COMPSCI 267 HW 1 - Optimizing Single Threaded Matmul

Group 77: Varun Jadia and Tejvir Jogani

1. Introduction

In this report, we elaborate on the steps we took to optimize the matrix multiplication operation, C = C + A * B using a single thread. We investigated a variety of methods, ranging from matrix blocking to writing a kernel using vector operations. We also detail how each method affected performance and potential reasons for the results we saw.

Optimizations

Blocking

The first optimization method we tried was matrix blocking, i.e., breaking down the multiplication operation into smaller matrix multiplication operations such that these smaller blocks fit into the cache. We tested block sizes ranging from 4 to 200 for both the naive blocking code (i.e., with only blocking implemented) and blocking with our SIMD kernel (discussed later) and found that a block size of 120 gave us the most optimal performance (using the 8x8 kernel). We believe this is because at this size, the blocks are slightly smaller than the L1 cache size, which means the entire block can fit into the L1 cache. At higher block sizes, we noticed that the performance plateaus and even starts to dip slightly, which is because the blocks are no longer small enough to fit into the L1 cache.

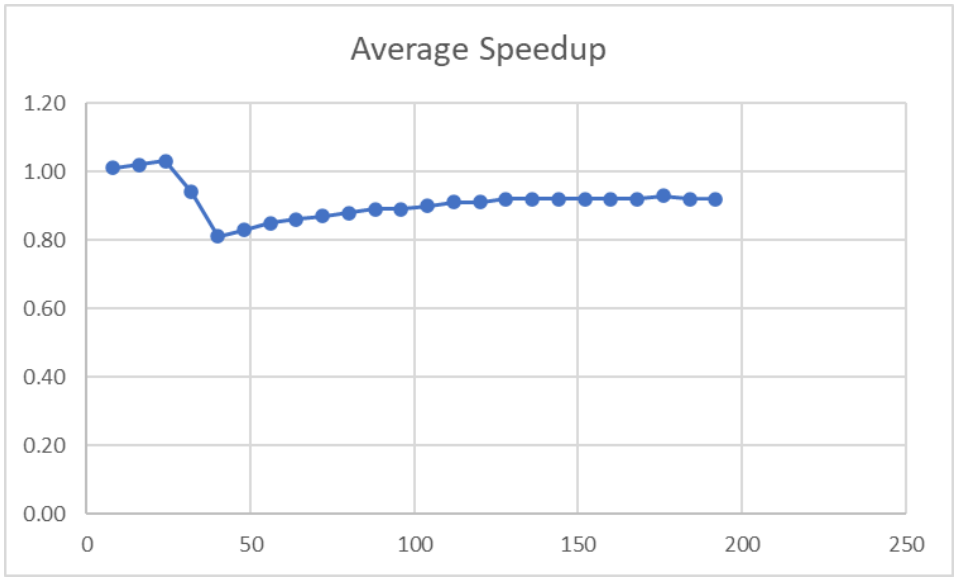

Fig 1: Average speedup trend for naive blocking code (% of peak vs matrix size)

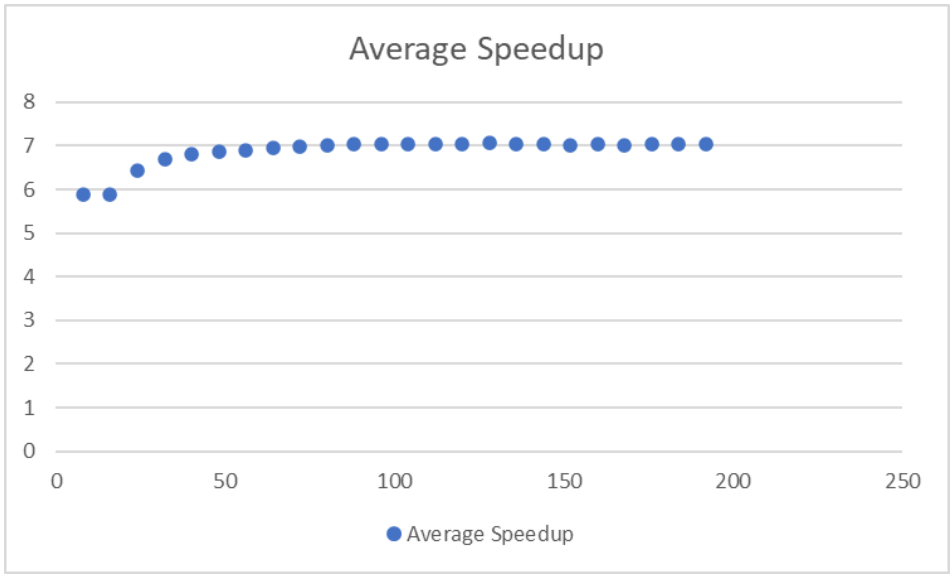

Fig 2: Average speedup trend for blocking + 8x8 SIMD kernel (% of peak vs matrix size)

We also tried multiple levels of blocking to use all levels of the cache but did not receive any speedup from doing so, potentially due to issues with our implementation. Due to time constraints, we chose not to pursue this further.

Copy optimizations/Matrix repacking + Alignment

To make use of spatial locality, we also tried repacking our matrices into row-major order. This gave us a speedup of ~1% on the naive blocking code - this is because converting to row major means that we can index elements of the matrix using a stride of 1 instead of lda, which allows us to make use of subsequent entries stored in the cache instead of having to evict and re-store elements in the cache each time we access an entry. To illustrate this, consider the following naive code:

```
void square_dgemm(int n, double* A, double* B, double* C) {
    for (int i = 0; i < n; ++i) {
        for (int j = 0; j < n; ++j) {
            // Compute C(i,j)
            double cij = C[i + j * n];
            for (int k = 0; k < n; k++) {
                cij += A[i + k * n] * B[k + j * n];
            }
            C[i + j * n] = cij;
        }
    }
}
```

Here both A and B are in column major order, and in the inner loop k, subsequent elements of A that are used for the multiplication are n (lda) strides away. Now, after repacking A:

```
void square_dgemm_repack(int n, double* A, double* B, double* C) {
    for (int i = 0; i < n; ++i) {
        for (int j = 0; j < n; ++j) {
            // Compute C(i,j)
            double cij = C[i + j * n];
            for (int k = 0; k < n; k++) {
                cij += A[k + i * n] * B[k + j * n];
            }
            C[i + j * n] = cij;
        }
    }
}
```

In this case, we access elements of A using only a stride of 1. In our case, we repacked all the matrices in row major.

Additionally, we also made sure that the matrices were cache aligned by using the intrinsic `_mm_malloc`. This makes a fractional difference to the performance, but makes it easier and more efficient to perform SIMD tasks. Each matrix was copied and aligned using a function called `copy_matrix`. The function essentially makes cache aligned space and then performs an $O(n^2)$ copy.

Kernels

We tried implementing kernels - first by simply writing a microkernel function - code that simply ran our dgemm logic without using any intrinsics. This didn't yield a large speed up but it laid a foundation for how the kernel is supposed to function.

A brief explanation of how our microkernel works:

Consider you have the given matrices in row major order. Now we can view matrix multiplication as a row wise operation. Formally, if x_i is row i of matrix X and x_{ia} is the a[th] element of row i of X, then:

$$[c_{i1}, c_{i2},..., c_{in}] = c_i + \sum_{j=1}^{n} a_{ij} b_j$$

SIMD Kernel

We also tried implementing a SIMD kernel to further speed up matrix multiplication. We started with a small kernel and then took that logic to write a kernel for 8x8 matrix multiplication (i.e., multiplying two 8x8 matrices) using the AVX512 registers.

Our formula mentioned above works extremely efficiently with registers since we can preload all of matrix B and C into 16 registers for an 8x8 kernel. We need 8 registers for the 8 rows of the matrix and then each register holds 8 doubles so that takes care of each column. For A, we broadcast the necessary element onto a register and then performed the FMA operation (parallel multiplication and add).

Since some of these operations can happen independently, we can iterate through and add to c0 first, then c1, and so on so forth. This speeds up the program a lot more since the time to access memory is extremely low since the data is on the CPU.

Moreover, to simplify dealing with the border cases (when the input matrices had dimensions that were not multiples of 8), we padded our matrices to the nearest multiple of 8 and then depadded them at the very end after completing all multiplications/additions.

Microkernels gave us a significant speedup increase of almost 7%, which was the most compared to the other techniques we tried. This is most likely because our kernel helped reduce the number of branch prediction operations/helped improve Instruction Level Parallelism in our code.

We also noticed that the 8x8 kernel gave the best performance for mid-large size matrices - we also measured performance with a 4x4 kernel, the results of which are recorded below:

Kernel	Best Performance	Block Size
4x4	5.71% (of peak)	140
8x8	7.76% (of peak)	120

This improvement can be attributed to more operations taking place in parallel since the kernel size is larger.

Loop optimization - reordering and unrolling

The primary loop optimization technique we tried was to reorder our loops, first in the outer square_dgemm function and then in the do_block code to potentially increase Instruction Level Parallelism. We tried all 36 loop orders and found that the 'ikj' loop order performed best for the outer function and 'kij' for the inner function. This is because of how the memory is brought to the cache and can have a large impact in general on the compute.

The results for the outer function were surprising because we expected 'kij' to perform the best, but the difference between each permutation was marginal (<1%), which probably means that the outer loop ordering mattered much less than the inner function's ordering. The increase in speedup we received from reordering the loops in the inner function to 'kij' was around 1%

We also tried loop unrolling, which was done through our kernel function. Essentially, our entire kernel didn't use loops - each step was an intrinsic, i.e. one line of assembly. We wrote a Python Script to generalize the creation of a kernel. We started with a 4x4 kernel and then unrolled it to a 8x8 kernel, giving us a speedup increase of 2%

Conclusion

Of all the techniques, the SIMD kernel gave us the most significant speedup because it helped with both ILP and reducing branch prediction operations. However, all our optimization ideas had many parameters we could tweak. For all of these tasks, we ran a parameter sweep to ensure we took the best parameter in order to get the largest speed up. Some important results are in the appendix.

Appendix of results

Block size 152 + 8x8 kernel

```
Description:    blocking and 8x8 simd kernel

Size: 31        Mflops/s: 1400.30       Percentage: 3.13
Size: 32        Mflops/s: 1541.62       Percentage: 3.44
Size: 96        Mflops/s: 3000.38       Percentage: 6.70
Size: 97        Mflops/s: 2373.66       Percentage: 5.30
Size: 127       Mflops/s: 2659.04       Percentage: 5.94
Size: 128       Mflops/s: 3171.21       Percentage: 7.08
Size: 129       Mflops/s: 2365.88       Percentage: 5.28
Size: 191       Mflops/s: 2991.11       Percentage: 6.68
Size: 192       Mflops/s: 3386.54       Percentage: 7.56
Size: 229       Mflops/s: 2926.48       Percentage: 6.53
Size: 255       Mflops/s: 2972.35       Percentage: 6.63
Size: 256       Mflops/s: 3420.95       Percentage: 7.64
Size: 257       Mflops/s: 2995.83       Percentage: 6.69
Size: 319       Mflops/s: 3185.01       Percentage: 7.11
Size: 320       Mflops/s: 3692.56       Percentage: 8.24
Size: 321       Mflops/s: 3080.82       Percentage: 6.88
Size: 417       Mflops/s: 3598.31       Percentage: 8.03
Size: 479       Mflops/s: 3534.24       Percentage: 7.89
Size: 480       Mflops/s: 4216.95       Percentage: 9.41
Size: 511       Mflops/s: 3299.01       Percentage: 7.36
Size: 512       Mflops/s: 2885.35       Percentage: 6.44
Size: 639       Mflops/s: 3813.31       Percentage: 8.51
Size: 640       Mflops/s: 4053.46       Percentage: 9.05
Size: 767       Mflops/s: 3847.66       Percentage: 8.59
Size: 768       Mflops/s: 3585.23       Percentage: 8.00
Size: 769       Mflops/s: 3902.26       Percentage: 8.71
Average percentage of Peak = 7.03
```

Block size 152 + 4x4 kernel

```
Size: 31        Mflops/s: 1322.13       Percentage: 2.95
Size: 32        Mflops/s: 1451.45       Percentage: 3.24
Size: 96        Mflops/s: 2546.96       Percentage: 5.69
Size: 97        Mflops/s: 2217.98       Percentage: 4.95
Size: 127       Mflops/s: 2294.47       Percentage: 5.12
Size: 128       Mflops/s: 2731.05       Percentage: 6.10
Size: 129       Mflops/s: 2156.01       Percentage: 4.81
Size: 191       Mflops/s: 2566.75       Percentage: 5.73
Size: 192       Mflops/s: 2887.72       Percentage: 6.45
Size: 229       Mflops/s: 2487.10       Percentage: 5.55
Size: 255       Mflops/s: 2170.61       Percentage: 4.85
Size: 256       Mflops/s: 2411.90       Percentage: 5.38
Size: 257       Mflops/s: 2583.13       Percentage: 5.77
Size: 319       Mflops/s: 2668.90       Percentage: 5.96
Size: 320       Mflops/s: 3042.17       Percentage: 6.79
Size: 321       Mflops/s: 2582.62       Percentage: 5.76
Size: 417       Mflops/s: 2881.54       Percentage: 6.43
Size: 479       Mflops/s: 2881.51       Percentage: 6.43
Size: 480       Mflops/s: 3328.10       Percentage: 7.43
Size: 511       Mflops/s: 2269.85       Percentage: 5.07
Size: 512       Mflops/s: 2147.35       Percentage: 4.79
Size: 639       Mflops/s: 3065.26       Percentage: 6.84
Size: 640       Mflops/s: 3254.81       Percentage: 7.27
Size: 767       Mflops/s: 2666.51       Percentage: 5.95
Size: 768       Mflops/s: 2497.86       Percentage: 5.58
Size: 769       Mflops/s: 3036.54       Percentage: 6.78
Average percentage of Peak = 5.68
```

```
Description:    blocking_120+8x8kernel+kij_inner

Size: 31        Mflops/s: 1530.77       Percentage: 3.42
Size: 32        Mflops/s: 1682.09       Percentage: 3.75
Size: 96        Mflops/s: 3354.19       Percentage: 7.49
Size: 97        Mflops/s: 2593.23       Percentage: 5.79
Size: 127       Mflops/s: 2963.43       Percentage: 6.61
Size: 128       Mflops/s: 3576.22       Percentage: 7.98
Size: 129       Mflops/s: 2592.95       Percentage: 5.79
Size: 191       Mflops/s: 3264.09       Percentage: 7.29
Size: 192       Mflops/s: 3771.65       Percentage: 8.42
Size: 229       Mflops/s: 3165.02       Percentage: 7.06
Size: 255       Mflops/s: 3260.79       Percentage: 7.28
Size: 256       Mflops/s: 3739.19       Percentage: 8.35
Size: 257       Mflops/s: 3308.18       Percentage: 7.38
Size: 319       Mflops/s: 3520.25       Percentage: 7.86
Size: 320       Mflops/s: 4168.92       Percentage: 9.31
Size: 321       Mflops/s: 3447.93       Percentage: 7.70
Size: 417       Mflops/s: 3826.40       Percentage: 8.54
Size: 479       Mflops/s: 3982.51       Percentage: 8.89
Size: 480       Mflops/s: 4572.26       Percentage: 10.21
Size: 511       Mflops/s: 3628.54       Percentage: 8.10
Size: 512       Mflops/s: 3273.53       Percentage: 7.31
Size: 639       Mflops/s: 4035.53       Percentage: 9.01
Size: 640       Mflops/s: 4346.36       Percentage: 9.70
Size: 767       Mflops/s: 4493.11       Percentage: 10.03
Size: 768       Mflops/s: 4071.64       Percentage: 9.09
Size: 769       Mflops/s: 4265.16       Percentage: 9.52
Average percentage of Peak = 7.76
```

Blocking_120+8x8kernel+kij_inner+kij_outer

```
Size: 31         Mflops/s: 1437.93       Percentage: 3.21
Size: 32         Mflops/s: 1582.96       Percentage: 3.53
Size: 96         Mflops/s: 3110.84       Percentage: 6.94
Size: 97         Mflops/s: 2419.78       Percentage: 5.40
Size: 127        Mflops/s: 2621.42       Percentage: 5.85
Size: 128        Mflops/s: 3236.69       Percentage: 7.22
Size: 129        Mflops/s: 2344.04       Percentage: 5.23
Size: 191        Mflops/s: 3091.29       Percentage: 6.90
Size: 192        Mflops/s: 3518.35       Percentage: 7.85
Size: 229        Mflops/s: 2948.76       Percentage: 6.58
Size: 255        Mflops/s: 3138.31       Percentage: 7.01
Size: 256        Mflops/s: 3616.79       Percentage: 8.07
Size: 257        Mflops/s: 3142.90       Percentage: 7.02
Size: 319        Mflops/s: 3374.29       Percentage: 7.53
Size: 320        Mflops/s: 3980.10       Percentage: 8.88
Size: 321        Mflops/s: 3282.67       Percentage: 7.33
Size: 417        Mflops/s: 3729.35       Percentage: 8.32
Size: 479        Mflops/s: 3836.76       Percentage: 8.56
Size: 480        Mflops/s: 4451.53       Percentage: 9.94
Size: 511        Mflops/s: 3460.12       Percentage: 7.72
Size: 512        Mflops/s: 3062.20       Percentage: 6.84
Size: 639        Mflops/s: 4013.01       Percentage: 8.96
Size: 640        Mflops/s: 4280.92       Percentage: 9.56
Size: 767        Mflops/s: 4185.20       Percentage: 9.34
Size: 768        Mflops/s: 3774.44       Percentage: 8.43
Size: 769        Mflops/s: 4122.62       Percentage: 9.20
Average percentage of Peak = 7.36
```