Convolution Matrix Optimization with Various Techniques

Donghao Su, Haoji Liu

05/14/2018

I. Introduction

Matrix Convolution is widely used in image processing to apply effects onto a given image. Some examples can be blurring, edging, sharpening, cropping and scaling. The original image is represented as a matrix of pixels, where each cell contains a tuple of floating points. The second matrix is usually much smaller, it acts as the kernel to apply the effect onto each pixel. This is done by doing multiplication of the kernel matrix and each cell within the first matrix. Many kernels are free and ready for use, they are of various sizes normally ranging from 3x3 to 7x7.

In this project, we are doing a convolution operation of 7*7 and 1024*1024 matrices with 16 bits input and 64 bits output. There are four optimization techniques used in this project: OpenMp, SIMD, Loop Unrolling and MPI. All the entries of the result matrix go over with different parallelization technique. The performance is then measured and compared by the execution time with the naive sequential execution.

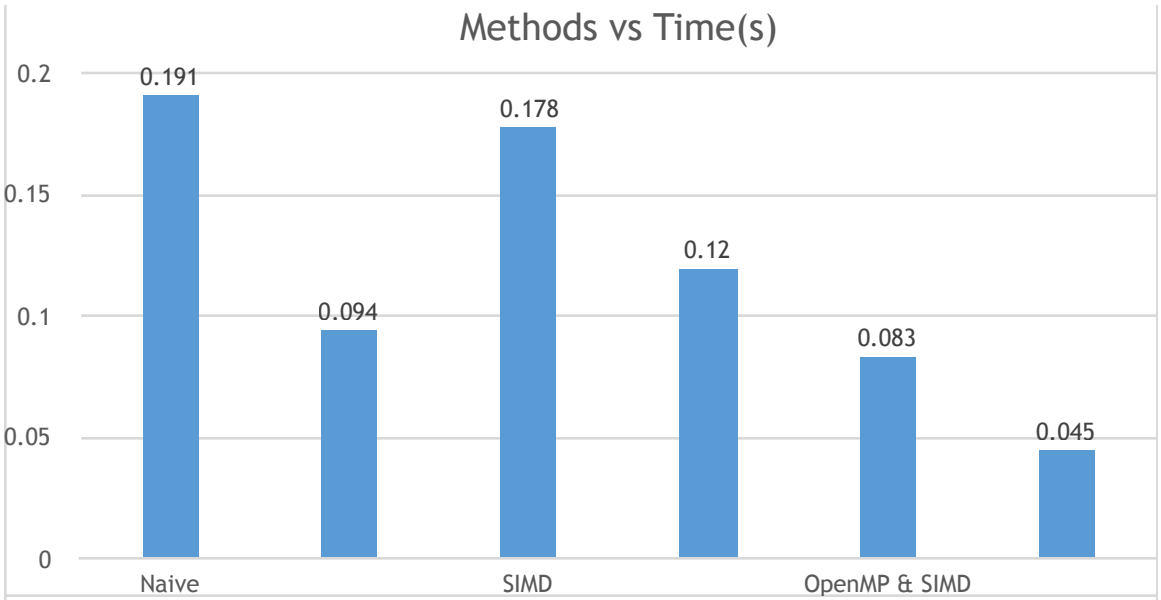
In addition, another performance boost is achieved by supplying flag “-mfma”, which asks the compiler to take advantage of the fused multiply-add instruction set that’s supported by Intel x86. The performance is increased by < 5%, which is still a visible improvement with such a simple trick.

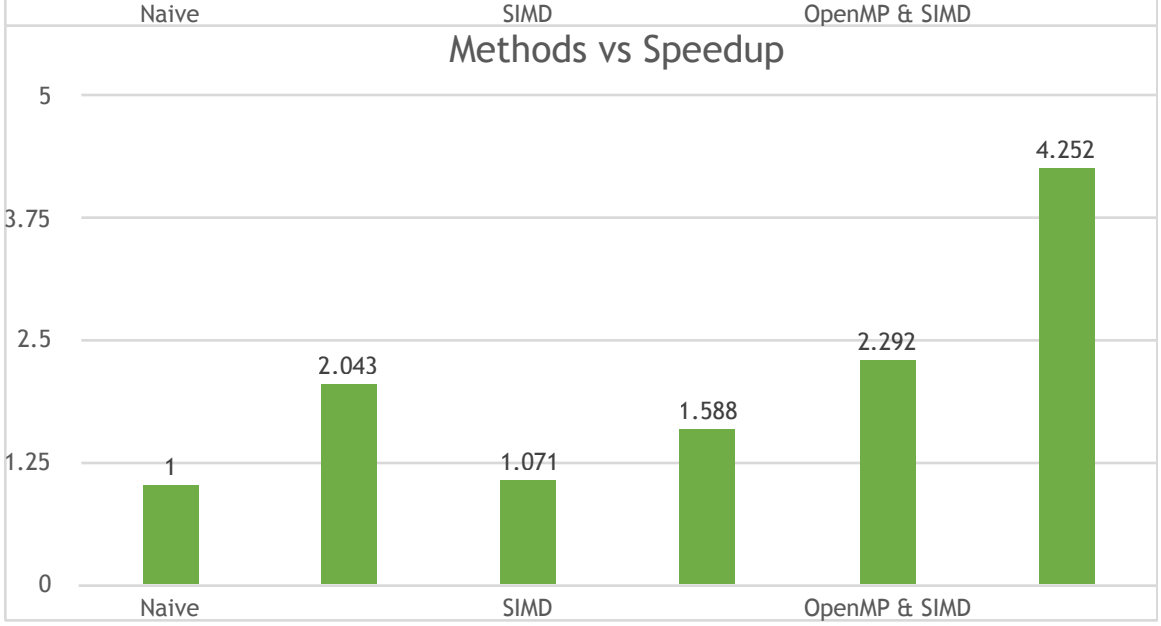
II. Testing Environment

MacBook Pro (13-inch, 2017, Two Thunderbolt 3 ports)
CPU: 2.3 GHz Intel Core i5
Number of Processors: 1
Total Number of Cores: 2
L2 Cache (per Core): 256 KB
L3 Cache: 4 MB
Memory: 8 GB 2133 MHz LPDDR3
Operating System: MacOS High Sierra 10.13.1 (17B1003)
Compiler: gcc-7
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	0.191	1
OpenMP	0.094	2.0426
SIMD	0.178	1.0712
Loop Unrolling	0.120	1.5883
OpenMP & SIMD	0.083	2.2922
OpenMP & SIMD & Loop Unrolling	0.045	4.2515


Methods vs Time(s)
0.2
0.15
0.1
0.05
0
0.191
0.094
0.178
0.12
0.083
0.045
Naive
SIMD
OpenMP & SIMD


Methods vs Speedup
5
3.75
2.5
1.25
0
1
2.043
1.071
1.588
2.292
4.252
Naive
SIMD
OpenMP & SIMD

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.

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)];
                    }}}}}
```

Since there are 2 cores in the CPU, the theoretical maximum speedup is 2. Somehow the speed gain exceeds 200% which shouldn't happen theoretically. The error is small so we consider this as a flute.

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);
            }}}}}
```

The Intel CPU supports the Intel AVX2 and FMA extensions. The size of result matrix is 64bits wide, so 4 operations can be done in parallel. This will give a speedup of 4 theoretically. 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.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];
}}}}}
```

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.

5. 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);
                }}}}}
```

The speedup is very close to the speedups of openmp and simd simply muptiplied. This shows that openmp and simd can work together quite well.

6. 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);
                                        }}}}}
```

It shows that loop unrolling works worse with simd and openmp. The guess is that SIMD has some sort of loop unrolling logic as well, which offsets the effectiveness of following loop unrolling optimizations.

V. Conclusion

Many methods can be used to speed up a computationally intensive operation such as matrix convolution. Here we explored a few well-known methods and the result is encouraging. We didn't experiment with several other ways of optimization such as GPU utilization, MPI and cross-language comparison. Nonetheless, this project still provides some useful insights into how to speed u p seemingly non-trivial production works with very simple methods and yet achieve very impressive results.