CS205-2022Fall Project 4

Optimizing Matrix Multiplication

Name: 陈康睿

SID: 12110524

Part 1 - Analysis & Implementation

In this project, I try to use localization, SIMD, Multithreading, Memory alignment and compile options to accelerate matrix multiplication. I also tried Stressen algorithm.

To simplify and concentrate on optimization, I only implemented functions supporting square matrices. (For Stressen algorithm, square matrices with order 2^n especially)

Libraries

```
#include <stddef.h>
#include <stdbool.h>
#include <stdio.h>
#include <string.h>
#include <immintrin.h>
#include <omp.h>
#include <sys/time.h>
#include <time.h>
#include <stdlib.h>
#include <cblas.h> // OpenBLAS
```

Data Structure & Basic Operations

It is worth a mention that the memory allocated for entry is aligned for acceleration.

```
typedef struct Matrix{
    size_t n;
    float *entry;
} Matrix;

Matrix *createMatrix(const size_t n, const float *const entry) {
    Matrix *mat = (Matrix *)malloc(sizeof(Matrix));
    mat->n = n;
    size_t siz = n*n;
    mat->entry = (float *)aligned_alloc(256, sizeof(float)*siz);
    if (entry)
        memcpy(mat->entry, entry, sizeof(float)*siz);
    return mat;
}

void deleteMatrix(Matrix *const mat) {
    free(mat->entry);
    free(mat);
}
```

Plain Multiplication

Simple and basic ijk 3-layer loops implication

```
Matrix *mul_plain(const Matrix *const a, const Matrix *const b) {
    size_t n = a->n;
    Matrix *ret = createMatrix(n, NULL);
    bzero(ret->entry, sizeof(float)*(n*n));
    for (size_t i = 0; i < n; ++i) {
        for (size_t j = 0; j < n; ++j)
            for (size_t k = 0; k < n; ++k)
                ret->entry[i*n+j] += a->entry[i*n+k]*b->entry[k*n+j];
    }
    return ret;
}
```

1^{st} Optimization - Loop order & Localization

Noted that in the original loops, each time we need to calculate $k * n$, which is time-consuming. We are using row majored storage so we can just exchange j and k so that $i * n$ and $i * k$, which can be pre-calculated, then we only need to plus j each time.

```
Matrix *mul_order(const Matrix *const a, const Matrix *const b) {
    size_t n = a->n;
    Matrix *ret = createMatrix(n, NULL);
    bzero(ret->entry, sizeof(float)*(n*n));
    for (size_t i = 0; i < n; ++i) {
        size_t in = i*n;
        for (size_t k = 0; k < n; ++k) {
            float a_ik = a->entry[in+k];
            size_t kn = k*n;
            for (size_t j = 0; j < n; ++j)
                ret->entry[in+j] += a_ik*b->entry[kn+j];
        }
    }
    return ret;
}
```

2^{nd} Optimization - SIMD

Since my computer CPU is Intel® i7-10875H, which supports AVX2, so I used Intel® Intrinsics to vectorize multiplication. (When allocate memory for matrix entries, I aligned their address. So we can use `_mm256_###_p#()` other than slower ones `_mm256_###u_p#()`)

```
Matrix *mul_order_avx_omp(const Matrix *const a, const Matrix *const b) {
    size_t n = a->n;
    Matrix *ret = createMatrix(a->n, NULL);
    bzero(ret->entry, sizeof(float)*(n*n));
    for (size_t i = 0; i < n; ++i) {
        size_t in = i*n;
        for (size_t k = 0; k < n; ++k) {
            float aik = a->entry[in+k];
            size_t kn = k*n;
```

```
            // all b[k][j] needs to be multiplied by a[i][k] for c[i][j]
            __m256 a_ik = _mm256_set_ps(aik, aik, aik, aik, aik, aik, aik, aik);
            for (size_t j = 0; j < n; j += 8)
                // ret->entry[in+j] += a_ik*b->entry[kn+j]
                // directly call functions to calculate to save more time
                // though do not read elegant...
                _mm256_store_ps(ret->entry+in+j,
                                _mm256_add_ps(
                                    _mm256_load_ps(ret->entry+in+j),
                                                _mm256_mul_ps(a_ik,
                                                _mm256_load_ps(b-
>entry+kn+j))));
        }
    }
    return ret;
}
```

3^{rd} Optimization - Multithreading

By the independence inside the loop, we can use OpenMP and put `#pragma omp parallel for` right before the outermost `for`.

```
Matrix *mul_order_avx_omp(const Matrix *const a, const Matrix *const b) {
    size_t n = a->n;
    Matrix *ret = createMatrix(a->n, NULL);
    bzero(ret->entry, sizeof(float)*(n*n));
    #pragma omp parallel for
    for (size_t i = 0; i < n; ++i) {
        size_t in = i*n;
        for (size_t k = 0; k < n; ++k) {
            float aik = a->entry[in+k];
            size_t kn = k*n;
            __m256 a_ik = _mm256_set_ps(aik, aik, aik, aik, aik, aik, aik, aik);
            for (size_t j = 0; j < n; j += 8)
                _mm256_store_ps(ret->entry+in+j,
                                _mm256_add_ps(_mm256_load_ps(ret->entry+in+j),
                                            _mm256_mul_ps(a_ik,
_mm256_load_ps(b->entry+kn+j))));
        }
    }
    return ret;
}
```

An Unsuccessful Attempt on Stressen Algorithm

This algorithm recusively divide the matrix and merge up with about $\Theta(n^{2.7})$. I try to implements this algorithm but it still doesn't work properly and with recursion I don't no whether OpenMP really works. Despite the accuracy, though I try my best to used SIMD and multithreading to optimize it and it's supporting functions, it still can't beat others when compile with -O3.

设A和B是俩个n x n的矩阵，其中n可以写成2的幂。将A和B分别等分成4个小矩阵，此时如果把A和B都当成2x2矩阵来看,每个元素就是一个（n/2）x（n/2）矩阵，而A和B的成积就可以写成

$$\begin{pmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{pmatrix} \begin{pmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{pmatrix} = \begin{pmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{pmatrix}$$

其中 利用斯特拉森方法得到7个小矩阵，分别定义为：

$$M_1 = (A_{12} - A_{22})(B_{21} + B_{22})$$

$$M_2 = (A_{11} + A_{22})(B_{11} + B_{22})$$

$$M_3 = (A_{21} - A_{11})(B_{11} + B_{12})$$

$$M_4 = (A_{11} + A_{12})B_{22}$$

$$M_5 = A_{11}(B_{12} - B_{22})$$

$$M_6 = A_{22}(B_{21} - B_{11})$$

$$M_7 = (A_{21} + A_{22})B_{11}$$

矩阵 $M_1 \sim M_7$ 可以通过7次矩阵乘法、6次矩阵加法和4次矩阵减法计算得出，前述4个小矩阵 $C_1 \sim C_4$ 可以由矩阵 $M_1 \sim M_7$ 通过6次矩阵加法和2次矩阵减法得出，方法如下：

$$C_{11} = M_1 + M_2 + M_6 - M_4$$

$$C_{12} = M_4 + M_5$$

$$C_{21} = M_6 + M_7$$

$$C_{22} = M_2 + M_3 + M_5 - M_7$$

用上述方案解n=2;矩阵乘法；假定施特拉斯矩阵分割方案仅用于n>=8的矩阵乘法，而对于小于8的矩阵直接利用公式计算；n的值越大，斯拉特森方法更方便；设T(n)表示斯特拉森分治运算法所需时间，因为大的矩阵会被递归分成小矩阵直到每个矩阵的大小小于或等于k，所以T(n)的递归表达式为T(n)=d(n<=k);T(n)=7*t(n/2)+cn2(n平方）（n>k),其中cn2表示完成18次（n/2)(n/2)接矩阵的加减法，以及把大小为N的矩阵分割成小矩阵所需的时间；


```
/**
 * For stressen
 * Use SIMD and OpenMP
*/
Matrix *addMatrix(const Matrix *const a, const Matrix *const b) {
    size_t n = a->n;
    Matrix *ret = createMatrix(n, NULL);
    size_t siz = n*n;
    #pragma omp parallel for
    for (size_t i = 0; i < siz; i += 8)
        _mm256_store_ps(ret->entry+i, _mm256_add_ps(_mm256_load_ps(a->entry+i),
                                                    _mm256_load_ps(b-
>entry+i)));
    return ret;
}

/**
 * For stressen
 * Use SIMD and OpenMP
*/
Matrix *subMatrix(const Matrix *const a, const Matrix *const b) {
    size_t n = a->n;
    Matrix *ret = createMatrix(n, NULL);
    size_t siz = n*n;
```

```
    #pragma omp parallel for
    for (size_t i = 0; i < siz; i += 8)
        _mm256_store_ps(ret->entry+i, _mm256_sub_ps(_mm256_load_ps(a->entry+i),
                                                    _mm256_load_ps(b-
>entry+i)));
    return ret;
}

/**
 * For stressen
 * Use SIMD and OpenMP
*/
Matrix **divide_matrix(const Matrix *const mat) {
    size_t N = mat->n, n = mat->n>>1;
    Matrix **ret = malloc(sizeof(Matrix *)*4);

    #pragma omp parallel for
    for (size_t i = 0; i < 4; ++i) ret[i] = createMatrix(n, NULL);

    #pragma omp parallel for
    for (size_t i = 0; i < n; ++i)
        memcpy(ret[0]->entry+i*n, mat->entry+i*N, sizeof(float)*n);
    #pragma omp parallel for
    for (size_t i = 0; i < n; ++i)
        memcpy(ret[1]->entry+i*n, mat->entry+i*N+n, sizeof(float)*n);
    #pragma omp parallel for
    for (size_t i = 0; i < n; ++i)
        memcpy(ret[2]->entry+i*n, mat->entry+(n+i)*N, sizeof(float)*n);
    #pragma omp parallel for
    for (size_t i = 0; i < n; ++i)
        memcpy(ret[3]->entry+i*n, mat->entry+(n+i)*N+n, sizeof(float)*n);

    return ret;
}

/**
 * For stressen
 * Use SIMD and OpenMP
*/
Matrix *merge_matrix(Matrix **mat) {
    size_t n = mat[0]->n, N = mat[0]->n<<1;
    Matrix *ret = createMatrix(N, NULL);

    #pragma omp parallel for
    for (size_t i = 0; i < n; ++i)
        memcpy(ret->entry+i*N, mat[0]->entry+i*n, sizeof(float)*n);
    #pragma omp parallel for
    for (size_t i = 0; i < n; ++i)
        memcpy(ret->entry+i*N+n, mat[1]->entry+i*n, sizeof(float)*n);
    #pragma omp parallel for
    for (size_t i = 0; i < n; ++i)
        memcpy(ret->entry+(n+i)*N, mat[2]->entry+i*n, sizeof(float)*n);
    #pragma omp parallel for
    for (size_t i = 0; i < n; ++i)
        memcpy(ret->entry+(n+i)*N+n, mat[3]->entry+i*n, sizeof(float)*n);
```

```
    return ret;
}

/**
 * Reccursively divide and solve the multiplication
 * Use OpenMP
*/
Matrix *stressen(const Matrix *const a, const Matrix *const b) {
    size_t n = a->n;
    if (n <= 64)
        return mul_order_avx_omp(a, b);

    Matrix *ret, *mat[7], *c[4],
           **sub_a = divide_matrix(a),
           **sub_b = divide_matrix(b);
    #pragma omp parallel sections
    {
        #pragma omp section
        {
            Matrix *t[2];
            t[0] = subMatrix(sub_a[1], sub_a[3]);
            t[1] = addMatrix(sub_b[2], sub_b[3]);
            mat[0] = stressen(t[0], t[1]);
            deleteMatrix(t[0]);
            deleteMatrix(t[1]);
        }
        #pragma omp section
        {
            Matrix *t[2];
            t[0] = addMatrix(sub_a[0], sub_a[3]);
            t[1] = addMatrix(sub_b[0], sub_b[3]);
            mat[1] = stressen(t[0], t[1]);
            deleteMatrix(t[0]);
            deleteMatrix(t[1]);
        }
        #pragma omp section
        {
            Matrix *t[2];
            t[0] = subMatrix(sub_a[2], sub_a[0]);
            t[1] = addMatrix(sub_b[0], sub_b[1]);
            mat[2] = stressen(t[0], t[1]);
            deleteMatrix(t[0]);
            deleteMatrix(t[1]);
        }
        #pragma omp section
        {
            Matrix *temp;
            temp = addMatrix(sub_a[0], sub_a[1]);
            mat[3] = stressen(temp, sub_b[3]);
            deleteMatrix(temp);
        }
        #pragma omp section
        {
            Matrix *temp;
```

```
            temp = subMatrix(sub_b[1], sub_b[3]);
            mat[4] = stressen(temp, sub_a[0]);
            deleteMatrix(temp);
        }
        #pragma omp section
        {
            Matrix *temp;
            temp = subMatrix(sub_b[2], sub_b[0]);
            mat[5] = stressen(temp, sub_a[3]);
            deleteMatrix(temp);
        }
        #pragma omp section
        {
            Matrix *temp;
            temp = addMatrix(sub_a[2], sub_a[3]);
            mat[6] = stressen(temp, sub_b[0]);
            deleteMatrix(temp);
        }
    }

    #pragma omp parallel sections
    {
        #pragma omp section
        {
            Matrix *t[2];
            t[0] = addMatrix(mat[0], mat[1]);
            t[1] = subMatrix(mat[5], mat[3]);
            c[0] = addMatrix(t[0], t[1]);
            deleteMatrix(t[0]);
            deleteMatrix(t[1]);
        }
        #pragma omp section
        {
            c[1] = addMatrix(mat[3], mat[4]);
        }
        #pragma omp section
        {
            c[2] = addMatrix(mat[5], mat[6]);
        }
        #pragma omp section
        {
            Matrix *t[2];
            t[0] = addMatrix(mat[1], mat[2]);
            t[1] = subMatrix(mat[4], mat[6]);
            c[3] = addMatrix(t[0], t[1]);
            deleteMatrix(t[0]);
            deleteMatrix(t[1]);
        }
    }

    #pragma omp parallel for
    for (size_t i = 0; i < 4; ++i) {
        deleteMatrix(sub_a[i]);
        deleteMatrix(sub_b[i]);
    }
```

```
    free(sub_a);
    free(sub_b);

    ret = merge_matrix(c);
    #pragma omp parallel for
    for(size_t i = 0; i < 4; ++i)
        deleteMatrix(c[i]);

    return ret;
}
```

Part 2 - Results & Verification

16*16

```
root@ArtanisaxLEGION:~/GitHub/CS205-2022Fall/Project4/build# ./Mul
Initiallized
Plain: 1651.5 781.2  10ms
Order+OpenMP: 1651.5 781.2  2ms
Order+OpenMP+avx2: 1651.5 781.2  0ms
OpenBLAS: 1651.5 781.2  1ms
```

128*128

```
root@ArtanisaxLEGION:~/GitHub/CS205-2022Fall/Project4/build# ./Mul
Initiallized
Plain: 65.5 0.0  0ms
Order+OpenMP: 65.5 0.0  0ms
Order+OpenMP+avx2: 65.5 0.0  0ms
OpenBLAS: 65.5 0.0  0ms
```

1024*1024

```
root@ArtanisaxLEGION:~/GitHub/CS205-2022Fall/Project4/build# ./Mul
Initiallized
Plain: 12622.0 634.2  10ms
Order+OpenMP: 12622.0 634.2  1ms
Order+OpenMP+avx2: 12622.0 634.2  0ms
Stressen: 808.7 -216.1  2ms
OpenBLAS: 12622.0 634.2  11ms
```

2048*2048

```
root@ArtanisaxLEGION:~/GitHub/CS205-2022Fall/Project4/build# ./Mul
Initiallized
Plain: 31650.8 39750.1  168620ms
Order+OpenMP: 31650.8 39750.1  2676ms
Order+OpenMP+avx2: 31650.8 39750.1  727ms
OpenBLAS: 31650.9 39750.1  36ms
```

4096*4096

```
root@ArtanisaxLEGION:~/GitHub/CS205-2022Fall/Project4/build# ./Mul
Initiallized
Order+OpenMP: 154765.0 65569.8  21701ms
Order+OpenMP+avx2: 154765.0 65569.8  6293ms
Stressen: 29343774.0 8115808.0  4754ms
OpenBLAS: 154765.2 65569.8  235ms
```

4096*4096 O3

```
root@ArtanisaxLEGION:~/GitHub/CS205-2022Fall/Project4/build# ./Mul
Initiallized
Order+OpenMP: 43625.9 51939.9  2072ms
Order+OpenMP+avx2: 43625.9 51939.9  2480ms
Stressen: 2370379.5 1093460.5  3167ms
OpenBLAS: 43625.8 51939.8  221ms
```

8192*8192 O3

```
root@ArtanisaxLEGION:~/GitHub/CS205-2022Fall/Project4/build# ./Mul
Initiallized
Order+OpenMP: 107024.0 164030.7  30757ms
Order+OpenMP+avx2: 107024.0 164030.7  35755ms
OpenBLAS: 107023.8 164030.7  1712ms
```

16384*16384 O3

```
root@ArtanisaxLEGION:~/GitHub/CS205-2022Fall/Project4/build# ./Mul
Initiallized
Order+OpenMP: 352072.5 287412.6  451020ms
Order+OpenMP+avx2: 352072.5 287412.6  440049ms
OpenBLAS: 352073.0 287413.2  13526ms
```

With `-O3`, avx2 seems useless. And OpenBLAS run much faster than my implementations.

A Fun Discovery

AVX2 is more precise than normal arithmetic operation with `float`.

```
size_t t = 1000000, 10000000, 100000000;
    float ans = 0;
    float *temp = (float *)aligned_alloc(256, sizeof(float)*8);
    for (int i = 0; i < 8; ++i) temp[i] = i;
    gettimeofday(&st, NULL);
    for (int i = 0; i < t; ++i)
        for (int j = 0; j < 8; j++)
            ans += temp[j];
    gettimeofday(&ed, NULL);
```

```
    printf(" %.1f %.0fms\n", ans, 1.0*((ed.tv_sec-st.tv_sec)*1e6+(ed.tv_usec-
st.tv_usec))/1e3);
    gettimeofday(&st, NULL);
    float sum[8];
    __m256 a = _mm256_setzero_ps(), b;
    for (int i = 0; i < t; ++i) {
        b = _mm256_load_ps(temp);
        a = _mm256_add_ps(a, b);
    }
    a = _mm256_hadd_ps(a, a);
    a = _mm256_hadd_ps(a, a);
    _mm256_store_ps(sum, a);
    putchar('\n');
    ans = sum[0]+sum[4];
    gettimeofday(&ed, NULL);
    printf(" %.1f %.0fms\n", ans, 1.0*((ed.tv_sec-st.tv_sec)*1e6+(ed.tv_usec-
st.tv_usec))/1e3);
```

```
root@ArtanisaxLEGION:~/GitHub/CS205-2022Fall/Project4/build# ./Mul
 2800000.0 1ms
 2800000.0 0ms
```

```
root@ArtanisaxLEGION:~/GitHub/CS205-2022Fall/Project4/build# ./Mul
 26396748.0 8ms
 28000000.0 1ms
```

```
root@ArtanisaxLEGION:~/GitHub/CS205-2022Fall/Project4/build# ./Mul
 134217728.0 76ms
 294181472.0 11ms
```

Part 3 - Self-review

Though my result is far far away from OpenBLAS, and even $\Theta(n^3)$ with optimizing can beat algorithm with $\Theta(n^{2.7})$. This project tells me our computer (both software and hardware) has much more potential to be squeezed and some times simple optimization can win better algorithm with bigger constants in reality, which only need limited data size. The depressing results dose encourage me to learn more about principles of computer composition and compilation of software.

Part 4 - Codes & Comments

Please see My GitHub Repository