CS 267 HW1

Group3: Yunzhe Liu, Du Xiang, Rohit Agarwal

February 2023

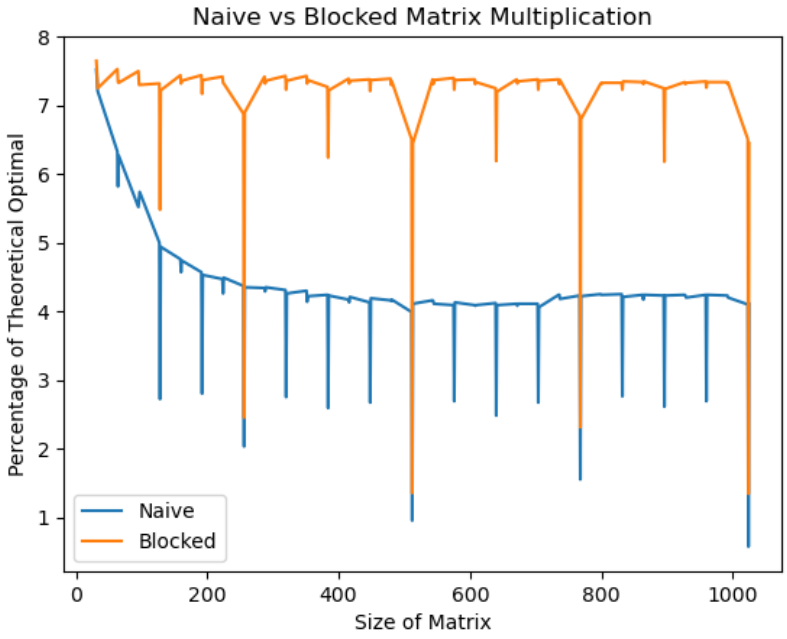
1 Optimizations

The following are optimizations we used. We implemented them in the following order:

1. Loop reordering
2. Block-size tuning
3. Micro-kernels & loop unrolling
4. Repacking

1.1 Naive Matrix Multiplication

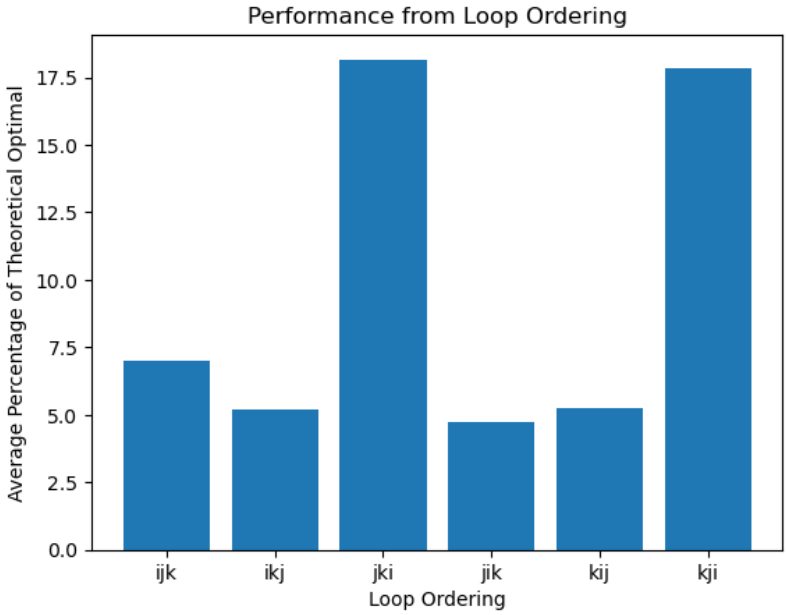
Here is a depiction of the naive and given blocked algorithms. Notice they are not very fast, but give a baseline we can iterate upon.



1.2 Loop Reordering

The first thing we tried was reordering the loops in the do_block method. Intuitively, we want to decrease the amount that each array strides per iteration. Since the matrices are stored in column major order, this means trying to only change the row number in every iteration. In our indexing convention, we are computing $C_{ij} = C_{ij} + A_{ik} * B_{kj}$ at every step.

The loop orders jki and kji achieve this, because the inner loop strides by 1 over A, 1 over B and 0 over C. This was confirmed by our testing. Here are the results of running the blocked algorithm given with all 6 loop orderings (with the block size changed to 32).

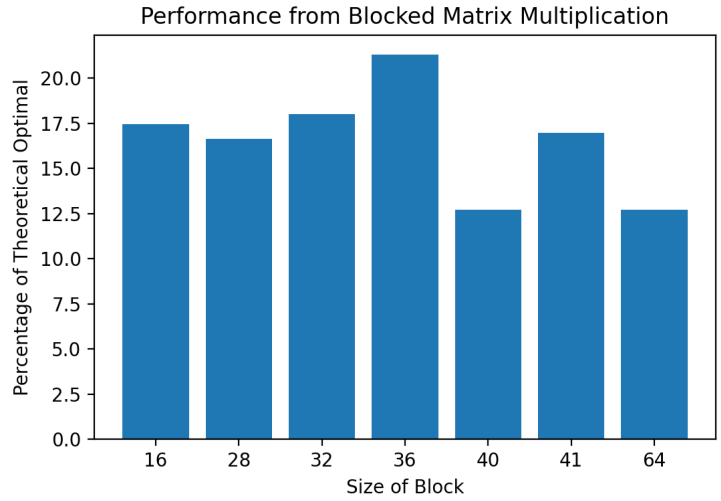


1.3 Blocked Matrix Multiplication

Recall that we wish to compute the problem $C = C + A * B$ for square matrices of size $n \times n$. We calculated the theoretical block size as the following:

$$
\begin{aligned}
b &\leq \sqrt{M/3} \\
b &\leq \sqrt{\text{L1 cache size}/3} \\
b &\leq \sqrt{32KB * 1024B/KB/8B/double/3} \\
b &\leq 36.95
\end{aligned}
$$

So, we obtained the theoretical block size to be less than 36 to fit into the L1 cache memory.



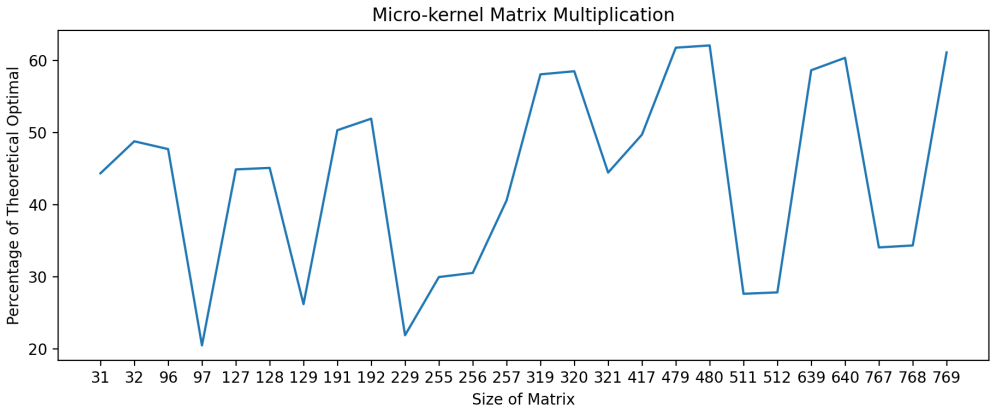
As seen from the experiments, when block size = 36, we obtained the highest performance from blocked matrix multiplication.

1.4 Micro-kernels & Loop Unrolling

In order to optimize the instructional level parallelism, we further optimize the Blocked matrix multiplication by vectorizing the blocked multiplication process. We applied the AVX SIMD intrinsics function _mm256_load_pd and _mm256_fmadd_pd to perform 4 doubles fuse multiplication and add. In addition, we unrolled the inner i loop, so that there will be less overhead with for-loop conditions.

```
#define microkernel(As, Cs, Br) { \
    Ar = _mm256_load_pd(As); \
    Cr = _mm256_load_pd(Cs); \
    _mm256_store_pd(Cs, _mm256_fmadd_pd(Ar, Br, Cr)); \
}
```

After experimentation with the block sizes, we obtained that b = 32 was the best option to perform 4x8 sized micro-kernels.



From the plot, we observed performance dips at matrix size = 97, 129, 229... Most of these numbers are prime numbers and they are 'awkwardly' padded to be the multiple of block size 32. For example, 97, we would have to pad 31 extra 0s on each side of the matrix which introduces lots of overhead.

1.5 Repacking

When doing block multiplications, elements in a block are not contiguous in memory when $N > 32$. So we repacked the matrices to put them contiguously. Specifically, the new position of the element in block (i, j) with in-block relative position (ii, jj) is:

$$\left(i + j\frac{N}{B}\right) B^2 + (ii + jjB)$$

where $0 \leq i, j < N/B$ and $0 \leq ii, jj < B$, $B = 32$ is the block size.

In practice, we enumerate the elements in the original matrix in block order and put them in the new matrix.

1.6 Special Optimizations

1. We observed a significant performance dip for $N = 97$, and all $N = 32k + 1$. That's because a lot of extra work is needed when we pad them to $32(k + 1)$. Therefore, when $N = 32k + 1$, we instead pad them to $32k + 4$ to reduce the extra work.

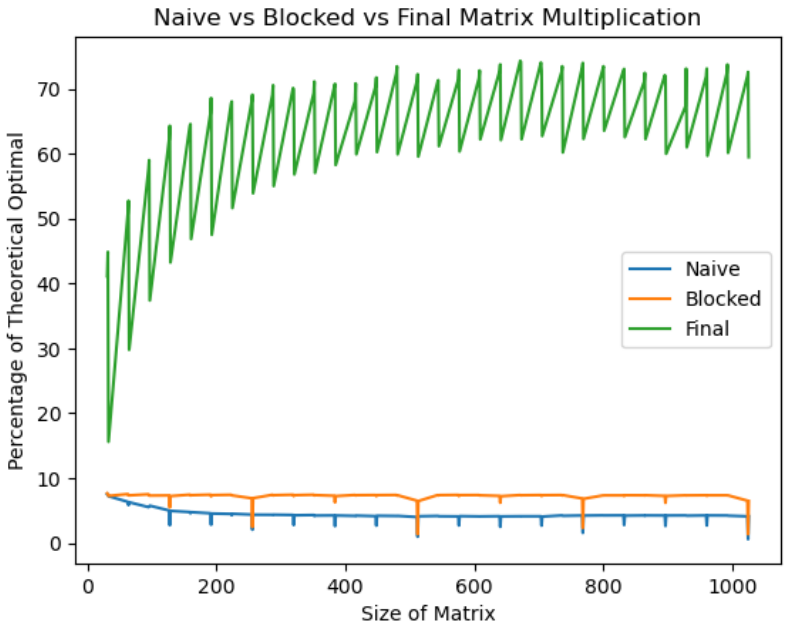
2. In the experiment, we observed that an extra magic malloc statement gives us considerable performance improvement. We failed to figure out the reason behind it. One guess is that some combination of the compiler and the operating system work together to increase the speed of scheduling. A more in-depth analysis of the assembly emitted is required to make any conclusions.

```
double *A_aligned = (double*) _mm_malloc(lda * lda * sizeof(double), 32);
double *B_aligned = (double*) _mm_malloc(lda * lda * sizeof(double), 32);
double *C_aligned = (double*) _mm_malloc(lda * lda * sizeof(double), 32);

// magic malloc
if(lda > 32){
    _mm_malloc(lda * lda * sizeof(double), 32);
}
```

2 Final Results

With all the optimizations we mentioned above, we finally achieved a 62% performance in the representative subset and 65% in all sizes. These were our final results.



Despite our attempts at optimizing, we still have dips at $32k+1$ due the fact that padding to $32k+4$ creates a less cache-efficient blocking. There is also the overhead of having tail cases with only one row/column (though these dips smooth out as we increase the size). These dips are not as killer (relatively) as those we observed in the naive and blocked algorithms, however.

3 Team Contributions

All three of the group members implemented most of these optimizations independently on their own and achieved speedups of about 50%. However, Yunzhe was able to find the "special" optimizations to push the performance to the next level and get us to 65%. The report was mainly assembled by Du and Rohit, and Yunzhe provided the final code to help Du and Rohit gather data on the individual effects of each optimization. All three members contributed to the coding and writing of the report.