CS 6210: Scientific and Data Computing I Fall 2024

Assignment 3

Milena Belianovich
November 25, 2024

For this assignment, various CUDA versions are to be implemented for transposed matrixmultiplication, whose sequential C code is shown below. For all the versions, use a 1D 256 x 1 thread-block. To ease the implementation effort, the codes only need to pass the correctness test for a fixed problem size of 1024. Since CHPC GPUs are under a heavy load, use CADE Lab for this assignment.

Transposed-Transposed Matrix-Matrix Multiplication ($C = A^T B^T$):

```
for (i=0;i<1024;i++)
    for (k=0;k<1024;k++)
        for (j=0;j<1024;j++)
        // C[i][j] += A[k][i]*B[j][k];
        C[i*1024+j] += A[k*1024+i]*B[j*1024+k];
```

1 Problems

1. (20 points) Replace "FIXMEs" in mmtt.cu to create a version that passes the correctness test. It should use 256 x 1 thread blocks and a distinct thread to compute each output element of C and achieve coalesced access of A and B from global memory. Expected performance on CADE: around 180 GFLOPs.

   Solution.

   Implementation.

   The kernel computes one element of the result matrix 'C' per thread, based on the formula:

$$C[i, j] = \sum_{k=0}^{n} A[k, i] \cdot B[j, k]$$

   Key Features:

   (a) Thread Mapping: each thread computes one element of C using its threadIdx and blockIdx to determine its position in the matrix.

(b) Memory Access: global memory reads for A and B are structured to ensure coalescing.

(c) Thread Block: configured as 256×1 as per the problem requirements.

Results.

The implementation was tested on CADE for a matrix size of 1024×1024. The following results were observed:

Trial	Elapsed Time (ms)	GFLOPS
1	23.830879	90.11
2	12.137920	176.92
3	11.520352	186.41

Analysis:

(a) Correctness: the output matched the reference results computed on the CPU (h Cref).

(b) Performance: the performance improved over successive trials, reaching a peak of 186.41 GFLOPS, surpassing the expected 180 GFLOPS.

Conclusion.

The implementation meets the problem's requirements:

1. Each thread computes a distinct element of C.
2. Coalesced memory access was achieved for A and B.
3. The implementation exceeds the expected performance target of 180 GFLOPS.

2. (10 points) Reverse the mapping from the thread-grid to output data space from the above. What performance differences are observed?

Solution.

Implementation.

The kernel was updated to reverse the thread-to-output mapping. Instead of assigning rows to threads first, columns were assigned as follows:

$$\text{int col} = \text{blockIdx.x} * \text{blockDim.x} + \text{threadIdx.x};$$

$$\text{int row} = \text{blockIdx.y} * \text{blockDim.y} + \text{threadIdx.y};$$

This swapped the roles of rows and columns compared to Problem 1. The rest of the kernel logic and global memory accesses remained unchanged.

Results.

The implementation was tested on CADE for a matrix size of 1024×1024. The following results were observed:

Trial	Elapsed Time (ms)	GFLOPS
1	147.097626	14.60
2	60.045055	35.76
3	56.455040	38.04

Analysis:

(a) Performance Drop: the reversed mapping caused a significant drop in performance compared to Problem 1, where the best GFLOPS reached 186.41. In this case, the peak performance was 38.04 GFLOPS.

(b) Thread Mapping Impact: the reversed thread-grid mapping resulted in irregular access patterns to A and B in global memory, leading to increased memory transaction overhead and slower performance.

Conclusion.

Reversing the thread-grid mapping negatively impacted performance due to inefficient memory access patterns. This experiment demonstrates the importance of designing thread-grid mappings that optimize memory coalescing for high-performance CUDA applications.

3. (10 points) Starting with the version of mmtt.cu (from part 1) that achieves the expected performance, implement a version that performs 4-way unrolling along k in mmtt k4.cu. Is any performance improvement achieved over mmtt?

Solution.

Implementation.

The kernel was modified to perform 4-way unrolling along the k-dimension. Instead of processing one iteration at a time, the loop was unrolled to process four iterations in a single step. Any remaining iterations (when k is not divisible by 4) were handled separately. The updated kernel is shown below:

```
for (k = 0; k <= ds - 4; k += 4) {
    value += A[k * ds + row] * B[col * ds + k];
    value += A[(k + 1) * ds + row] * B[col * ds + (k + 1)];
    value += A[(k + 2) * ds + row] * B[col * ds + (k + 2)];
    value += A[(k + 3) * ds + row] * B[col * ds + (k + 3)];
}
// Handle remaining iterations
for (; k < ds; k++) {
    value += A[k * ds + row] * B[col * ds + k];
}
```

Results.

The implementation was tested on CADE for a matrix size of 1024×1024. The following results were observed:

Trial	Elapsed Time (ms)	GFLOPS
1	30.676031	70.01
2	10.881504	197.35
3	10.698112	200.73

Analysis:

(a) Correctness: the output matched the reference results computed on the CPU (h Cref).

(b) Performance: the performance improved significantly, achieving a peak of 200.73 GFLOPS, which exceeds the results from Problem 1 (peak 186.41 GFLOPS).

Conclusion.

The 4-way unrolling along the k-dimension resulted in a noticeable performance improvement. This optimization reduces loop overhead and enhances instruction-level parallelism, making better use of the GPU's computational resources.

4. (15 points) Implement a version that performs 4-way unrolling along j in mmtt j4.cu. Is performance improvement achieved over mmtt?

Solution.

Implementation.

The kernel was modified to perform 4-way unrolling along the j-dimension. Instead of processing one element of C per loop iteration, the loop over j was unrolled to compute four consecutive elements of C for the same row, as shown below:

```
for (j = 0; j < ds; j += 4) {
    float value0 = 0.0f, value1 = 0.0f, value2 = 0.0f, value3 = 0.0f;

    for (k = 0; k < ds; k++) {
        value0 += A[k * ds + row] * B[j * ds + k];
        value1 += A[k * ds + row] * B[(j + 1) * ds + k];
        value2 += A[k * ds + row] * B[(j + 2) * ds + k];
        value3 += A[k * ds + row] * B[(j + 3) * ds + k];
    }

    C[row * ds + j] = value0;
    if (j + 1 < ds) C[row * ds + (j + 1)] = value1;
    if (j + 2 < ds) C[row * ds + (j + 2)] = value2;
    if (j + 3 < ds) C[row * ds + (j + 3)] = value3;
}
```

Results.

The implementation was tested on CADE for a matrix size of 1024×1024. The following results were observed:

Trial	Elapsed Time (ms)	GFLOPS
1	108.290688	19.83
2	42.128834	50.97
3	42.189281	50.90

Analysis:

(a) Correctness: the output matched the reference results computed on the CPU (h Cref).

(b) Performance: the j-unrolling implementation achieved a peak performance of 50.97 GFLOPS, significantly lower than the original implementation (186.41 GFLOPS) and k-unrolling (200.73 GFLOPS).

Conclusion.

The 4-way unrolling along the j-dimension did not lead to significant performance improvement. The reason for this is likely due to memory access inefficiencies along the column-major indexing of B, which disrupt coalesced memory access patterns.

5. (15 points) Implement a version that performs 4-way unrolling along i in mmtt i4.cu. Is performance improvement achieved over mmtt?

Solution.

Implementation.

The kernel was modified to perform 4-way unrolling along the i-dimension. Instead of processing one row per loop iteration, the loop was unrolled to compute four rows of C for each column, as shown below:

```
for (int k = 0; k < ds; k++) {
    value0 += A[k * ds + (row_base + 0)] * B[col * ds + k];
    if (row_base + 1 < ds) value1 += A[k * ds + (row_base + 1)] *
    B[col * ds + k];
    if (row_base + 2 < ds) value2 += A[k * ds + (row_base + 2)] *
    B[col * ds + k];
    if (row_base + 3 < ds) value3 += A[k * ds + (row_base + 3)] *
    B[col * ds + k];
}
C[(row_base + 0) * ds + col] = value0;
if (row_base + 1 < ds) C[(row_base + 1) * ds + col] = value1;
if (row_base + 2 < ds) C[(row_base + 2) * ds + col] = value2;
if (row_base + 3 < ds) C[(row_base + 3) * ds + col] = value3;
```

Results.

The implementation was tested on CADE for a matrix size of 1024×1024. The following results were observed:

Trial	Elapsed Time (ms)	GFLOPS
1	438.59	4.90
2	380.87	5.64
3	378.97	5.67

Analysis:

(a) Correctness: the output matched the reference results computed on the CPU (h Cref).

(b) Performance: the i-unrolling implementation achieved a peak performance of 5.67 GFLOPS, significantly lower than the original implementation (186.41 GFLOPS) and k-unrolling (200.73 GFLOPS).

(c) Memory Access Patterns: the inefficiency arises from irregular and non-coalesced memory access patterns for A when processing multiple rows simultaneously.

Conclusion.

The 4-way unrolling along the i-dimension did not lead to significant performance improvement. Instead, it resulted in a drastic performance degradation.

6. (15 points) Implement a version that performs 4-way unrolling along i and j in mmtt i4j4.cu. Is performance improvement achieved over previous versions?

Solution.

Implementation.

The kernel performs 4-way unrolling along both the i- and j-dimensions of the output matrix C. Each thread is responsible for computing contributions for a 4x4 submatrix of C, as specified by the unrolling strategy. The kernel uses nested loops to accumulate contributions for all k-values.

Key Features:

(a) Thread Mapping: threads compute 4 rows and 4 columns of C, based on $threadIdx.x$, $threadIdx.y$, $blockIdx.x$, and $blockIdx.y$.

(b) Memory Access: the kernel accesses A and B directly from global memory, and stores results in C.

(c) Unrolling Strategy: contributions to a 4x4 submatrix are calculated in a single kernel invocation.

Results.

The implementation was tested on CADE for a matrix size of 1024×1024. The following results were observed:

Trial	Elapsed Time (ms)	GFLOPS
1	7183.276367	0.30
2	7170.738770	0.30
3	7173.199707	0.30

Analysis:

(a) Correctness: the kernel produces results that match the CPU-computed reference output.

(b) Performance: the performance is significantly lower than the baseline implementation (186.41 GFLOPS) due to poor memory access patterns and lack of shared memory optimization.

Conclusion.

The kernel implementation fulfills the requirements of 4-way unrolling along i and j, but it is highly inefficient.

7. (15 points) Implement a version that uses shared memory to buffer elements of A and B in mmtt sm.cu. No loop unrolling is required. Is any performance improvement achieved over mmtt?

Solution.

Implementation.

The kernel uses shared memory to buffer tiles of A and B, reducing global memory accesses and improving performance. Each tile of A and B is loaded into shared memory by threads in the block, and results are computed for the corresponding elements of C within that tile.

Key Features:

(a) Shared Memory: tiles of size 16×16 for A and B are buffered in shared memory, minimizing global memory access overhead.

(b) Thread Mapping: threads in a block collaboratively compute elements of C, using the corresponding tiles from A and B.

(c) Synchronization: each tile computation is synchronized using `__syncthreads()` to ensure consistent memory operations.

Results.

The implementation was tested on CADE for a matrix size of 1024×1024. The following results were observed:

Trial	Elapsed Time (ms)	GFLOPS
1	20.636032	104.06
2	3.754304	572.01
3	3.751776	572.39

Analysis.

(a) Correctness: The output matched the CPU-computed reference, validating the correctness of the implementation.

(b) Performance:

- The first trial showed 104.06 GFLOPS, which is better than the first trial of the baseline implementation (from Problem 1).
- Subsequent trials demonstrated a significant improvement, achieving 572 GFLOPS due to the reduced overhead of shared memory.
- The consistent performance across trials highlights the stability and efficiency of the shared memory optimization.

Conclusion.

The shared memory implementation successfully minimizes global memory access overhead, achieving substantial performance improvement over the baseline. The approach demonstrates the power of leveraging shared memory for matrix computations, achieving up to 572.39 GFLOPS.