ECE 587 – Hardware/Software Co-Design
Lecture 19 CUDA III: Synchronization

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

March 23, 2026

Outline

Shared Memory and Synchronization

Register Blocking

Reading Assignment

- This lecture: CUDA Synchronization
- Next lecture: CUDA Warp Execution

Outline

Shared Memory and Synchronization

Register Blocking

Ideas for Block Matrix Multiplication on CUDA

```
// The two outer loops should be parallelized by CUDA blocks
for (std::size_t ti = 0; ti < tiles; ++ti) {
    for (std::size_t tj = 0; tj < tiles; ++tj) {
        const std::size_t baseC_tile = (ti * tiles + tj) * M2;
        // CUDA threads in a block then follow the same inner loop.
        for (std::size_t tk = 0; tk < tiles; ++tk) {
            const std::size_t baseA_tile = (ti * tiles + tk) * M2;
            const std::size_t baseB_tile = (tk * tiles + tj) * M2;
            // The next three loops are parallelized by these threads.
            ...
```

- Each CUDA block works on one pair of (`ti`, `tj`).
- Threads in the same block collaborate on all `tk` steps.
  - For each step, they reuse the same tile of $\boldsymbol{A}$ and $\boldsymbol{B}$.
- That means data loaded by one thread must become visible to the other threads in the same block.

Tiled GEMM Kernel: Shared Tiles

```
__global__ void gemm_tiled(const float* A, const float* B, float* C, int N) {
  extern __shared__ float shared[];
  const int M = static_cast<int>(blockDim.x);

  float* tile_A = shared;
  float* tile_B = shared + M * M;
  ...
}
```

- All threads in a block access the same tile of A and B.
  - Shared memory `extern __shared__` provides on-chip buffer visible to all threads in the block.
- Each CUDA kernel can name at most one dynamically allocated shared memory buffer.
  - You can name the buffer anyway you like, e.g. `shared` here.
  - Use pointer arithmetic to slice the buffer into desired number of arrays. Be careful with pointer types.
  - We will allocate the buffer later at the kernel launch.

Block and Thread Mapping

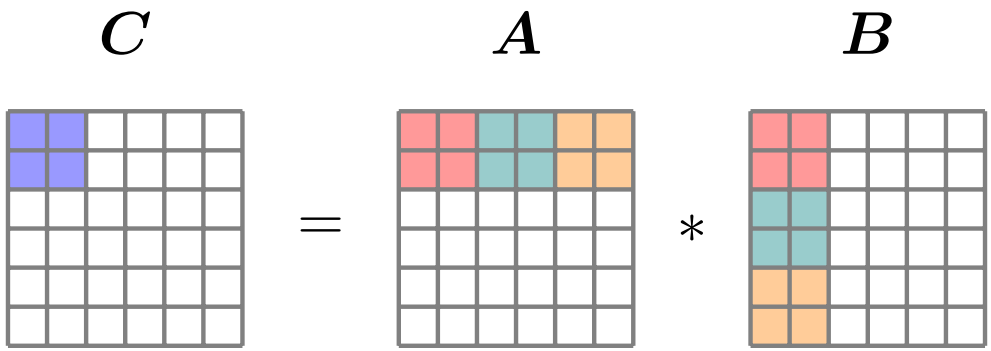
```
const int ti = static_cast<int>(blockIdx.y);
const int tj = static_cast<int>(blockIdx.x);
const int ii = static_cast<int>(threadIdx.y);
const int jj = static_cast<int>(threadIdx.x);
const int row = ti * M + ii;
const int col = tj * M + jj;

float sum = 0.0f;
const int tiles = (N + M - 1) / M;

for (int tk = 0; tk < tiles; ++tk) {
  ...
}
```

- The block and thread mapping is the same as `gemm_simple`
  - `ti` and `tj` identify which output tile of C this block computes.
  - `ii` and `jj` identify one thread's position inside the tile.
  - Each thread computes one element of C at `row` and `col`.
- Unlike the earlier CPU tiled example, our CUDA kernel keeps A, B, and C in row-major storage for simplicity.

What Should Each Thread Load?



- However, we would like to optimize the memory access pattern for these threads based on block GEMM.
- For one `tk` step, these threads operate on data from the same pair of tiles from $\boldsymbol{A}$ and $\boldsymbol{B}$.
  - With a tile size of $M \times M$, there will be $M \times M$ threads.
  - On average, each thread should at least read one element each from the pair of tiles from $\boldsymbol{A}$ and $\boldsymbol{B}$.
- Let each thread load one element from $\boldsymbol{A}$ and one from $\boldsymbol{B}$.
  - The `gemm_simple` kernel loads M times more, while coalesced memory access may save traffic depending on block shape.

Loading A into Shared Memory

```
// const int jj = static_cast<int>(threadIdx.x);
for (int tk = 0; tk < tiles; ++tk) {
  const int col_A = tk * M + jj;
  if (row < N && col_A < N) {
    tile_A[ii * M + jj] = A[row * N + col_A];
  } else {
    tile_A[ii * M + jj] = 0.0f;
  }
  ...
}
```

- At a fixed `tk`, each thread loads one element of A.
  - Into an element of the shared `tile_A`.
  - The element of A is located at the same `row` as the element of C the thread computes, and at the column `col_A` depending on `tk` its column `jj` within the block.
- A warp of threads use neighboring `jj`.
  - Global memory loads from A may be coalesced into the same cache line. Very efficient!

Loading B into Shared Memory

```
for (int tk = 0; tk < tiles; ++tk) {
  ...
  const int row_B = tk * M + ii;
  if (row_B < N && col < N) {
    tile_B[ii * M + jj] = B[row_B * N + col];
  } else {
    tile_B[ii * M + jj] = 0.0f;
  }
  ...
}
```

- The indices look different because this tile comes from B.
- However, the access pattern is the same as that of A.
  - A warp of threads use neighboring `jj`.
  - Global memory loads from B may be efficiently coalesced into the same cache line.

Can We Start Computing Immediately?

```
for (int kk = 0; kk < M; ++kk) {
  sum += tile_A[ii * M + kk] * tile_B[kk * M + jj];
}
```

- After the loads, each thread should update its own `sum`.
  - Use the row from $\boldsymbol{A}$ in `tile_A`.
  - Use the column of $\boldsymbol{B}$ in `tile_B`.
  - But since each thread only loads two elements, are all the data ready by now in `tile_A` and `tile_B`?
- Yes if the data is loaded by threads in the same warp.
  - Threads in the same warp always execute the same instruction.
- No if the data is loaded by threads in different warps.
  - A slower warp of threads are still loading data, while a faster warp of threads read them.
  - Now the faster warp of threads read stale or undefined data.

Synchronization via Barrier

```
for (int tk = 0; tk < tiles; ++tk) {
  ... // load A into tile_A and B into tile_B
  __syncthreads();
  for (int kk = 0; kk < M; ++kk) {
    sum += tile_A[ii * M + kk] * tile_B[kk * M + jj];
  }
}
```

- `__syncthreads()` is a barrier for all threads in a block.
  - A thread will wait when it reaches `__syncthreads()`,
  - Until all threads reach `__syncthreads()`.
- This barrier guarantees that all shared-memory writes for the current tile finish before the reads to compute `sum`.
- Do we need more barriers?
  - What if faster warps finish computing `sum`, proceed to the next `tk`, and load the next `tile_A` and `tile_B`, while slower warps still computing the current `sum`?
  - Now the slower warps read future data which is incorrect.

Complete Tiled GEMM Kernel

```
for (int tk = 0; tk < tiles; ++tk) {
  ... // load A into tile_A and B into tile_B
  __syncthreads();
  for (int kk = 0; kk < M; ++kk) {
    sum += tile_A[ii * M + kk] * tile_B[kk * M + jj];
  }
  __syncthreads();
}
if (row < N && col < N) C[row * N + col] = sum;
```

- Use two barriers.
  - The first barrier guarantees reads to only happen after all writes finish for the same `tk`.
  - The second barrier guarantees writes to only happen after all reads finish for a previous `tk`.
- Each thread updates its C element as needed.
- While CPU barriers synchronize threads among cores and thus are costly, `__syncthreads()` synchronizes threads running on the same SM and is implemented on hardware efficiently.

Launching Kernel with Dynamic Shared Memory

```
dim3 dim_threads(M, M);
dim3 dim_blocks((N + M - 1) / M, (N + M - 1) / M);
const size_t shared_bytes = 2ULL * static_cast<size_t>(M * M) * sizeof(float);
gemm_tiled<<<dim_blocks, dim_threads, shared_bytes>>>(d_A, d_B, d_C, N);
```

- CUDA allows to use the third launch parameter to specify the size of dynamic shared memory.
  - No need to allocate or deallocate it manually.
- Recall we need space for two `M x M` tiles `tile_A` and `tile_B`.
- Also note how we pass `M` as `dim_threads` and retrieve it in the kernel to avoid mismatched tile size and thread numbers.

Experimental Results: Tiled GEMM

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

tile size M	N=4096	N=8192	N=16384
8	0.38/361.36	3.30/333.28	31.96/275.19
16	0.25/545.25	2.12/518.90	20.91/420.57
32	0.20/693.26	1.65/667.77	14.53/605.50

- For $M = 32$, results from tiled version are substantially faster than the simple GEMM kernel.
  - `gemm_simple` never reaches 600 GFLOPS.
  - Performance decreases as N increases, mostly due to L2 misses.
  - Less performance loss for $N = 16384$.
- Larger tiles perform better.
  - Global memory accesses are efficiently coalesced when a warp of threads reads/writes the same row.
  - Can we use 64×64 tiles? e.g. for $M = 64$ or larger?

Outline

Increase Work per Thread

```
sum += tile_A[ii * M + kk] * tile_B[kk * M + jj];
```

- Using `tile_A` and `tile_B` reduces the total traffic between global memory/L2 cache and shared memory.
  - Still, SMs need to load data to registers to compute `sum`.
- Computation vs communication
  - Registers/shared memory communication: 2 loads.
  - Computation: 1 multiplication, 1 addition.
  - A 1:1 ratio.
- We cannot reduce computation, but can we reduce loads?
  - Keep `tile_A[ii * M + kk]` in register longer and reuse it to increase the computation to communication ratio.

Register Blocking

```
// sum += tile_A[ii * M + kk] * tile_B[kk * M + jj];
const float a = tile_A[ii * M + kk];
sum0 += a * tile_B[kk * (2 * M) + jj];
sum1 += a * tile_B[kk * (2 * M) + M + jj];
```

- Allow each thread to work with more data.
  - Make it possible to reuse data in registers.
- 1×2 Register Blocking
  - Each thread computes two elements of $\boldsymbol{C}$.
  - Each block of threads reads 1 tile of $\boldsymbol{A}$ and 2 tiles of $\boldsymbol{B}$ per `tk`, and computes 2 tiles of $\boldsymbol{C}$ at the end.
- Computation vs communication
  - Registers/shared memory communication: 3 loads.
  - Computation: 2 multiplications, 2 additions.
  - A 4:3 ratio. More computation per communication.

1x2 Block and Thread Mapping

```
const int ti = static_cast<int>(blockIdx.y);
const int tj = static_cast<int>(blockIdx.x);
const int ii = static_cast<int>(threadIdx.y);
const int jj = static_cast<int>(threadIdx.x);
const int row = ti * M + ii;
const int col0 = tj * (2 * M) + jj;
const int col1 = col0 + M;
```

- Each block still has $M \times M$ threads.
- Each block now computes 2 tiles of C.
  - Identified as (`ti`, `tj`*2) and (`ti`, `tj`*2+1)
  - Each thread computes `C[row, col0]` and `C[row, col1]`.

Larger Shared Tile for B

```
float* tile_A = shared; // this array is still M*M
float* tile_B = shared + M * M; // this array is M*2M now
...
for (int tk = 0; tk < tiles; ++tk) {
  ...
  const int row_B = tk * M + ii;
  if (row_B < N && col0 < N) {
    tile_B[ii * (2 * M) + jj] = B[row_B * N + col0];
    ...
  }
  if (row_B < N && col1 < N) {
    tile_B[ii * (2 * M) + M + jj] = B[row_B * N + col1];
    ...
  }
  ...
}
```

- Each block needs 2 tiles of B now.
- `tile_B` becomes an $M \times 2M$ array.
- Each thread loads two elements of B.
- Spread the two loads in two tiles so that a warp of threads still access neighboring elements at a time.

Reuse Data in Registers

```
float sum0 = 0.0f;
float sum1 = 0.0f;
for (int tk = 0; tk < tiles; ++tk) {
  ...
  __syncthreads();
  for (int kk = 0; kk < M; ++kk) {
    const float a = tile_A[ii * M + kk];
    sum0 += a * tile_B[kk * (2 * M) + jj];
    sum1 += a * tile_B[kk * (2 * M) + M + jj];
  }
  __syncthreads();
}
if (row < N && col0 < N) C[row * N + col0] = sum0;
if (row < N && col1 < N) C[row * N + col1] = sum1;
```

- The synchronization pattern does not change.
- Save one load of `tile_A[ii * M + kk]`
- At the end, each thread writes two output elements.

Launching the 1x2 Kernel

```
dim3 dim_threads(M, M);
dim3 dim_blocks((N + 2 * M - 1) / (2 * M), (N + M - 1) / M);
const size_t shared_bytes = 3ULL * static_cast<size_t>(M * M) * sizeof(float);
gemm_tiled_1x2<<<dim_blocks, dim_threads, shared_bytes>>>(d_A, d_B, d_C, N);
```

- `dim_threads` remains unchanged (`M x M`).
- The grid is different now.
  - Each block computes $2M$ columns and M rows.
  - So there are only half the number of blocks along the x direction.
- Shared memory now needs `3*M*M` floats.
  - 1 tile for $\boldsymbol{A}$ and 2 tiles for $\boldsymbol{B}$.

Experimental Results: 1x2 Register Blocking

Table: `gemm_tiled_1x2` vs. `gemm_tiled` time(s)/GFLOPS, warm-up+single run, data transfer time excluded, on one Tesla T4 GPU.

kernel / M	N=4096	N=8192	N=16384
tiled, 8	0.38/361.36	3.30/333.28	31.96/275.19
1x2, 8	0.30/454.36	2.63/417.71	25.63/343.23
tiled, 16	0.25/545.25	2.12/518.90	20.91/420.57
1x2, 16	0.19/721.49	1.60/688.75	16.33/538.77
tiled, 32	0.20/693.26	1.65/667.77	14.53/605.50
1x2, 32	0.15/918.41	1.23/891.72	10.84/811.73

- For every matrix size and every tile size, `gemm_tiled_1x2` is much faster than `gemm_tiled`.
- What about more aggressive register blocking?
  - And other optimizations?

Summary

- Shared memory lets threads in the same block cooperatively access and reuse data.
- The `__syncthreads()` barrier synchronizes threads in a block and guarantees the correctness regarding the ordering of data loads and stores.
- Register blocking increases computation to communication ratio and improves performance further.