CMOR 421 Homework 1: Serial Optimization

Hubert King

March 1st, 2024

1 Matrix-Matrix Multiplication (10 points)

1.1 Changes for Column Major Format

The following pseudocode illustrates the loop reordering for matrix-matrix multiplication where matrices A and B are stored in column-major format:

```
for j = 1 to n
    for k = 1 to n
        for i = 1 to n
            C[i, j] = C[i, j] + A[i, k] * B[k, j]
        end
    end
end
```

The outer loop over j iterates through each column of matrix B and column of matrix C. This ensures contiguous memory access for $B[k, j]$. The middle loop over k iterates through the columns of matrix A. It allows the elements $A[i, k]$ to be accessed sequentially, which is cache-efficient for column-major storage. The innermost loop over i computes the contributions of the elements of column k from matrix A and column j from matrix B to column j of matrix C. This loop fills up column j of C, one element at a time. The multiplication $A[i, k] * B[k, j]$ and the accumulation into $C[i, j]$ are performed for each element i in the current columns k and j, respectively. This computation benefits from the preceding loops' column-wise traversal.

1.2 Performance Expectation

Optimal performance is achieved when matrix access during multiplication aligns with the storage order, enhancing CPU cache efficiency. With **row-major storage**, multiplication like AB is typically faster since this format favors row-wise access. Conversely, with **column-major storage**, the operation $A^T B$ can be more efficient. This is because $A^T B$ involves accessing the columns of A (which correspond to rows of A^T) and columns of B sequentially, matching the column-major storage scheme and thus optimizing cache utilization. The contiguous memory access of the columns in both matrices reduces cache misses and leverages the cache line's full potential, accelerating the computation.

2 Optimizing Matrix-Matrix Multiplication (20 points)

2.1 Timing Generation

Timings for various matrix and block sizes are summarized in Tables 1 and 2. Overall, a block size of 8 shows the most significant performance improvement across all tested matrix sizes, indicating its effectiveness is independent of matrix size. This suggests that the choice of a block size of 8 is optimal for the range of matrix dimensions tested.

For our roofline plot, we use a block size of 8 and generate timings with matrix dimensions 2^i for $i = 4, \ldots, 10$. We convert timings (ms) to GFLOPs/second using $\frac{2n^3}{10^9 \times \text{runtime}}$.

Table 1: Timings (ms) for Naive Matrix-Matrix Multiplication

Block Size	n = 128	n = 256	n = 512
4	1.88	22.67	327.85
8	1.82	18.14	404.86
16	1.79	20.40	392.94
32	1.82	16.52	351.49
64	1.85	16.45	337.93

Table 2: Timings (ms) for Blocked Matrix-Matrix Multiplication

Block Size	n = 128	n = 256	n = 512
4	0.72	8.12	81.76
8	0.45	3.97	45.44
16	1.16	9.92	92.54
32	1.46	12.55	125.79
64	1.59	12.81	141.89

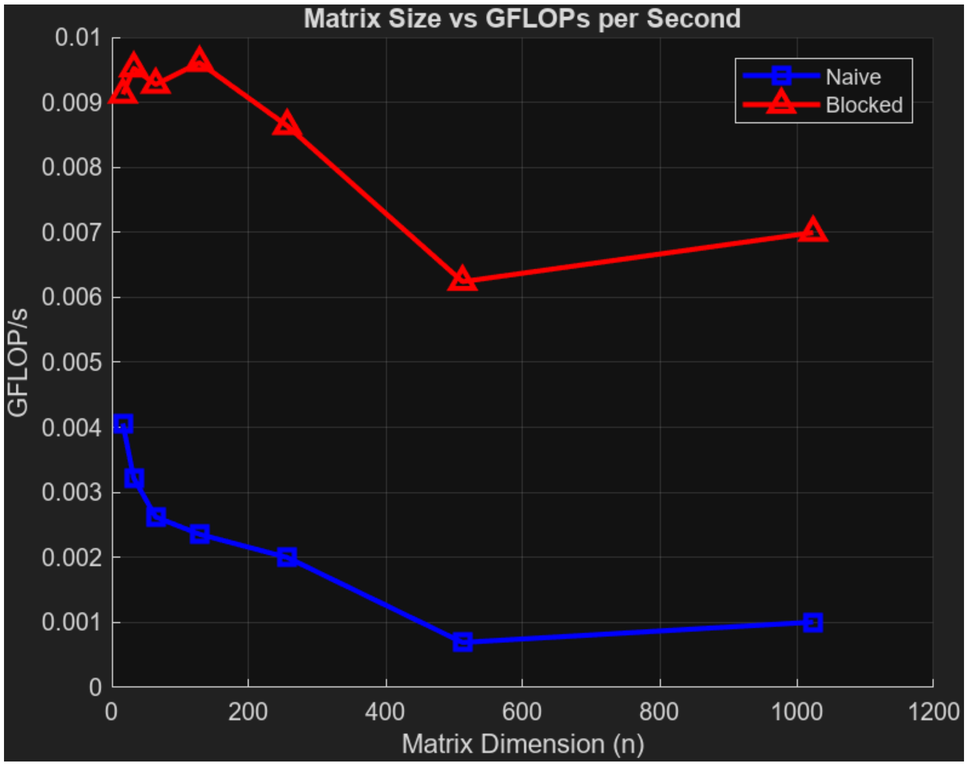

Figure 1: Roofline plot for Naive and Blocked Matrix-Matrix Multiplication on NOTS using block size of 8.

3 Recursive Matrix-Matrix Multiplication (70 points)

3.1 Implementation Details and Block Size Optimization

The *rmatmul* function performs recursive matrix multiplication. It divides the input matrices A and B into smaller sub-matrices and recursively multiplies them. Once the size of these sub-matrices reaches a predefined threshold denoted by `BLOCK_SIZE`, the multiplication for these smaller blocks is executed by a *microkernel* function. Optimal `BLOCK_SIZE` on NOTS was determined to be tie between 8 and 16.

3.2 Correctness Check

The correctness of the recursive matrix multiplication implementation is verified by setting both matrices $A = B = I$ and verifying $C = I$, considering the constraints of machine precision. C is reset to zero to ensure a clean state for the multiplication. The recursive matrix multiplication is then executed once using matrices A and B. To validate the result, the code iterates C element-wise, employing a tolerance level defined by the machine epsilon for floating-point numbers (`numeric_limits<float>::epsilon()`). It checks if the diagonal elements of C are approximately equal to 1.0, and the off-diagonal elements are approximately equal to 0.0, within the specified tolerance. This element-wise verification is conducted using the `almost_equal` function, which compares two floating-point numbers considering a given tolerance. If any element of C does not adhere to the expected values, the flag `is_identity` is set to `false`, and the verification process is terminated early. Finally, the result of this verification is displayed in the terminal.

3.3 Performance Timing and Roofline Plot

Timings for Recursive Matrix-Matrix multiplication using various matrix and block sizes are summarized in Table 3. Overall, a block sizes of 8 and 16 jointly had the best performance. For our roofline plot, we use a block size of 16 and generate timings with matrix dimensions 2^i for $i = 4, \ldots, 10$, from which to convert to GFLOPs/sec using the same conversion as previously.

Table 3: Timings (ms) for Recursive Matrix-Matrix Multiplication

Block Size	n = 128	n = 256	n = 512
4	1.54	12.84	116.40
8	1.40	11.25	96.11
16	1.28	10.52	95.20
32	1.55	16.21	153.99
64	1.67	13.03	146.88

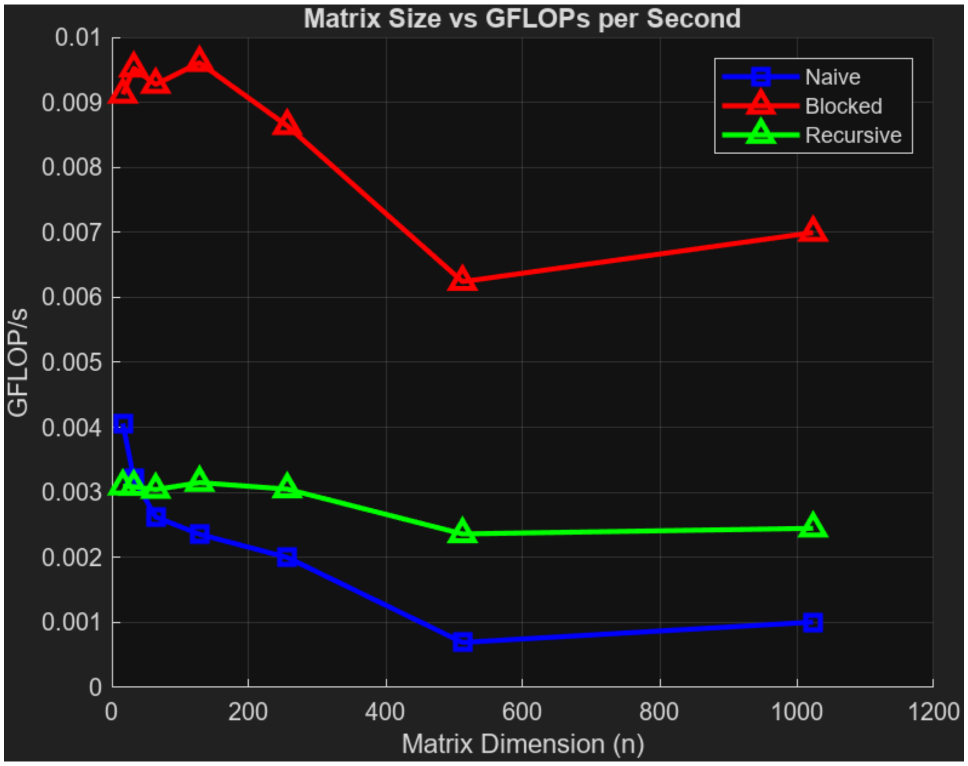

Figure 2: Roofline plot for Naive, Blocked, and Recursive Matrix-Matrix Multiplication on NOTS using block size of 16.

4 Build and Run Instructions

This section provides detailed instructions for compiling and executing rmatmul.cpp. The program is designed to work with matrices of a dimension specified by the user through a command-line argument.

4.1 Prerequisites

Ensure the following requirements are met:

- The instructions are intended for users on the NOTS system.
- GCC version 13.1.0 is necessary for successful compilation and execution of the rmatmul.cpp.

4.2 Loading the Required GCC Module

Access NOTS on an interactive node. To compile the program, the GCCcore/13.1.0 module must be loaded.

```
module load GCCcore/13.1.0
```

Verify that the module is loaded correctly and the correct version of GCC is being used:

```
gcc --version
```

The output should confirm that GCC version 13.1.0 is active.

4.3 Building the Program

To compile the program, ensure you are in the directory containing ‘main.cpp‘. Use the following command to compile the program, which incorporates source files from the ‘src‘ directory and includes header files from the ‘include‘ directory:

```
g++ -O3 -o main main.cpp src/functions.cpp -I./include
```

This command instructs the G++ compiler to optimize the compilation at level 3, produce an executable named ‘main‘ , compile together ‘main.cpp‘ and ‘src/functions.cpp‘, and look for header files within the ‘include‘ directory.

4.4 Running the Program

After successful compilation, the program can be executed by running the following command in the terminal, where dimension is replaced by the desired matrix size.

```
./main <dimension>
```

4.5 Example Usage

Before compiling and running the program, ensure you are on an interactive node or environment that has access to the necessary compilers. Here’s how you can compile and run the program for a matrix operation $A \times B = C$, where A, B, and C are identity matrices, and I is a matrix in $\mathbb{R}^{128\times128}$:

```
module load GCCcore/13.1.0
g++ -O3 -o main main.cpp src/functions.cpp -I./include
./main 128
```