Songklanakarin J. Sci. Technol.
42 (6), 1368-1376, Nov. - Dec. 2020

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

Original Article

An effective implementation of Strassen's algorithm using AVX intrinsics for a multicore architecture

Nwe Zin Oo and Panyayot Chaikan*

*Department of Computer Engineering, Faculty of Engineering,
Prince of Songkla University, Hat Yai, Songkhla, 90110 Thailand*

Received: 23 April 2019; Revised: 24 August 2019; Accepted: 26 September 2019

Abstract

This paper proposes an effective implementation of Strassen's algorithm with AVX intrinsics to augment matrix-matrix multiplication in a multicore system. AVX-2 and FMA3 intrinsic functions are utilized, along with OpenMP, to implement the multiplication kernel of Strassen's algorithm. Loop tiling and unrolling techniques are also utilized to increase the cache utilization. A systematic method is proposed for determining the best stop condition for the recursion to achieve maximum performance on specific matrix sizes. In addition, an analysis method makes fine-tuning possible when our algorithm is adapted to another machine with a different hardware configuration. Performance comparisons between our algorithm and the latest versions of two well-known open-source libraries have been carried out. Our algorithm is, on average, 1.52 and 1.87 times faster than the Eigen and the OpenBLAS libraries, respectively, and can be scaled efficiently when the matrix becomes larger.

Keywords: advanced vector extension, AVX, AVX-2, matrix-matrix multiplication, FMA, Strassen's algorithm

1. Introduction

In recent years, the Advanced Vector Extension (AVX) instruction set has been bundled with all the CPUs produced by Intel and AMD. It allows multiple pieces of floating-point data to be processed at the same time, resulting in very high performance. Its successor, AVX-2, added 256-bit integer operations and fused-multiply-accumulate operations for floating-point data, useful for scientific applications. Many researchers have reported on its use to augment processing performance. For example, Kye, Lee, and Lee (2018) increased the processing speed of matrix transposition, Al Hasib, Cebrian, and Natvig (2018) proposed an implementation of k-means clustering for a compressed dataset, and Bramas and Kus (2018) speeded up the processing of a sparse matrix-vector product. Hassan, Mahmoud, Hemeida, and Saber (2018) utilized AVX and OpenMP to accelerate vector-matrix multiplication, Barash, Guskova and Shchur (2017) improved the performance of random number generators, and Bramas (2017) boosted the speed of the quicksort algorithm.

We employ AVX intrinsics for an effective implementation of Strassen's algorithm for single precision matrix-matrix multiplication. We decided to utilize AVX-2 with its FMA3 capabilities, which are available in reasonably priced CPUs, from both Intel and AMD. Our aim is to augment the speed of applications that rely on matrix-matrix multiplication using off-the-shelf CPUs.

2. Matrix-Matrix Multiplication

Matrix-matrix multiplication, defined as $c = a \times b$, where a, b, and c are $n \times n$, requires $2n^3$ floating-point operations. The basic sequential algorithm is shown in Figure 1. The performance of the algorithm in Giga Floating-Point Operation per Second (GFLOPS) is

$$GFLOPS = \frac{2*n*n*n}{s*10^9}, \qquad (1)$$

where s is the execution time of the program in seconds.

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

```
for(i=0; i<n; i++)
  for (k = 0; k<n; k++)
    for (j = 0; j<n; j ++)
      c[i][j] = c[i][j] + a[i][k] * b[k][j];
```

Figure 1. Basic sequential algorithm to calculate matrix-matrix multiplication of size *n*×*n*.

3. AVX Instruction Set and the FMA3

The third generation of Intel's Advanced Vector Extensions (AVX) (Intel, 2011) comprises sixteen 256-bit registers, YMM0-YMM15, supporting both integer and floating-point operations. The AVX instructions allow eight single-precision floating-point operations to be processed simultaneously, twice the number supported by Streaming SIMD Extensions (SSE). There are four main ways to take advantage of AVX instructions: 1) using assembly language to call AVX instructions directly; 2) using the AVX inline assembly in C or C++; 3) using compiler intrinsics; or 4) utilizing the compiler's automatic vectorization feature. We employ compiler intrinsics to implement Strassen's algorithm because this gives better performance than auto-vectorization, but is not as cumbersome or error prone as assembly language. Using AVX inline assembly in a high level language is not significantly different from utilizing compiler intrinsics (Hassana, Hemeida, & Mahmoud, 2016).

The syntax of AVX intrinsic functions follows the pattern _mm256_<*operation*>_ <*suffix*> (Mitra, Johnston, Rendell, McCreath, & Zhou, 2013) where the *operation* can be load, store, arithmetic, or logical operation, and the *suffix* is the type of data used. For example, *_mm256_add_ps* and *_mm256_add_pd* add 32-bit and 64-bit floating-point data respectively. Figure 2 shows more function prototype examples.

Floating-point matrix-matrix multiplication relies on the fused-multiply-add operation, which can be implemented using the *_mm256_mul_ps* and *_mm256_add_ps* functions. However, replacing these two functions with a single *_mm256 _fmadd_ps* call can speed up the computation. This fused-multiply-add (FMA) operation performs the multiplication and addition of the 64-bit floating-point data in a single step with rounding. Intel Haswell processors have supported FMA since 2013 (Intel, 2019), and the processors currently produced by AMD also support it (Advanced Micro Devices, 2019).

4. Optimization Methods for Parallel Matrix-Matrix Multiplication

The single-instruction-multiple-data (SIMD) processing of the AVX gives higher performance than using scalar instructions, and every processing core has an AVX unit. As a consequence, very high performance is expected by utilizing AVX instructions on a multi-core machine. Also, to maximize the performance of the parallel application utilizing AVX, OpenMP is employed in conjunction with two optimization techniques: loop tiling and loop unrolling.

4.1 Loop tiling

If the size of the data is very large, it is impossible to keep it all inside the cache. Data movements between the cache and main memory may be required very often, leading to many cache miss penalties. To reduce this effect, the data can be split into smaller chunks, and each chunk is loaded by the processor and kept inside the cache automatically by its cache controller. Increased reuse of these data from the cache leads to improved performance. In the case of matrix-matrix multiplication, $c=a\times b$, the matrices are stored in 3 arrays, and each data element of *a* and *b* will be accessed multiple times. If the matrix size is $n\times n$, then each data element from each matrix will be accessed at least *n* times. When loop tiling is applied, the outer loop keeps a chunk of the first source matrix inside the cache, while a series of chunks taken from the second matrix are processed by the inner loop. This pattern allows the chunk of the first matrix to be reused many times before being flushed from the cache. The next chunk from the first matrix will then be processed using the same pattern, and so on. The chunk size in the outer loop must be large enough to minimize memory accesses and to increase temporal locality. However, it must not be larger than the L1 data cache to prevent some of the data being evicted to the higher cache level. The programmer must find an appropriate chunk size for each implementation.

4.2 Loop unrolling

Loop unrolling is an attempt to increase program execution speed by sacrificing program code size. By reducing the number of loop iterations, branch penalties will be potentially decreased, but the loop body will become larger as it is unrolled. Also modern CPUs use superscalar and out-of-order execution techniques, which enable many independent instructions to be executed simultaneously, producing in-

```
__m256  _mm256_set1_ps (float a);
__m256  _mm256_add_ps (__m256 a, __m256 b);
__m256  _mm256_mul_ps (__m256 a, __m256 b);
__m256d  _mm256_add_pd (__m256d a, __m256d b);
__m256  _mm256_fmadd_ps (__m256 a, __m256 b, __m256 c);
__m256d  _mm256_shuffle_pd (__m256d a, __m256d b, const int imm8);
__m256d  _mm256_permute4x64_pd (__m256d a, const int imm8);
__m256 _mm256_broadcast_ss (float const * mem_addr);
void  _mm256_store_ps (float * mem_addr, __m256 a);
```

Figure 2. Example AVX intrinsic function prototypes.

struction level parallelism. Therefore, instruction execution units inside the processor are better utilized if the unrolled statements are independent of each other. However, because loop unrolling increases the size of the program, the code may become larger than the L1 code cache, which could degrade performance due to cache misses. Moreover, the unrolled code may be more difficult for the programmer to understand.

4.3 OpenMP

The OpenMP API for shared memory architectures allows the programmer to easily develop a parallel application for multiprocessors and multi-core machines. OpenMP directives allow a program to create multiple concurrent threads that are assigned to different processing cores through the runtime environment. It allows the data to be split between processing cores to encourage data parallelism, with each core processing only its part of the data. OpenMP also provides an easy way for the programmer to create different tasks executed by different threads, thereby supporting task parallelism.

To efficiently apply loop tiling, loop unrolling, and AVX intrinsics in OpenMP, there are several considerations that must be kept in mind. The first is that the data must be kept in the CPU registers as long as possible, before being moved to higher latency memory (the L1 cache, and then the higher cache). Also, loop tiling must allow at least one larger block of data to be used by every core at the same time. This reduces memory contention between the processing cores, while smaller blocks are accessed by different cores. However, to provide fast access, the larger block must not be bigger than the cache. Finally, unrolling must produce as many independent operations as possible, but should not result in register spilling and movement of the code to the L2 cache.

5. Advanced AVX-Intrinsic based Matrix-Matrix Multiplication

The algorithm for single precision floating-point matrix-matrix multiplication ($c = a \times b$) using AVX intrinsics is shown in Figure 3. The variables *r0*, *r1*, *r2*, *r3*, *a0*, *b0* and *b1* are of type __m256, and will be mapped to the YMM registers inside the CPU. Sixteen consecutive data elements from matrix *b* are preloaded using *_mm256_load_ps*, and kept in variables *b0* and *b1*. The data from matrix *a* are loaded one element at a time before being broadcast to all eight elements of the target AVX register. This broadcast data will be multiplied with the preloaded *b0* and *b1* values, and the products kept in two registers, *r0* and *r1*. To reduce memory accesses, the preloaded data from matrix *b* will be reused and multiplied with the next data element read from the next row of matrix *a* using the same pattern. Two additional registers are required to keep the appended results, *r2* and *r3*. The products will be accumulated in variables *r0* to *r3* before being stored in the destination matrix *c* at the end of the innermost loop. The *VectorSize* parameter is set to 8 because this is the number of 32-bit floating-point operations that the AVX instructions can perform in parallel. The data between *a*[*i*][*k*] and *a*[*i*+1][*k*] are multiplied with the data from matrix *b*, and four results are written into matrix *c*. This approach means that the unrolling factor, defined as U_F, is set to 2 in Figure 3, but if greater performance is required, then this

```
/*the UF, VectorSize and TileSize are set to 2, 8 and 16
respectively. */
__m256 r0, r1, r2, r3, a0, b0, b1;
for (int i = 0; i < n; i += UF){
  for (int kk = 0; kk < n; kk += TileSize){
     for (int j = 0; j < n; j += VectorSize*2){
        r0 = _mm256_setzero_ps();
         r1 = _mm256_setzero_ps();
         r2 = _mm256_setzero_ps();
         r3 = _mm256_setzero_ps();
         for (int k = kk; k< kk + TileSize; k += 2){
           _mm_prefetch((char *)&b[k + PrefetchDistance][j],
_MM_HINT_NTA);
           b0 = _mm256_load_ps(&b[k][j]);
           b1 = _mm256_load_ps(&b[k][j + VectorSize]);
           a0 = _mm256_broadcast_ss(&a[i][k]);
           r0 = _mm256_fmadd_ps(a0, b0, r0);
           r1 = _mm256_fmadd_ps(a0, b1, r1);

           a0 = _mm256_broadcast_ss(&a[i + 1][k]);
           r2 = _mm256_fmadd_ps(a0, b0, r2);
           r3 = _mm256_fmadd_ps(a0, b1, r3);
           }
         b0 = _mm256_load_ps(&c[i][j]);
         b1 = _mm256_load_ps(&c[i][j + VectorSize]);
         r0 = _mm256_add_ps(r0, b0);
         r1 = _mm256_add_ps(r1, b1);

         b0 = _mm256_load_ps(&c[i + 1][j]);
         b1 = _mm256_load_ps(&c[i + 1][j + VectorSize]);
         r2 = _mm256_add_ps(r2, b0);
         r3 = _mm256_add_ps(r3, b1);
         _mm256_store_ps(&c[i][j], r0);
         _mm256_store_ps(&c[i][j + VectorSize], r1);
         _mm256_store_ps(&c[i + 1][j], r2);
         _mm256_store_ps(&c[i + 1][j + VectorSize], r3)
        }
     }
  }
}
```

Figure 3. Matrix-matrix multiplication of size $n \times n$ using AVX intrinsics.

factor could be increased. For example, if U_F is increased to 4, then four data elements from matrix *a* (*a*[*i*][*k*], *a*[*i*+1][*k*], *a*[*i*+2][*k*], *a*[*i*+3][*k*]) are multiplied with the preloaded data from matrix *b*, and eight products are stored in 8 variables, named *r0* to *r7*, before being written to matrix *c*. By increasing the U_F value, the preloaded data from matrix *b* will be utilized repeatedly, thus reducing memory accesses and augmenting performance.

Loop tiling is utilized in this algorithm by splitting the *kk*-loop counter into blocks whose size is specified by the *TileSize* parameter. The appropriate values for *TileSize* and U_F need to be determined for several matrix sizes to achieve maximum performance. A multi-core version of this algorithm is created by applying a "#pragma omp parallel for" statement to the outermost loop, and the list of private variables for each thread must be declared explicitly.

Due to its use of AVX instructions, loop tiling, and unrolling, our proposed algorithm is called "AVX-Tiling" for easier reference in the rest of this paper.

6. Strassen's Algorithm and the AVX Instruction Set

The computation complexity for the conventional matrix-matrix multiplication of matrices of size $n \times n$ is $O(n^3)$, and Strassen algorithm reduces this to $O(n^{2.8074})$ (Stothers, 2010). Many algorithms, based on Coppersmith-Winograd's method (1990), have been proposed to achieve better performance. For example, Davie and Stothers reduces the complexity of Coppersmith-Winograd from $O(n^{2.375477})$ to $O(n^{2.3736897})$ (2013), and Le Gall's method is $O(n^{2.3728639})$ (2014). However, unlike Strassen, the Coppersmith-Winograd based algorithms are rarely used in practice because they are difficult to implement (Le Gall, 2012).

The Strassen algorithm splits each input matrix into 4 chunks, each one fourth of the original's size. The addition, subtraction, and multiplication operations are applied to these sub-matrices, as shown in Figure 4, and the result is kept in an output matrix *c*. If multiplication operation between these sub-matrices is required, then they are recursively split again using the same mechanism until a stop condition is reached. In theory, this condition is determined by the smallest size that the multiplication routine can support, but usually it is unnecessary to use Strassen down to this level. Instead, the programmer must determine what number of recursive levels gives the best performance for their implementation. At each level, it is necessary to allocate memory for 21 temporary matrices, i.e. M_1, M_{1A}, M_{1B}, M_2, M_{2A}, B_{11}, M_3, M_{3B}, A_{11}, M_4, M_{4B}, A_{22}, M_5, M_{5A}, B_{22}, M_6, M_{6A}, M_{6B}, M_7, M_{7A}, and M_{7B}, all of which are the same size. After the matrix *c* at the current level is complete, memory deallocation of these matrices prevents memory overflow.

Figure 5 shows pseudo code for implementing Strassen's algorithm using AVX intrinsics. To create the sub-matrices at each level of recursion, two steps are required, and both use OpenMP to enable parallel operations to take place. The first step creates sub-matrices that use data from matrix *a*, and the second creates sub-matrices from matrix *b*. Figure 6 shows the pseudo code for creating A_{11}, A_{22}, and all of M_{xA}, and this pseudo code can also be applied to creating B_{11}, B_{22}, and all of M_{xB}. The algorithm passes its sub-matrices through 7 recursive calls, to produce the output data in *M1-M7*. Then C_{xy} is generated from *M1-M7* using the same programming style as in Figure 6.

7. Experimental Results and Discussions

The performance of single precision floating-point matrix-matrix multiplication was evaluated across several configurations. The test machine was a 2.5GHz Core i7-4710HQ (Haswell microarchitecture) with 4 processing cores, with hyperthreading turned on. The programs were written using Microsoft Visual C++ 2017 and OpenMP. Every test matrix was of size *n*×*n*. Sixteen different values for *n* were investigated, starting from 1024, and incremented by 1024 in each subsequent configuration, until the last value of *n* was 16380. To obtain the GFLOPS performance of each configuration, the RDTSC command (Intel, 1997) was executed before and after each matrix multiplication to measure the number of clock cycles. The *s* variable in equation (1) was obtained by dividing this number of clock cycles by the CPU frequency.

We implemented a simple matrix-matrix multiplication, as shown in Figure 1, and a parallel version was obtained by adding the "pragma omp parallel for" statement, along with the declaration of the local variables that were utilized in each thread. When the compiler's optimization and automatic vectorization were turned off, the obtained performance was as shown in Figure 7. The parallel version is about 2.99 times faster than the serial version, on an average. The best performance of 2.09 GFLOPS came from the parallel version when the matrix size was 1024×1024. After automatic vectorization was turned on, and the optimization option of the compiler set to /O2 for maximum speed, the performance of the simple matrix-matrix multiplication increased dramatically. The best performance was now 42.37 GFLOPS when the matrix size was 2048×2048.

$$\begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix} \times \begin{bmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{bmatrix} = \begin{bmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{bmatrix}$$

$$\begin{aligned}
M_1 &= M_{1A}*M_{1B} = (A_{11}+A_{22})*(B_{11}+B_{22}) \\
M_2 &= M_{2A}*B_{11} = (A_{21}+A_{22})*B_{11} \\
M_3 &= A_{11}*M_{3B} = A_{11}*(B_{12}-B_{22}) \\
M_4 &= A_{22}*M_{4B} = A_{22}*(B_{21}-B_{11}) \\
M_5 &= M_{5A}*B_{22} = (A_{11}+A_{12})*B_{22} \\
M_6 &= M_{6A}*M_{6B} = (A_{21} - A_{11})*(B_{11}+B_{12}) \\
M_7 &= M_{7A}*M_{7B} = (A_{12}-A_{22})*(B_{21}+B_{22}) \\
C_{11} &= M_1+M_4-M_5+M_7 \\
C_{12} &= M_3+M_5 \\
C_{21} &= M_2+M_4 \\
C_{22} &= M_1-M_2+M_3+M_6
\end{aligned}$$

Figure 4. The Strassen algorithm.

```
void strassen(int n, float *a, float *b, float *c)
{
  if (n == stop_condition)
    AVX_Tiling(a, b, c)
  else {
    allocate memory for 21 matrices
    create $A_{11}$, $A_{22}$, and all of $M_{xA}$
    create $B_{11}$, $B_{22}$, and all of $M_{xB}$
    clear all data in {$M_1$,…, $M_7$}
    strassen(n/2, $M_{1A}$, $M_{1B}$ , $M_1$) ⎫
    strassen(n/2, $M_{2A}$, $B_{11}$ , $M_2$) ⎬create $M_1$...$M_7$
    …                                          ⎪
    strassen(n/2, $M_{7A}$, $M_{7B}$ , $M_7$) ⎭
    generate all of $C_{xy}$ in matrix c using {$M_1$,…, $M_7$}
  }
}
```

Figure 5. Our proposed AVX based Strassen algorithm.

```
__m256 a11, a12, a21, a22, m1, m2, m5, m6, m7;
#pragma omp parallel for private( j, a11, a12, a21, a22, m1, m2, m5, m6, m7)
for (i = 0; i< n/2; i += 8) {
  for (j = 0; j< n/2; j += Vector_Size){
    a11 and a12 are loaded from a[i][j] and a[i][j+n/2] respectively
    a21 and a22 are loaded from a[i+n/2][j] and a[i+n/2][j+n/2] respectively
    create m1, m2, m5, m6, m7 from {a11, a12, a21, a22} using AVX arithmetic
instructions
    store a11, a22, and m1 to $A_{11}$[i][j], $A_{22}$[i][j], and $M_{1A}$[i][j] respectively
    store m2, m5,m6 and m7 to $M_{2A}$[i][j], $M_{5A}$[i][j], $M_{6A}$[i][j] and $M_{7A}$[i][j]
respectively
    }
}
```

Figure 6. Create A_{11}, A_{22}, and the M_{xA} sub-matrices.

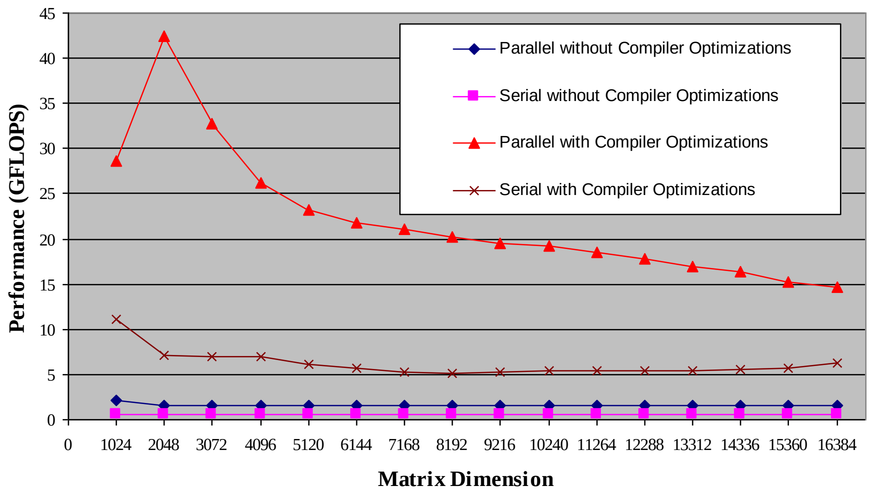

Figure 7. Performance of simple matrix-matrix multiplication.

There are two things to note. The first is that utilizing OpenMP speeds up matrix-matrix multiplication quite efficiently. Although the average speedup factor (3.6) is less than the number of processing cores (4), some cases attained speedup factors higher than the number of cores. For example, when the matrix dimension was 2048, 3072, or 8168, the speedups were 5.97, 4.73, or 4.04 respectively. Secondly, the performances of the optimizations and the automatic vectorization are excellent when we compare the results to the parallel versions of matrix-matrix multiplication. As a consequence, all the implementations in the rest of this section use compiler's optimizations.

Our AVX-Tiling algorithm employs several parameters which can be adjusted to obtain better performance. To make AVX-Tiling more suitable for the recursion in Strassen's algorithm, the unrolling factor U_F should be a power of two. As a consequence, our algorithm employed four different U_F values – 2, 4, 8, and 16 – combined with four different *TileSize* values – 8, 16, 32, and 64 – resulting in 16 combinations. The best {U_F, *TileSize*} pair was {4, 16}, and its performance is shown in Figure 8. The best performance was 107.39 GFLOPS when the matrix size was 2048×2048, but the performance tends to decrease as the matrix size gets larger. The lowest performance, as shown in Figure 8, was 34.14 GFLOPS.

Why did a U_F value of 4 give better results than 2, 8, or 16? Disassembly of the compiled program showed that if U_F was set to 2 or 4, then the required number of the accumulate registers did not exceed the number of available AVX registers, which were 4 and 8 respectively. The total number of independent calculations using the FMA instruction should be as large as possible to provide a maximum throughput of the FMA engine inside the AVX unit. Therefore setting U_F to 4 gave better performance than 2, but setting the U_F to 8 or 16 required 16 or 32 accumulated registers respectively. This combined with the three registers needed to store the preloaded value from matrix *b* and the broadcast value from matrix *a*, exceeded the number of hardware registers. This caused the compiler to use register spilling, which resulted in reduced performance.

Two more related questions are: 1) what was the reason for the decrease of performance when matrix size was larger, and 2) why was maximum performance obtained for a matrix size of 2048×2048?

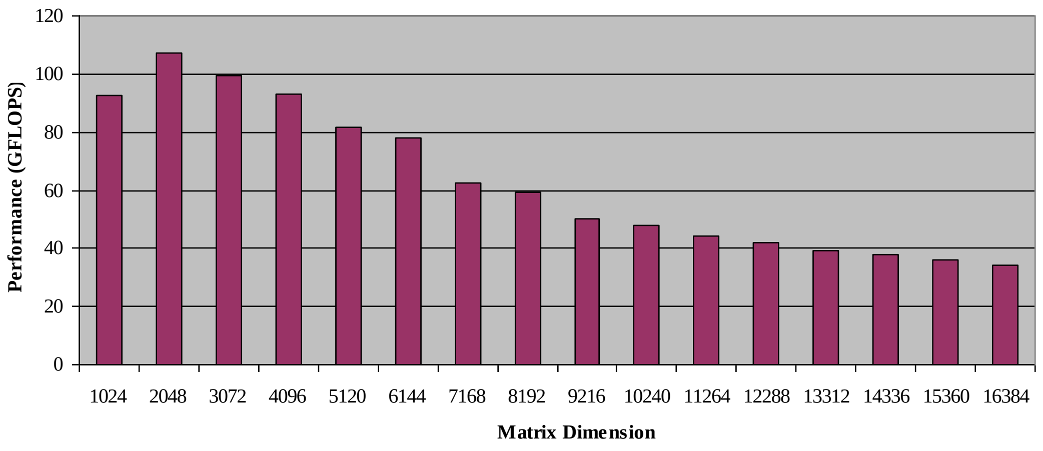

Figure 8. Performance of our AVX-Tiling matrix-matrix multiplication when the {U_F, *TileSize*} was set to {4, 16}.

By profiling the program, we found that OpenMP parallelizes the *i*-loop of the AVX-Tiling algorithm among the processing cores. Let *A*, *B*, and *C* be the tiles of data for the matrices *a*, *b*, *c* in the *j*-loop, as shown in Figure 9. The total number of bytes in the *A* tile can be calculated from U_F**TileSize**sizeof(float), and the total number of bytes in the *B* tile can be determined using *TileSize***n**sizeof(float). The number of bytes in the *C* tile is obtained from U_F **n**sizeof(float). When the *TileSize* is set to 16, the size of the *A* tile is 256 bytes, and is not dependent on the size of the source matrices, but the size of the *B* and *C* tiles does depend on the size. Table 1 shows the memory required for the *B* and *C* tiles for different matrix sizes. Since hyperthreading was turned on, each processing core is responsible for 2 threads at a time, so each thread in the same processing core will read the same data from the *B* tile. However, two threads requires two *A* tiles and two *C* tiles. As mentioned before, the size of the *A* tile is quite small, so reading two separate *A* tiles for each thread is not a problem because both tiles can be stored inside the core's L1 cache. However, the size of the *B* and *C* tiles depends on the size of the destination matrix, so it is impossible to keep two *C* tiles for both threads, and a *B* tile in the L1 cache simultaneously. As shown in Table 1, when the matrix size is less than 2048×2048, all the tiles from matrices *b* and *c* can be kept inside the L2 cache, which is 256 KB. This size restriction is the main reason why our AVX-Tiling algorithm gives the best result for a matrix size of 2048×2048, when the *TileSize* is set to 16.

Once the best values for U_F and *TileSize* had been determined, AVX-Tiling algorithm was utilized as a multiplication kernel for Strassen's algorithm. Our implementation of Strassen's algorithm using the AVX, as outlined in Figures 4 and 5, relies on the creation of sub-matrices at each level of recursion, along with SIMD addition/subtraction functions, and multiplication kernels. The algorithm divides the matrix into four parts at each level of recursion, stopping when the level reaches the stop condition. Then the multiplication kernel is utilized to obtain a result for that level. Four different recursive levels were tested, and the results are shown in Figure 10 and Table 2.

Table 2 shows the performance of our implementation of Strassen's algorithm. Four different recursive levels were tested for each matrix size, and performance was obtained in GFLOPS. The level of recursion that gave the best performance for each matrix size is shown in bold italics in the respective row of the table. For a matrix of size 5120×5120 or smaller, the best performance was obtained with one level of recursion. For matrix sizes of 6144×6144 to 10240×10240, our algorithm gave best results at two levels of recursion. However, for very large matrices, of sizes larger than 10240×10240, three levels of recursion gave the optimum performance.

Why do different matrix sizes need different levels of recursion to attain peak performance, as shown in Figure 10 and Table 2? We calculated the size of the sub-matrices at each recursive level of the tested matrix sizes, and the results are shown in Table 3. We found that almost all the sub-matrices that deliver maximum performance are of size 2048×2048 or smaller. This result conforms with what happened when we used the AVX-Tiling algorithm alone, as shown in Table 1. However, another sub-matrix size, 2560×2560, delivered the best performance. After calculating

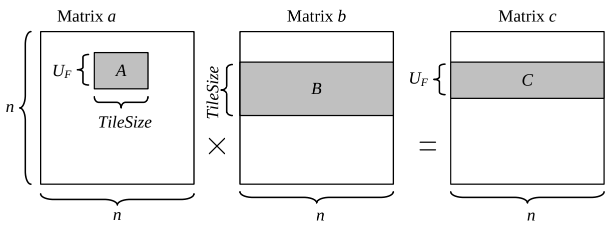

Figure 9. Tiles for the source and destination matrices in the *j*-loop of the AVX-Tiling algorithm.

Table 1. Memory requirements for each tile, and total memory required for (*B*+2**C*) when the {U_F, *TileSize*} is {4, 16}.

Matrix size	Memory required for each tile (kilobytes)		
	B	*C*	*B*+2**C*
1024×1024	64	16	96
2048×2048	128	32	192
3072×3072	192	48	288
4096×4096	256	64	384

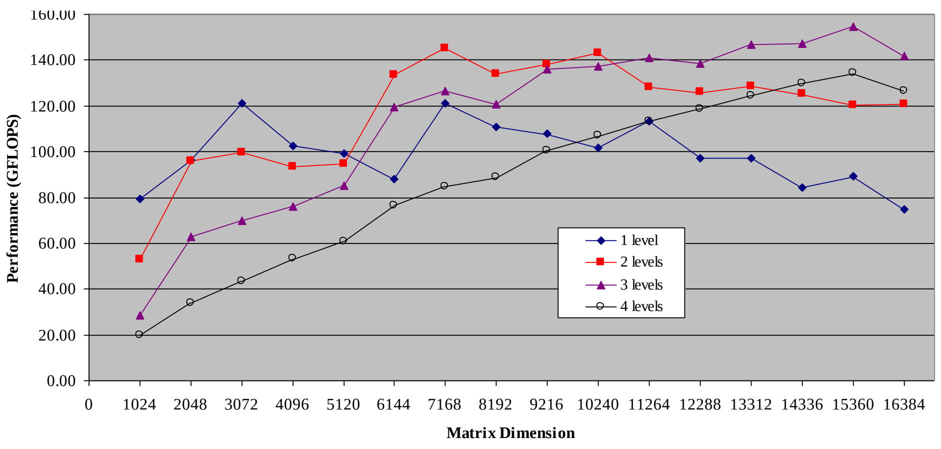

Figure 10. Performance of Strassen matrix-matrix multiplication for different levels of recursion.

Table 2. Performance of matrix-matrix multiplication using Strassen algorithm.

Matrix size	Performance (GFLOPS) at different levels of recursion			
	1 level	2 levels	3 levels	4 levels
1024×1024	*79.29*	53.06	28.34	20.00
2048×2048	*96.21*	95.89	63.03	33.87
3072×3072	*121.25*	99.78	69.74	43.54
4096×4096	*102.51*	93.64	75.92	53.15
5120×5120	*99.13*	94.87	85.21	60.72
6144×6144	88.15	*133.48*	119.54	76.49
7168×7168	121.14	*145.02*	126.34	84.64
8192×8192	110.88	*133.99*	120.81	88.83
9216×9216	107.72	*138.29*	136.12	100.43
10240×10240	101.54	*143.20*	137.33	106.98
11264×11264	113.45	128.18	*141.06*	113.34
12288×12288	97.01	126.24	*138.45*	118.65
13312×13312	97.05	128.69	*146.82*	124.58
14336×14336	84.16	125.24	*147.07*	129.85
15360×15360	89.40	120.48	*154.62*	134.29
16384×16384	74.77	120.62	*141.80*	126.67

Table 3. Sizes of the sub-matrices at different recursive levels.

Matrix size	Sizes of the sub-matrices to be calculated with the AVX-Tiling algorithm at different recursive levels			
	1 level	2 levels	3 levels	4 levels
1024×1024	*512×512*	256×256	128×128	64×64
2048×2048	*1024×1024*	512×512	256×256	128×128
3072×3072	*1536×1536*	768×768	384×384	192×192
4096×4096	*2048×2048*	1024×1024	512×512	256×256
5120×5120	*2560×2560*	1280×1280	640×640	320×320
6144×6144	3072×3072	*1536×1536*	768×768	384×384
7168×7168	3584×3584	*1792×1792*	896×896	448×448
8192×8192	4096×4096	*2048×2048*	1024×1024	512×512
9216×9216	4608×4608	*2304×2304*	1152×1152	576×576
10240×10240	5120×5120	*2560×2560*	1280×1280	640×640
11264×11264	5632×5632	2816×2816	*1408×1408*	704×704
12288×12288	6144×6144	3072×3072	*1536×1536*	768×768
13312×13312	6656×6656	3328×3328	*1664×1664*	832×832
14336×14336	7168×7168	3584×3584	*1792×1792*	896×896
15360×15360	7680×7680	3840×3840	*1920×1920*	960×960
16384×16384	8192×8192	4096×4096	*2048×2048*	1024×1024

the required memory for the *B* and *C* tiles, we found that the total memory required for (*B*+2**C*) equaled 240 KB, which was smaller than the L2 cache size of our machine.

These results highlight two important considerations: the first is that when the matrix size is small (≤ 2048), the use of AVX-Tiling alone (see Figure 8 and Figure 10) gives better performance at all recursive levels compared to Strassen+AVX-Tiling. The second point is that when the matrix size becomes larger, Strassen+AVX-Tiling gives better performance, but the number of recursive levels needs to be determined based on the size of the matrix and the size of the L2 cache.

To give our algorithm the flexibility to be implemented on processors with different sizes of L2 cache, we propose an algorithm that obtains the optimal recursive level value for the Strassen algorithm when utilizing AVX-Tiling as a multiplication kernel:

$$L = \arg\min_{L\geq 0} \left\{ C - \frac{\sqrt{n}}{2^L}(4TileSize + 8U_F) : C \geq \frac{\sqrt{n}}{2^L}(4TileSize + 8U_F) \right\}, \tag{2}$$

where *C* is the size of the L2 cache and *n* is the size of the destination matrix.

If *L* from equation (2) equals 0, then AVX-Tiling should be applied to the input matrix directly. However, when *L* is greater than zero, Strassen+AVX-Tiling is utilized with *L* recursive call levels.

We compared the performance of our algorithm utilizing equation (2) with the two latest versions of the open-source libraries, Eigen (version 3.3.7) and OpenBLAS (version 0.3.5). These libraries were compiled on our test machine, with /O2 optimization, multithreaded support, and other optimizations for the Haswell microarchitecture, and Figure 11 shows their GFLOPS performance. Our algorithm

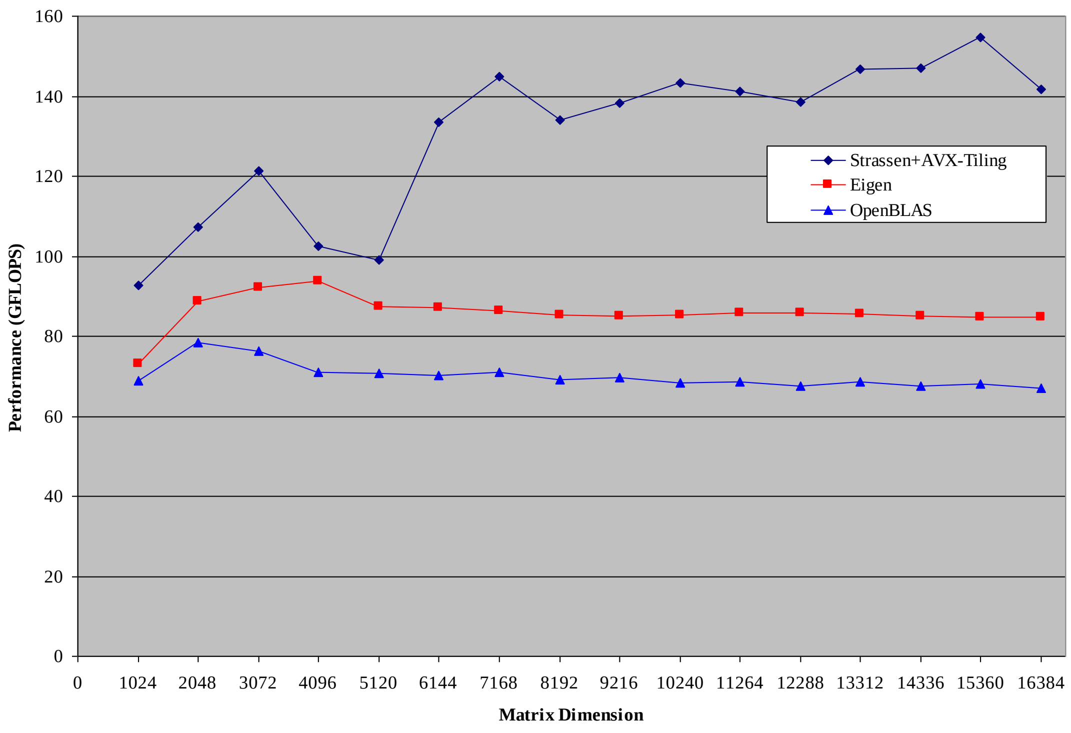

Figure 11. Performance comparison of our AVX-based Strassen algorithm versus the Eigen and the OpenBLAS libraries.

is, on average, 1.52 and 1.87 times faster than Eigen and OpenBLAS respectively.

8. Conclusions

We have proposed an effective implementation of Strassen's algorithm for matrix-matrix multiplication that utilizes AVX instructions and OpenMP on a multi-core architecture. The results show that our algorithm is, on average, 1.52 and 1.87 times faster than the latest versions of the Eigen and the OpenBLAS libraries, respectively. In addition, the performance of our algorithm increases as the matrix size becomes larger, while the performance of both the open-source libraries remains flat.

Even though Strassen requires less multiplications compared to conventional matrix multiplication, it does require additional additions and subtractions, and comes with the overhead of memory copying between the current source matrix and its sub-matrices. Therefore, too many recursive calls in Strassen's algorithm may reduce performance. We have proposed a systematic method for determining an optimum number of recursive levels to obtain the best performance for each matrix size. It ensures that sub-matrix size does not exceed the size of the L2 cache.

Acknowledgements

This work was supported by the Higher Education Research Promotion and the Thailand's Education Hub for Southern Region of ASEAN Countries Project Office of the Higher Education Commission. The authors are grateful to Dr. Andrew Davison for his kind help in polishing the language of this paper.

References

Advanced Micro Devices, Inc. (2019). AMD64 architecture programmer's manual volume 4: 128-Bit and 256-Bit media instructions. Retrieved from https://www.amd.com/system/files/TechDocs/26568.pdf.

Al Hasib, A., Cebrian, J. M., & Natvig, L. (2018). A vectorized k-means algorithm for compressed datasets: Design and experimental analysis. *Journal of Supercomputing*, *74*(6), 2705-2728. doi:10.1007/s11227-018-2310-0

Bramas, B. (2017). A novel hybrid quicksort algorithm vectorized using AVX-512 on intel skylake. *International Journal of Advanced Computer Science and Applications*, *8*(10), 337-344. doi:10.14569/IJACSA.2017.081044

Bramas, B., & Kus, P. (2018). Computing the sparse matrix vector product using block-based kernels without zero padding on processors with AVX-512 instructions. *PEERJ Computer Science*, e151. doi:10.7717/peerj-cs.151

Barash, L. Y., Guskova, M. S., & Shchur, L. N. (2017). Employing AVX vectorization to improve the performance of random number generators. *Programming and Computer Software*, *43*(3), 145-160. doi:10.1134/S0361768817030033

Coppersmith, D., & Winograd, S. (1990). Matrix Multiplication via Arithmetic Progressions. *Journal of Symbolic Computation*, *9*(3), 251-280. doi:10.1016/S0747-7171(08)80013-2

Davie, A. M., & Stothers, A. J. (2013). Improved bound for complexity of matrix multiplication. *Proceedings of* *the Royal Society of Edinburgh*, 351-369. doi:10. 1017/S0308210511001648

Hassana, S. A., Hemeida, A. M., & Mahmoud, M. M. (2016). Performance evaluation of matrix-matrix multiplications using Intel's advanced vector extensions (AVX). *Microprocessors and Microsystems*, *47*(SI), 369-374. doi:10.1016/ j.micpro.2016. 10.002

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. (1997). Using the RDTSC instruction for performance monitoring. Retrieved from http:// developer.intel.com/drg/pentiumII/appnotes/RDTSC PM1.HTM.

Intel Corporation. (2011). Intel® 64 and IA-32 Architectures Software Developer's Manual. Retrieved from https: //software.intel.com/sites/default/files/managed/ 39/ c5/325462-sdm-vol-1-2abcd-3abcd.pdf.

Intel Corporation. (2019). Intel® 64 and IA-32 Architectures Software Developer's Manual Volume 1: Basic Architecture. Retrieved from https://software.intel. com/ sites/default/files/managed/a4/60/253665-sdm-vol-1.pdf.

Kye, H., Lee, S. H., & Lee, J. (2018). CPU-based real-time maximum intensity projection via fast matrix transposition using parallelization operations with AVX instruction set. *Multimedia Tools and Applications*, *77*(12), 15971-15994. doi:10.1007/s11042-017-5171-2

Le Gall, F. (2012). Faster algorithms for rectangular matrix multiplication. *Proceedings of the 53rd Annual IEEE Symposium on Foundations of Computer Science*, 514-523. doi:10.1109/FOCS.2012.80

Le Gall, F. (2014). Powers of tensors and fast matrix multiplication. *Proceedings of the 39th International Symposium on Symbolic and Algebraic Computation*, 296-303. doi:10.1145/2608628.2608664

Mitra, G., Johnston, B., Rendell, A. P., McCreath, E., & Zhou, J. (2013). Use of SIMD vector operations to accelerate application code performance on low-powered ARM and Intel platforms. *IEEE 27th International Parallel and Distributed Processing Symposium Workshops and PhD Forum*, 1107-1116. doi:10.1109/IPDPSW.2013.207

Stothers, A. J. (2010). *On the Complexity of Matrix Multiplication* (Doctoral thesis, University of Edinburgh, Edinburgh, Scotland). Retrieved from https://www. era.lib.ed.ac.uk/bitstream/handle/1842/4734/Stothers 2010.pdf.