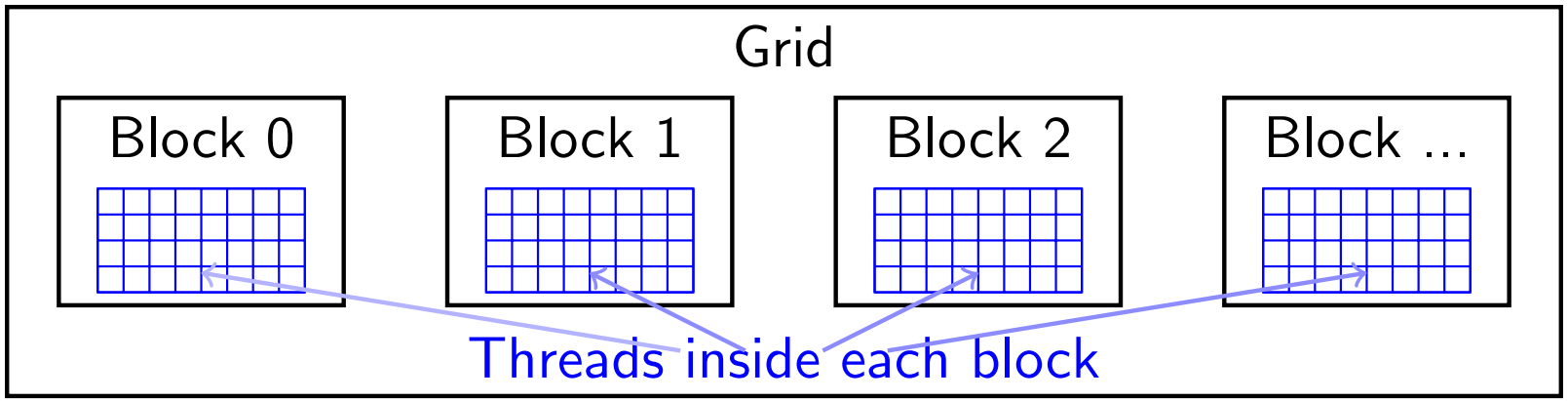
- Vector addition can be easily parallelized on CPU using OpenMP.
- Usually GPU has more cores to exploit, but requires explicit management of parallelism and memory.

CUDA Kernel for Vector Addition with 1D Mapping

```
__global__ void vec_add(
  const float* a, const float* b, float* c, int n) {
  int idx = blockIdx.x * blockDim.x + threadIdx.x;
  if (idx < n) {
    c[idx] = a[idx] + b[idx];
  }
}
```

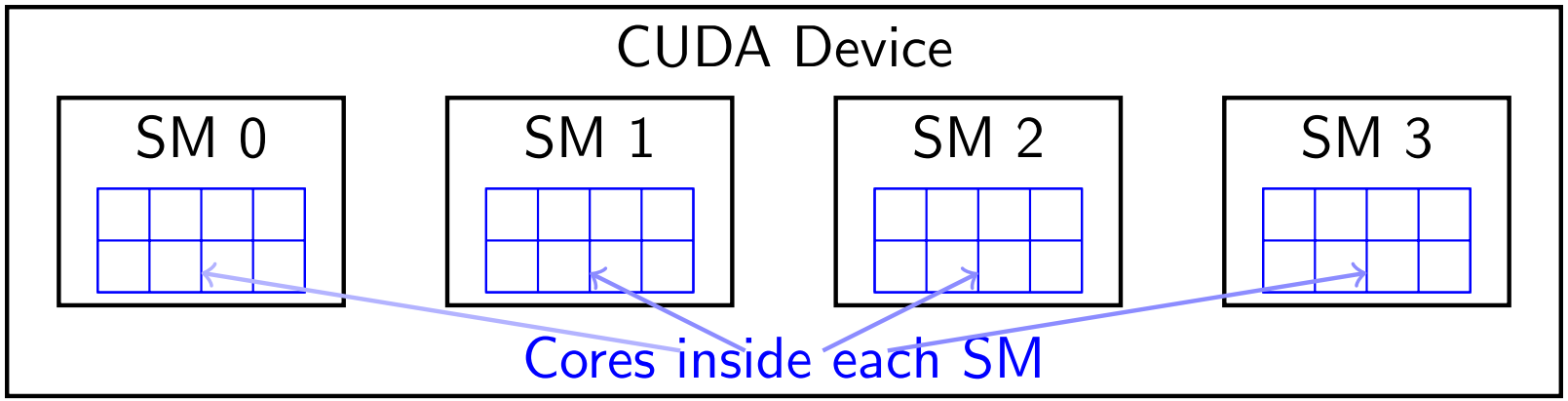
- Allow each CUDA thread to compute one element of the output vector along the 1D index space.
  - Much more CUDA threads than CPU threads!
  - Bound check is needed to avoid out-of-bounds access.
- What are `blockIdx`, `blockDim`, and `threadIdx`?
  - In order to compute `idx`.
- `__global__` indicates this function is a kernel to be executed on the device.
  - Where are the input and output vectors `a`, `b`, and `c` located?

Threads and Blocks



- The massive number of CUDA threads are organized hierarchically for management.
  - Threads are grouped into blocks.
  - Blocks are then grouped into a grid.
- `blockIdx`: the block index within the grid.
- `blockDim`: number of threads in a block.
- `threadIdx`: the thread index within the block.
- In 1D mapping, `blockIdx.x*blockDim.x+threadIdx.x` computes the global thread index.

Streaming Multiprocessors and Cores



- Each CUDA device contains a fixed number of streaming multiprocessors (SMs).
  - Each SM execute a block of threads at a time.
  - Different SM may execute different kernels at the same time.
- Each CUDA SM contains a fixed number of CUDA cores.
  - Each core executes one thread at a time.
  - Cores in the same SM execute threads from the same kernel.
- For best performance we usually should have much more blocks than SMs and threads than cores.

Device Data Management

```
// __global__ void vec_add(
//   const float* a, const float* b, float* c, int n)

// 1) allocate device memory
cudaMalloc((void**)&d_a, n * sizeof(float));
cudaMalloc((void**)&d_b, n * sizeof(float));
cudaMalloc((void**)&d_c, n * sizeof(float));

// 2) copy input data H2D
cudaMemcpy(d_a, h_a, n * sizeof(float), cudaMemcpyHostToDevice);
cudaMemcpy(d_b, h_b, n * sizeof(float), cudaMemcpyHostToDevice);

// 3) launch kernel (see next slide)
// vec_add<<<..., ...>>>(d_a, d_b, d_c, n);

// 4) copy output data D2H
cudaMemcpy(h_c, d_c, n * sizeof(float), cudaMemcpyDeviceToHost);
```

- All kernels including `vec_add` operate on device memory.
  - Pass-by-value parameters like `n` are copied automatically.
  - Pointers like `a`, `b`, and `c` refer to device memory and we need to explicitly allocate them and manage data transfer.

Kernel Launch

```
const int threads_per_block = 256;
const int blocks = (n + threads_per_block - 1) / threads_per_block;
vec_add<<<blocks, threads_per_block>>>(dev_a, dev_b, dev_c, n);
```

- We could schedule more threads per block than cores per SM.
  - Let's use 256 for now.
  - We will spend a lot of time discussing how threads are scheduled in SM and how that affects performance later.
- Notice how we compute `blocks` to cover all elements.
  - This is a common pattern to handle cases where `n` is not a multiple of `threads_per_block`.
  - No floating point division, no rounding, just integer arithmetic.
  - Recall the bound check in the kernel `vec_add` to avoid out-of-bounds access.

Vector Addition Performance

Table: `vec_add` kernel time (ms)/GFLOPS, warm-up+single run, data transfer time excluded, on one Tesla T4 GPU.

ths/blk	N=1M	N=2M	N=4M	N=8M
64	0.06/18.39	0.10/20.05	0.20/21.02	0.39/21.35
128	0.06/18.30	0.11/19.69	0.20/20.70	0.40/21.11
256	0.06/18.17	0.11/19.69	0.20/20.69	0.39/21.25
512	0.06/17.68	0.11/19.69	0.20/20.48	0.40/21.13

- Compile and run `vec_add.cu` on a CUDA GPU.
  - Add error checking, warm-up, and measurement to our sample code as needed.
- GFLOPS increase slightly with N.
- `threads_per_block` (ths/blk) has very little impact to performance on this simple kernel.

Outline

ijk Ordering Revisited

```
for (std::size_t i = 0; i < N; ++i) {
  for (std::size_t j = 0; j < N; ++j) {
    float sum = 0.0f;
    for (std::size_t k = 0; k < N; ++k) {
        sum += A[i * N + k] * B[k * N + j];
    }
    C[i * N + j] = sum;
  }
}
```

- While the ijk ordering is not the best for CPU performance, it exposes a lot of paralellism.
- Each element of C is computed independently.
  - We can use a CUDA thread to compute each element of C.

From 1D Mapping to 2D Mapping

```
int row = blockIdx.y * blockDim.y + threadIdx.y;
int col = blockIdx.x * blockDim.x + threadIdx.x;
```

- We can simply use the 1D mapping since we can locate each element of $\boldsymbol{A}$, $\boldsymbol{B}$, and $\boldsymbol{C}$ using the 1D index.
  - E.g. for the row-major order.
- However, it is more natural to use 2D mapping to compute each element of $\boldsymbol{C}$.
  - As we will discuss later, 2D mapping also makes it easier to reason with memory access patterns and to optimize them.
  - CUDA in addition supports 3D mapping.
- Similar to the 1D mapping, we compute the row and column indices for threads globally from the block and thread indices.

Simple GEMM Kernel

```
__global__ void gemm_simple(
  const float* A, const 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 sum = 0.0f;
    for (int k = 0; k < N; ++k) {
      sum += A[row * N + k] * B[k * N + col];
    }
    C[row * N + col] = sum;
  }
}
```

- Assume all matrices are stored in row-major order.
- Run the k loop sequentially within each thread.
- Don't forget the Bound check.

Simple GEMM Kernel Launch

```
dim3 dim_threads(16, 16); // or dim_threads(1, 64), dim_threads(64, 1), etc.
dim3 dim_blocks(
  (N + dim_threads.x - 1) / dim_threads.x,
  (N + dim_threads.y - 1) / dim_threads.y);
gemm_simple<<<dim_blocks, dim_threads>>>(d_A, d_B, d_C, N);
```

- `dim3` allows us to specify shapes up to 3D.
  - Providing a 2D shape if two parameters are given.
- `dim_threads(32, 32)` gives 1024 threads per block.
  - Each block computes a 32×32 submatrix of $\boldsymbol{C}$.
- `dim_threads(16, 64)` also gives 1024 threads per block.
  - Each block computes a 64-row 16-column submatrix of $\boldsymbol{C}$.
- `dim_threads(1, 64)` gives 64 threads per block.
  - Each block computes 64 elements in the same column of $\boldsymbol{C}$.

Simple GEMM Performance

Table: `gemm_simple` kernel time (s)/GFLOPS, warm-up+single run, data transfer time excluded, on one Tesla T4 GPU.

shape	N=4096	N=8192	N=16384
(64,16)	0.24/566.85	2.05/537.26	23.81/369.39
(32,32)	0.23/597.74	1.92/573.14	18.73/469.67
(16,64)	0.26/529.47	2.25/489.65	21.10/416.87
(64,4)	0.39/352.76	5.88/187.11	103.53/84.96
(32,8)	0.30/465.15	2.69/408.22	78.62/111.88
(16,16)	0.32/430.35	2.76/398.47	50.23/175.12
(8,8)	0.59/231.42	7.00/157.09	66.94/131.40
(1,64)	2.24/61.43	20.65/53.23	186.25/47.23

- We achieve much higher GFLOPS than vector addition.
- More threads help, and shape matters.
- Smaller matrices benefit from less cache misses.

Summary

- CUDA enables access tp massively parallel computational resources availabel on modern GPUs.
- Flexible thread mappings express parallelism in natural ways.
- How do shapes impact GEMM performance?
  - And other computations.