Project 4

0.链接

https://github.com/Haosonn/CS205-Project4-MatMul_Optimization

1.概述

本次Project需求是实现矩阵乘法的加速。

2.方法对比

- 测试环境：

```
Linux 5.10.102.1-microsoft-standard-WSL2
CPU: 8核 11th Gen Intel(R) Core(TM) i7-1165G7 @ 2.80GHz
Ubuntu 22.04.1 LTS
```

注，本次Project重点在于对矩阵乘法在效率上的提升，因此后续优化部分为了算法简短，考虑到代码的简洁度，仅考虑长宽均为8倍数的方矩阵（以此拓展为任意长宽的矩阵仅需做边界处理，对运算时间影响小）

- 最原始的方法

```
void matmul_plain_float(const float *A, const float *B, float *C, size_t n,
size_t m, size_t p) {
    memset(C, 0, sizeof(float) * n * p);
    for (size_t i = 0; i < n; i++)
        for (size_t j = 0; j < m; j++)
            for (size_t k = 0; k < p; k++)
                C[i * n + j] += A[i * n + k] * B[k * m + j];
}
```

Matrix size n	16	128	1024	2048	4096
(matmul_plain_float) Time (s)	0.000036	0.016746	5.793855	152.820706	619.889226

- ikj访存优化，提高三倍速度左右

```
void matmul_plain_without_ikj_float(const float *A, const float *B, float *C,
size_t n, size_t m, size_t p) {
    memset(C, 0, sizeof(float) * n * p);
    for (size_t i = 0; i < n; i++)
        for (size_t k = 0; k < p; k++)
            for (size_t j = 0; j < m; j++)
                C[i * n + j] += A[i * n + k] * B[k * m + j];
}
```

Matrix size n	16	128	1024	2048	4096
(matmul_plain_ikj_float) Time (s)	0.000006	0.002723	2.555389	49.206797	378.815709

- omp并行优化，矩阵较小的时候用时多，但矩阵较大时比ikj再快三倍

```
void matmul_omp_float(const float *A, const float *B, float *C, size_t n,
size_t m, size_t p) {
    memset(C, 0, sizeof(float) * n * p);
    omp_set_num_threads(omp_get_num_procs());
#pragma omp parallel for schedule(dynamic) default(none) shared(A, B, C, n,
m, p)
    for (size_t i = 0; i < n; i++)
        for (size_t k = 0; k < p; k++)
            for (size_t j = 0; j < m; j++)
                C[i * n + j] += A[i * n + k] * B[k * m + j];
}
```

Matrix size n	16	128	1024	2048	4096	8192
(matmul_plain_omp_float) Time (s)	0.059787	0.011641	2.002420	14.363277	116.972946	1016.564343

- 编译加-O3选项，再次发生质的提升（以下均开启O3编译选项）

Matrix size n	16	128	1024	2048	4096	8192
(matmul_plain_float) Time (s)	0.000036	0.016746	5.793855	152.820706	619.889226	TLE
(matmul_plain_float + O3) Time (s)	0.000019	0.002347	4.072855	52.945550	619.648461	TLE
(matmul_plain_ikj_float) Time (s)	0.000006	0.002723	2.555389	49.206797	378.815709	TLE
(matmul_plain_ikj_float + O3) Time (s)	0.007297	0.009015	0.239604	2.101517	19.914656	164.013545
(matmul_plain_omp_float) Time (s)	0.059787	0.011641	2.002420	14.363277	116.972946	1016.564343
(matmul_plain_omp_float + O3) Time (s)	0.007297	0.021984	0.061817	0.515482	18.101178	60.896478

```
O3
Matrix size: 16
matmul_cblas_float:0.006191
Matrix size: 128
matmul_cblas_float:0.021984
Matrix size: 1024
matmul_cblas_float:0.057892
Matrix size: 2048
matmul_cblas_float:0.057494
Matrix size: 4096
matmul_cblas_float:0.460099
Matrix size: 8192
matmul_cblas_float:4.947546
```

- 在omp的基础上拆开for循环，提升一定效率

```
void matmul_omp_unroll_square_float(const float *A, const float *B, float *C,
size_t n) {
    memset(C, 0, sizeof(float) * n * n);
    omp_set_num_threads(omp_get_num_procs());
#pragma omp parallel for schedule(dynamic) default(none) shared(A, B, C, n)
    for (size_t i = 0; i < n; i++) {
        for (size_t k = 0; k < n; k+=8) {
            float a1 = A[i * n + k];
            float a2 = A[i * n + k + 1];
            float a3 = A[i * n + k + 2];
            float a4 = A[i * n + k + 3];
            float a5 = A[i * n + k + 4];
            float a6 = A[i * n + k + 5];
            float a7 = A[i * n + k + 6];
            float a8 = A[i * n + k + 7];
            for (size_t j = 0; j < n; j+=8) {
                C[i * n + j] += a1 * B[k * n + j];
                C[i * n + j + 1] += a2 * B[(k + 1) * n + j + 1];
                C[i * n + j + 2] += a3 * B[(k + 2) * n + j + 2];
                C[i * n + j + 3] += a4 * B[(k + 3) * n + j + 3];
                C[i * n + j + 4] += a5 * B[(k + 4) * n + j + 4];
                C[i * n + j + 5] += a6 * B[(k + 5) * n + j + 5];
                C[i * n + j + 6] += a7 * B[(k + 6) * n + j + 6];
                C[i * n + j + 7] += a8 * B[(k + 7) * n + j + 7];
            }
        }
    }
}
```

Matrix size n	16	128	1024	2048	4096	8192
(matmul_omp_unroll_square_float) Time (s)	0.015534	0.001324	0.055677	0.939038	6.826273	59.988599

- 采用avx指令集优化，提升效果不显著，甚至内存未对齐的情况下表现差于omp优化版本。（分别检验内存对齐与否的运算效率）

```
//内存已对齐的代码
void matmul_avx2_omp_square_float(const float *A, const float *B, float *C,
size_t n);
```

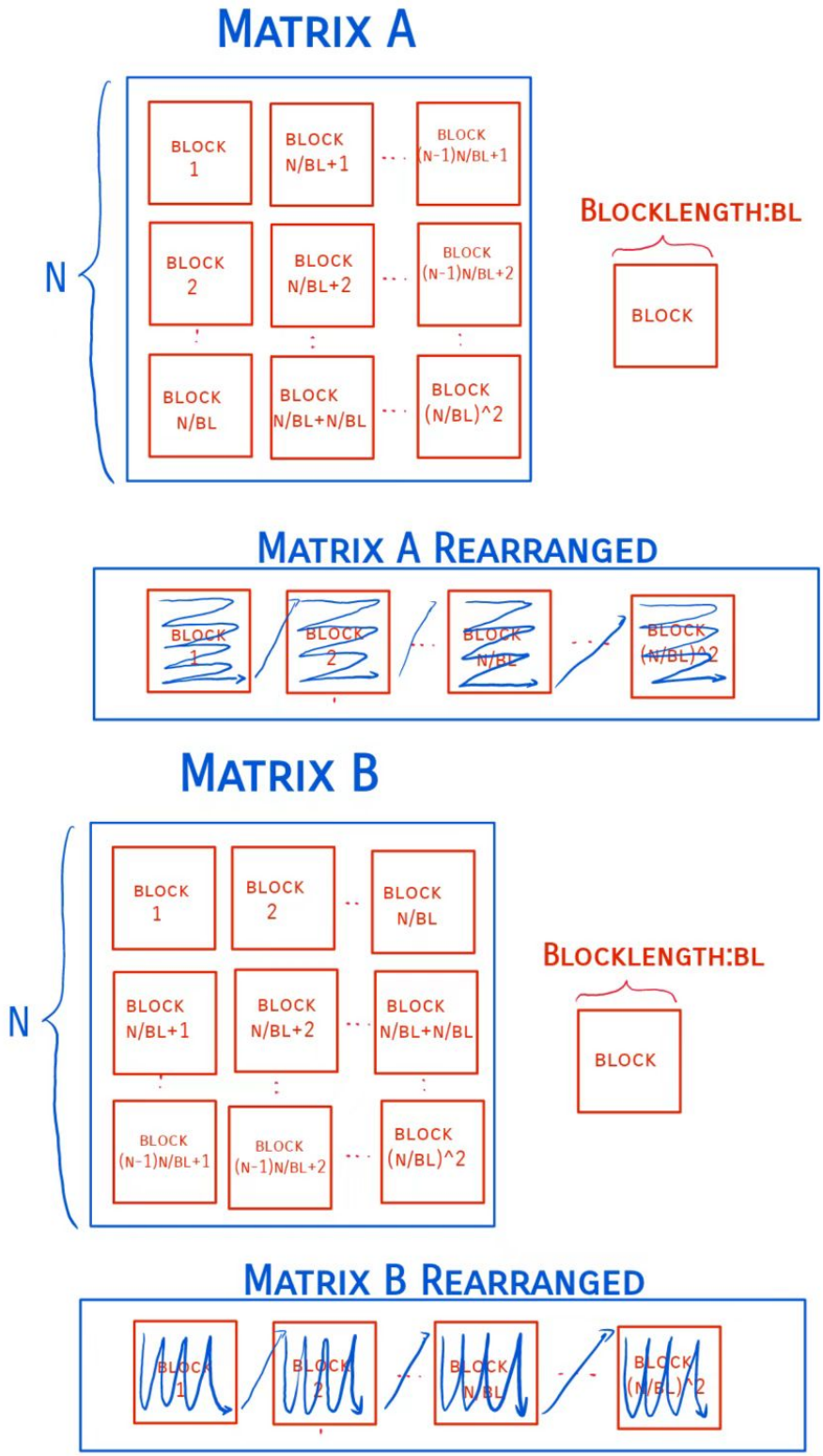
```
void matmul_avx2_omp_square_ufloat(const float *A, const float *B, float *C,
size_t n);
//对比load, store 与 loadu, storeu

//仅贴出内存对齐的版本
void matmul_avx2_omp_square_float(const float *A, const float *B, float *C,
size_t n) {
    memset(C, 0, sizeof(float) * n * n);
    __m256 sum = _mm256_setzero_ps();
    __m256 vec1_B = _mm256_setzero_ps();
    __m256 vec2_B = _mm256_setzero_ps();
    __m256 vec3_B = _mm256_setzero_ps();
    __m256 vec4_B = _mm256_setzero_ps();
    __m256 vecC = _mm256_setzero_ps();
    __m256 scalar1 = _mm256_setzero_ps();
    __m256 scalar2 = _mm256_setzero_ps();
    __m256 scalar3 = _mm256_setzero_ps();
    __m256 scalar4 = _mm256_setzero_ps();
    omp_set_num_threads(omp_get_num_procs());
#pragma omp parallel for schedule(dynamic) default(none)
private(vec1_B,vec2_B,vec3_B,vec4_B,vecC,sum,scalar1,scalar2,scalar3,scalar4)
shared(A,B,C,n)
    for (size_t i = 0; i < n; i++)
    {
        for (size_t k = 0; k < n; k+=4)
        {
            scalar1 = _mm256_set1_ps(A[i * n + k]);
            scalar2 = _mm256_set1_ps(A[i * n + k + 1]);
            scalar3 = _mm256_set1_ps(A[i * n + k + 2]);
            scalar4 = _mm256_set1_ps(A[i * n + k + 3]);
            for (size_t j = 0; j < n; j+=8)
            {
                vec1_B = _mm256_load_ps((B + (k * n + j)));
                vec1_B = _mm256_mul_ps(vec1_B, scalar1);
                vec2_B = _mm256_load_ps((B + ((k + 1) * n + j)));
                vec2_B = _mm256_mul_ps(vec2_B, scalar2);
                vec3_B = _mm256_load_ps((B + ((k + 2) * n + j)));
                vec3_B = _mm256_mul_ps(vec3_B, scalar3);
                vec4_B = _mm256_load_ps((B + ((k + 3) * n + j)));
                vec4_B = _mm256_mul_ps(vec4_B, scalar4);
                vecC = _mm256_load_ps((C + (i * n + j)));
                vecC = _mm256_add_ps(vec1_B, vecC);
                vecC = _mm256_add_ps(vec2_B, vecC);
                vecC = _mm256_add_ps(vec3_B, vecC);
                vecC = _mm256_add_ps(vec4_B, vecC);
                _mm256_store_ps((C + (i * n + j)), vecC);
            }
        }
    }
}
//----------测试load store对齐内存方式-------------
Amem = (float *) malloc((size[i] * size[i] + 255) * sizeof(float));
Bmem = (float *) malloc((size[i] * size[i] + 255) * sizeof(float));
Cmem = (float *) malloc((size[i] * size[i] + 255) * sizeof(float));
A = (float *) (((uintptr_t) Amem + 255) & ~(uintptr_t) 0xFF);
```

```
B = (float *) (((uintptr_t) Bmem + 255) & ~(uintptr_t) 0xFF);
C = (float *) (((uintptr_t) Cmem + 255) & ~(uintptr_t) 0xFF);
```

Matrix size n	16	128	1024	2048	4096	8192
(matmul_avx2_omp_square_float) Time (s)	0.058760	0.005985	0.027502	0.300629	5.805694	56.609771
(matmul_avx2_omp_square_ufloat) Time (s)	0.071480	0.007909	0.078220	1.213298	8.361659	68.485738

```
Matrix size: 16
matmul_transposed_omp_square_float:0.091341
Matrix size: 128
matmul_transposed_omp_square_float:0.000847
Matrix size: 1024
matmul_transposed_omp_square_float:0.025034
Matrix size: 2048
matmul_transposed_omp_square_float:0.417102
Matrix size: 4096
matmul_transposed_omp_square_float:5.036560
Matrix size: 8192
matmul_transposed_omp_square_float:39.275778
```

- 最终尝试，加入分块矩阵优化，引用类似转置的方法，使得矩阵乘法的点乘向量都是连续内存区间，减少cache miss（提速一倍多）。



记矩阵A Rearranged 为AR，矩阵B Rearranged 为BR，如上图，以(N/BlockLength)个Block为一组，记组号i (0<=i<=N/BlockLength-1)，AR的第i组Block的数据为第[i*BlockLength, (i+1)*BlockLength-1] 列，BR的第i组Block的数据为第[i*BlockLength,(i+1)*BlockLength-1] 行，因此，对于AR第i组的所有元素，所有可能的另外乘算元素都在BR的第i组中，我们就可以每次按组进行矩阵乘法。

另一优化是加速 __m256 八个元素求和的方法

```
extern inline float hsum128_ps(__m128 v) {
    __m128 shuf = _mm_movehdup_ps(v);
    __m128 sums = _mm_add_ps(v, shuf);
    shuf        = _mm_movehl_ps(shuf, sums);
    sums        = _mm_add_ss(sums, shuf);
    return        _mm_cvtss_f32(sums);
}
extern inline float hsum256_ps(__m256 v) {
    __m128 vlow  = _mm256_castps256_ps128(v);
    __m128 vhigh = _mm256_extractf128_ps(v, 1);
    vlow  = _mm_add_ps(vlow, vhigh);
    return hsum128_ps(vlow);
}
```

```
void mulBlock(size_t blockLength, const float * b1, const float * b2, float *
res) {
    omp_set_num_threads(omp_get_num_procs());
#pragma omp parallel for schedule(dynamic) default(none)
shared(b1,b2,res,blockLength)
    for (size_t i=0; i<blockLength; i++) {
        for (size_t j=0; j<blockLength; j+=8) {
            for (size_t k=0; k<blockLength; k+=8) {
                __m256 vecA = _mm256_loadu_ps((b1 + (i * blockLength + k)));
                __m256 vecB1 = _mm256_loadu_ps((b2 + (j * blockLength + k)));
                __m256 vecB2 = _mm256_loadu_ps((b2 + ((j + 1) * blockLength +
k)));
                __m256 vecB3 = _mm256_loadu_ps((b2 + ((j + 2) * blockLength +
k)));
                __m256 vecB4 = _mm256_loadu_ps((b2 + ((j + 3) * blockLength +
k)));
                __m256 vecB5 = _mm256_loadu_ps((b2 + ((j + 4) * blockLength +
k)));
                __m256 vecB6 = _mm256_loadu_ps((b2 + ((j + 5) * blockLength +
k)));
                __m256 vecB7 = _mm256_loadu_ps((b2 + ((j + 6) * blockLength +
k)));
                __m256 vecB8 = _mm256_loadu_ps((b2 + ((j + 7) * blockLength +
k)));
                vecB1 = _mm256_mul_ps(vecA, vecB1);
                vecB2 = _mm256_mul_ps(vecA, vecB2);
                vecB3 = _mm256_mul_ps(vecA, vecB3);
                vecB4 = _mm256_mul_ps(vecA, vecB4);
                vecB5 = _mm256_mul_ps(vecA, vecB5);
                vecB6 = _mm256_mul_ps(vecA, vecB6);
                vecB7 = _mm256_mul_ps(vecA, vecB7);
                vecB8 = _mm256_mul_ps(vecA, vecB8);
                __m256 sum = _mm256_setr_ps(hsum256_ps(vecB1), hsum256_ps(vecB2),
hsum256_ps(vecB3), hsum256_ps(vecB4), hsum256_ps(vecB5), hsum256_ps(vecB6),
hsum256_ps(vecB7), hsum256_ps(vecB8));
                __m256 vecC = _mm256_loadu_ps((res + i * blockLength + j ));
                vecC = _mm256_add_ps(vecC, sum);
                _mm256_storeu_ps((res + (i * blockLength + j)), vecC);
            }
        }
```

```
    }
}

void addBlock(size_t blockLength, const float *resBlock, float *C, size_t row,
size_t col, size_t n) {
    omp_set_num_threads(omp_get_num_procs());
#pragma omp parallel for schedule(dynamic) default(none)
shared(resBlock,C,blockLength,row,col,n)
    for (size_t i=0; i<blockLength; i++) {
        for (size_t j=0; j<blockLength; j+=8) {
            __m256 vecC = _mm256_loadu_ps((C + (row + i) * n + col + j));
            __m256 vecRes = _mm256_loadu_ps((resBlock + i * blockLength + j));
            vecC = _mm256_add_ps(vecC, vecRes);
            _mm256_storeu_ps((C + (row + i) * n + col + j), vecC);
        }
    }
}

void matmul_avx2_div_square_float(const float *A, const float *B, float *C,
size_t n) {
    size_t MAXSIZE = 512;
    memset(C, 0, n * n * sizeof(float));
    size_t blockLength = n;
    if (n > MAXSIZE)
        blockLength = MAXSIZE;
    size_t blockSize = blockLength * blockLength;
    size_t blockNumber = n / blockLength; // number of blocks in one row or
column

    float * ARearrange = (float *)malloc(n * n * sizeof(float));
    float * BRearrange = (float *)malloc(n * n * sizeof(float));
    size_t firstIndexK = 0;
    for (size_t k = 0; k<n; k+=blockLength) {
        for (size_t i = 0; i<n; i++) {
            for (size_t j = 0; j<blockLength;j++) {
                ARearrange[firstIndexK + i * blockLength + j] = A[i * n + k + j];
//                BRearrange[firstIndexK + i * blockLength + j] = BTrans[i * n +
k + j];
                BRearrange[firstIndexK + i * blockLength + j] = B[(k + j) * n +
i];
            }
        }
        firstIndexK += blockLength * n;
    }

#ifndef UNIX
    float * b1 = (float *)malloc(blockSize * sizeof(float));
    float * b2 = (float *)malloc(blockSize * sizeof(float));
    float * resBlock = (float *)malloc(blockSize * sizeof(float));
    float * blockGroupA = (float *)malloc(blockNumber * blockSize *
sizeof(float));
    float * blockGroupB = (float *)malloc(blockNumber * blockSize *
sizeof(float));
#else
    float * b1 = (float *)aligned_alloc(256, blockSize * sizeof(float));
```

```
    float * b2 = (float *)aligned_alloc(256, blockSize * sizeof(float));
    float * resBlock = (float *)aligned_alloc(256, blockSize * sizeof(float));
    float * blockGroupA = (float *)aligned_alloc(256, blockNumber * blockSize *
sizeof(float));
    float * blockGroupB = (float *)aligned_alloc(256, blockNumber * blockSize *
sizeof(float));
#endif


    for (size_t gr = 0; gr < blockNumber; gr++) { //group
        memcpy(blockGroupA, ARearrange + gr * blockSize * blockNumber,
blockNumber * blockSize * sizeof(float));
        memcpy(blockGroupB, BRearrange + gr * blockSize * blockNumber,
blockNumber * blockSize * sizeof(float));
        for (size_t ag = 0; ag < blockNumber; ag++) {
            memcpy(b1, blockGroupA + ag * blockSize, blockSize * sizeof(float));
            for (size_t bg = 0; bg < blockNumber; bg++) {
                memcpy(b2, blockGroupB + bg * blockSize, blockSize *
sizeof(float));
                memset(resBlock, 0, blockSize * sizeof(float));
                mulBlock(blockLength, b1, b2, resBlock);
                addBlock(blockLength, resBlock, C, ag * blockLength, bg *
blockLength, n);
            }
        }
    }
    free(ARearrange);
    free(BRearrange);
    free(b1);
    free(b2);
    free(resBlock);
    free(blockGroupA);
    free(blockGroupB);
}
```

Matrix size n	16	128	1024	2048	4096	8192
(matmul_avx2_div_square_float) Time (s)	0.030221	0.043129	0.107231	0.764453	3.212300	24.931550

- 最后的matmul_improve函数，小矩阵通过omp实现，大矩阵通过最后的方法实现，与CBLAS对比。