Course Name

Homework Number

Tailai Zhang, AAAAAAAAAA

BBBBBBBBB, CCCCCCCCC

January 3, 2024

Please note that this document has been modified. Contents from the other contributors have been removed to reflect my personal contribution to the write-up.

Problem 1.1: Matrix Multiplication

Strassen's Algorithm

To further optimize matrix multiplication, we also attempted implementing Strassen's Algorithm, which is an advanced method for matrix multiplication that divides the input matrices into smaller submatrices, recursively applies the multiplication process, and then combines the results. This divide-and-conquer approach reduces the computational complexity from $O(n^3)$ in the standard matrix multiplication to $O(n^{\log_2 7})$. Formulae for Strassen's algorithm are listed below:

$$A \times B = C$$

$$A = \begin{pmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{pmatrix}, \quad B = \begin{pmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{pmatrix}, \quad C = \begin{pmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{pmatrix}$$

$$\begin{aligned}
M_1 &= (A_{11} + A_{22}) \cdot (B_{11} + B_{22}) \\
M_2 &= (A_{21} + A_{22}) \cdot B_{11} \\
M_3 &= A_{11} \cdot (B_{12} - B_{22}) \\
M_4 &= A_{22} \cdot (B_{21} - B_{11}) \\
M_5 &= (A_{11} + A_{12}) \cdot B_{22} \\
M_6 &= (A_{21} - A_{11}) \cdot (B_{11} + B_{12}) \\
M_7 &= (A_{12} - A_{22}) \cdot (B_{21} + B_{22}) \\
C_{11} &= M_1 + M_4 - M_5 + M_7 \\
C_{12} &= M_3 + M_5 \\
C_{21} &= M_2 + M_4 \\
C_{22} &= M_1 - M_2 + M_3 + M_6
\end{aligned}$$

Despite its improved asymptotic time complexity, in practice, Strassen's algorithm is always less efficient for especially smaller matrices. The recursive nature of the algorithm(i.e. many more function calls and memory allocation) and the additional matrix additions and subtractions introduced often make it less favorable compared to the straightforward approach for small matrices.

Given the varying efficiencies of Strassen's algorithm and naive multiplication across different matrix sizes, we attempted a hybrid approach instead. i.e. use Strassen's algorithm for large matrices, and use the optimized naive approach when the input matrix is below a certain smaller size. The crossover point for switching will be found empirically by testing our Strassen implementation compared to the optimized naive approach. Our implementation of Strassen's algorithm can be found in `strassen_matmul.c`

Strassen's Implementation

Some notable details for our Strassen's Implementation are listed below:

- In our implementation, for matrix size that is not a multiple of 2, matrices are padded to the nearest size of 2^n that is bigger than the input size. This ensures compatibility of the algorithm with arbitrary input size but creates significant overhead in memory allocation and calculations. Possible improvements are proposed in later sections. We did not implement those due to their complexity and the time constraints we are facing.
- Instead of the recursive down to element-wise calculation in the original Strassen's Algorithm, we choose to stop recursion at 4×4 matrices and use SIMD to directly calculate 4×4 matrix multiplication, this contributes significantly to improving running speed and reducing recursion overheads.
- When partitioning matrix to sub-matrices, we avoid memory allocations and instead create pointers to the relevant sections of the original matrices. This approach reduces the memory overhead and computational by copying and allocating new submatrices.
- An attempt was made to use a common buffer for each of the matrices m1 to m7 to reduce memory allocations. However, this led to challenges where, after recursion, some pointers incorrectly pointed to the same matrix, resulting in over-adding and over-subtracting that affected the result correctness, we were unable to resolve this issue.
- In calculating $A \times B$ in our code, we transposed B to B^T in advance so that we can have better cache performance, and by doing so, when doing 4×4 multiplication in the base case, columns of matrix B can be extracted with a single `_mm256_load_pd` operation. Note that the pre-transpose of B requires a slight adjustment to the original Strassen formula to maintain the correctness of calculation for the algorithm. This is why, if we examine the code carefully, the calculation sequence and matrices are slightly different from the original formula.

Comparison with Naive Result, and possible improvements

Unfortunately, our current implementation's efficiency is worse than that of the naive approach for matrix sizes ranging from 4 to 4096, and potentially beyond. We conclude the main reason to be the excessive memory overhead, the recursion nature of this algorithm, and the extra calculations that are introduced by padding. We propose possible improvements to our code as follows:

1. **Padding and Memory Usage:** Reducing the overhead from padding zeros is crucial. There are dynamic methods to handle odd-sized matrices proposed in research papers, like 'dynamic peeling' or 'dynamic overlap'.
2. **Data Layout for Cache Efficiency:** Implementing a data layout that optimizes cache usage, such as Morton ordering, can significantly improve performance. This layout organizes data in a way that aligns with the recursive nature of Strassen's algorithm, potentially reducing cache misses.

3. **More Efficient Base Case**: Currently we optimized the base case to 4×4 matrix utilizing the `AVX` family, we can potentially improve this, say to 8×8 matrix multiplication based on the specific characteristics of the hardware.

4. **Strassen-Winograd Variant** The Strassen-Winograd variant further reduces the number of multiplications, which could lead to performance improvements. Similar to Strassen's algorithm, we still need to adopt the hybrid approach and examine the cross-over point, since this variant is also recursive and the constant factor is likely to be high as well despite it being asymptotically better.