Songklanakarin J. Sci. Technol.
43 (4), 1230-1236, Jul. - Aug. 2021

http://www.sjst.psu.ac.th

Original Article

High performance 2D convolution utilizing the AVX512 on a multi-core architecture

Isamail Masamae and Panyayot Chaikan*

*Department of Computer Engineering, Faculty of Engineering,
Prince of Songkla University, Hat Yai, Songkhla, 90112 Thailand*

Received: 20 February 2020; Revised: 19 August 2020; Accepted: 25 September 2020

Abstract

Convolution is a time consuming operation, especially for signal and image processing, which led us to develop an efficient implementation of 2D convolution for a multi-core architecture utilizing AVX512 intrinsics and OpenMP. For single precision convolution, our algorithm is on average 2.30, 3.88, 5.75, and 19.95 times faster than the IPP, OpenCV, Baziotis's algorithm, and MKL libraries. For double precision convolution, our algorithm is on average 3.12, 5.10, and 16.95 times faster than the OpenCV, Baziotis's algorithm, and MKL libraries. We have also developed a hybrid 2D convolution algorithm, written in C and assembly, to further augment the processing speeds for small kernel sizes.

Keywords: AVX512, Advanced Vector Extension, 2D convolution, OpenMP, x64 assembly language

1. Introduction

The Advanced Vector Extension (AVX) has become a ubiquitous part of the 80x86 architecture. It consists of sixteen 256-bit registers that allow 8 single precision or 4 double precision floating point pieces of data to be processed simultaneously. Its successor, the AVX512, doubles the size of each register from 256 to 512 bits, allowing 16 single precision or 8 double precision floating point data items to be processed at the same time. Several researchers have looked at using the AVX512 to augment processing speeds. For example, Kodama and Ishiyama (2019) accelerated the calculation of quadrupole terms in the Barnes-Hut tree code, Rucci *et al.* (2019) enhanced the computational speed of the Smith-Waterman (SW) algorithm, and Watanabe and Nakagawa (2019) employed AVX2 and AVX512 to calculate the Lennard-Jones force potential.

Two dimensional (2D) convolution is a computationally intensive image processing operation required in many filtering applications. Although modern processors are equipped with powerful SIMD instructions, the traditional 2D convolution algorithm and the automatic vectorization capabilities of compilers cannot fully exploit the computing capacity of the SIMD execution engines in multi-core architectures. In this paper, we employ AVX512 intrinsics to augment the convolution speed.

This paper is organized as follows: section 2 introduces background and related work, section 3 describes our algorithm and a hybrid 2D convolution implementation written in C and assembly, section 4 presents experimental results, and section 5 concludes the paper.

*Corresponding author
Email address: panyayot@coe.psu.ac.th

2. Background and Related Work

The 2D convolution of a template *T* of size $T_{x\_sz} \times T_{y\_sz}$ with a kernel *K* of size *M*×*N* is given by the expression:

$$OP(x,y) = \sum_{m=-a}^{a} \sum_{n=-b}^{b} K(m,n) \cdot T(x-m, y-n) \qquad (1)$$

where *OP* is the output of the convolution, and *a* = (*M*-1) / 2 and b = (*N*-1) / 2.

In timing critical applications, special hardware accelerators using FPGAs have been proposed as a way of optimizing convolution. For non-timing critical applications, software optimization is often preferred. For example, Bipin and Nair (2016) optimized convolution using a sparse matrix vector multiplication technique, while Baziotis (2018) accelerated convolution by means of a collaboration between AVX2 and MPI.

Jin and Finkel (2019) compared the performance of 2D convolution on three platforms: the CPU, the GPU, and the FPGA. They reported that the FPGA outperformed both the CPU and GPU for larger kernel sizes, but that the GPU was faster when the kernel size was smaller. Also, the FPGA proved to be unsuitable for double precision convolution in terms of performance and resource utilization. Although the GPU delivered good performance, it consumed more power than the other approaches.

Amiri and Shahbahrami (2017) showed that manual vectorization using the compiler's intrinsics outperformed automatic vectorization by both the GCC and LLVM compilers. They also proposed a way to utilize AVX2 instructions to increase convolution speed, but their method did not employ a multicore architecture, and fine tuning and optimization methods were not investigated.

Many APIs support SIMD instructions, for example, the OpenCV library has supported AVX512 since version 3.4.1 (Alekhin, 2020) while the convolution function of Intel's MKL library utilizes AVX512 instructions if the CPU supports them (Intel, 2019a), as does the IPP library (Intel, 2019b).

In this paper, we focus on the utilization of AVX512 to augment convolution speed, using two programming models on a multi-core architecture: (1) compiler intrinsics and (2) calling AVX512 instructions directly via assembly language. Optimization and fine-tuning methods are proposed to maximize the utilization of the FMA engines in each processing core, which allows the developers to adapt our algorithm to process different sizes of data on different numbers of cores.

3. Our Proposed AVX512 Convolution Algorithms

Let *T*, *K*, and *OP* be the template, the kernel, and the output of the convolution of size $T_{x\_sz}\times T_{y\_sz}$, $K_{x\_sz}\times K_{y\_sz}$, and $OP_{x\_sz}\times OP_{y\_sz}$ respectively. The convolution operation, defined as $OP = T * K$, is obtained by performing three steps:

Step 1: $SIMD_{sz}$ is the number of data elements that the AVX512 instructions can process at a time, which is 16 and 8 for single and double precision data types. Zero padding is applied to the template initially, producing *TZ* of size $TZ_{x\_sz}\times TZ_{y\_sz}$, which has two main benefits. Firstly, it lets the convolution kernel be applied to the edges of a template more easily with more general code. Also, the AVX512 instruction reads 64 bytes of memory at a time, which means that T_{x_sz} should be divisible by $SIMD_{sz}$. If it isn't then the remaining template data on each row must be manipulated using a normal SISD load instruction, which reduces the performance. The TZ_{x_sz} value is determined by:

$$TZ_{x\_sz} = \begin{cases} T_{x\_sz} + 2(K_{x\_sz}/2), & \text{if } ((T_{x\_sz} + 2(K_{x\_sz}/2))\,\%\,SIMD_{sz}) = 0 \\ \\ T_{x\_sz} + 2(K_{x\_sz}/2) + SIMD_{sz} - (T_{x\_sz} + 2(K_{x\_sz}/2))\,\%\,SIMD_{sz}, & \text{otherwise,} \end{cases} \tag{2}$$

where / and % are integer division and modulo. The TZ_{y_sz} value is determined using:

$$TZ_{y\_sz} = T_{y\_sz} + 2(K_{y\_sz}/2). \tag{3}$$

Once memory of size $TZ_{x_sz}\times TZ_{y_sz}\times$sizeof(data type) has been allocated, the *T* data is copied into *TZ* using the following condition:

$$TZ[y][x] = \begin{cases} T[y-K_{y\_sz}/2][x-K_{x\_sz}/2], & \text{if } (K_{y\_sz}/2 \le y < TZ_{y\_sz} - K_{y\_sz}/2) \text{ and} \\ & (K_{x\_sz}/2 \le x < TZ_{x\_sz} - K_{x\_sz}/2), \\ \\ 0, \quad \text{otherwise.} \end{cases} \tag{4}$$

Step 2: Memory of size $K_{x_sz}\times K_{y_sz}\times$sizeof(data type) is allocated, producing a space for a flipped kernel, named *KF*. All of the original kernel, *K*, is copied into *KF* using the following condition:

$$KF[i] = K[K_{x\_sz} * \left(K_{y\_sz} - i - 1\right)]. \tag{5}$$

Step 3: The convolution algorithm shown in Figure 1, or the alternative in Figure 4, is applied to the data in *TZ* and *KF*, producing the convolution result, *OP*.

```
#define SIMDsz 8
#pragma omp parallel num_threads(N_THREADS)
{
  int tid, i, x, y;
  tid = omp_get_thread_num();
  __m512 out_vect, x_vect, kernel_vector;
  for (int y = tid; y<Ty_sz; y += NUM_THREADS)
  {
    for (int x = 0; x < Tx_sz; x += SIMDsz)
    {
      out_vect = _mm512_setzero_pd();
      for (int i = 0; i < Kx_sz*Ky_sz; i++)
      {
        x_vect = _mm512_loadu_pd(&TZ[y+(i/Ky_sz)][x+(i%Kx_sz)]);
        kernel_vect = _mm512_set1_pd(KF[i]);
        out_vect = _mm512_fmadd_pd(x_vect, kernel_vect, out_vect);
      }
    _mm512_storeu_pd(&OP[y][x], out_vect);
    }
  }
}
```

Figure 1. The basic convolution algorithm utilizing AVX512 intrinsics

3.1 Basic convolution

Figure 1 shows our basic convolution algorithm using AVX512 intrinsics on double precision data. In this case, $SIMD_{SZ}$ is set to be 8 because the AVX512 can process 8 double precision data elements at a time. In each iteration of the *x* variable loop, all eight elements in *out_vect* are initialized to zero. In the innermost loop, eight elements from *TZ* are loaded into *x_vect* at a time, before being multiplied with the broadcasted kernel data, and the product is added to *out_vect*. These operations can be implemented using a single multiply-accumulate operation called *_mm512_fmadd_pd*. After all the elements in the kernel have been processed at the end of the innermost loop, *out_vect* is stored in the *OP* array.

The "#pragma omp" directive enables the compiled program to create multiple concurrent threads that are assigned to different processing cores. This lets the data be split between processing cores to encourage data parallelism, as shown in Figure 2. The ID of each thread is obtained by means of the *omp_get_thread_num* function, and are used to split the template data among the processing threads. For example, if T_{y_sz} = 20 and *NUM_THREADS* = 4, then the threads number 0, 1, 2, and 3 will process the data at y = {0, 4, 8, 12, 16}, {1, 5, 9, 13, 17}, {2, 6, 10, 14, 18}, and {3, 7, 11, 15, 19} respectively, as shown in Type I of Figure 2. This decomposition allows data to be shared between the processing cores at a higher level than the block-wise decomposition employed by Type II in Figure 2.

The Figure 1 code only supports double precision data. For it to handle single precision data, all the *__m512d* variables must be changed to be of type *__m512*, and all the AVX512 operations of the form *_mm512_<op>_pd* must be changed to *_mm_512_<op>_ps*. Also, the $SIMD_{SZ}$ value must be changed from 8 to 16 to reflect the number of elements that the AVX512 can process simultaneously.

3.2 Unrolled convolution

Although the algorithm in Figure 1 speeds up the calculation by allowing 8 data elements to be processed simultaneously by each AVX512 instruction, the calculation still relies on storing the result of each multiply-accumulate operation in one *out_vect* variable. This data dependency between each loop iteration leads to inefficient usage of the FMA engines in each processing core. To overcome this problem, we utilize an unrolling technique to minimize the data dependencies between the loop iterations, although two important questions are raised. The first is how to unroll the AVX intrinsic code efficiently to maximize the utilization of the FMA engines. The second is how to define a general unrolling method to support any unrolling size factor for the various sizes of template and kernel.

Let *UF* be an unrolling factor which specifies the number of independent FMA intrinsic functions to be explicitly called in the C code. In order to perform a convolution between the kernel and all the data in each row of the template, multiple iterations are required to calculate the data of size $UF \times SIMD_{SZ}$. The general form of our unrolling method is shown in Figure 3. In each template row, the shaded part is the data that can be processed in the innermost loop, and the unshaded part is the remainder, of size $R \times SIMD_{SZ}$, for which the AVX intrinsic function need to be called again to perform the calculation at the end of the loop.

Figure 4 shows the unrolled version of our AVX algorithm. The *UF* value in this code is actually the number of independent FMA calculations allowed to take place in each iteration of the *x*-loop. The *out_vect* numbers must match this factor, as does *x_vect*. At each iteration of the *x*-loop, *x* is increased by $UF \times SIMD_{SZ}$, while the upper bound, x_{ub}, is:

$$x_{ub} = T_{x\_sz} - \left(T_{x\_sz} \,\%\, \left(UF \times SIMD_{SZ}\right)\right). \tag{6}$$

At the end of the *x*-loop, if the values of T_{x_sz} and x_{ub} are not equal, then there must be some remaining elements of *TZ* that have not been processed. *R* numbers of *out_vect* are cleared again, and the remaining elements of *TZ* are multiplied with the broadcasted kernel before being accumulated in the *out_vect* variables. The FMA intrinsic function is called *R* times to complete the multiply-accumulate operation for the

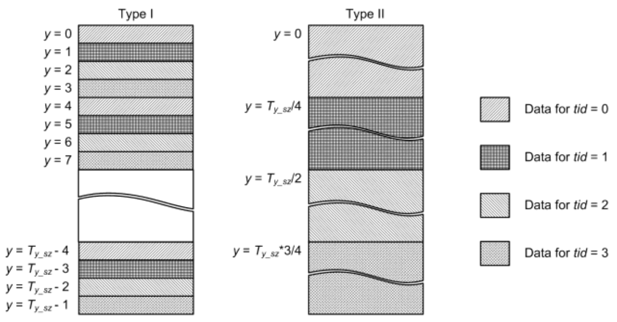


Figure 2. Two examples of data decomposition among four processing threads

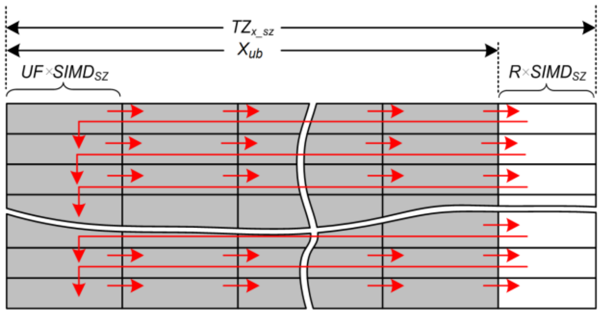
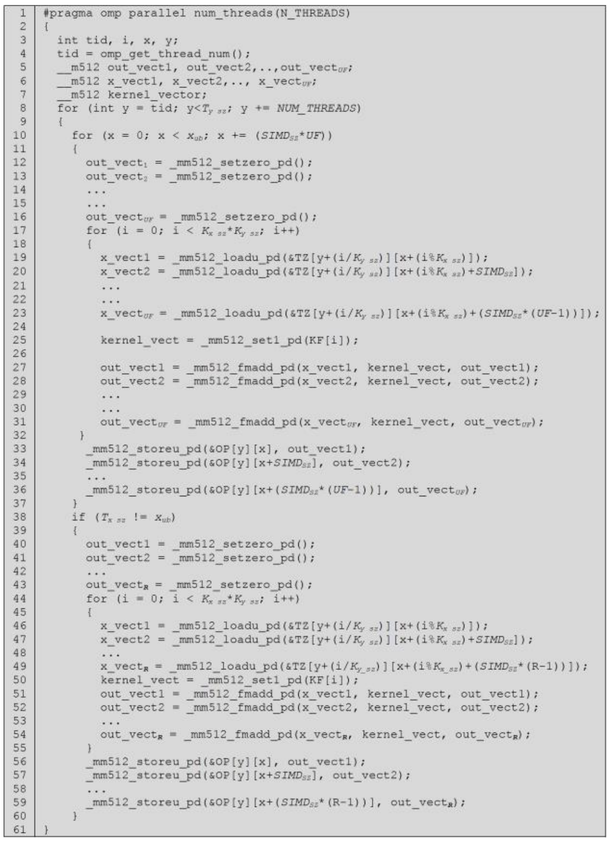


Figure 3. The general form of our AVX512 convolution utilizing unrolling



```
#pragma omp parallel num_threads(N_THREADS)
{
  int tid, i, x, y;
  tid = omp_get_thread_num();
  __m512 out_vect1, out_vect2,..,out_vect_UF;
  __m512 x_vect1, x_vect2,.., x_vect_UF;
  __m512 kernel_vector;
  for (int y = tid; y<T_y_sz; y += NUM_THREADS)
  {
    for (x = 0; x < x_ub; x += (SIMD_SZ*UF))
    {
      out_vect_1 = _mm512_setzero_pd();
      out_vect_2 = _mm512_setzero_pd();
      ...
      ...
      out_vect_UF = _mm512_setzero_pd();
      for (i = 0; i < K_x_sz*K_y_sz; i++)
      {
        x_vect1 = _mm512_loadu_pd(&TZ[y+(i/K_y_sz)][x+(i%K_x_sz)]);
        x_vect2 = _mm512_loadu_pd(&TZ[y+(i/K_y_sz)][x+(i%K_x_sz)+SIMD_SZ]);
        ...
        ...
        x_vect_UF = _mm512_loadu_pd(&TZ[y+(i/K_y_sz)][x+(i%K_x_sz)+(SIMD_SZ*(UF-1))]);

        kernel_vect = _mm512_set1_pd(KF[i]);

        out_vect1 = _mm512_fmadd_pd(x_vect1, kernel_vect, out_vect1);
        out_vect2 = _mm512_fmadd_pd(x_vect2, kernel_vect, out_vect2);
        ...
        ...
        out_vect_UF = _mm512_fmadd_pd(x_vect_UF, kernel_vect, out_vect_UF);
     }
      _mm512_storeu_pd(&OP[y][x], out_vect1);
      _mm512_storeu_pd(&OP[y][x+SIMD_SZ], out_vect2);
      ...
      _mm512_storeu_pd(&OP[y][x+(SIMD_SZ*(UF-1))], out_vect_UF);
    }
    if (T_x_sz != x_ub)
    {
      out_vect1 = _mm512_setzero_pd();
      out_vect2 = _mm512_setzero_pd();
      ...
      out_vect_R = _mm512_setzero_pd();
      for (i = 0; i < K_x_sz*K_y_sz; i++)
      {
        x_vect1 = _mm512_loadu_pd(&TZ[y+(i/K_y_sz)][x+(i%K_x_sz)]);
        x_vect2 = _mm512_loadu_pd(&TZ[y+(i/K_y_sz)][x+(i%K_x_sz)+SIMD_SZ]);
        ...
        x_vect_R = _mm512_loadu_pd(&TZ[y+(i/K_y_sz)][x+(i%K_x_sz)+(SIMD_SZ*(R-1))]);
        kernel_vect = _mm512_set1_pd(KF[i]);
        out_vect1 = _mm512_fmadd_pd(x_vect1, kernel_vect, out_vect1);
        out_vect2 = _mm512_fmadd_pd(x_vect2, kernel_vect, out_vect2);
        ...
        out_vect_R = _mm512_fmadd_pd(x_vect_R, kernel_vect, out_vect_R);
      }
      _mm512_storeu_pd(&OP[y][x], out_vect1);
      _mm512_storeu_pd(&OP[y][x+SIMD_SZ], out_vect2);
      ...
      _mm512_storeu_pd(&OP[y][x+(SIMD_SZ*(R-1))], out_vect_R);
    }
}
```

Figure 4. The general form of our algorithm using unrolling

remaining data loaded from *TZ*, kept in *x_vect*, and the kernel. The value of *R* is defined by:

$$R = \left(T_{x\_sz} \,\%\, \left(UF \times SIMD_{SZ}\right)\right) / \, SIMD_{SZ}. \tag{7}$$

Figure 5 shows the dataflow of our AVX512 convolution algorithm utilizing unrolling. In each *i*-loop, *UF* independently fused multiply-add operations are fed to the FMA pipelines of each CPU core.

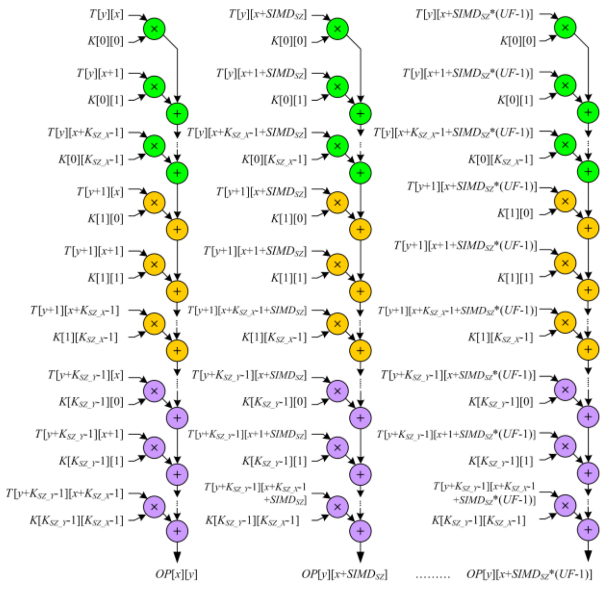


Figure 5. Dataflow graph of our AVX512 convolution with unrolling

3.3 Hybrid 2D convolution using assembly language

The 2D convolution algorithms proposed in section 3.1 and 3.2 are written in C, with the utilization of AVX512 intrinsic functions carried out by the C compiler. Calling intrinsic functions in C gives better performance than automatic vectorization, as confirmed by Amiri and Shahbahrami (2017). However, there are two other ways to augment the performance of the AVX512 intrinsics C code: (1) employing inline assembly language, or (2) reimplementing the entire algorithm in assembly. Since some compilers do not support inline assembly for 64-bit applications, we chose to rewrite the algorithm.

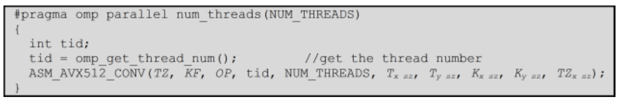
Writing a convolution algorithm in assembly using the AVX512 machine instructions for a multi-core architecture is both cumbersome and error prone. We decided on a compromise method which offered the performance of assembly, but supported the multi-core architecture using OpenMP.

Figure 6 shows C code for calling a function named *ASM_AVX512_CONV* inside an OpenMP directive. This function is implemented using assembly, but most C compilers allow an .asm file to be added to a project as an "extern" statement, such as:

extern "C" { void *ASM_AVX512_CONV*(type argument1, type argument2, ...); }.

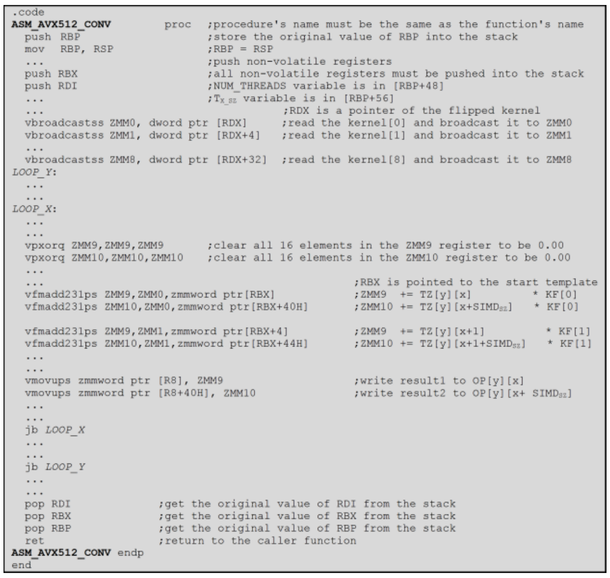
The "pragma omp" statement in Figure 6 creates *NUM_THREADS* parallel threads, which will each call the *ASM_AVX512_CONV* function. Each thread will determine which segment of data to process based on the thread ID that it receives as an argument, as shown in Figure 2.

For *ASM_AVX512_CONV*, the values for the algorithm's *TZ*, *KF*, *OP*, and the thread ID are passed via the registers RCX, RDX, R8, and R9, while the other parameters are passed using the stack. The length of the assembly source (over 250 lines) means that only key parts are shown in Figure 7.



```
#pragma omp parallel num_threads(NUM_THREADS)
{
  int tid;
  tid = omp_get_thread_num();          //get the thread number
  ASM_AVX512_CONV(TZ, KF, OP, tid, NUM_THREADS, Tx sz, Ty sz, Kx sz, Ky sz, TZx sz);
}
```

Figure 6. Calling an assembly language function inside an OpenMP directive



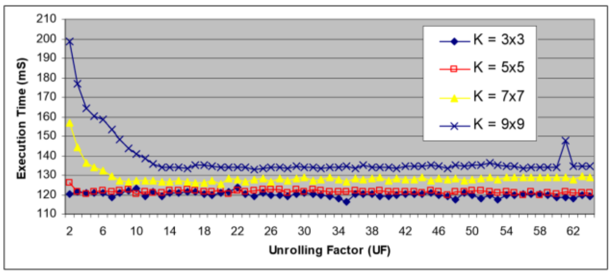
```
.code
ASM_AVX512_CONV        proc  ;procedure's name must be the same as the function's name
  push  RBP                  ;store the original value of RBP into the stack
  mov   RBP, RSP             ;RBP = RSP
  ...                        ;push non-volatile registers
  push RBX                   ;all non-volatile registers must be pushed into the stack
  push RDI                   ;NUM_THREADS variable is in [RBP+48]
  ...                        ;Tx_sz variable is in [RBP+56]
  ...                                    ;RDX is a pointer of the flipped kernel
  vbroadcastss ZMM0, dword ptr [RDX]     ;read the kernel[0] and broadcast it to ZMM0
  vbroadcastss ZMM1, dword ptr [RDX+4]   ;read the kernel[1] and broadcast it to ZMM1
  ...
  vbroadcastss ZMM8, dword ptr [RDX+32]  ;read the kernel[8] and broadcast it to ZMM8
LOOP_Y:
  ...
  ...
LOOP_X:
  ...
  ...
  vpxorq ZMM9,ZMM9,ZMM9      ;clear all 16 elements in the ZMM9 register to be 0.00
  vpxorq ZMM10,ZMM10,ZMM10   ;clear all 16 elements in the ZMM10 register to be 0.00
  ...
  ...                                            ;RBX is pointed to the start template
  vfmadd231ps ZMM9,ZMM0,zmmword ptr[RBX]         ;ZMM9  += TZ[y][x]           * KF[0]
  vfmadd231ps ZMM10,ZMM0,zmmword ptr[RBX+40H]    ;ZMM10 += TZ[y][x+SIMDsz]    * KF[0]

  vfmadd231ps ZMM9,ZMM1,zmmword ptr[RBX+4]       ;ZMM9  += TZ[y][x+1]          * KF[1]
  vfmadd231ps ZMM10,ZMM1,zmmword ptr[RBX+44H]    ;ZMM10 += TZ[y][x+1+SIMDsz]   * KF[1]
  ...
  ...
  vmovups zmmword ptr [R8], ZMM9                 ;write result1 to OP[y][x]
  vmovups zmmword ptr [R8+40H], ZMM10            ;write result2 to OP[y][x+ SIMDsz]
  ...
  ...
  jb LOOP_X
  ...
  ...
  jb LOOP_Y
  ...
  ...
  pop RDI              ;get the original value of RDI from the stack
  pop RBX              ;get the original value of RBX from the stack
  pop RBP              ;get the original value of RBP from the stack
  ret                  ;return to the caller function
ASM_AVX512_CONV endp
end
```

Figure 7. Part of the ASM_AVX512_CONV function

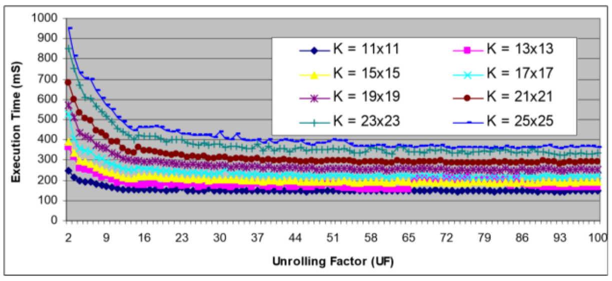
4. Experimental Results

The performance of our algorithm was tested with C code written using Microsoft Visual Studio C++ 2017. The test machine was a 3.3GHz Core i9-7900X with hyperthreading turned on. The test template was fixed at 10240×10240, with a convolution kernel of *K*×*K*, with twelve different *K* values, starting from 3, incremented by 2, until equal to 25. To find the best unrolling factor, we evaluated our algorithm across 99 *UF* values, ranging from 2 to 100. The performance results are shown in Figures 8 and 9.

Figure 8(a) shows the performance of our algorithm when the kernel size was between 3×3 and 9×9. As the unrolling factor increased, performance improved, but stopped when the factor exceeded 35. As a consequence, we fixed the factor at 35 for kernel sizes between 3×3 and 9×9, and only the performance of unrolling factors between 2 and 64 are shown in Figure 8(a).



(a)



(b)

Figure 8. Performance of our single precision algorithm for various unrolling factors, tested with (a) kernel sizes smaller than 11×11, and (b) kernel sizes between 11×11 and 25×25

For kernel sizes between 11×11 and 25×25, the relationship between performance and the unrolling factor trended in the same direction, but the best performance point changed, as shown in Figure 8(b). For kernel sizes between 11×11 and 21×21, the best performance was reached when the factor was 62, and there was no significant improvement with larger values. An unrolling factor of 93 gave the optimum performance for kernel sizes of 23×23 and 25×25.

Figure 9 illustrates the performance of our double precision algorithm. The optimum unrolling factors for different kernel sizes are shown in Table 1.

The performance of our algorithm was compared with three well known libraries: Intel's MKL (version 2019, update 5), Intel's IPP (version 2019, update 5), and OpenCV (version 4.2.0). In addition, we also compared the performance with the algorithm developed by Baziotis (2018). The performance was evaluated on a template of size 10240×10240. As before, these algorithms and libraries were tested using 12 different kernel sizes, and the performance results are shown in Figures 10 to 13.

Figure 10(a) shows that our algorithm is faster than the MKL library for all kernel sizes. OpenCV is also slower than our algorithm, but by a smaller amount even though it also utilizes AVX512 instructions. Baziotis's algorithm is slower than OpenCV for almost every configurations because it only supports AVX2, which is theoretically slower than AVX512.

Our algorithm outperforms IPP for kernel sizes of 5×5 or larger, but an IPP kernel size of 3×3 is 5 percent faster than our algorithm. Figure 10(b) shows the speed-up ratios of our algorithm compared to MKL, IPP, and OpenCV when performing single precision convolution. Our algorithm is on average 2.30, 3.88, 5.75, and 19.95 times faster than IPP, OpenCV, Baziotis's algorithm, and MKL respectively.

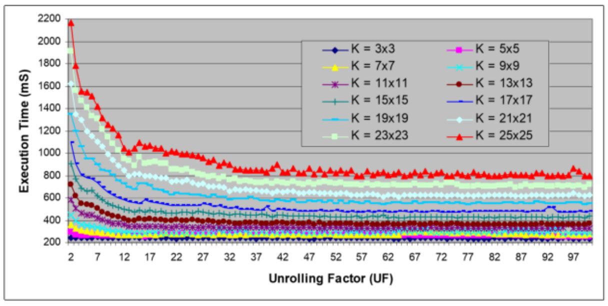
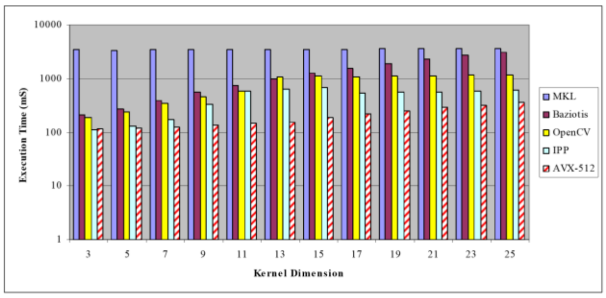


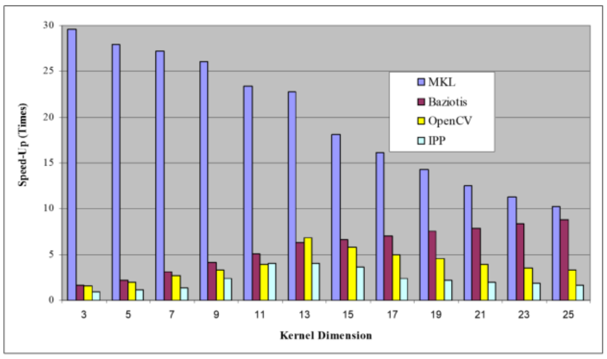
Figure 9. Performance of our double precision algorithm for various unrolling factors, tested with kernel sizes between 3×3 and 25×25

Table 1. The optimum unrolling factors for our convolution algorithm

Convolution type	Kernel dimension	Optimum unrolling factor (*UF*)
Single precision	3, 5, 7, 9	35
	11, 13, 15, 17, 19, 21	62
	23, 25	93
Double precision	3, 5, 7, 9	16
	11, 13, 15	40
	17, 19, 21, 23, 25	81



(a)



(b)

Figure 10. Performance comparisons of a single precision convolution with our algorithm versus Baziotis's algorithm, MKL, IPP, and the OpenCV libraries: (a) execution time, and (b) speed-up

Figure 11(a) shows the performance of our double precision convolution using our algorithm compared to MKL,

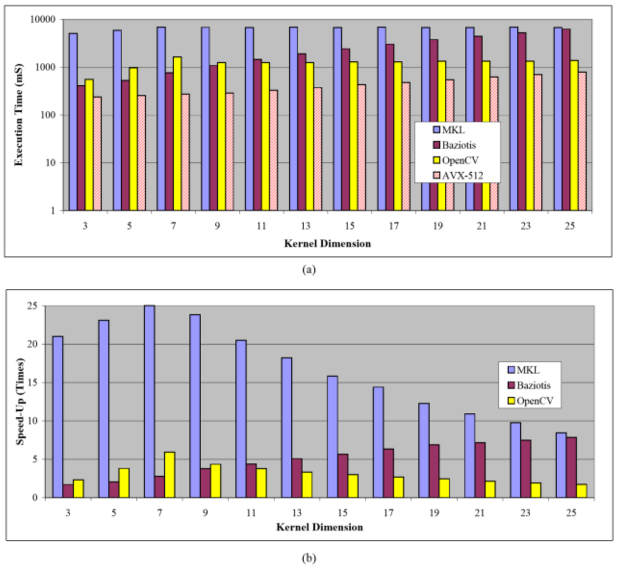


Figure 11. Performance comparisons of a double precision convolution with our algorithm versus MKL, Baziotis's algorithm, and the OpenCV libraries: (a) execution time, and (b) speed-up

Baziotis's algorithm, and OpenCV. IPP was not included because its convolution function does not support double precision data. Our algorithm outperforms MKL in every configuration. When no unrolling was applied, our algorithm was faster than OpenCV for kernel sizes of 15×15 or larger, but slower for all smaller sizes. However, after unrolling was applied with the optimum factor shown in Table 1, our algorithm outperformed OpenCV, Baziotis's algorithm, and MKL in all configurations. On average, our algorithm with unrolling was 3.12, 5.10, and 16.95 times faster than OpenCV, Baziotis's algorithm, and MKL respectively.

For single precision convolution, our algorithm was superior to both OpenCV and IPP with a kernel size of 5×5 or larger. However, for a kernel of size 3×3, our algorithm was still faster than OpenCV, but 5 percent slower than IPP. Since a kernel of size 3×3 is frequently used in image processing applications, we looked at ways to improve the performance of our algorithm at this size. We investigated two approaches: loop tiling and data decomposition. Although loop tiling has proved effective for some problems, such as matrix-vector multiplication (Hassan, Mahmoud, Hemeida & Saber, 2018), it made our algorithm slower. Similarly, different forms of data decomposition among the processing threads only degraded performance.

Since the Microsoft Visual Studio 2017 does not support inline assembly for 64-bit applications, a complete rewrite in assembly was carried out.

Figure 12 shows the modified single precision convolution optimized for a kernel size of 3×3. To reduce reading overheads, the kernel elements are read only once and then broadcasted to 9 variables of type *__m512* before entering the main loop. This code was initially written in C, and ran a little faster than the previous code, but still slower than the IPP. When it was reimplemented in assembly, its performance exceeded the IPP. We fixed the kernel size at 3×3, with a template of *T*×*T*, with ten different *T* values,

```
 1  int UF = 2;            //the UF is set to 2 in this example
 2  __m512 k0 = _mm512_set1_ps(Flip_kernel[0]);
 3  __m512 k1 = _mm512_set1_ps(Flip_kernel[1]);
 4  ...
 5  __m512 k8 = _mm512_set1_ps(Flip_kernel[8]);
 6  for (int y=tid*(Ty_sz/NUM_THREADS); y<((tid+1)*(Ty_sz/NUM_THREADS)); y++)
 7  {
 8    for (int x=0; x < xub; x += (SIMDSZ*UF))
 9    {
10      out_vect1 = _mm512_setzero_ps();
11      out_vect2 = _mm512_setzero_ps();
12
13      x_vect1 = _mm512_loadu_ps(&TZ[y][x]);
14      x_vect2 = _mm512_loadu_ps(&TZ[y][x+ SIMDSZ]);
15      out_vect1 = _mm512_fmadd_ps(x_vect1, k0, out_vect1);
16      out_vect2 = _mm512_fmadd_ps(x_vect2, k0, out_vect2);
17
18      x_vect1 = _mm512_loadu_ps(&TZ[y][x+1]);
19      x_vect2 = _mm512_loadu_ps(&TZ[y][x+1+ SIMDSZ]);
20      out_vect1 = _mm512_fmadd_ps(x_vect1, k1, out_vect1);
21      out_vect2 = _mm512_fmadd_ps(x_vect2, k1, out_vect2);
22      ...
23      ...
24      x_vect1 = _mm512_loadu_ps(&TZ[y+2][x+2]);
25      x_vect2 = _mm512_loadu_ps(&TZ[y+2][x+2+ SIMDSZ]);
26      out_vect1 = _mm512_fmadd_ps(x_vect1, k8, out_vect1);
27      out_vect2 = _mm512_fmadd_ps(x_vect2, k8, out_vect2);
28
29      _mm512_storeu_ps(&OP[y][x], out_vect1);
30      _mm512_storeu_ps(&OP[y][x + SIMDSZ], out_vect2);
31    }
32    if (Tx_sz != xub)
33    {
34       ...//calculate the remaining elements the same way
35       ...//as described in lines 10-30
36    }
37  }
```

Figure 12. The AVX512 convolution optimized for a kernel size of 3×3

starting from 1024, incremented by 1024, until it was equal to 10240, and the performance is shown in Figure 13. Our assembly language convolution, optimized for kernel size of 3×3, is 1.04, 1.84, 2.28, and 24.46 times faster than IPP, OpenCV, Baziotis' algorithm, and MKL respectively.

Since our double precision convolution, which was written in C, already outperformed all the other algorithms, it was unnecessary to reimplement it in assembly. When tested with a kernel size of 3×3, with ten different template sizes, the performance results are illustrated in Figure 14. Our algorithm is 1.80, 2.10, and 18.08 times faster than Baziotis's algorithm, OpenCV, and MKL respectively.

5. Conclusions

Our 2D convolution algorithm achieves high performance, not only from the utilization of AVX512

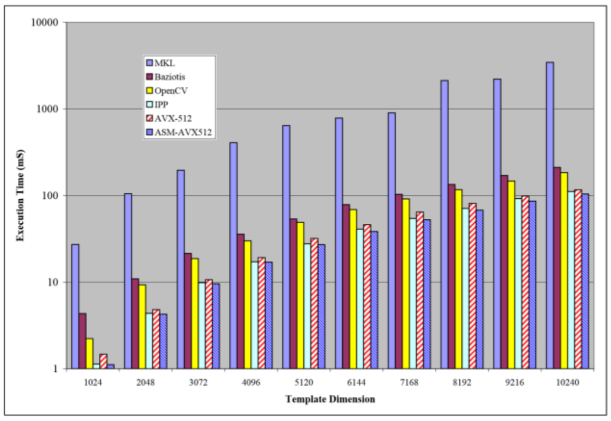


Figure 13. Performance comparison of our single precision convolution algorithms — optimized for a kernel size of 3×3 — versus MKL, Baziotis's algorithm, OpenCV, and IPP

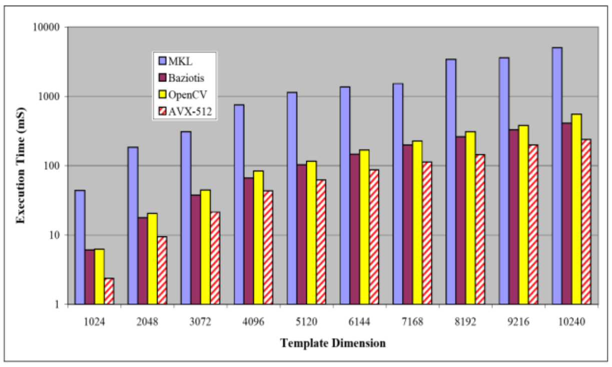


Figure 14. Performance comparison of our double precision convolution algorithms — optimized for a kernel size of 3×3 — versus MKL, Baziotis's algorithm, OpenCV, and IPP

instructions, but also from an awareness of data decomposition between the processing cores, combined with an unrolling mechanism that aims to maximize the utilization of the FMA engines in each core. This means that our algorithm is faster than convolution carried out by the OpenCV, the IPP, and the MKL libraries even though they also utilize AVX512 instructions. The main reasons that our single precision convolution with a kernel size of 5×5 or larger, and our double precision convolution algorithms, outperform these libraries is due to a high unrolling factor, combined with an optimum unrolling selected for each kernel size. These features enable our algorithm to achieve higher parallelism than these libraries, even though it is implemented in C. For single precision convolution, our algorithm is on average 2.30, 3.88, 5.75, and 19.95 times faster than the IPP, OpenCV, Baziotis's algorithm, and MKL libraries. For double precision convolution, our algorithm is on average 3.12, 5.10, and 16.95 times faster than the OpenCV, Baziotis's algorithm, and MKL libraries.

C with a high unrolling factor causes the compiler to use register spilling which requires data swapping between the AVX512 registers and memory. However, this situation does not occur in assembly code, so our assembly convolution is faster than the C version mainly because it reduces memory bottlenecks. The 3×3 kernel convolution in our assembly convolution is 1.04 times faster than IPP.

Acknowledgements

The authors are grateful to Dr. Andrew Davison for his kind help in polishing the language of this paper.

References

Alekhin, A. (2020). OpenCV change logs. Retrieved from https://github.com/opencv/opencv/ wiki/ChangeLog

Amiri, H., & Shahbahrami, A. (2017). High performance implementation of 2D convolution using intel's advanced vector extensions. *International Symposium on Artificial Intelligence and Signal Processing (AISP2017*), 25-30. doi:10.1109/AISP. 2017.8324097

Baziotis, S. (2018). 2D image convolution using MPI and AVX instructions. Retrieved from https://github. com/baziotis/2D-Image-Convolution-MPI-SIMD

Bipin, B., & Nair, J. J. (2016). Image convolution optimization using sparse matrix vector multiplication technique, *International Conference on Advances in Computing, Communications and Informatics (ICACCI2016)*, 1453-1457. doi:10. 1109/ICACCI.2016.7732252

Hassan, S. A., Mahmoud, M. M., Hemeida, A. M. & Saber, M. A. (2018). Effective Implementation of Matrix-Vector Multiplication on Intel's AVX Multicore Processor. *Computer Languages Systems and Structures*, *51*, 158-175. doi: 10.1016/j.cl.2017.06. 003

Intel Corporation. (2019a). Intel® Math Kernel Library Developer Reference, MKL 2019 Revision: 024. Retrieved from https://software.intel.com/sites/ default/files/mkl-2019-developer-reference-c.pdf

Intel Corporation. (2019b). Intel® Integrated Performance Primitives 2019, Retrieved from https://software. intel.com/sites/default/files/ipp-devguide.pdf

Jin, Z., & Finkel, H. (2019). Exploration of OpenCL 2D convolution Kernels on Intel FPGA, CPU, and GPU Platforms, *IEEE International Conference on Big Data (IEEE Big Data 2019)*. doi:10.1109/BigData 47090.2019.9006494

Kodama, T., & Ishiyama, T. (2019). Acceleration of the tree method with a SIMD instruction set, *Publications of the Astronomical Society of Japan*, *71*(2), Article No. 35. doi:10.1093/pasj/psy151

Rucci, E., Sanchez, C.G., Juan, G.B., De Giusti, A., Naiouf, M., & Prieto-Matias, M. (2019). SWIMM 2.0: Enhanced smith-waterman on intel's multicore and manycore architectures based on AVX-512 vector extensions. *International Journal of Parallel Programming*, 47(2), 296-316. doi:10.1007/s10766-018-0585-7

Watanabe, H., & Nakagawa, K. M. (2019). SIMD vectorization for the Lennard-Jones potential with AVX2 and AVX-512 instructions. *Computer Physics Communications*, *237*, 1-7. doi:10.1016/ j.cpc.2018.10.028