**Optimization of Matrix Convolution on Intel Machine**

Jason Su, Harry He

June 1st, 2015

**I. Introduction**

Matrix Convolution is widely used in image processing,and it is basicly multiplying matrices of two different size. In this project, we are doing a non-separable convolution of 7\*7 and 1024\*1024 matrices with 16 bits input and 64 bits output. There are four optimization technics used in this project: OpenMp, SIMD,  Loop Unrolling and Register Blocking. All the entries of the resultant matrix are calculated both sequentially and parallelly and their performance and efficiency are compared on the excution time of the naive version.

**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

Cache write policy: write back

L1 instruction cache: 32KB per core, 8-way

L1 data cache: 32KB per core, 8-way

L2 cache: 256KB per core, 8-way

L3 cache: 6MB unified, 12-way

L4 cache: 128MB unified, 16-way

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 | 0.04342 | 1.00000 |
| OpenMP | 0.01848 | 2.3491 |
| SIMD | 0.01730 | 2.5101 |
| Cache Blocking | 0.04863 | 0.8930 |
| Loop Unrolling | 0.01964 | 2.2111 |
| Register Blocking | 0.02040 | 2.1281 |
| OpenMP & SIMD | 0.00739 | 5.8779 |
| OpenMP & SIMD & Loop Unrolling | 0.00580 | 7.4817 |
| OpenMP & SIMD & Loop Unrolling & Register Blocking | 0.00407 | 10.6685 |

**IV Analysis**

1. Naive

void naive(uint64\_t\* restrict result,

const uint16\_t\* restrict matrix1, const uint16\_t\* restrict matrix2) {

memset(result, 0, WIDTH2\*HEIGHT2\*sizeof(uint64\_t));

for (int i = 0; i < WIDTH2; i++)

{

for (int m = 0; m < WIDTH1; m++)

{

           for (int n = 0; n < HEIGHT1; n++)

           {

               uint64\_t t = matrix1[m\*WIDTH1+n];

               for (int j = 0; j < HEIGHT2; j++)

               {

                   result[i\*WIDTH2+j] += t\*matrix2[(i-m)\*WIDTH2+(j-n)];

               }}}}}

Using formula result(i,j) += matrix1(m,n)\*matrix2(i-m,j-n), writing the naive version as j in the most inner loop 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. Since results consists of 64 bits=8 bytes number, matrix2 consists of 16 bits = 2bytes number and the cache line which is 64 bytes, the miss rate is only about 0.5\*(8/64+2/64)=7.8%. This gives the best performance.

 2. Openmp

void openmp(uint64\_t\* restrict result,

const uint16\_t\* restrict matrix1, const uint16\_t\* restrict matrix2) {

memset(result, 0, WIDTH2\*HEIGHT2\*sizeof(uint64\_t));

#pragma omp parallel for

for (int i = 0; i < WIDTH2; i++)

{

for (int m = 0; m < WIDTH1; m++)

{

           for (int n = 0; n < HEIGHT1; n++)

           {

               uint64\_t t = matrix1[m\*WIDTH1+n];

               for (int j = 0; j < HEIGHT2; j++)

               {

                   result[i\*WIDTH2+j] += t\*matrix2[(i-m)\*WIDTH2+(j-n)];

              }}}}}

Speedup: 2.35

Since there are 4 cores in the CPU, the theoretical maximum speedup is 4. There are two reasons why that speedup is not observed: The main reason is that the workload is not enough to make use of the full potential of multiple cores. One of the source matrices is only 7\*7. Second, the optimization function is not completely parallelizable. Then by the Amdahl’s Law, the speedup is limited. Third, 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(uint64\_t\* restrict result,

const uint16\_t\* restrict matrix1, const uint16\_t\* restrict matrix2) {

memset(result, 0, WIDTH2\*HEIGHT2\*sizeof(uint64\_t));

for (int i = 0; i < WIDTH2; i++)

{

for (int m = 0; m < WIDTH1; m++)

{

           for (int n = 0; n < HEIGHT1; n++)

           {

               \_\_m256i m1 = \_mm256\_set1\_epi32(((uint32\_t)(matrix1[m\*WIDTH1+n])));

               for (int j = 0; j < HEIGHT2; j+=4)

               {

                \_\_m256i r = \_mm256\_loadu\_si256((\_\_m256i\*)(result+i\*WIDTH2+j));

                \_\_m256i m2 = \_mm256\_loadu\_si256((\_\_m256i\*)(matrix2+(i-m)\*WIDTH2+j-n));

                m2 = \_mm256\_cvtepu16\_epi32(\_mm256\_extracti128\_si256(m2, 0));

                m2 = \_mm256\_mullo\_epi32(m1, m2);

                r = \_mm256\_add\_epi64(r, \_mm256\_cvtepu32\_epi64(\_mm256\_extracti128\_si256(m2, 0)));

                \_mm256\_storeu\_si256((\_\_m256i\*)(result+i\*WIDTH2+j), r);

               }}}}}

Speedup: 2.51

The Intel CPU supports the Intel AVX2 and FMA extensions, which uses 256bits SIMD registers. The size of result matrix is 64bits wide, so 4 operations can be done in parallel. This will give a speedup of 4 theatrically. However, because the source matrices are 16bits wide but the destination matrix is 64bits wide. CPU has to spend lots of time to resolve the width difference. This reduces the speedup. Moreover, because not all code is not able to be written in SIMD instructions, such as branch instructions, the lower speedup should be expected.

4. Cache Blocking

void cacheBlock(uint64\_t\* restrict result,

   const uint16\_t\* restrict matrix1, const uint16\_t\* restrict matrix2) {

   memset(result, 0, WIDTH2\*HEIGHT2\*sizeof(uint64\_t));

   const int BLOCK = 512;

   for (int jj = 0; jj < WIDTH2; jj+=BLOCK) {

   for (int i = 0; i < WIDTH2; i++)

   {

   for (int m = 0; m < WIDTH1; m++)

   {

            for (int n = 0; n < HEIGHT1; n++)

            {

                uint64\_t t = matrix1[m\*WIDTH1+n];

                for (int j = jj; j < jj+BLOCK; j++)

                {

                    result[i\*WIDTH2+j] += t\*matrix2[(i-m)\*WIDTH2+(j-n)];

                }

            }

   }

   }

   }

}

Cache Blocking is not a useful optimization techniques in this project because the cache is big enough to hold all reusable data. Since matrix1 is only 7\*7, the cache only needs to be able to afford 7 lines of result and 7 lines of matrix2. Because both result of matrix2 is 1024\*1024 and each element in result is 8 bytes and each element in matrix2 is 2 bytes, the size of cache just needs to be at least 8\*7\*1024+2\*7\*1024=70KB, which is much less than the size of L2 cache.

 5.Loop Unrolling

void loopUnroll(uint64\_t\* restrict result,

const uint16\_t\* restrict matrix1, const uint16\_t\* restrict matrix2) {

memset(result, 0, WIDTH2\*HEIGHT2\*sizeof(uint64\_t));

for (int i = 0; i < WIDTH2; i++)

{

       uint64\_t\* dest = result + i\*WIDTH2;

for (int m = 0; m < WIDTH1; m++)

{

           for (int n = 0; n < HEIGHT1; n++)

           {

               uint64\_t t = matrix1[m\*WIDTH1+n];

               const uint16\_t\* mat2 = matrix2 + (i-m)\*WIDTH2-n;

               for (int j = 0; j < HEIGHT2; 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: 2.21

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(uint64\_t\* restrict result,

const uint16\_t\* restrict matrix1, const uint16\_t\* restrict matrix2) {

memset(result, 0, WIDTH2\*HEIGHT2\*sizeof(uint64\_t));

for (int i = 0; i < WIDTH2; i++)

{

for (int m = 0; m < WIDTH1; m++)

{

uint64\_t m1\_0 = matrix1[m\*WIDTH1+0];

uint64\_t m1\_1 = matrix1[m\*WIDTH1+1];

uint64\_t m1\_2 = matrix1[m\*WIDTH1+2];

uint64\_t m1\_3 = matrix1[m\*WIDTH1+3];

uint64\_t m1\_4 = matrix1[m\*WIDTH1+4];

uint64\_t m1\_5 = matrix1[m\*WIDTH1+5];

uint64\_t m1\_6 = matrix1[m\*WIDTH1+6];

for (int j = 0; j < HEIGHT2; j+=4)

{

uint64\_t m2\_\_6 = matrix2[(i-m)\*WIDTH2+j-6];

uint64\_t m2\_\_5 = matrix2[(i-m)\*WIDTH2+j-5];

uint64\_t m2\_\_4 = matrix2[(i-m)\*WIDTH2+j-4];

uint64\_t m2\_\_3 = matrix2[(i-m)\*WIDTH2+j-3];

uint64\_t m2\_\_2 = matrix2[(i-m)\*WIDTH2+j-2];

uint64\_t m2\_\_1 = matrix2[(i-m)\*WIDTH2+j-1];

uint64\_t m2\_0  = matrix2[(i-m)\*WIDTH2+j+0];

uint64\_t m2\_1  = matrix2[(i-m)\*WIDTH2+j+1];

uint64\_t m2\_2  = matrix2[(i-m)\*WIDTH2+j+2];

uint64\_t m2\_3  = matrix2[(i-m)\*WIDTH2+j+3];

result[i\*WIDTH2+j]   += m1\_0\*m2\_0 + m1\_1\*m2\_\_1 + m1\_2\*m2\_\_2 + m1\_3\*m2\_\_3 + m1\_4\*m2\_\_4 + m1\_5\*m2\_\_5 + m1\_6\*m2\_\_6;

result[i\*WIDTH2+j+1] += m1\_0\*m2\_1 + m1\_1\*m2\_0  + m1\_2\*m2\_\_1 + m1\_3\*m2\_\_2 + m1\_4\*m2\_\_3 + m1\_5\*m2\_\_4 + m1\_6\*m2\_\_5;

result[i\*WIDTH2+j+2] += m1\_0\*m2\_2 + m1\_1\*m2\_1  + m1\_2\*m2\_0  + m1\_3\*m2\_\_1 + m1\_4\*m2\_\_2 + m1\_5\*m2\_\_3 + m1\_6\*m2\_\_4;

result[i\*WIDTH2+j+3] += m1\_0\*m2\_3 + m1\_1\*m2\_2  + m1\_2\*m2\_1  + m1\_3\*m2\_0  + m1\_4\*m2\_\_1 + m1\_5\*m2\_\_2 + m1\_6\*m2\_\_3;

                 }}}}

Speedup: 2.12

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,m,n,j) pair in the most inner loop. However, with register blocking, 24 (i,j,k) pairs are done at once in the most inner loop, and this visit cache 14 times. Therefore, for each (i,j,k) pair, cache is only accessed 14/24 = 0.58 times, which gives the speedup observed.

 7. Openmp & SIMD

void openmp\_simd(uint64\_t\* restrict result,

const uint16\_t\* restrict matrix1, const uint16\_t\* restrict matrix2) {

   memset(result, 0, WIDTH2\*HEIGHT2\*sizeof(uint64\_t));

   #pragma omp parallel for

   for (int i = 0; i < WIDTH2; i++)

   {

       for (int m = 0; m < WIDTH1; m++)

       {

           for (int n = 0; n < HEIGHT1; n++)

           {

               \_\_m256i m1 = \_mm256\_set1\_epi32(((uint32\_t)(matrix1[m\*WIDTH1+n])));

               for (int j = 0; j < HEIGHT2; j+=4)

               {

                \_\_m256i r = \_mm256\_loadu\_si256((\_\_m256i\*)(result+i\*WIDTH2+j));

                \_\_m256i m2 = \_mm256\_loadu\_si256((\_\_m256i\*)(matrix2+(i-m)\*WIDTH2+j-n));

                m2 = \_mm256\_cvtepu16\_epi32(\_mm256\_extracti128\_si256(m2, 0));

                m2 = \_mm256\_mullo\_epi32(m1, m2);

                r = \_mm256\_add\_epi64(r, \_mm256\_cvtepu32\_epi64(\_mm256\_extracti128\_si256(m2, 0)));

                \_mm256\_storeu\_si256((\_\_m256i\*)(result+i\*WIDTH2+j), r);

               }}}}}

Speedup: 5.88

The speedup is very close to, or even better than the product of speedups of openmp and simd, which is 2.35\*2.51=5.89 This shows that openmp and simd can work together perfectly. The two optimization technologies are strongly independent.

 8. OpenMp & SIMD & Loop Unrolling

void openmp\_simd\_loopUnroll(uint64\_t\* restrict result,

const uint16\_t\* restrict matrix1, const uint16\_t\* restrict matrix2) {

   memset(result, 0, WIDTH2\*HEIGHT2\*sizeof(uint64\_t));

   #pragma omp parallel for

   for (int i = 0; i < WIDTH2; i++)

   {

       for (int m = 0; m < WIDTH1; m++)

       {

           for (int n = 0; n < HEIGHT1; n++)

           {

               \_\_m256i m1 = \_mm256\_set1\_epi32(((uint32\_t)(matrix1[m\*WIDTH1+n])));

               for (int j = 0; j < HEIGHT2; j+=32)

               {

                \_\_m256i r0 = \_mm256\_loadu\_si256((\_\_m256i\*)(result+i\*WIDTH2+j));

                \_\_m256i r1 = \_mm256\_loadu\_si256((\_\_m256i\*)(result+i\*WIDTH2+j+4));

                \_\_m256i r2 = \_mm256\_loadu\_si256((\_\_m256i\*)(result+i\*WIDTH2+j+8));

                \_\_m256i r3 = \_mm256\_loadu\_si256((\_\_m256i\*)(result+i\*WIDTH2+j+12));

                \_\_m256i m2 = \_mm256\_loadu\_si256((\_\_m256i\*)(matrix2+(i-m)\*WIDTH2+j-n));

                \_\_m256i m2\_0 = \_mm256\_cvtepu16\_epi32(\_mm256\_extracti128\_si256(m2, 0));

                \_\_m256i m2\_1 = \_mm256\_cvtepu16\_epi32(\_mm256\_extracti128\_si256(m2, 1));

                m2\_0 = \_mm256\_mullo\_epi32(m1, m2\_0);

                m2\_1 = \_mm256\_mullo\_epi32(m1, m2\_1);

                r0 = \_mm256\_add\_epi64(r0, \_mm256\_cvtepu32\_epi64(\_mm256\_extracti128\_si256(m2\_0, 0)));

                r1 = \_mm256\_add\_epi64(r1, \_mm256\_cvtepu32\_epi64(\_mm256\_extracti128\_si256(m2\_0, 1)));

                r2 = \_mm256\_add\_epi64(r2, \_mm256\_cvtepu32\_epi64(\_mm256\_extracti128\_si256(m2\_1, 0)));

                r3 = \_mm256\_add\_epi64(r3, \_mm256\_cvtepu32\_epi64(\_mm256\_extracti128\_si256(m2\_1, 1)));

                \_mm256\_storeu\_si256((\_\_m256i\*)(result+i\*WIDTH2+j), r0);

                \_mm256\_storeu\_si256((\_\_m256i\*)(result+i\*WIDTH2+j+4), r1);

                \_mm256\_storeu\_si256((\_\_m256i\*)(result+i\*WIDTH2+j+8), r2);

                \_mm256\_storeu\_si256((\_\_m256i\*)(result+i\*WIDTH2+j+12), r3);

                r0 = \_mm256\_loadu\_si256((\_\_m256i\*)(result+i\*WIDTH2+j+16));

                r1 = \_mm256\_loadu\_si256((\_\_m256i\*)(result+i\*WIDTH2+j+20));

                r2 = \_mm256\_loadu\_si256((\_\_m256i\*)(result+i\*WIDTH2+j+24));

                r3 = \_mm256\_loadu\_si256((\_\_m256i\*)(result+i\*WIDTH2+j+28));

                m2 = \_mm256\_loadu\_si256((\_\_m256i\*)(matrix2+(i-m)\*WIDTH2+j-n+16));

                m2\_0 = \_mm256\_cvtepu16\_epi32(\_mm256\_extracti128\_si256(m2, 0));

                m2\_1 = \_mm256\_cvtepu16\_epi32(\_mm256\_extracti128\_si256(m2, 1));

                m2\_0 = \_mm256\_mullo\_epi32(m1, m2\_0);

                m2\_1 = \_mm256\_mullo\_epi32(m1, m2\_1);

                r0 = \_mm256\_add\_epi64(r0, \_mm256\_cvtepu32\_epi64(\_mm256\_extracti128\_si256(m2\_0, 0)));

                r1 = \_mm256\_add\_epi64(r1, \_mm256\_cvtepu32\_epi64(\_mm256\_extracti128\_si256(m2\_0, 1)));

                r2 = \_mm256\_add\_epi64(r2, \_mm256\_cvtepu32\_epi64(\_mm256\_extracti128\_si256(m2\_1, 0)));

                r3 = \_mm256\_add\_epi64(r3, \_mm256\_cvtepu32\_epi64(\_mm256\_extracti128\_si256(m2\_1, 1)));

                \_mm256\_storeu\_si256((\_\_m256i\*)(result+i\*WIDTH2+j+16), r0);

                \_mm256\_storeu\_si256((\_\_m256i\*)(result+i\*WIDTH2+j+20), r1);

                \_mm256\_storeu\_si256((\_\_m256i\*)(result+i\*WIDTH2+j+24), r2);

                \_mm256\_storeu\_si256((\_\_m256i\*)(result+i\*WIDTH2+j+28), r3);

                                              }}}}}

Comparing to its speedup of 7.48 and the speedups of openmp&simd&cache blocking and loop unrolling ,5.88\*2.21=13.00, it shows that loop unrolling works worse with simd and openmp. The code of simd is not short so it involves some degree of loop unrolling, reducing the effectiveness of following loop unrolling optimizations.

     9. OpenMp & SIMD & Loop Unrolling & Register Blocking

void openmp\_simd\_loopUnroll\_registerBlock(uint64\_t\* restrict result,

const uint16\_t\* restrict matrix1, const uint16\_t\* restrict matrix2) {

   memset(result, 0, WIDTH2\*HEIGHT2\*sizeof(uint64\_t));

   #pragma omp parallel for

   for (int i = 0; i < WIDTH2; i++)

   {

       for (int m = 0; m < WIDTH1; m++)

       {

\_\_m256i m1\_0 = \_mm256\_set1\_epi32(((uint32\_t)(matrix1[m\*WIDTH1])));

\_\_m256i m1\_1 = \_mm256\_set1\_epi32(((uint32\_t)(matrix1[m\*WIDTH1+1])));

\_\_m256i m1\_2 = \_mm256\_set1\_epi32(((uint32\_t)(matrix1[m\*WIDTH1+2])));

\_\_m256i m1\_3 = \_mm256\_set1\_epi32(((uint32\_t)(matrix1[m\*WIDTH1+3])));

\_\_m256i m1\_4 = \_mm256\_set1\_epi32(((uint32\_t)(matrix1[m\*WIDTH1+4])));

\_\_m256i m1\_5 = \_mm256\_set1\_epi32(((uint32\_t)(matrix1[m\*WIDTH1+5])));

\_\_m256i m1\_6 = \_mm256\_set1\_epi32(((uint32\_t)(matrix1[m\*WIDTH1+6])));

for (int j = 0; j < HEIGHT2; j+=32)

{

\_\_m256i r0 = \_mm256\_loadu\_si256((\_\_m256i\*)(result+i\*WIDTH2+j));

\_\_m256i r1 = \_mm256\_loadu\_si256((\_\_m256i\*)(result+i\*WIDTH2+j+4));

\_\_m256i r2 = \_mm256\_loadu\_si256((\_\_m256i\*)(result+i\*WIDTH2+j+8));

\_\_m256i r3 = \_mm256\_loadu\_si256((\_\_m256i\*)(result+i\*WIDTH2+j+12));

\_\_m256i m2 = \_mm256\_loadu\_si256((\_\_m256i\*)(matrix2+(i-m)\*WIDTH2+j));

\_\_m256i m2\_0 = \_mm256\_cvtepu16\_epi32(\_mm256\_extracti128\_si256(m2, 0));

\_\_m256i m2\_1 = \_mm256\_cvtepu16\_epi32(\_mm256\_extracti128\_si256(m2, 1));

m2\_0 = \_mm256\_mullo\_epi32(m1\_0, m2\_0);

m2\_1 = \_mm256\_mullo\_epi32(m1\_0, m2\_1);

r0 = \_mm256\_add\_epi64(r0, \_mm256\_cvtepu32\_epi64(\_mm256\_extracti128\_si256(m2\_0, 0)));

r1 = \_mm256\_add\_epi64(r1, \_mm256\_cvtepu32\_epi64(\_mm256\_extracti128\_si256(m2\_0, 1)));

r2 = \_mm256\_add\_epi64(r2, \_mm256\_cvtepu32\_epi64(\_mm256\_extracti128\_si256(m2\_1, 0)));

r3 = \_mm256\_add\_epi64(r3, \_mm256\_cvtepu32\_epi64(\_mm256\_extracti128\_si256(m2\_1, 1)));

m2 = \_mm256\_loadu\_si256((\_\_m256i\*)(matrix2+(i-m)\*WIDTH2+j-1));

m2\_0 = \_mm256\_cvtepu16\_epi32(\_mm256\_extracti128\_si256(m2, 0));

m2\_1 = \_mm256\_cvtepu16\_epi32(\_mm256\_extracti128\_si256(m2, 1));

m2\_0 = \_mm256\_mullo\_epi32(m1\_1, m2\_0);

m2\_1 = \_mm256\_mullo\_epi32(m1\_1, m2\_1);

r0 = \_mm256\_add\_epi64(r0, \_mm256\_cvtepu32\_epi64(\_mm256\_extracti128\_si256(m2\_0, 0)));

r1 = \_mm256\_add\_epi64(r1, \_mm256\_cvtepu32\_epi64(\_mm256\_extracti128\_si256(m2\_0, 1)));

r2 = \_mm256\_add\_epi64(r2, \_mm256\_cvtepu32\_epi64(\_mm256\_extracti128\_si256(m2\_1, 0)));

r3 = \_mm256\_add\_epi64(r3, \_mm256\_cvtepu32\_epi64(\_mm256\_extracti128\_si256(m2\_1, 1)));

m2 = \_mm256\_loadu\_si256((\_\_m256i\*)(matrix2+(i-m)\*WIDTH2+j-2));

m2\_0 = \_mm256\_cvtepu16\_epi32(\_mm256\_extracti128\_si256(m2, 0));

m2\_1 = \_mm256\_cvtepu16\_epi32(\_mm256\_extracti128\_si256(m2, 1));

m2\_0 = \_mm256\_mullo\_epi32(m1\_2, m2\_0);

m2\_1 = \_mm256\_mullo\_epi32(m1\_2, m2\_1);

r0 = \_mm256\_add\_epi64(r0, \_mm256\_cvtepu32\_epi64(\_mm256\_extracti128\_si256(m2\_0, 0)));

r1 = \_mm256\_add\_epi64(r1, \_mm256\_cvtepu32\_epi64(\_mm256\_extracti128\_si256(m2\_0, 1)));

r2 = \_mm256\_add\_epi64(r2, \_mm256\_cvtepu32\_epi64(\_mm256\_extracti128\_si256(m2\_1, 0)));

r3 = \_mm256\_add\_epi64(r3, \_mm256\_cvtepu32\_epi64(\_mm256\_extracti128\_si256(m2\_1, 1)));

m2 = \_mm256\_loadu\_si256((\_\_m256i\*)(matrix2+(i-m)\*WIDTH2+j-3));

m2\_0 = \_mm256\_cvtepu16\_epi32(\_mm256\_extracti128\_si256(m2, 0));

m2\_1 = \_mm256\_cvtepu16\_epi32(\_mm256\_extracti128\_si256(m2, 1));

m2\_0 = \_mm256\_mullo\_epi32(m1\_3, m2\_0);

m2\_1 = \_mm256\_mullo\_epi32(m1\_3, m2\_1);

r0 = \_mm256\_add\_epi64(r0, \_mm256\_cvtepu32\_epi64(\_mm256\_extracti128\_si256(m2\_0, 0)));

r1 = \_mm256\_add\_epi64(r1, \_mm256\_cvtepu32\_epi64(\_mm256\_extracti128\_si256(m2\_0, 1)));

r2 = \_mm256\_add\_epi64(r2, \_mm256\_cvtepu32\_epi64(\_mm256\_extracti128\_si256(m2\_1, 0)));

r3 = \_mm256\_add\_epi64(r3, \_mm256\_cvtepu32\_epi64(\_mm256\_extracti128\_si256(m2\_1, 1)));

m2 = \_mm256\_loadu\_si256((\_\_m256i\*)(matrix2+(i-m)\*WIDTH2+j-4));

m2\_0 = \_mm256\_cvtepu16\_epi32(\_mm256\_extracti128\_si256(m2, 0));

m2\_1 = \_mm256\_cvtepu16\_epi32(\_mm256\_extracti128\_si256(m2, 1));

m2\_0 = \_mm256\_mullo\_epi32(m1\_4, m2\_0);

m2\_1 = \_mm256\_mullo\_epi32(m1\_4, m2\_1);

r0 = \_mm256\_add\_epi64(r0, \_mm256\_cvtepu32\_epi64(\_mm256\_extracti128\_si256(m2\_0, 0)));

r1 = \_mm256\_add\_epi64(r1, \_mm256\_cvtepu32\_epi64(\_mm256\_extracti128\_si256(m2\_0, 1)));

r2 = \_mm256\_add\_epi64(r2, \_mm256\_cvtepu32\_epi64(\_mm256\_extracti128\_si256(m2\_1, 0)));

r3 = \_mm256\_add\_epi64(r3, \_mm256\_cvtepu32\_epi64(\_mm256\_extracti128\_si256(m2\_1, 1)));

m2 = \_mm256\_loadu\_si256((\_\_m256i\*)(matrix2+(i-m)\*WIDTH2+j-5));

m2\_0 = \_mm256\_cvtepu16\_epi32(\_mm256\_extracti128\_si256(m2, 0));

m2\_1 = \_mm256\_cvtepu16\_epi32(\_mm256\_extracti128\_si256(m2, 1));

m2\_0 = \_mm256\_mullo\_epi32(m1\_5, m2\_0);

m2\_1 = \_mm256\_mullo\_epi32(m1\_5, m2\_1);

r0 = \_mm256\_add\_epi64(r0, \_mm256\_cvtepu32\_epi64(\_mm256\_extracti128\_si256(m2\_0, 0)));

r1 = \_mm256\_add\_epi64(r1, \_mm256\_cvtepu32\_epi64(\_mm256\_extracti128\_si256(m2\_0, 1)));

r2 = \_mm256\_add\_epi64(r2, \_mm256\_cvtepu32\_epi64(\_mm256\_extracti128\_si256(m2\_1, 0)));

r3 = \_mm256\_add\_epi64(r3, \_mm256\_cvtepu32\_epi64(\_mm256\_extracti128\_si256(m2\_1, 1)));

m2 = \_mm256\_loadu\_si256((\_\_m256i\*)(matrix2+(i-m)\*WIDTH2+j-6));

m2\_0 = \_mm256\_cvtepu16\_epi32(\_mm256\_extracti128\_si256(m2, 0));

m2\_1 = \_mm256\_cvtepu16\_epi32(\_mm256\_extracti128\_si256(m2, 1));

m2\_0 = \_mm256\_mullo\_epi32(m1\_6, m2\_0);

m2\_1 = \_mm256\_mullo\_epi32(m1\_6, m2\_1);

r0 = \_mm256\_add\_epi64(r0, \_mm256\_cvtepu32\_epi64(\_mm256\_extracti128\_si256(m2\_0, 0)));

r1 = \_mm256\_add\_epi64(r1, \_mm256\_cvtepu32\_epi64(\_mm256\_extracti128\_si256(m2\_0, 1)));

r2 = \_mm256\_add\_epi64(r2, \_mm256\_cvtepu32\_epi64(\_mm256\_extracti128\_si256(m2\_1, 0)));

r3 = \_mm256\_add\_epi64(r3, \_mm256\_cvtepu32\_epi64(\_mm256\_extracti128\_si256(m2\_1, 1)));

\_mm256\_storeu\_si256((\_\_m256i\*)(result+i\*WIDTH2+j), r0);

\_mm256\_storeu\_si256((\_\_m256i\*)(result+i\*WIDTH2+j+4), r1);

\_mm256\_storeu\_si256((\_\_m256i\*)(result+i\*WIDTH2+j+8), r2);

\_mm256\_storeu\_si256((\_\_m256i\*)(result+i\*WIDTH2+j+12), r3);

r0 = \_mm256\_loadu\_si256((\_\_m256i\*)(result+i\*WIDTH2+j+16));

r1 = \_mm256\_loadu\_si256((\_\_m256i\*)(result+i\*WIDTH2+j+20));

r2 = \_mm256\_loadu\_si256((\_\_m256i\*)(result+i\*WIDTH2+j+24));

r3 = \_mm256\_loadu\_si256((\_\_m256i\*)(result+i\*WIDTH2+j+28));

m2 = \_mm256\_loadu\_si256((\_\_m256i\*)(matrix2+(i-m)\*WIDTH2+j+16));

m2\_0 = \_mm256\_cvtepu16\_epi32(\_mm256\_extracti128\_si256(m2, 0));

m2\_1 = \_mm256\_cvtepu16\_epi32(\_mm256\_extracti128\_si256(m2, 1));

m2\_0 = \_mm256\_mullo\_epi32(m1\_0, m2\_0);

m2\_1 = \_mm256\_mullo\_epi32(m1\_0, m2\_1);

r0 = \_mm256\_add\_epi64(r0, \_mm256\_cvtepu32\_epi64(\_mm256\_extracti128\_si256(m2\_0, 0)));

r1 = \_mm256\_add\_epi64(r1, \_mm256\_cvtepu32\_epi64(\_mm256\_extracti128\_si256(m2\_0, 1)));

r2 = \_mm256\_add\_epi64(r2, \_mm256\_cvtepu32\_epi64(\_mm256\_extracti128\_si256(m2\_1, 0)));

r3 = \_mm256\_add\_epi64(r3, \_mm256\_cvtepu32\_epi64(\_mm256\_extracti128\_si256(m2\_1, 1)));

m2 = \_mm256\_loadu\_si256((\_\_m256i\*)(matrix2+(i-m)\*WIDTH2+j-1+16));

m2\_0 = \_mm256\_cvtepu16\_epi32(\_mm256\_extracti128\_si256(m2, 0));

m2\_1 = \_mm256\_cvtepu16\_epi32(\_mm256\_extracti128\_si256(m2, 1));

m2\_0 = \_mm256\_mullo\_epi32(m1\_1, m2\_0);

m2\_1 = \_mm256\_mullo\_epi32(m1\_1, m2\_1);

r0 = \_mm256\_add\_epi64(r0, \_mm256\_cvtepu32\_epi64(\_mm256\_extracti128\_si256(m2\_0, 0)));

r1 = \_mm256\_add\_epi64(r1, \_mm256\_cvtepu32\_epi64(\_mm256\_extracti128\_si256(m2\_0, 1)));

r2 = \_mm256\_add\_epi64(r2, \_mm256\_cvtepu32\_epi64(\_mm256\_extracti128\_si256(m2\_1, 0)));

r3 = \_mm256\_add\_epi64(r3, \_mm256\_cvtepu32\_epi64(\_mm256\_extracti128\_si256(m2\_1, 1)));

m2 = \_mm256\_loadu\_si256((\_\_m256i\*)(matrix2+(i-m)\*WIDTH2+j-2+16));

m2\_0 = \_mm256\_cvtepu16\_epi32(\_mm256\_extracti128\_si256(m2, 0));

m2\_1 = \_mm256\_cvtepu16\_epi32(\_mm256\_extracti128\_si256(m2, 1));

m2\_0 = \_mm256\_mullo\_epi32(m1\_2, m2\_0);

m2\_1 = \_mm256\_mullo\_epi32(m1\_2, m2\_1);

r0 = \_mm256\_add\_epi64(r0, \_mm256\_cvtepu32\_epi64(\_mm256\_extracti128\_si256(m2\_0, 0)));

r1 = \_mm256\_add\_epi64(r1, \_mm256\_cvtepu32\_epi64(\_mm256\_extracti128\_si256(m2\_0, 1)));

r2 = \_mm256\_add\_epi64(r2, \_mm256\_cvtepu32\_epi64(\_mm256\_extracti128\_si256(m2\_1, 0)));

r3 = \_mm256\_add\_epi64(r3, \_mm256\_cvtepu32\_epi64(\_mm256\_extracti128\_si256(m2\_1, 1)));

m2 = \_mm256\_loadu\_si256((\_\_m256i\*)(matrix2+(i-m)\*WIDTH2+j-3+16));

m2\_0 = \_mm256\_cvtepu16\_epi32(\_mm256\_extracti128\_si256(m2, 0));

m2\_1 = \_mm256\_cvtepu16\_epi32(\_mm256\_extracti128\_si256(m2, 1));

m2\_0 = \_mm256\_mullo\_epi32(m1\_3, m2\_0);

m2\_1 = \_mm256\_mullo\_epi32(m1\_3, m2\_1);

r0 = \_mm256\_add\_epi64(r0, \_mm256\_cvtepu32\_epi64(\_mm256\_extracti128\_si256(m2\_0, 0)));

r1 = \_mm256\_add\_epi64(r1, \_mm256\_cvtepu32\_epi64(\_mm256\_extracti128\_si256(m2\_0, 1)));

r2 = \_mm256\_add\_epi64(r2, \_mm256\_cvtepu32\_epi64(\_mm256\_extracti128\_si256(m2\_1, 0)));

r3 = \_mm256\_add\_epi64(r3, \_mm256\_cvtepu32\_epi64(\_mm256\_extracti128\_si256(m2\_1, 1)));

m2 = \_mm256\_loadu\_si256((\_\_m256i\*)(matrix2+(i-m)\*WIDTH2+j-4+16));

m2\_0 = \_mm256\_cvtepu16\_epi32(\_mm256\_extracti128\_si256(m2, 0));

m2\_1 = \_mm256\_cvtepu16\_epi32(\_mm256\_extracti128\_si256(m2, 1));

m2\_0 = \_mm256\_mullo\_epi32(m1\_4, m2\_0);

m2\_1 = \_mm256\_mullo\_epi32(m1\_4, m2\_1);

r0 = \_mm256\_add\_epi64(r0, \_mm256\_cvtepu32\_epi64(\_mm256\_extracti128\_si256(m2\_0, 0)));

r1 = \_mm256\_add\_epi64(r1, \_mm256\_cvtepu32\_epi64(\_mm256\_extracti128\_si256(m2\_0, 1)));

r2 = \_mm256\_add\_epi64(r2, \_mm256\_cvtepu32\_epi64(\_mm256\_extracti128\_si256(m2\_1, 0)));

r3 = \_mm256\_add\_epi64(r3, \_mm256\_cvtepu32\_epi64(\_mm256\_extracti128\_si256(m2\_1, 1)));

m2 = \_mm256\_loadu\_si256((\_\_m256i\*)(matrix2+(i-m)\*WIDTH2+j-5+16));

m2\_0 = \_mm256\_cvtepu16\_epi32(\_mm256\_extracti128\_si256(m2, 0));

m2\_1 = \_mm256\_cvtepu16\_epi32(\_mm256\_extracti128\_si256(m2, 1));

m2\_0 = \_mm256\_mullo\_epi32(m1\_5, m2\_0);

m2\_1 = \_mm256\_mullo\_epi32(m1\_5, m2\_1);

r0 = \_mm256\_add\_epi64(r0, \_mm256\_cvtepu32\_epi64(\_mm256\_extracti128\_si256(m2\_0, 0)));

r1 = \_mm256\_add\_epi64(r1, \_mm256\_cvtepu32\_epi64(\_mm256\_extracti128\_si256(m2\_0, 1)));

r2 = \_mm256\_add\_epi64(r2, \_mm256\_cvtepu32\_epi64(\_mm256\_extracti128\_si256(m2\_1, 0)));

r3 = \_mm256\_add\_epi64(r3, \_mm256\_cvtepu32\_epi64(\_mm256\_extracti128\_si256(m2\_1, 1)));

m2 = \_mm256\_loadu\_si256((\_\_m256i\*)(matrix2+(i-m)\*WIDTH2+j-6+16));

m2\_0 = \_mm256\_cvtepu16\_epi32(\_mm256\_extracti128\_si256(m2, 0));

m2\_1 = \_mm256\_cvtepu16\_epi32(\_mm256\_extracti128\_si256(m2, 1));

m2\_0 = \_mm256\_mullo\_epi32(m1\_6, m2\_0);

m2\_1 = \_mm256\_mullo\_epi32(m1\_6, m2\_1);

r0 = \_mm256\_add\_epi64(r0, \_mm256\_cvtepu32\_epi64(\_mm256\_extracti128\_si256(m2\_0, 0)));

r1 = \_mm256\_add\_epi64(r1, \_mm256\_cvtepu32\_epi64(\_mm256\_extracti128\_si256(m2\_0, 1)));

r2 = \_mm256\_add\_epi64(r2, \_mm256\_cvtepu32\_epi64(\_mm256\_extracti128\_si256(m2\_1, 0)));

r3 = \_mm256\_add\_epi64(r3, \_mm256\_cvtepu32\_epi64(\_mm256\_extracti128\_si256(m2\_1, 1)));

\_mm256\_storeu\_si256((\_\_m256i\*)(result+i\*WIDTH2+j+16), r0);

\_mm256\_storeu\_si256((\_\_m256i\*)(result+i\*WIDTH2+j+20), r1);

\_mm256\_storeu\_si256((\_\_m256i\*)(result+i\*WIDTH2+j+24), r2);

\_mm256\_storeu\_si256((\_\_m256i\*)(result+i\*WIDTH2+j+28), r3);

                   }}}}

Speedup: 10.6877

Comparing to its speedup and the speedups of openmp&simd&loop unrolling and register blocking, 7.48\*2.12 = 15.85, it shows that register blocking has a huge diminishing returns when working with other optimizations such as simd. The main reason is that when working in simd, the registers have to store several elements continuously. In comparion, without simd, each register can store a single element. This reduce the effectiveness of register blocking with simd because extra cache operations have to be done in order to follow the property of simd.

**V. Other Code**

1. header file

#ifndef MATRIX\_HEADER

#define MATRIX\_HEADER

#include <string.h>

#include <immintrin.h>

#include <stdint.h>

#define ENABLE 1

#define DISABLE 0

#define PAD (8\*1024)

#define WIDTH1 7

#define HEIGHT1 7

#define WIDTH2 1024

#define HEIGHT2 1024

#define LINE\_SIZE 64

#define L1\_SIZE (32\*1024)

#define L2\_SIZE (256\*1024)

#define L3\_SIZE (6144\*1024)

#define L4\_SIZE (131072\*1024)

#define NUM\_OF\_OPTIMIZATIONS 9

extern int compare\_matrix(const uint64\_t\* sample, const uint64\_t\* reference);

extern void naive(uint64\_t\* restrict result,

const uint16\_t\* restrict matrix1, const uint16\_t\* restrict matrix2);

extern void openmp(uint64\_t\* restrict result,

const uint16\_t\* restrict matrix1, const uint16\_t\* restrict matrix2);

extern void simd(uint64\_t\* restrict result,

const uint16\_t\* restrict matrix1, const uint16\_t\* restrict matrix2);

extern void cacheBlock(uint64\_t\* restrict result,

const uint16\_t\* restrict matrix1, const uint16\_t\* restrict matrix2);

extern void loopUnroll(uint64\_t\* restrict result,

const uint16\_t\* restrict matrix1, const uint16\_t\* restrict matrix2);

extern void registerBlock(uint64\_t\* restrict result,

const uint16\_t\* restrict matrix1, const uint16\_t\* restrict matrix2);

extern void openmp\_simd(uint64\_t\* restrict result,

const uint16\_t\* restrict matrix1, const uint16\_t\* restrict matrix2);

extern void openmp\_simd\_loopUnroll(uint64\_t\* restrict result,

const uint16\_t\* restrict matrix1, const uint16\_t\* restrict matrix2);

extern void openmp\_simd\_loopUnroll\_registerBlock(uint64\_t\* restrict result,

const uint16\_t\* restrict matrix1, const uint16\_t\* restrict matrix2);

#endif

2. main function

/\*

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

Name        : main.c

Author      : Jason Su and Harry He

Version     : 1.0.0

Copyright   : GNU

Description : 7\*7 1024\*1024 2-D non-separable convolution

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

\*/

#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);

uint16\_t\* matrix1s[NUM\_OF\_OPTIMIZATIONS] = {};

uint16\_t\* matrix2s[NUM\_OF\_OPTIMIZATIONS] = {};

uint64\_t\* results[NUM\_OF\_OPTIMIZATIONS] = {}; /\* Store the outputs of optimizations \*/

uint64\_t\* 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])(uint64\_t\*, const uint16\_t\*, const uint16\_t\*) ={ /\* The optimization functions \*/

naive,

openmp,

simd,

cacheBlock,

loopUnroll,

registerBlock,

openmp\_simd,

openmp\_simd\_loopUnroll,

openmp\_simd\_loopUnroll\_registerBlock};

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

"naive",

"openmp",

"simd",

"cache block",

"loop unroll",

"register block",

"openmp & simd",

"openmp & simd & loop unroll",

"openmp & simd & 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,

ENABLE,

ENABLE,

ENABLE,

ENABLE,

    ENABLE,

ENABLE,

ENABLE,

ENABLE};\*/

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

const int NUM\_OF\_EXPERIMENTS = 50;

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(WIDTH1\*HEIGHT1\*sizeof(uint16\_t), 64);

matrix2s[j] = \_mm\_malloc((WIDTH2\*HEIGHT2+2\*PAD)\*sizeof(uint16\_t), 64);

for (int i = 0; i < PAD; i++) {

matrix2s[j][i] = 0;

}

matrix2s[j] += PAD;

for (int k = 0; k < WIDTH1\*HEIGHT1; k++) {

matrix1s[j][k] = rand()%65535;

}

for (int k = 0; k < WIDTH2\*HEIGHT2; k++) {

matrix2s[j][k] = rand()%65535;

}

results[j] = \_mm\_malloc(WIDTH2\*HEIGHT2\*sizeof(uint64\_t), 64);

references[j] = \_mm\_malloc(WIDTH2\*HEIGHT2\*sizeof(uint64\_t), 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]-PAD);

\_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**

After combining all the methods, we achieve a significant speed up of 10.6685. We believe that there are more and better ways to optimize matrix convolution and other algorithms, and we are willing to learn more about optimization during the summer research in the ASPIRE Lab.