ECE 587 – Hardware/Software Co-Design
Lecture 18 CUDA II: Memory Access

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

March 11, 2026

Outline

Reading Assignment

- This lecture: CUDA Memory Access
- Next lecture (3/23): CUDA Synchronization

Outline

Recap: Dependence of Performance on Shape

Table: `gemm_simple` kernel at N=8192 (time(s)/GFLOPS)

shape	N=8192
(64,16)	2.05/537.26
(32,32)	1.92/573.14
(16,64)	2.25/489.65
(64,4)	5.88/187.11
(32,8)	2.69/408.22
(16,16)	2.76/398.47

- Better performance when there are more threads per block.
- Blocks of threads perform differently as shape changes.
  - Same amount of computation per block if the number of threads per block is the same.
  - Shape choices must have changed memory access patterns and resulted in different communication costs.

Warp Execution

- Unlike CPU cores, CUDA cores are so simple that they cannot independently execute different instructions.
- Instead, a CUDA SM controls its cores to execute the same instruction across a group of 32 threads called a warp.
  - A block of threads is divided into warps.
  - E.g. a shape of $(16, 16)$ has 256 threads and thus 8 warps.
- Threads in the same warp execute the same instruction.
  - What if there is a branch? Not a concern for GEMM in general, but we will discuss this in more details later.
- For memory instructions, threads in the same warp issue memory requests together.
  - Huge opportunity for CUDA hardware to optimize memory access as long as the kernel makes it possible.

Warp Execution in Simple GEMM

```
// int row = blockIdx.y * blockDim.y + threadIdx.y;
// int col = blockIdx.x * blockDim.x + threadIdx.x;
sum += A[row * N + k] * B[k * N + col];
```

- A warp of threads execute the same set of instructions to update their own `sum` for a particular `k`.
- For 2D mapping, threads in the same warp have neighboring `threadIdx.x` values before `threadIdx.y` values change.
  - E.g. for $(16, 16)$, the first wrap includes all threads with `threadIdx.y` being 0 or 1.
- Therefore, threads in the same warp
  - Read as many `A[row * N + k]` as `threadIdx.y` varies.
  - Read neighboring `B[k * N + col]` from the same row.

Coalesced Memory Access

- CUDA uses cache to improve global memory performance.
  - One L1 cache per SM with 128-byte cache lines.
  - One L2 cache shared among all SMs.
- Coalesced memory access: when multiple threads in a warp request memory locations within a cache line, CUDA hardware will combine them into
  - If L1 hits (contains the data), one L1 transaction.
  - Otherwise, up to four transactions to L2 and global memory.

Hide Memory Access Latency

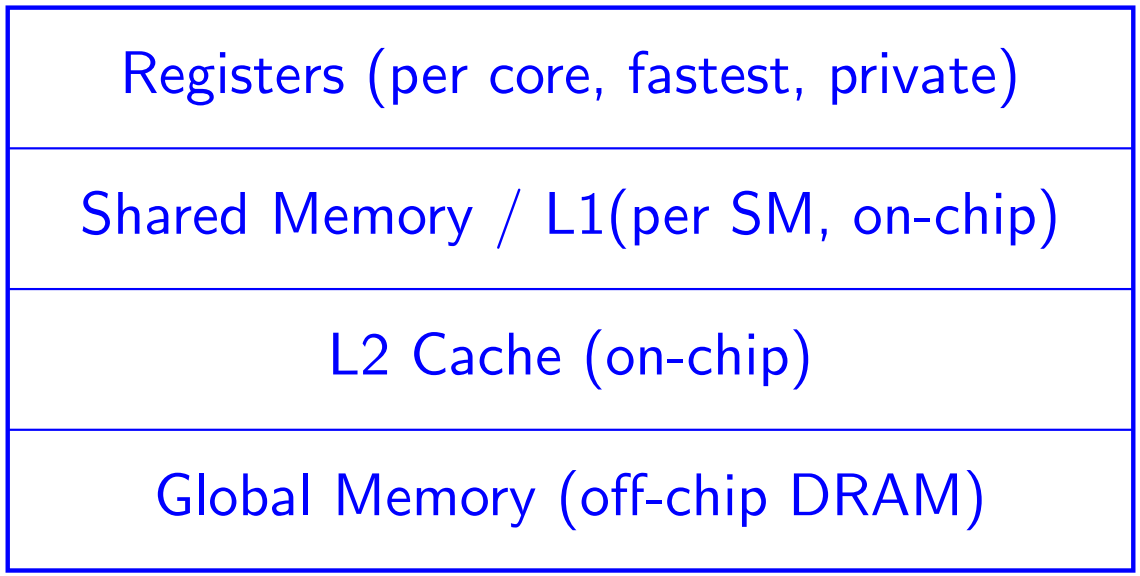
- With coalesced memory access, a warp of threads may still need to wait for the data to become available.
- Instead of letting cores waiting, SM will schedule another warp in the same block.
- Eventually this new warp may need to wait for its data.
- So SM will need to find a third warp to make cores busy.
- This is only possible if we have a lot of threads and thus a lot of warps per block.

Understand Simple GEMM Performance

- For a warp of threads in our simple GEMM kernel,
  - One transaction per `row` for `A[row * N + k]`.
  - One transaction for all `B[k * N + col]` if L1 hits.
- As SM schedules more warps, they will work on similar `k`.
  - Help to improve L1 hit rates in particular for $\boldsymbol{B}$.
- $(32, *)$ shapes perform better
  - Each warp of threads reads a cache line of $\boldsymbol{B}$ if L1 hits.
  - The whole cache line are useful since that's exactly 32 `float`s.
  - More threads per block help to hide memory access latency better so $(32, 32)$ is the best.
- For $(16, *)$ shapes, only half of the cache line is useful.
  - Less efficient memory transaction leads to worse performance.
- For $(64, *)$ shapes, each warp reads and makes full use of the cache line.
  - However, as warps in the same block now read two cache lines of data of $\boldsymbol{B}$, cache behavior changes so sometimes the performance is worse than $(16, *)$ shapes.

Outline

CUDA Memory Hierarchy



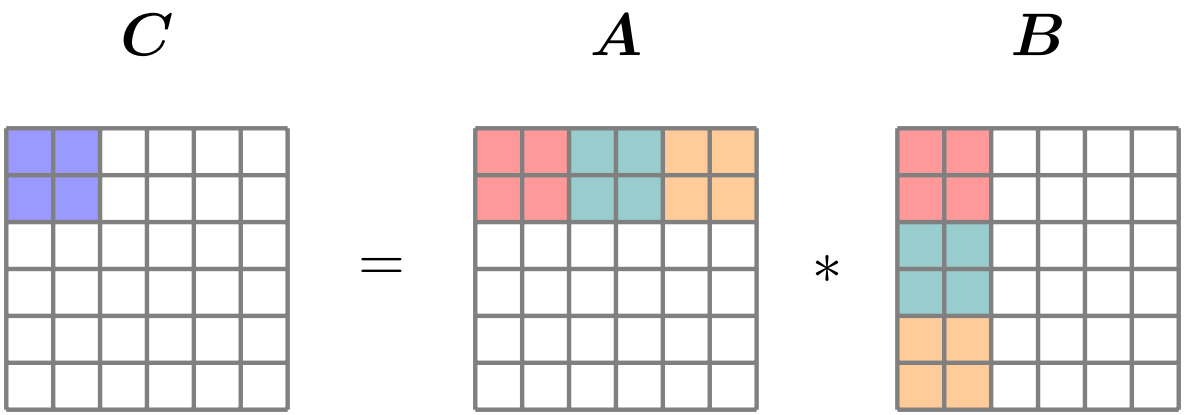
- Top levels (register/shared/L1) are fast but small.
- Global memory is large but slow.
- While recent versions of CUDA hardware allow developers to have more control over L2, we will focus on optimizing accesses at top levels.

Simple GEMM Kernel Memory Accesses

```
for (int k = 0; k < N; ++k) {
  sum += A[row * N + k] * B[k * N + col];
}
```

- Focus on total traffic and efficiency.
  - Assume there are enough threads to hide latency.
- For shape of $(32, 32)$,
  - Consider coalesced memory accesses.
  - $\boldsymbol{A}$ is accessed $\frac{N}{32}$ times, with a 25% efficiency since each element needs one 32-byte transaction.
  - $\boldsymbol{B}$ is accessed N times, with 100% efficiency.
  - Both may hit different levels within the memory hierarchy, result in different performance numbers.
- Overall cache misses increase as N increase.
  - GFLOPS drop as N increases, in particular for $N = 16384$.

Can Tiling Help Again?



- Each CUDA block still computes a submatrix of C.
- Make CUDA threads to work on a subset of `k` at a time,
  - Modify accesses to A so they can be coalesced.
  - Ensure the elements from submatrices of A, B, C to stay in L1 for higher bandwidth.

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.
            ...
```

- Synchronize threads in a block so they work on the same set of `k`.
- Have explict control over L1 to store submatrices of $\boldsymbol{A}$, $\boldsymbol{B}$, and $\boldsymbol{C}$.
  - Indeed, CUDA allows to repurpose L1 into shared memory for individual SMs so developers can manually optimize memory traffic for all threads in a block.

Summary

- Block shape strongly affects memory behavior and communication cost.
- Memory coalescing and layout-aware mapping are key for GPU performance.
- While simple GEMM kernel is able to leverage some memory optimizations, tiling is the next step to improve GEMM performance.