HW1: Optimizing Matrix Multiply Report

Lian Liao, ll987, ll987(Perlmutter)

Results:

```
Description:    Lian's simple blocked dgemm.

Size: 31    Mflops/s: 18215.76  Percentage: 32.53
Size: 32    Mflops/s: 21380.83  Percentage: 38.18
Size: 96    Mflops/s: 24341.68  Percentage: 43.47
Size: 97    Mflops/s: 22339.48  Percentage: 39.89
Size: 127   Mflops/s: 21275.31  Percentage: 37.99
Size: 128   Mflops/s: 24527.16  Percentage: 43.80
Size: 129   Mflops/s: 21952.85  Percentage: 39.20
Size: 191   Mflops/s: 22069.91  Percentage: 39.41
Size: 192   Mflops/s: 24809.92  Percentage: 44.30
Size: 229   Mflops/s: 23531.40  Percentage: 42.02
Size: 255   Mflops/s: 22415.87  Percentage: 40.03
Size: 256   Mflops/s: 23466.10  Percentage: 41.90
Size: 257   Mflops/s: 22671.96  Percentage: 40.49
Size: 319   Mflops/s: 23010.51  Percentage: 41.09
Size: 320   Mflops/s: 24729.34  Percentage: 44.16
Size: 321   Mflops/s: 23424.00  Percentage: 41.83
Size: 417   Mflops/s: 24012.34  Percentage: 42.88
Size: 479   Mflops/s: 22569.95  Percentage: 40.30
Size: 480   Mflops/s: 24623.90  Percentage: 43.97
Size: 511   Mflops/s: 22161.70  Percentage: 39.57
Size: 512   Mflops/s: 21316.95  Percentage: 38.07
Size: 639   Mflops/s: 23728.70  Percentage: 42.37
Size: 640   Mflops/s: 24337.85  Percentage: 43.46
Size: 767   Mflops/s: 23745.19  Percentage: 42.40
Size: 768   Mflops/s: 23453.67  Percentage: 41.88
Size: 769   Mflops/s: 23736.84  Percentage: 42.39
Average percentage of Peak = 41.06
```

```
Description:    Lian's simple blocked dgemm.

Size: 31    Mflops/s: 19580.72  Percentage: 34.97
Size: 32    Mflops/s: 21468.34  Percentage: 38.34
Size: 96    Mflops/s: 17899.13  Percentage: 31.96
Size: 97    Mflops/s: 14008.13  Percentage: 25.01
Size: 127   Mflops/s: 16502.12  Percentage: 29.47
Size: 128   Mflops/s: 16858.31  Percentage: 30.10
Size: 129   Mflops/s: 14870.02  Percentage: 26.55
Size: 191   Mflops/s: 18898.74  Percentage: 33.75
Size: 192   Mflops/s: 19207.89  Percentage: 34.30
Size: 229   Mflops/s: 19511.36  Percentage: 34.84
Size: 255   Mflops/s: 16171.19  Percentage: 28.88
Size: 256   Mflops/s: 16349.90  Percentage: 29.20
Size: 257   Mflops/s: 19309.90  Percentage: 34.48
Size: 319   Mflops/s: 21811.90  Percentage: 38.95
Size: 320   Mflops/s: 22010.27  Percentage: 39.30
Size: 321   Mflops/s: 20843.82  Percentage: 37.22
Size: 417   Mflops/s: 22353.49  Percentage: 39.92
Size: 479   Mflops/s: 23821.97  Percentage: 42.54
Size: 480   Mflops/s: 23995.03  Percentage: 42.85
Size: 511   Mflops/s: 16456.49  Percentage: 29.39
Size: 512   Mflops/s: 16590.15  Percentage: 29.63
Size: 639   Mflops/s: 23424.52  Percentage: 41.83
Size: 640   Mflops/s: 23564.24  Percentage: 42.08
Size: 767   Mflops/s: 20161.97  Percentage: 36.00
Size: 768   Mflops/s: 20272.97  Percentage: 36.20
Size: 769   Mflops/s: 28075.46  Percentage: 50.13
Average percentage of Peak = 35.30
```

My best try is around 41% of the peak. To get this result, I have tried two main approaches. The main difference is that one initially realigns and fixes the size to memory; the other one doesn’t realign it and keeps the size then deals with a tail. For me, the approach of dealing with the tail is better, which is 40%. And I combine two of them to get the 41%

optimizations attempt:

Based on the HW1 doc. I tried to implement multi-level blocking. I just simply added one more layer with 3 for-loop.

```
// For each block-row of A
for (int i = 0; i < fixlda; i += BLOCK_SIZE) {
    // For each block-row of A
    for (int j = 0; j < fixlda; j += BLOCK_SIZE) {
        // For each block-column of B
        for (int k = 0; k < fixlda; k += BLOCK_SIZE) {
            int M = min(BLOCK_SIZE, fixlda - i);
            int N = min(BLOCK_SIZE, fixlda - j);
            int K = min(BLOCK_SIZE, fixlda - k);
            //   do_block(fixlda, M, N, K, A_block + i + k * fixlda,
            // Perform individual block dgemm
            for (int f = i; f < i + M; f += SMALL_BLOCK) {
                for (int g = j; g < j + N; g += SMALL_BLOCK) {
                    for (int h = k; h < k + K; h += SMALL_BLOCK) {
                        int Q = min(SMALL_BLOCK, i + M - f);
                        int W = min(SMALL_BLOCK, j + N - g);
                        int E = min(SMALL_BLOCK, k + K - h);
```

Use the same logic to add a second layer. The result is positive, but not increase speed much from just one level of blocking.

Next one is Repack and Realign

```
double* A_block = (double*)_mm_malloc(size:M_mod_4 * K * sizeof(double), align:32);
double* B_block = (double*)_mm_malloc(size:N_mod_4 * K * sizeof(double), align:32);
```

I set 32 bytes because my avx is 256. 256/32=8 which is double's size.

I didn't do the Repack. I think it can potentially increase the speed with blocking.

```
for (j = 0 ; j < N - 3; j += 4) {
    b_ptr = &B_block[j * K];

    for (int m = 0; m < K; m++) {
        for (int n = 0; n < 4; n++) {
            b_ptr[m * 4 + n] = B[(j + n) * lda + m];
        }
    }

    for (i = 0; i < M - 3; i += 4) {
        a_ptr = &A_block[i * K];

        if (j == 0) {
            double* a_src = A + i;
            for (int u = 0; u < K; u++) {
                memcpy(dest:a_ptr + u * 4, src:a_src, n:4 * sizeof(double));
                a_src += lda;
            }
        }

        c_ptr = C + i + j * lda;
        kernel(lda, K, A:a_ptr, B:b_ptr, C:c_ptr);

    }
}
```

Here is in small block(C is M-by-N, A is M-by-K, and B is K-by-N)

I copied the matrix within the loop. Step is 4 because memory size is 32 and double is 8, 32/8=4. I only do Realign for A and B because it is slower to add C. I have tried to implement C_block in small blocks or initials, but they are slower. I guess because they are not really aligned since I didn't fix the size before getting the microkernel.

My micro-kernel is inspired by doc

```
void micro_kernel (double* A, double* B, double* C) {
    // Declare
    __m512d Ar;
    __m512d Br;
    __m512d Cr;

    // Load
    Ar = _mm512_load_pd(A);
    Br = _mm512_load_pd(B);

    // Compute
    Cr = _mm512_add_pd(Ar, Br);

    // Store
    _mm512_store_pd(C, Cr);
}
```

But I use __m256d is not supported in Perlmutter. My loop for K is unrolling for every 2 steps.

```
for (int i = 0; i < KK; i += 2){
    aa1 = _mm256_load_pd( p: A);
    A += 4;

    bb01 = _mm256_broadcast_sd( a: B++);
    bb02 = _mm256_broadcast_sd( a: B++);
    bb03 = _mm256_broadcast_sd( a: B++);
    bb04 = _mm256_broadcast_sd( a: B++);

    cc01 = _mm256_fmadd_pd( A: aa1, B: bb01, C: cc01);
    cc02 = _mm256_fmadd_pd( A: aa1, B: bb02, C: cc02);
    cc03 = _mm256_fmadd_pd( A: aa1, B: bb03, C: cc03);
    cc04 = _mm256_fmadd_pd( A: aa1, B: bb04, C: cc04);

    aa2 = _mm256_load_pd( p: A);
    A += 4;

    bb11 = _mm256_broadcast_sd( a: B++);
    bb12 = _mm256_broadcast_sd( a: B++);
    bb13 = _mm256_broadcast_sd( a: B++);
    bb14 = _mm256_broadcast_sd( a: B++);

    cc11 = _mm256_fmadd_pd( A: aa2, B: bb11, C: cc11);
    cc12 = _mm256_fmadd_pd( A: aa2, B: bb12, C: cc12);
    cc13 = _mm256_fmadd_pd( A: aa2, B: bb13, C: cc13);
    cc14 = _mm256_fmadd_pd( A: aa2, B: bb14, C: cc14);

}
```

I added them while in the loop and combined them in the end, then wrote to C.
After implementing the Micro-kernel, the speed increased a lot. This verified the results from some HPC papers which blocking and micro-kernel are most effective.

```
if (M%4 != 0){
    for (; i < M; ++i){
        for (int p = 0; p < N; p++){
            tem = C[i + p * lda];
            for (int k = 0; k < K; k++){
                tem += A[i + k * lda] * B[k + p * lda];
            }
            C[i + p * lda] = tem;
        }
    }
}
if (N%4 != 0){
    for (; j < N; j++){
        for (int p = 0; p < M_mod_4; p++){
            tem = C[p + j * lda];
            for (int k = 0; k < K; k++){
                tem += A[p + k * lda] * B[k + j * lda];
            }
            C[p + j * lda] = tem;
        }
    }
}
```

For the tail. Just simply access the original A,B, and C to deal. I tried to make them aligned to avoid tail, but it didn’t work out.

Other small optimizations:
I read that an “inline” function can help with optimization because the compiler attempts to embed the function's code directly into the calling site, rather than generating a function call. However, I don’t see any difference after adding them to my function.
Same as “restrict” I don’t see any difference after adding them to my Matrix.

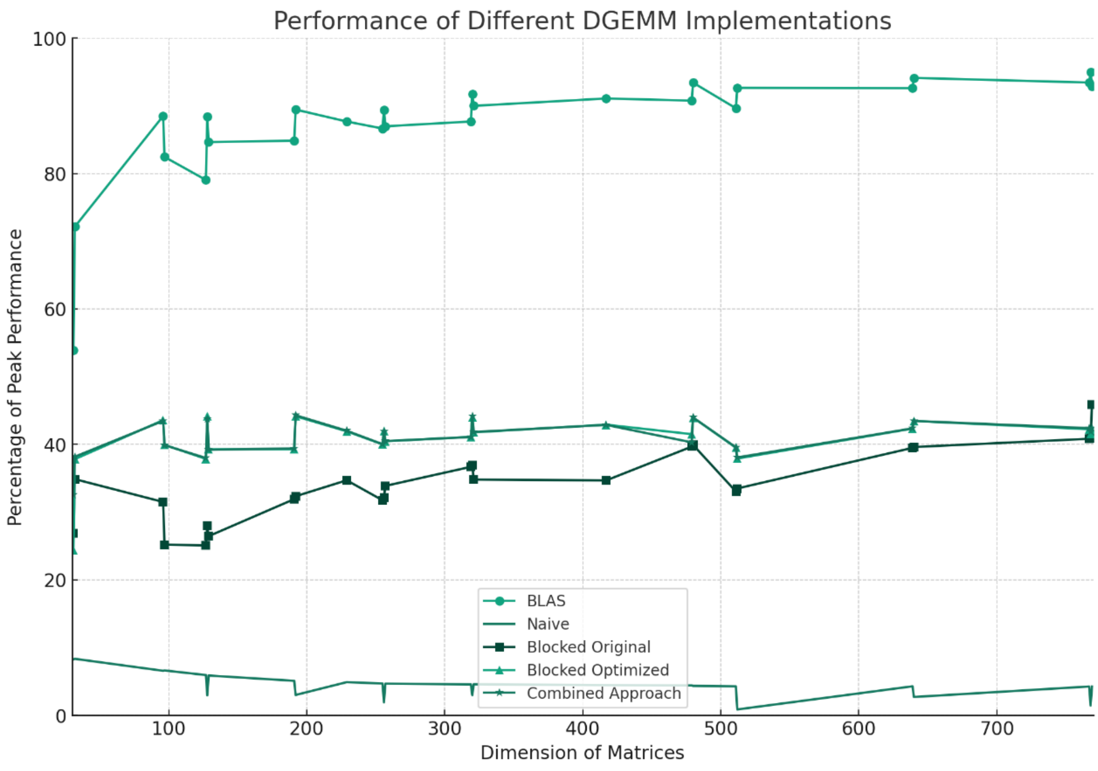
```
#ifndef BLOCK_SIZE
#define BLOCK_SIZE 128
#define SMALL_BLOCK 64
#endif
```

This is the critical factor for performance. Is decided whether the blocking size or represented as L1 and L2 cache. (128,64) is my best set to my code.

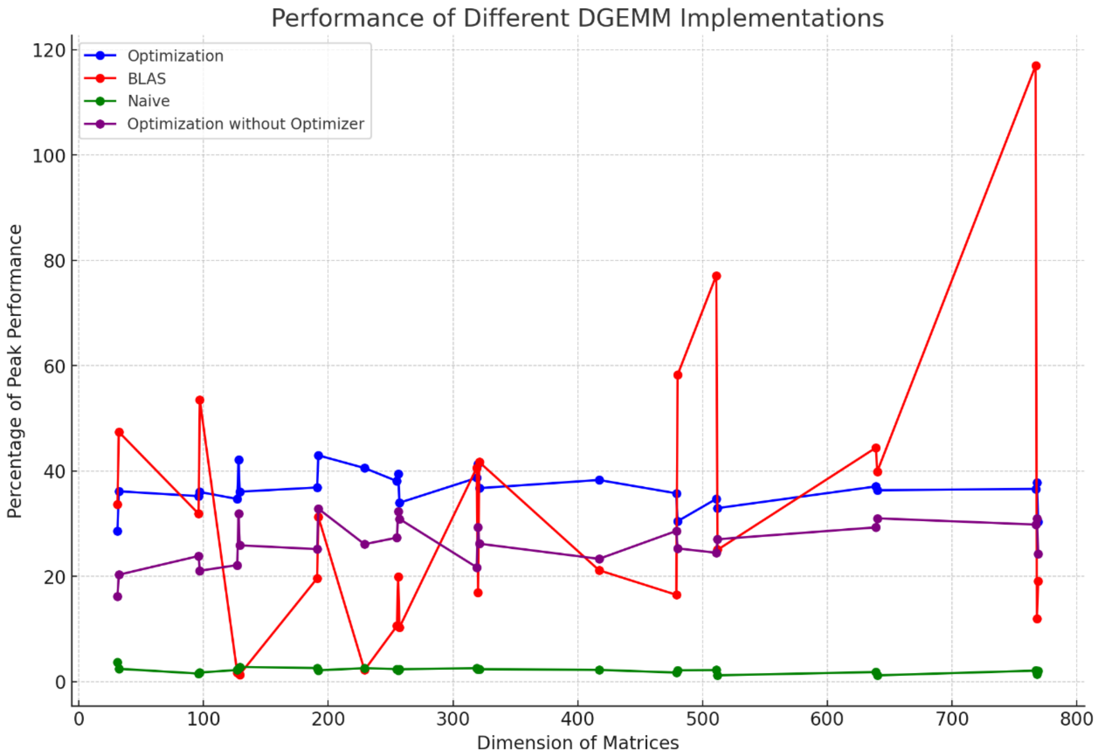
Note: because my other approach is better on size 31. So I only run it if the size is smaller than 32

```
#pragma GCC optimize ("peel-loops")
#pragma GCC optimize("inline")
#pragma GCC optimize("unroll-loops")
#pragma GCC optimize("Ofast")
```

Those are optimizers I found. They can help my local computer dramatically speed up, but not for Perlmutter. The interesting thing is that with optimizers, my computer is faster than in Perlmutter; without them, it is way slower than in Perlmutter.
Update: I didn't edit the max_speed on my desktop. After I changed it, the percentage of the peak was around 36%, which is a little bit slower than Perlmutter. My max_speed calculation is:
3.9 Processor frequency * 2 cores *2 vector pipelines * 2 flops for FMA=93.6



This is my plot for the final performance on Perlmutter. My curve is flat at around 40%



My performance on my desktop. It has a lot of fluctuations compared to Perlmutter

Conclusion:
The optimization attempts on matrix multiplication yielded a significant performance increase, achieving approximately 41% of the peak. The critical factor for performance optimization was the cache-aware blocking size, indicating the profound impact of memory hierarchy on computational efficiency (speed would drop a lot if I forgot to free it). Despite the various strategies employed, a performance gap remains at the theoretical peak.

References

Jiang, Xuan, et al. “Optimizing Matrix Multiplication on Nersc’s High Performance Computer Cori.” OSF Preprints, 26 Feb. 2022. Web.