**Optimization of Matrix Multiplication on Intel Machine**

Jason Su, Harry He

June 1st, 2015

**I. Introduction**

Matrix multiplication is widely used in image processing and many other scientific fields, so it is important to optimize it fast enough to fit applications nowadays. In this project, we optimize a 1024\*1024 double-precision matrix multiplication over random double precision floating points from 0 to 1. There are five optimization technics used in this project: OpenMp, SIMD, Cache Blocking, Loop Unrolling and Register Blocking.

**II. Testing Environment**

CPU: Intel(R) Core(TM) i7-4850HQ CPU @ 2.30GHz. 4 cores, 8 threads

Supported SIMD sets: SSE SSE2 SSE3 SSE4.1 SSE4.2 AVX AVX2 FMA

Cache line size: 64 Bytes for all caches

Associativity: 8 for all caches

L1 instruction cache: 32KB per core

L1 data cache: 32KB per core

L2 cache: 256KB per core

L3 cache: 6MB unified

L4 cache: 128MB unified

Operating System: Ubuntu 14.04 LTS

Compiler: Gcc 4.8.4

Compiling Flag: -Wall -mavx2 -mfma -O1 -std=c99 -fopenmp

Measurement Method: Close all user applications. 20 Measurements. Use average number.

**III. Result**

|  |  |  |
| --- | --- | --- |
| Method | Time(s) | Speedup |
| Naive | 1.069 | 1.00000 |
| OpenMP | 0.307 | 3.4814 |
| SIMD | 0.338 | 3.1587 |
| Cache Blocking | 1.125 | 0.9495 |
| Loop Unrolling | 0.725 | 1.4741 |
| Register Blocking | 0.447 | 2.3926 |
| OpenMP & SIMD | 0.095 | 11.2838 |
| OpenMP & SIMD & Cache Blocking | 0.077 | 13.9368 |
| OpenMP & SIMD & Cache Blocking & Loop Unrolling | 0.050 | 21.4819 |
| OpenMP & SIMD & Cache Blocking & Register Blocking | 0.043 | 25.1387 |

**IV Analysis**

1. Naive

void naive(double\* restrict result,

const double\* restrict matrix1, const double\* restrict matrix2) {

memset(result, 0, WIDTH\*HEIGHT\*sizeof(double));

for (int i = 0; i < WIDTH; i++)

{

for (int k = 0; k < WIDTH; k++)

{

double t = matrix1[i\*WIDTH+k];

for (int j = 0; j < HEIGHT; j++)

{

result[i\*WIDTH+j] += t\*matrix2[k\*WIDTH+j];

}}}}

Using formula result(i,j) += matrix1(i,k)\*matrix2(k,j), writing the naive version as the loop order of ikj or kij gives the best cache locality. Because only j changes in the most inner loop and result and matrix2 are accessed continuously, the lowest cache miss rate is observed. Given the size of double precision number which is 8 bytes and the cache line which is 64 bytes, the miss rate is only about ⅛=12.5%. In comparion, the loop order of ijk or jik gives the miss rate of 5/16 = 31.25%. Meanwhile, the loop order of jki or kji gives the miss rate of ½ = 50%

2. Openmp

void openmp(double\* restrict result,

const double\* restrict matrix1, const double\* restrict matrix2) {

memset(result, 0, WIDTH\*HEIGHT\*sizeof(double));

// split work into all 8 threads

#pragma omp parallel for

for (int i = 0; i < WIDTH; i++)

{

for (int k = 0; k < WIDTH; k++)

{

double t = matrix1[i\*WIDTH+k];

for (int j = 0; j < HEIGHT; j++)

{

result[i\*WIDTH+j] += t\*matrix2[k\*WIDTH+j];

}}}}

Speedup: 3.48

Since there are 4 cores in the CPU, the theoretical maximum speedup is 4. There are two reasons why that speedup is not observed: First, the optimization function is not completely parallelizable. Then by the Amdahl’s Law, the speedup is limited. Second, this program is not the only program running on the CPU. CPU also runs system software. This creates imbalance between cores.

3. SIMD

void simd(double\* restrict result,

const double\* restrict matrix1, const double\* restrict matrix2) {

memset(result, 0, WIDTH\*HEIGHT\*sizeof(double));

for (int i = 0; i < WIDTH; i++)

{

for (int k = 0; k < WIDTH; k++)

{

\_\_m256d t = \_mm256\_broadcast\_sd(matrix1+i\*WIDTH+k); // AVX

for (int j = 0; j < HEIGHT; j+=4)

{

\_\_m256d r = \_mm256\_load\_pd(result+i\*WIDTH+j); // AVX

r = \_mm256\_fmadd\_pd(\_mm256\_load\_pd(matrix2+k\*WIDTH+j), t, r); // FMA

\_mm256\_store\_pd(result+i\*WIDTH+j, r); // AVX

}}}}

Speedup: 3.15

The Intel CPU supports the Intel AVX2 and FMA extensions, which uses 256bits SIMD registers. The size of double-precision number is 64bits each, so 4 operations can be done in parallel. This will give a speedup of 4 theatrically. However, because not all code is not able to be written in SIMD instructions, such as branch instructions, the speedup lower than 4 should be expected.

4. Cache Blocking

void cacheBlock(double\* restrict result,

const double\* restrict matrix1, const double\* restrict matrix2) {

memset(result, 0, WIDTH\*HEIGHT\*sizeof(double));

const int BLOCK1 = 512;

for (int kk = 0; kk < WIDTH; kk+=BLOCK1) {

for (int i = 0; i < WIDTH; i++)

{

for (int k = kk; k < kk+BLOCK1; k++)

{

double t = matrix1[i\*WIDTH+k];

for (int j = 0; j < HEIGHT;j++) {

result[i\*WIDTH+j] += t\*matrix2[k\*WIDTH+j];

}}}}}

Speedup:0.95

The reason behind the block size of 512 is that we want to maximize the cache hit rate. For a specific i, the data goes into cache is one line of matrix1(1024 double-precision numbers), one line of results(1024 double-precision numbers) and one block of matrix2(512\*1024 double-precision numbers). We want cache can afford the data so that when I changes, the data of matrix2 can be reused. The data is roughly 4 Mega Bytes, which can be afforded by the 6MB L3 cache. However, the effect of cache blocking is worse than our expectation because we add one more loop to the naive version. Moreover, all the matrixes, which are the size of 24M can fit in the L4 cache of 128M. Therefore, cache blocking is not very effective.

5.Loop Unrolling

void loopUnroll(double\* restrict result,

const double\* restrict matrix1, const double\* restrict matrix2) {

memset(result, 0, WIDTH\*HEIGHT\*sizeof(double));

for (int i = 0; i < WIDTH; i++)

{

double\* dest = result + i\*WIDTH;

for (int k = 0; k < WIDTH; k++)

{

double t = matrix1[i\*WIDTH+k];

const double\* mat2 = matrix2+k\*WIDTH;

for (int j = 0; j < HEIGHT; j+=32)

{

dest[j] += t\*mat2[j];

dest[j+1] += t\*mat2[j+1];

dest[j+2] += t\*mat2[j+2];

dest[j+3] += t\*mat2[j+3];

dest[j+4] += t\*mat2[j+4];

dest[j+5] += t\*mat2[j+5];

dest[j+6] += t\*mat2[j+6];

dest[j+7] += t\*mat2[j+7];

dest[j+8] += t\*mat2[j+8];

dest[j+9] += t\*mat2[j+9];

dest[j+10] += t\*mat2[j+10];

dest[j+11] += t\*mat2[j+11];

dest[j+12] += t\*mat2[j+12];

dest[j+13] += t\*mat2[j+13];

dest[j+14] += t\*mat2[j+14];

dest[j+15] += t\*mat2[j+15];

dest[j+16] += t\*mat2[j+16];

dest[j+17] += t\*mat2[j+17];

dest[j+18] += t\*mat2[j+18];

dest[j+19] += t\*mat2[j+19];

dest[j+20] += t\*mat2[j+20];

dest[j+21] += t\*mat2[j+21];

dest[j+22] += t\*mat2[j+22];

dest[j+23] += t\*mat2[j+23];

dest[j+24] += t\*mat2[j+24];

dest[j+25] += t\*mat2[j+25];

dest[j+26] += t\*mat2[j+26];

dest[j+27] += t\*mat2[j+27];

dest[j+28] += t\*mat2[j+28];

dest[j+29] += t\*mat2[j+29];

dest[j+30] += t\*mat2[j+30];

dest[j+31] += t\*mat2[j+31];

}}}}

Speedup: 1.47

By doing loop unrolling, the amount of branch instructions are significantly decreased. According to Intel’s documentation, the throughput of branch instructions are much smaller than that of arithmetic instructions. Therefore, a significant speedup is observed.

6. Register Blocking

void registerBlock(double\* restrict result,

const double\* restrict matrix1, const double\* restrict matrix2) {

memset(result, 0, WIDTH\*HEIGHT\*sizeof(double));

for (int i = 0; i < WIDTH; i++)

{

const double\* mat1 = matrix1+i\*WIDTH;

double\* dest = result+i\*WIDTH;

for (int k = 0; k < WIDTH; k+=4)

{

double t0 = mat1[k];

double t1 = mat1[k+1];

double t2 = mat1[k+2];

double t3 = mat1[k+3];

const double\* mat2 = matrix2+k\*WIDTH;

for (int j = 0; j < HEIGHT; j+=4) {

double sum0 = dest[j];

double sum1 = dest[j+1];

double sum2 = dest[j+2];

double sum3 = dest[j+3];

sum0 += t0\*mat2[j];

sum1 += t0\*mat2[j+1];

sum2 += t0\*mat2[j+2];

sum3 += t0\*mat2[j+3];

sum0 += t1\*mat2[j+WIDTH];

sum1 += t1\*mat2[j+WIDTH+1];

sum2 += t1\*mat2[j+WIDTH+2];

sum3 += t1\*mat2[j+WIDTH+3];

sum0 += t2\*mat2[j+WIDTH\*2];

sum1 += t2\*mat2[j+WIDTH\*2+1];

sum2 += t2\*mat2[j+WIDTH\*2+2];

sum3 += t2\*mat2[j+WIDTH\*2+3];

sum0 += t3\*mat2[j+WIDTH\*3];

sum1 += t3\*mat2[j+WIDTH\*3+1];

sum2 += t3\*mat2[j+WIDTH\*3+2];

sum3 += t3\*mat2[j+WIDTH\*3+3];

dest[j] = sum0;

dest[j+1] = sum1;

dest[j+2] = sum2;

dest[j+3] = sum3;

}}}}

Speedup: 2.39

There are 16 general registers. We try to make the most use of them to reduce the word load of cache. In the naive version, cache is visited twice for each (i,j,k) pair in the most inner loop. However, with register blocking, 16 (i,j,k) pairs are done at once in the most inner loop, and this visit cache 20 times. Therefore, for each (i,j,k) pair, cache is only accessed 20/16 = 1.25 times, which gives the speedup observed.

7. Openmp & SIMD

void openmp\_simd(double\* restrict result,

const double\* restrict matrix1, const double\* restrict matrix2) {

memset(result, 0, WIDTH\*HEIGHT\*sizeof(double));

#pragma omp parallel for

for (int i = 0; i < WIDTH; i++)

{

for (int k = 0; k < WIDTH; k++)

{

\_\_m256d t = \_mm256\_broadcast\_sd(matrix1+i\*WIDTH+k);

for (int j = 0; j < HEIGHT; j+=4)

{

\_\_m256d r = \_mm256\_load\_pd(result+i\*WIDTH+j);

r = \_mm256\_fmadd\_pd(\_mm256\_load\_pd(matrix2+k\*WIDTH+j), t, r);

\_mm256\_store\_pd(result+i\*WIDTH+j, r);

}}}}

Speedup: 11.28

The speedup is very close to, or even better than the product of speedups of openmp and simd, which is 3.4814\*3.1587= 10.9967. This shows that openmp and simd can work together perfectly. The two optimization technologies are strongly independent.

8.OpenMP & SIMD & Cache Blocking

void openmp\_simd\_cacheBlock(double\* restrict result,

const double\* restrict matrix1, const double\* restrict matrix2) {

const int BLOCK = 512;

memset(result, 0, WIDTH\*HEIGHT\*sizeof(double));

#pragma omp parallel

{

for (int kk = 0; kk < WIDTH; kk += BLOCK) {

#pragma omp for

for (int i = 0; i < WIDTH; i++)

{

for (int k = kk; k < kk+BLOCK; k++)

{

\_\_m256d t = \_mm256\_broadcast\_sd(matrix1+i\*WIDTH+k);

for (int j = 0; j < HEIGHT; j+=4)

{

\_\_m256d r = \_mm256\_load\_pd(result+i\*WIDTH+j);

r = \_mm256\_fmadd\_pd(\_mm256\_load\_pd(matrix2+k\*WIDTH+j), t, r);

\_mm256\_store\_pd(result+i\*WIDTH+j, r);

}}}}}}

The speedup of OpenMP & SIMD is 11.2838, while the speedup of OpenMP & SIMD & cache blocking is 13.9368. We can see that cache blocking start to show it function. It is more sensetive in multiple application of optimization.

9. OpenMp & SIMD & Cache Blocking & Loop Unrolling

void openmp\_simd\_cacheBlock\_loopUnroll(double\* restrict result,

const double\* restrict matrix1, const double\* restrict matrix2) {

const int BLOCK = 512;

memset(result, 0, WIDTH\*HEIGHT\*sizeof(double));

#pragma omp parallel

{

for (int kk = 0; kk < WIDTH; kk += BLOCK) {

#pragma omp for

for (int i = 0; i < WIDTH; i++)

{

for (int k = kk; k < kk+BLOCK; k++)

{

\_\_m256d t = \_mm256\_broadcast\_sd(matrix1+i\*WIDTH+k);

for (int j = 0; j < HEIGHT; j+=32)

{

double\* dest = result+i\*WIDTH+j;

const double\* mat2 = matrix2+k\*WIDTH+j;

\_\_m256d r;

r = \_mm256\_load\_pd(dest);

r = \_mm256\_fmadd\_pd(\_mm256\_load\_pd(mat2), t, r);

\_mm256\_store\_pd(dest, r);

r = \_mm256\_load\_pd(dest+4);

r = \_mm256\_fmadd\_pd(\_mm256\_load\_pd(mat2+4), t, r);

\_mm256\_store\_pd(dest+4, r);

r = \_mm256\_load\_pd(dest+8);

r = \_mm256\_fmadd\_pd(\_mm256\_load\_pd(mat2+8), t, r);

\_mm256\_store\_pd(dest+8, r);

r = \_mm256\_load\_pd(dest+12);

r = \_mm256\_fmadd\_pd(\_mm256\_load\_pd(mat2+12), t, r);

\_mm256\_store\_pd(dest+12, r);

r = \_mm256\_load\_pd(dest+16);

r = \_mm256\_fmadd\_pd(\_mm256\_load\_pd(mat2+16), t, r);

\_mm256\_store\_pd(dest+16, r);

r = \_mm256\_load\_pd(dest+20);

r = \_mm256\_fmadd\_pd(\_mm256\_load\_pd(mat2+20), t, r);

\_mm256\_store\_pd(dest+20, r);

r = \_mm256\_load\_pd(dest+24);

r = \_mm256\_fmadd\_pd(\_mm256\_load\_pd(mat2+24), t, r);

\_mm256\_store\_pd(dest+24, r);

r = \_mm256\_load\_pd(dest+28);

r = \_mm256\_fmadd\_pd(\_mm256\_load\_pd(mat2+28), t, r);

\_mm256\_store\_pd(dest+28, r);

}}}}}}

Comparing to its speedup of 21.4819 and the speedups of openmp&simd&cache blocking and loop unrolling ,13.93\*1.47=20.47, it shows that loop unrolling works much better with other optimization. In naive version, loop unrolling cannot eliminate a for loop. However, in this version, due to the use of cache blocking, loop unrolling can eliminate a for loop, reducing the execution of branch instructions. Due to the increased code complexity of code compared to naive version, loop unrolling also enables compiler to optimize code to work better with Intel’s out-of-order-execution technology.

10. OpenMp & SIMD & Cache Blocking & Loop Unrolling & Register Blocking

void openmp\_simd\_cacheBlock\_loopUnroll\_registerBlock(double\* restrict result,

const double\* restrict matrix1, const double\* restrict matrix2) {

const int BLOCK = 512;

memset(result, 0, WIDTH\*HEIGHT\*sizeof(double));

#pragma omp parallel

{

for (int kk = 0; kk < WIDTH; kk += BLOCK) {

#pragma omp for

for (int i = 0; i < WIDTH; i++)

{

const double\* mat1 = matrix1+i\*WIDTH;

double\* dest = result+i\*WIDTH;

for (int k = kk; k < kk+BLOCK; k+=8)

{

const double\* mat2 = matrix2+k\*WIDTH;

\_\_m256d t0 = \_mm256\_broadcast\_sd(mat1+k);

\_\_m256d t1 = \_mm256\_broadcast\_sd(mat1+k+1);

\_\_m256d t2 = \_mm256\_broadcast\_sd(mat1+k+2);

\_\_m256d t3 = \_mm256\_broadcast\_sd(mat1+k+3);

\_\_m256d t4 = \_mm256\_broadcast\_sd(mat1+k+4);

\_\_m256d t5 = \_mm256\_broadcast\_sd(mat1+k+5);

\_\_m256d t6 = \_mm256\_broadcast\_sd(mat1+k+6);

\_\_m256d t7 = \_mm256\_broadcast\_sd(mat1+k+7);

for (int j = 0; j < HEIGHT; j+= 32)

{

\_\_m256d sum0 = \_mm256\_load\_pd(dest+j);

\_\_m256d sum1 = \_mm256\_load\_pd(dest+j+4);

\_\_m256d sum2 = \_mm256\_load\_pd(dest+j+8);

\_\_m256d sum3 = \_mm256\_load\_pd(dest+j+12);

\_\_m256d sum4 = \_mm256\_load\_pd(dest+j+16);

\_\_m256d sum5 = \_mm256\_load\_pd(dest+j+20);

\_\_m256d sum6 = \_mm256\_load\_pd(dest+j+24);

\_\_m256d sum7 = \_mm256\_load\_pd(dest+j+28);

sum0 = \_mm256\_fmadd\_pd(\_mm256\_load\_pd(mat2+j), t0, sum0);

sum1 = \_mm256\_fmadd\_pd(\_mm256\_load\_pd(mat2+j+4), t0, sum1);

sum2 = \_mm256\_fmadd\_pd(\_mm256\_load\_pd(mat2+j+8), t0, sum2);

sum3 = \_mm256\_fmadd\_pd(\_mm256\_load\_pd(mat2+j+12), t0, sum3);

sum4 = \_mm256\_fmadd\_pd(\_mm256\_load\_pd(mat2+j+16), t0, sum4);

sum5 = \_mm256\_fmadd\_pd(\_mm256\_load\_pd(mat2+j+20), t0, sum5);

sum6 = \_mm256\_fmadd\_pd(\_mm256\_load\_pd(mat2+j+24), t0, sum6);

sum7 = \_mm256\_fmadd\_pd(\_mm256\_load\_pd(mat2+j+28), t0, sum7);

sum0 = \_mm256\_fmadd\_pd(\_mm256\_load\_pd(mat2+j+WIDTH), t1, sum0);

sum1 = \_mm256\_fmadd\_pd(\_mm256\_load\_pd(mat2+j+4+WIDTH), t1, sum1);

sum2 = \_mm256\_fmadd\_pd(\_mm256\_load\_pd(mat2+j+8+WIDTH), t1, sum2);

sum3 = \_mm256\_fmadd\_pd(\_mm256\_load\_pd(mat2+j+12+WIDTH), t1, sum3);

sum4 = \_mm256\_fmadd\_pd(\_mm256\_load\_pd(mat2+j+16+WIDTH), t1, sum4);

sum5 = \_mm256\_fmadd\_pd(\_mm256\_load\_pd(mat2+j+20+WIDTH), t1, sum5);

sum6 = \_mm256\_fmadd\_pd(\_mm256\_load\_pd(mat2+j+24+WIDTH), t1, sum6);

sum7 = \_mm256\_fmadd\_pd(\_mm256\_load\_pd(mat2+j+28+WIDTH), t1, sum7);

sum0 = \_mm256\_fmadd\_pd(\_mm256\_load\_pd(mat2+j+WIDTH\*2), t2, sum0);

sum1 = \_mm256\_fmadd\_pd(\_mm256\_load\_pd(mat2+j+4+WIDTH\*2), t2, sum1);

sum2 = \_mm256\_fmadd\_pd(\_mm256\_load\_pd(mat2+j+8+WIDTH\*2), t2, sum2);

sum3 = \_mm256\_fmadd\_pd(\_mm256\_load\_pd(mat2+j+12+WIDTH\*2), t2, sum3);

sum4 = \_mm256\_fmadd\_pd(\_mm256\_load\_pd(mat2+j+16+WIDTH\*2), t2, sum4);

sum5 = \_mm256\_fmadd\_pd(\_mm256\_load\_pd(mat2+j+20+WIDTH\*2), t2, sum5);

sum6 = \_mm256\_fmadd\_pd(\_mm256\_load\_pd(mat2+j+24+WIDTH\*2), t2, sum6);

sum7 = \_mm256\_fmadd\_pd(\_mm256\_load\_pd(mat2+j+28+WIDTH\*2), t2, sum7);

sum0 = \_mm256\_fmadd\_pd(\_mm256\_load\_pd(mat2+j+WIDTH\*3), t3, sum0);

sum1 = \_mm256\_fmadd\_pd(\_mm256\_load\_pd(mat2+j+4+WIDTH\*3), t3, sum1);

sum2 = \_mm256\_fmadd\_pd(\_mm256\_load\_pd(mat2+j+8+WIDTH\*3), t3, sum2);

sum3 = \_mm256\_fmadd\_pd(\_mm256\_load\_pd(mat2+j+12+WIDTH\*3), t3, sum3);

sum4 = \_mm256\_fmadd\_pd(\_mm256\_load\_pd(mat2+j+16+WIDTH\*3), t3, sum4);

sum5 = \_mm256\_fmadd\_pd(\_mm256\_load\_pd(mat2+j+20+WIDTH\*3), t3, sum5);

sum6 = \_mm256\_fmadd\_pd(\_mm256\_load\_pd(mat2+j+24+WIDTH\*3), t3, sum6);

sum7 = \_mm256\_fmadd\_pd(\_mm256\_load\_pd(mat2+j+28+WIDTH\*3), t3, sum7);

sum0 = \_mm256\_fmadd\_pd(\_mm256\_load\_pd(mat2+j+WIDTH\*4), t4, sum0);

sum1 = \_mm256\_fmadd\_pd(\_mm256\_load\_pd(mat2+j+4+WIDTH\*4), t4, sum1);

sum2 = \_mm256\_fmadd\_pd(\_mm256\_load\_pd(mat2+j+8+WIDTH\*4), t4, sum2);

sum3 = \_mm256\_fmadd\_pd(\_mm256\_load\_pd(mat2+j+12+WIDTH\*4), t4, sum3);

sum4 = \_mm256\_fmadd\_pd(\_mm256\_load\_pd(mat2+j+16+WIDTH\*4), t4, sum4);

sum5 = \_mm256\_fmadd\_pd(\_mm256\_load\_pd(mat2+j+20+WIDTH\*4), t4, sum5);

sum6 = \_mm256\_fmadd\_pd(\_mm256\_load\_pd(mat2+j+24+WIDTH\*4), t4, sum6);

sum7 = \_mm256\_fmadd\_pd(\_mm256\_load\_pd(mat2+j+28+WIDTH\*4), t4, sum7);

sum0 = \_mm256\_fmadd\_pd(\_mm256\_load\_pd(mat2+j+WIDTH\*5), t5, sum0);

sum1 = \_mm256\_fmadd\_pd(\_mm256\_load\_pd(mat2+j+4+WIDTH\*5), t5, sum1);

sum2 = \_mm256\_fmadd\_pd(\_mm256\_load\_pd(mat2+j+8+WIDTH\*5), t5, sum2);

sum3 = \_mm256\_fmadd\_pd(\_mm256\_load\_pd(mat2+j+12+WIDTH\*5), t5, sum3);

sum4 = \_mm256\_fmadd\_pd(\_mm256\_load\_pd(mat2+j+16+WIDTH\*5), t5, sum4);

sum5 = \_mm256\_fmadd\_pd(\_mm256\_load\_pd(mat2+j+20+WIDTH\*5), t5, sum5);

sum6 = \_mm256\_fmadd\_pd(\_mm256\_load\_pd(mat2+j+24+WIDTH\*5), t5, sum6);

sum7 = \_mm256\_fmadd\_pd(\_mm256\_load\_pd(mat2+j+28+WIDTH\*5), t5, sum7);

sum0 = \_mm256\_fmadd\_pd(\_mm256\_load\_pd(mat2+j+WIDTH\*6), t6, sum0);

sum1 = \_mm256\_fmadd\_pd(\_mm256\_load\_pd(mat2+j+4+WIDTH\*6), t6, sum1);

sum2 = \_mm256\_fmadd\_pd(\_mm256\_load\_pd(mat2+j+8+WIDTH\*6), t6, sum2);

sum3 = \_mm256\_fmadd\_pd(\_mm256\_load\_pd(mat2+j+12+WIDTH\*6), t6, sum3);

sum4 = \_mm256\_fmadd\_pd(\_mm256\_load\_pd(mat2+j+16+WIDTH\*6), t6, sum4);

sum5 = \_mm256\_fmadd\_pd(\_mm256\_load\_pd(mat2+j+20+WIDTH\*6), t6, sum5);

sum6 = \_mm256\_fmadd\_pd(\_mm256\_load\_pd(mat2+j+24+WIDTH\*6), t6, sum6);

sum7 = \_mm256\_fmadd\_pd(\_mm256\_load\_pd(mat2+j+28+WIDTH\*6), t6, sum7);

sum0 = \_mm256\_fmadd\_pd(\_mm256\_load\_pd(mat2+j+WIDTH\*7), t7, sum0);

sum1 = \_mm256\_fmadd\_pd(\_mm256\_load\_pd(mat2+j+4+WIDTH\*7), t7, sum1);

sum2 = \_mm256\_fmadd\_pd(\_mm256\_load\_pd(mat2+j+8+WIDTH\*7), t7, sum2);

sum3 = \_mm256\_fmadd\_pd(\_mm256\_load\_pd(mat2+j+12+WIDTH\*7), t7, sum3);

sum4 = \_mm256\_fmadd\_pd(\_mm256\_load\_pd(mat2+j+16+WIDTH\*7), t7, sum4);

sum5 = \_mm256\_fmadd\_pd(\_mm256\_load\_pd(mat2+j+20+WIDTH\*7), t7, sum5);

sum6 = \_mm256\_fmadd\_pd(\_mm256\_load\_pd(mat2+j+24+WIDTH\*7), t7, sum6);

sum7 = \_mm256\_fmadd\_pd(\_mm256\_load\_pd(mat2+j+28+WIDTH\*7), t7, sum7);

\_mm256\_store\_pd(dest+j, sum0);

\_mm256\_store\_pd(dest+j+4, sum1);

\_mm256\_store\_pd(dest+j+8, sum2);

\_mm256\_store\_pd(dest+j+12, sum3);

\_mm256\_store\_pd(dest+j+16, sum4);

\_mm256\_store\_pd(dest+j+20, sum5);

\_mm256\_store\_pd(dest+j+24, sum6);

\_mm256\_store\_pd(dest+j+28, sum7);

}}}}}}

Comparing to its speedup of 25.1387 and the speedups of individual optimizations, 3.4814\*3.1587\*0.9495\*1.4741\*2.3926 = 36.8259, it shows that this version has a huge diminishing returns. It is mainly caused by the similarity between loop unrolling and register blocking. Register blocking always involves repeation of similar code in order to make use of all registers, which is the same as loop unrolling. Therefore, the effect of loop unrolling is almost included by the register blocking, causing the huge diminishing returns.

**V. Other Code**

1. header file

#ifndef MATRIX\_OPTIMIZATION\_HEADER

#define MATRIX\_OPTIMIZATION\_HEADER

#include <string.h>

#include <immintrin.h>

#define WIDTH 1024

#define HEIGHT 1024

extern int compare\_matrix(double\* sample, double\* reference);

extern void optimization\_naive(double\* restrict result,

const double\* restrict matrix1, const double\* restrict matrix2);

extern void optimization\_openmp(double\* restrict result,

const double\* restrict matrix1, const double\* restrict matrix2);

extern void optimization\_simd(double\* restrict result,

const double\* restrict matrix1, const double\* restrict matrix2);

extern void optimization\_cache\_blocking(double\* restrict result,

const double\* restrict matrix1, const double\* restrict matrix2);

extern void optimization\_loop\_unrolling(double\* restrict result,

const double\* restrict matrix1, const double\* restrict matrix2);

extern void optimization\_register\_blocking(double\* restrict result,

const double\* restrict matrix1, const double\* restrict matrix2);

extern void optimization\_openmp\_simd(double\* restrict result,

const double\* restrict matrix1, const double\* restrict matrix2);

extern void optimization\_openmp\_simd\_cache\_blocking(double\* restrict result,

const double\* restrict matrix1, const double\* restrict matrix2);

#endif

1. main function

/\*

============================================================================

Name : main.c

Author : Jason Su and Harry He

Version : 1.0.0

Copyright : GNU

Description : 1024\*1024 double-precision matrix multliplication

============================================================================

\*/

#include <omp.h>

#include <stdio.h>

#include <stdlib.h>

#include <stdint.h>

#include <sys/time.h>

#include <time.h>

#include <immintrin.h>

#include <xmmintrin.h>

#include "header.h"

/\* Return the current system time in micro seconds \*/

static inline uint64\_t timestamp\_us()

{

struct timeval tv;

gettimeofday(&tv, NULL);

return 1000000L \* tv.tv\_sec + tv.tv\_usec;

}

int main(int argc, char \*argv[])

{

srand (time(NULL));

omp\_set\_num\_threads(8);

double\* matrix1s[NUM\_OF\_OPTIMIZATIONS] = {};

double\* matrix2s[NUM\_OF\_OPTIMIZATIONS] = {};

double\* results[NUM\_OF\_OPTIMIZATIONS] = {}; /\* Store the outputs of optimizations \*/

double\* references[NUM\_OF\_OPTIMIZATIONS] = {};

double times[NUM\_OF\_OPTIMIZATIONS] = {}; /\* Store the time measurement of optimizations \*/

int errors[NUM\_OF\_OPTIMIZATIONS] = {}; /\* Store whether the optimization has the correct result \*/

void (\*functions[NUM\_OF\_OPTIMIZATIONS])(double\*, const double\*, const double\*) ={ /\* The optimization functions \*/

naive,

openmp,

simd,

cacheBlock,

loopUnroll,

registerBlock,

openmp\_simd,

openmp\_simd\_cacheBlock,

openmp\_simd\_cacheBlock\_loopUnroll,

openmp\_simd\_cacheBlock\_loopUnroll\_registerBlock};

const char\* names[NUM\_OF\_OPTIMIZATIONS] ={ /\* optimization names \*/

"naive",

"openmp",

"simd",

"cache block",

"loop unroll",

"register block",

"openmp & simd",

"openmp & simd & cache block",

"openmp & simd & cache block & loop unroll",

"openmp & simd & cache block & loop unroll & register block"};

const int enables[NUM\_OF\_OPTIMIZATIONS] = { /\* whether or not enable the test of some optimizations \*/

ENABLE,

ENABLE,

ENABLE,

ENABLE,

ENABLE,

ENABLE,

ENABLE,

ENABLE,

ENABLE,

ENABLE};

/\* Do multiple experiments. Measure the average runtime. \*/

const int NUM\_OF\_EXPERIMENTS = 5;

for (int i = 0; i < NUM\_OF\_EXPERIMENTS; i++) {

for (int j = 0; j < NUM\_OF\_OPTIMIZATIONS; j++) {

if (enables[j]) {

matrix1s[j] = \_mm\_malloc(WIDTH\*HEIGHT\*sizeof(double), 64);

matrix2s[j] = \_mm\_malloc(WIDTH\*HEIGHT\*sizeof(double), 64);

for (int k = 0; k < WIDTH\*HEIGHT; k++) {

matrix1s[j][k] = rand()/((double)RAND\_MAX);

matrix2s[j][k] = rand()/((double)RAND\_MAX);

}

results[j] = \_mm\_malloc(WIDTH\*HEIGHT\*sizeof(double), 64);

references[j] = \_mm\_malloc(WIDTH\*HEIGHT\*sizeof(double), 64);

uint64\_t start = timestamp\_us();

functions[j](results[j], matrix1s[j], matrix2s[j]);

times[j] += (timestamp\_us() - start) / 1000000.0 / NUM\_OF\_EXPERIMENTS;

if (i == 0) {

naive(references[j], matrix1s[j], matrix2s[j]);

errors[j] = compare\_matrix(results[j], references[j]);

}

\_mm\_free(matrix1s[j]);

\_mm\_free(matrix2s[j]);

\_mm\_free(results[j]);

\_mm\_free(references[j]);

}

}

}

for (int i = 0; i < NUM\_OF\_OPTIMIZATIONS; i++) {

/\* Output time measurement results \*/

if (enables[i]) {

if (!enables[0])

printf("%-65s:%.3f\n", names[i], times[i]);

else

printf("%-65s:%.3f speedup: %.4f\n", names[i], times[i], times[0]/times[i]);

}

/\* Error Handling \*/

if (errors[i]) {

printf ("The result of %s is wrong\n", names[i]);

}

}

return 0;

}

**VI. Conclusion**

In the progress of writing this report, we find that the optimization flag of compiler makes a huge difference in code run. After reading the assembly code, we find that -O1 is the best choice that the compiler produces the assembly code we intends and meanwhile assembly code does not make extra optimization to the unoptimized version.

Meanwhile, we don’t get a complete stable result because we run the program on Ubuntu which is a multiple processing operating system. We should try to run it on a single processing operating system to get more precise result.

After combining all the methods, we achieve a significant speed up of 25.1387. We believe that there are more and better ways to optimize matrix multiplication and other algorithms, and we are willing to learn more about optimization during the summer research in the ASPIRE Lab.