ECE 587 – Hardware/Software Co-Design
Lecture 04 General Matrix Multiplication

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

January 23, 2026

Outline

Reading Assignment

- This lecture: General Matrix Multiplication (GEMM)
- Next lecture: 3.1

Outline

General Matrix Multiplication (GEMM)

$$\boldsymbol{C} = \boldsymbol{A}\boldsymbol{B} + \boldsymbol{D}$$

- Dimensions
  - $\boldsymbol{C}$, $\boldsymbol{D}$: $M \times N$ matrix
  - $\boldsymbol{A}$: $M \times K$ matrix
  - $\boldsymbol{B}$: $K \times N$ matrix
- For simplicity, assume $\boldsymbol{D} = 0$ and $M = N = K$.

$$\begin{pmatrix} c_{11} & c_{12} & \dots & c_{1N} \\ c_{21} & c_{22} & \dots & c_{2N} \\ \dots & \dots & \dots & \dots \\ c_{N1} & c_{N2} & \dots & c_{NN} \end{pmatrix} = \begin{pmatrix} a_{11} & a_{12} & \dots & a_{1N} \\ a_{21} & a_{22} & \dots & a_{2N} \\ \dots & \dots & \dots & \dots \\ a_{N1} & a_{N2} & \dots & a_{NN} \end{pmatrix} \begin{pmatrix} b_{11} & b_{12} & \dots & b_{1N} \\ b_{21} & b_{22} & \dots & b_{2N} \\ \dots & \dots & \dots & \dots \\ b_{N1} & b_{N2} & \dots & b_{NN} \end{pmatrix}$$

- How to write code to compute $\boldsymbol{C} = \boldsymbol{A}\boldsymbol{B}$? And efficiently?

Matrix Storage

- Map from a 2-D matrix to a 1-D array in memory.
- Row-major order: store a matrix row by row.
  - Let's use 0-based indexing for arrays and matrices to match that of most programming languages.
  - Allocate an array of size $N * N$ for matrix $\boldsymbol{A}$, and then a_{ij} for $0 \leq i, j < N$ is stored at the index $i * N + j$.
- There are other ways to store matrices.
  - E.g. column-major order that stores a matrix column by column.
  - We may prefer to store different matrices in different orders for different operations.
  - Practically, we assume all matrices are stored in the same way so that we don't underestimate the cost to convert from one order to another.

A Simple Implementation

```
void gemm_ijk(const float* A, const float* B, float* C, std::size_t N) {
    std::fill(C, C + N * N, 0.0f);
    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;
        }
    }
}
```

- Let's assume elements are 4-byte floating point numbers.
- Use three loops, usually known as the ijk ordering.
  - i refers to rows of $\boldsymbol{A}$, j refers to columns of $\boldsymbol{B}$, k refers to the inner-product
  - The inner loop computes the inner-product of a row in $\boldsymbol{A}$ and a column in $\boldsymbol{B}$ to obtain an element in $\boldsymbol{C}$.

Performance Analysis

- For $N = 1024$, the ijk loops take a few seconds to run.
  - With proper compiler optimizations.
- How many operations are performed?
  - N^3 floating point multiplications and additions.
  - For $N = 1024$, 1 billion multiply-accumulate (MAC)
  - Compilers are powerful enough to optimize integer index calculations so they don't count.
- Assume it completes in 3 seconds and the CPU runs at 2GHz.
  - Each MAC takes 3ns, which is 6 clock cycles.
  - This includes time for the other part of the loops.
  - Also note that this is a measure of throughput (not latency).
- How good is the performance?
  - Modern CPUs can issue multiple instructions per cycle and process multiple data per instruction.
  - Is it possible to speed up by 10 times or more so that multiple MAC can be completed in each cycle?

Outline

The Memory System

- Obviously, to compute a MAC, we need to read two operands
  - The cost of writing back the result can be amortized as we only need to write once after $N = 1024$ MACs.
- If both operands are read from the main memory, this could easily take 100 cycles.
  - No, we are doing MUCH better than that.
- The cache heirarchy helps a lot.
  - It takes much less time to read from cache.
  - Depending on where in the heirarchy the data becomes available, it takes from few cycles to 10s of cycles.
  - For $N = 1024$, $\boldsymbol{A}$ and $\boldsymbol{B}$ are 4MB each, and can entirely fit into the cache.
- However, if we would like to achieve multiple MACs per cycle, this doesn't seem enough.

Memory Access Optimizations

- Spatial locality and cache line
  - Each time data is read from memory to cache, a fixed-size size block named cache line, usually 64 bytes, is fetched.
  - Accessing more data in the same cache line hits the cache and doesn't need more memory accesses.
- Instruction-level parallelism (ILP)
  - An instruction can proceed before a previous instruction completes, as long as it doesn't depend the results.
  - Help to hide memory access latency by computing the current MAC and reading data for the next MAC at the same time.
- Prefetching
  - CPU can predict future memory accesses and fetch data into cache before instructions actually requesting them.
  - Work well for regular memory access patterns like that in GEMM and other dense matrix operations.
  - Hide memory access latency further.

Analyzing Memory Access for ijk Loops

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

- In the inner loop, we need to access row i of $\boldsymbol{A}$ and column j of $\boldsymbol{B}$.
  - Compilers should be smart enough to optimize the dependency on `sum` so that the elements can be fetched in parallel.
- For row i of $\boldsymbol{A}$, i.e. $A[i * N + k]$
  - Perfect spacial locality as k iterates from 0 to $N - 1$ – the whole cache line can be utilized.
  - Prefetching can easily predict future accesses along the row.
  - Most likely the memory access latency is completly hidden.

Analyzing Memory Access for ijk Loops (Cont.)

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

- For column j of $\boldsymbol{B}$, i.e. $B[k * N + j]$
  - Almost no spacial locality – access 4 bytes in every cache line.
  - Prefetching should work, but bandwidth is wasted.
  - Most likely the 3ns (6 cycles) used by each MAC is spent on fetching the column of $\boldsymbol{B}$ from the cache heirarchy.
  - Indeed, with 64 bytes cache line, this corresponds to about 22GB/s bandwidth – but only 1/16 is used.
- Can we optimize the memory access pattern for $\boldsymbol{B}$?

Outline

Loop Reordering

```
void gemm_ikj(const float* A, const float* B, float* C, std::size_t N) {
    std::fill(C, C + N * N, 0.0f);
    for (std::size_t i = 0; i < N; ++i) {
        for (std::size_t k = 0; k < N; ++k) {
            float a_ik = A[i * N + k];
            for (std::size_t j = 0; j < N; ++j) {
                C[i * N + j] += a_ik * B[k * N + j];
            }
        }
    }
}
```

- What if we reorder the loops to ikj?
- The inner loop updates a row of $\boldsymbol{C}$ from a row in $\boldsymbol{B}$ and an element of $\boldsymbol{A}$.
- How does the memory access pattern change?

Analyzing Memory Access for ikj Loops

```
for (std::size_t j = 0; j < N; ++j) {
    C[i * N + j] += a_ik * B[k * N + j];
}
```

- Access to `a_ik` is amortized for N MAC but need read row k of $\boldsymbol{B}$ and read/write row i of $\boldsymbol{C}$.
- For row k of $\boldsymbol{B}$, i.e. $B[k * N + j]$
  - Similar to that of row i of $\boldsymbol{A}$ in the ijk loops, most likely the memory access latency is completly hidden.
- For row i of $\boldsymbol{C}$, i.e. $C[i * N + j]$
  - Each element along the row is written back right after it is read – the operation will happen in the cache.
  - There will be some additional cost to write back to main memory but it could be further hidden.
  - Overall most memory access latency could be hidden.

Performance Improvement

- For $N = 1024$, the ikj loops take a fraction of second to run.
  - With proper compiler optimizations.
  - More than 10 times faster than the ijk loops.
- Assume it completes in 125ms and the CPU runs at 2GHz.
  - Each MAC takes 0.125ns, which is 1/4 clock cycle.
  - Indeed, SIMD (Single Instruction Multiple Data) instructions can work on 4 4-byte floating point numbers at a time, making it possible to update 4 elements of a row of $\boldsymbol{C}$ from 4 elements of a row of $\boldsymbol{B}$ every cycle.
- Can we improve the performance further?
  - Other loop ordering?
  - Optimize memory access pattern further?
  - Make better use of CPU or other hardware devices?

Summary

- Optimizing software depends on understanding of the underlying hardware.
- Communication cost cannot be ignored.
- What about computations other than GEMM?