การปรับปรุงความเร็วการคูณเมทริกซ์ด้วยเมทริกซ์โดยใช้เทคนิคการปูกระเบื้องแบบ 2 มิติ และชุดคำสั่ง AVX512 สำหรับสถาปัตยกรรมมัลติคอร์

An Improvement of the Matrix-Matrix Multiplication Speed using 2D-Tiling and AVX512 Intrinsics for Multi-Core Architectures

หนู่-ซิน อู[1] และ ปัญญยศ ไชยกาฬ[2*]

Nwe Zin Oo[1] and Panyayot Chaikan[2*]

บทคัดย่อ

การคูณเมทริกซ์ด้วยเมทริกซ์เป็นการดำเนินการทางคณิตศาสตร์ที่ใช้เวลาในการประมวลผลอย่างมากในงานทางวิทยาศาสตร์และวิศวกรรม เมื่อเมทริกซ์มีขนาดใหญ่จะเสียเวลาในการประมวลผลมาก ส่งผลให้ซอฟต์แวร์ทำงานช้าซึ่งเป็นสิ่งที่รับไม่ได้ในโปรแกรมประยุกต์แบบเวลาจริง บทความนี้นำเทคนิคการปูกระเบื้องแบบ 2 มิติ การคลายการวนซ้ำ การเสริมเต็มข้อมูล ตัวชี้แนะโอเพนเอ็มพี และชุดคำสั่งเอวีเอ็กซ์ 512 มาใช้เพิ่มความเร็วในการคูณเมทริกซ์บนสถาปัตยกรรมการประมวลผลแบบหลายแกน ขั้นตอนวิธีที่นำเสนอเมื่อทดสอบบนเครื่อง Core i9-7900X พบว่ามีความเร็วมากกว่า 2 เท่าเมื่อเทียบกับการดำเนินการที่ใช้คลังโปรแกรมโอเพนบลาสและไอเกนสำหรับการประมวลผลเมทริกซ์แบบทศนิยมจุดลอยตัวทั้งชนิดความเที่ยงเท่าเดียวและความเที่ยงสองเท่า นอกจากนี้ยังได้มีการนำเสนอสมการสำหรับใช้ปรับค่าพารามิเตอร์เพื่อให้สามารถนำขั้นตอนวิธีที่นำเสนอไปประมวลผลบนเมทริกซ์ขนาดใด ๆ บนตัวประมวลผลอื่น ที่มีรูปแบบของหน่วยความจำแคชแตกต่างกัน

คำสำคัญ เอวีเอ็กซ์ 512 โอเพนเอ็มพี การปูกระเบื้องแบบสองมิติ การประมวลผลหลายแกน ไอเกน โอเพนบลาส

Abstract

Matrix-matrix multiplication is a time-consuming operation in scientific and engineering applications. When the matrix size is large, it will take a lot of computation time, resulting in slow software

[1] นักศึกษาปริญญาเอก สาขาวิชาวิศวกรรมคอมพิวเตอร์ คณะวิศวกรรมศาสตร์ มหาวิทยาลัยสงขลานครินทร์ สงขลา 90112

[2] ผศ, ดร., สาขาวิชาวิศวกรรมคอมพิวเตอร์ คณะวิศวกรรมศาสตร์ มหาวิทยาลัยสงขลานครินทร์ สงขลา 90112

[1] Ph.D. Student, Department of Computer Engineering, Faculty of Engineering, Prince of Songkla University, Songkhla, 90112, Thailand

[2] Asst. Prof., Dr., Department of Computer Engineering, Faculty of Engineering, Prince of Songkla University, Songkhla, 90112, Thailand

[*] Corresponding author: E-mail address: panyayot.c@psu.ac.th. Tel.: 074-287350

(Received: September 22, 2020; Revised: April 26, 2021; Accepted: April 27, 2021)

which is unacceptable in real-time applications. In this paper, 2D-tiling, loop unrolling, data padding, OpenMP directives, and AVX512 intrinsics are utilized to increase the speed of matrix-matrix multiplication on multi-core architectures. Our algorithm, tested on a Core i9-7900X machine, is more than two times faster than the operations offered by the OpenBLAS and Eigen libraries for single and double precision floating-point matrices. We also propose an equation for parameter tuning which allows our algorithm to be adapted to process any size of matrix on CPUs with different cache organizations.

Keywords: AVX512, OpenMP, 2D-tiling, Multicore Processing, Eigen, OpenBLAS

Introduction

Matrix-matrix multiplication is a time-consuming operation which can be addressed in several ways. The hardware can be improved by switching to a faster CPU or utilizing an FPGA [1], or software modification can be utilized alone, or some combinations of the two. For example, the utilization of a GPU and its APIs can speed up processing [2] or distributing it among various machines [3]. In this paper, we choose the second approach (software alone) because it is cheaper than the others, which is an important consideration in our environment.

When a standard multiplication algorithm is utilized, as shown in Figure 1(a), the performance can be very poor because most compilers cannot automatically parallelize this code efficiently among the cores. Programmers avoid this problem by calling ready-made functions provided in standard arithmetic libraries such as the Eigen [4] and OpenBLAS [5]. The latest versions of these libraries support AVX512 instructions [4], [5], but their performance decreases when the matrix size is large due to memory bottlenecks. This is illustrated in Figure 2 which shows a block diagram for a modern x86 processor. Each core includes a dedicated small L1 data cache which allows it to run as quickly as possible. The L2 cache is also dedicated to each core but is larger, and the L3 cache is shared between all the cores. Cache latency increases with level, for example, the latencies of the L1, L2, and L3 caches for the Intel Skylake microarchitecture are 4, 12, and 42 clock cycles respectively [6]. The performance of multiplication utilizing these libraries is highest when all the matrices are smaller than the L1 cache size, but crucially when the total space required for both the sources and destination matrices is larger than the L3 cache, the performance is considerably reduced due to memory bottlenecks.

In this paper, we utilize loop tiling to solve this problem, and employ AVX512 intrinsics to further augment the speed of single and double precision floating-point multiplication. We have focused on the AVX512 because tenth generation Intel processors support it, especially low price CPUs such as the Core i3-1000G [7]. Loop unrolling is also utilized to increase the instruction level parallelism in each processing core.

Background and Related Work

Figure 1(a) shows the standard sequential algorithm for matrix-matrix multiplication, defined as $C = C+A\times B$. It requires $2n^3$ arithmetic operations when A, B, and C are square matrices of size $n\times n$. Each result in C, defined as $C[i, j]$, can be expressed as shown in Equation (1).

$$C[i, j] = C[i, j] + \sum_{k=0}^{n-1} A[i, k] * B[k, j] \tag{1}$$

Figure 1(b) is a modification of the code in Figure 1(a). The loop counter k is divided by the constant T_1, resulting in a division of matrices A and B into the tiles of size $1\times T_1$ and $T_1\times 1$ respectively. Each result of the modified code, defined as $C[i, j]$, can be expressed as shown in Equation (2).

$$C[i, j] = C[i, j] + \sum_{k=0}^{n/T_1-1} \sum_{K=kT_1}^{kT_1+T_1-1} \left(A[i, K] * B[K, j] \right) \tag{2}$$

Equation (2) gives the same result as of Equation (1). The speed of the code in Figures 1(a) and 1(b) are not significantly different. However, if the k-loop is moved from its position and then be placed between the j-loop and the i-loop, as shown in Figure 1(c), then the performance will be improved. The B tile is reused for each increasing of the i variable, when it fits inside the cache, the main memory access for the B tile is considerably reduced.

The performance is further improved by applying 2D-tiling, as shown in Figure 1(d). The loop counter i is divided by the constant T_2, resulting in a division of matrices A, B, and C into the tiles of size $T_2\times T_1$, $T_1\times 1$, and $T_2\times 1$ respectively. When the C tile is loaded, its adjacent data is also read into the cache due to the cache mechanism. Let L be the cache line size, total memory of $T_2\times L$ is read and then be placed into the cache for the C tile of size $T_2\times 1$. This adjacent data of the C tile is reused for each increasing of the k variable, thereby increasing performance.

```
for (int j=0; j<n; j++)
 for (int i=0; i<n; i++)
  for (int k=0; k<n; k++)
   C[i][j] += A[i][k]*B[k][j];
```
(a)

```
for (int j=0; j<n; j++)
 for (int i=0; i<n; i++)
  for (int k=0; k<n/T1; k+=T1)
   for (int K=k; K<kT1; K++)
    C[i][j] += A[i][K]*B[K][j];
```
(b)

```
for (int j=0; j<n; j++)
 for (int k=0; k<n/T1; k+=T1)
  for (int i=0; i<n; i++)
   for (int K=k; K<kT1; K++)
    C[i][j] += A[i][K]*B[K][j];
```
(c)

```
for (int j=0; j<n; j++)
 for (int i=0; i<n/T2; i+=T2)
  for (int k=0; k<n/T1; k+=T1)
   for (int I=i; I<iT2; I++)
    for (int K=k; K<kT1; K++)
     C[i][j] += A[i][K]*B[K][j];
```
(d)

```
int i, j, k, I, K;
#pragma omp parallel for (private i, k, I, K)
for (int j=0; j<n; j++)
 for (int i=0; i<n/T2; i+=T2)
  for (int k=0; k<n/T1; k+=T1)
   for (int I=i; I<iT2; I++)
    for (int K=k; K<kT1; K++)
     C[i][j] += A[i][K]*B[K][j];
```
(e)

```
__m512 a_line, b1_line, b2_line;
__m512 r1_line, r2_line, c1_line, c2_line;

#pragma omp parallel for private(a_line, b1_line, b2_line,
                r1_line, r2_line, c1_line, c2_line)
for (int j = 0; j < n; j += (AVX512_SZ * 2)) {
 for (int i = 0; i < n; i += T2) {
  for (int k = 0; k < n; k += T1) {
   for (int I = i; I < i + T2; I++) {
    r1_line = _mm512_setzero_ps();
    r2_line = _mm512_setzero_ps();
    for (int K = k; K < k + T1; K++) {
     a_line = _mm512_set1_ps(A[I][K]);
     b1_line = _mm512_load_ps(&B[K][j]);
     b2_line = _mm512_load_ps(&B[K][j + AVX512_SZ]);
     r1_line = _mm512_fmadd_ps(a_line, b1_line, r1_line);
     r2_line = _mm512_fmadd_ps(a_line, b2_line, r2_line);
    }
    c1_line = _mm512_load_ps(&C[I][j]);
    c2_line = _mm512_load_ps(&C[I][j + AVX512_SZ]);
    r1_line = _mm512_add_ps(c1_line, r1_line);
    r2_line = _mm512_add_ps(c2_line, r2_line);
    _mm512_store_ps(&C[I][j], r1_line);
    _mm512_store_ps(&C[I][j + AVX512_SZ], r2_line);
   }}}}
```
(f)

```
__m512 a_line, b1_line;                    //two AVX512 variables for keeping the data from matrices A and B
__m512 r1_line, c1_line;                   //two AVX512 variables for keeping the multiplication results
#pragma omp parallel for private(a_line, b1_line,
                r1_line, c1_line)          //OpenMP directive call and its local variable declaration
for (int j = 0; j < n; j += AVX512_SZ) {   //the c tile is of size T2×AVX512_SZ
 for (int i = 0; i < n; i += T2) {         //the b tile is of size T1×AVX512_SZ
  for (int k = 0; k < n; k += T1) {        //the a tile is of size T2×T1
   for (int I = i; I < i + T2; I++) {      //
    r1_line = _mm512_setzero_ps();         //r1_line[x] = 0 for all x ∈ {0, 1, 2, .. 15}
    for (int K = k; K < k + T1; K++) {     //
     a_line = _mm512_set1_ps(A[I][K]);     //a_line[x] = A[I][K] for all x ∈ {0, 1, 2, .. 15}
     b1_line = _mm512_load_ps(&B[K][j]);   //read 16 data from the B matrix, starting at location [K][j]
     r1_line = _mm512_fmadd_ps(a_line, b1_line, r1_line);
                                           //r1_line[x] = a_line[x] * b_line[x] + r1_line[x] for all x ∈ {0, 1, 2, .. 15}
    }
    c1_line = _mm512_load_ps(&C[I][j]);    //read the previous values of 16 data, starting at [I][j] in the C matrix.
    r1_line = _mm512_add_ps(c1_line, r1_line);
                                           //
          _mm512_store_ps(&C[I][j], r1_line);
                                           //write 16 results into memory, starting at [I][j] in the C matrix.
   }}}}
```
(g)

Figure 1. (a) Standard sequential matrix-matrix multiplication. (b) Our 1D-tiling matrix-matrix multiplication. (c) Another implementation of 1D-tiling multiplication. (d) Our 2D-tiling matrix-matrix multiplication. (e) Parallel matrix-matrix multiplication utilizing 2D-tiling. (f) Our proposed 2D matrix-matrix multiplication for single precision floating-point matrices with an unrolling factor of two. (g) Our proposed 2D matrix-matrix multiplication with no unrolling for the same type of matrices.

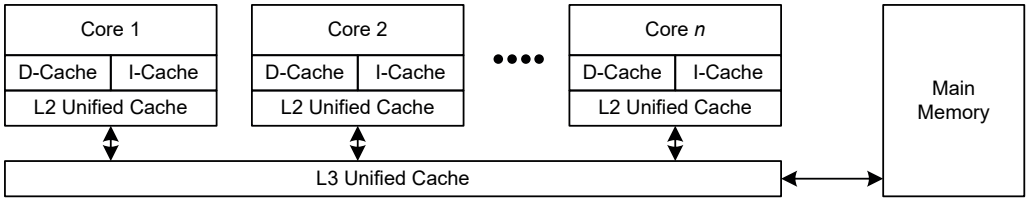


Figure 2. Typical organization of x86 multi-core CPUs.

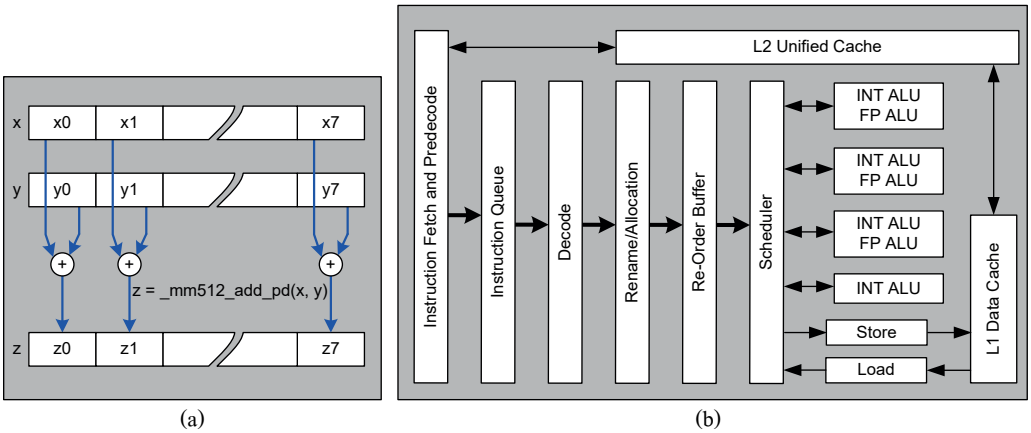


Figure 3. (a) Addition of two single precision floating-point vectors utilizing AVX512 intrinsics. (b) Block diagram of the Intel Skylake microarchitecture.

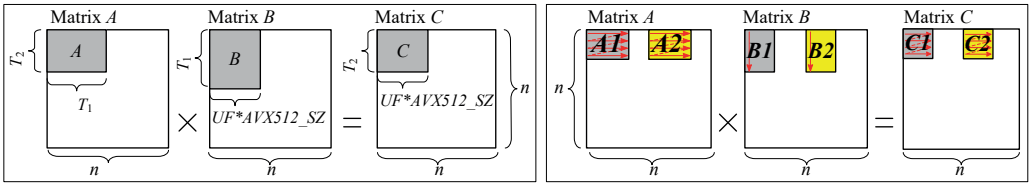
Even though the code in Figures 1(b), 1(c), and 1(d) utilizes different loop counters and loop sequences, they all comply with Equation (2) and produce the same results.

When the code in Figure 1(d) is compiled on a multi-core architecture, it fails to utilize the processing cores efficiently. Typically, an OpenMP directive is added, as in Figure 1(e), which allows multiple threads to be assigned to the cores at runtime. Although this splits the data among the cores, the arithmetic units in each one will not be fully utilized. SIMD instructions and loop unrolling technique can be employed to overcome this problem.

The SIMD instruction set has long been an integral part of most platforms, since it allows multiple data elements to be processed by a single machine instruction, resulting in higher performance. For example, SIMD appears as an extension in the NEON [8] and Altivec [9] architectures for the ARM and PowerPC respectively. For the x86-64, the AVX instruction set extension adds sixteen 256-bit registers supporting both integer and floating-point operations. Its successor, AVX512, doubles the register size to 512 bits, allowing eight double precision or sixteen single precision floating-point values to be processed simultaneously. Figure 3(a) shows the utilization of the *_mm512_add_pd* AVX intrinsic function, which simultaneously adds eight elements from *x* and *y*, producing eight results for the *z* variable.

Many researchers have described how AVX can augment processing performance. For example, optimized QcBits key encapsulation with AVX512 [10], boosted the efficiency of 2D shallow water equations calculations [11], reservoir simulations [12], and strong string dictionary compression [13].

Along with SIMD, loop unrolling is an important programming technique for further increasing instruction level parallelism. When each core contains multiple ALUs, as shown in Figure 3(b), then parallel execution of instructions among those ALUs becomes simpler if complex statements are unrolled so they are more independent of each other.



(a) (b)

Figure 4. (a) Matrix tiles with unrolling in our 2D-tiling multiplication, and (b) an example when there are two threads.

2D-Tiling for Parallel Matrix-matrix Multiplication Utilizing AVX512

Our 2D-tiling method for parallel matrix-matrix multiplication utilizing AVX512 intrinsics is more complex than conventional multiplication or matrix-vector multiplication [14, 15]. Figure 1(g) shows our proposed algorithm.

Instead of obtaining a single result during the last iteration of the *k*-loop, as in Figure 1(a), the source matrices are divided into tiles, and partial results are obtained by applying multiply-accumulate operations to those tiles.

In order to obtain each partial product in the *C* tile, the data elements from the *A* tile are loaded one element at a time, and then broadcast to the AVX512 variable, *a_line*. It is multiplied to the data obtained from the *B* tile, and the result accumulated in the *r1_line* variable. This process is repeated T_1 times, and the partial product in *r1_line* is added to each element of the *C* tile.

To further optimize performance, an unrolling technique increases the degree of instruction level parallelism (ILP).

For an unrolling factor *UF*, the data from the *B* tile is loaded *UF* times in each iteration of the *K*-loop. This means that the *B* tile will be of size $T_1 \times (UF \times AVX512\_SZ)$, as illustrated in Figure 4(a). The *AVX512_SZ* parameter is the amount of data that can be processed simultaneously by an AVX512 instruction, which is 16 and 8 values for single and double precision floating-point data respectively.

Figure 4(b) shows the unrolled version of the code with a *UF* value of two. Since the *B* tile will be reused in the next iteration of the *I*-loop, performance will be increased if we can store all the data in the cache because then the CPU will not have to read it from slow main memory. For the best performance, an appropriate *B* tile size must be determined.

To effectively distribute the workload in a multi-core architecture, an OpenMP directive is employed, as shown in Figures 1(f) and 1(g): "#pragma omp parallel for" parallelizes the *j*-loop among the cores.

Figure 4(b) illustrates how the algorithm works when there are two threads. Thread number 1 multiplies the *A1* tile to *B1* and stores the result in *C1*, while thread number 2 does the same with *A2*, *B2*, and *C2*.

The code in Figures 1(f) and 1(g) only supports single precision floating-point matrices. For double precision floating-point, *_mm512_<operation>_ps* must be changed to *_mm512_<operation>_pd*, and the *AVX512_SZ* parameter must be changed from 16 to 8.

Zero Padding

The L1 cache in modern x86 architectures is dedicated to each core, as shown in Figure 2. Since its latency is small compared to the higher levels cache, keeping the data in L1 as much as possible will improve the overall performance due to the reduction of memory bottlenecks.

Even though our loop tiling splits the source matrices into smaller tiles to fit inside the L1 cache, when the matrix dimension is a multiple of 2 Kbytes, the number of tile data lines stored in the cache is reduced due to the cache mapping. For example, consider when the cache size is 32 KBytes, and its mapping is an 8-way set associative with 64 bytes per cache line. When the single precision matrix size 4096×4096, all the *B* tile lines will be mapped to the same cache set, number 0. This means that the number of tile data lines simultaneously kept inside the L1 cache is reduced to only *W* lines, which is actually the number of cache way. To reduce this effect, zero padding is required.

Let W, L and C_{SZ} be a number of cache way, an L1 cache line size, and a cache size. The data size, D_{SZ}, is set to 4 or 8 bytes for single or double precision floating-point data respectively. For a matrix of size $n \times n$, an optimum zero padding size (in bytes), P_{SZ}, for our multiplication algorithm can be calculated using:

$$P_{SZ} = \arg\max_{P_{SZ}} \sum_{k=0}^{T_i-1} \left\{ \left(\left(\left(n + \frac{P_{SZ}}{D_{SZ}} \right) * D_{SZ} * k \right) \% \left(\frac{C_{SZ}}{W} \right) \right) / L \right\}. \quad (3)$$

Although AVX512 allows unaligned memory addresses to be loaded, performance is better if they are aligned to multiples of 32 bytes, which indicates that the P_{SZ} value should be a multiple of 32. To apply zero padding, five steps are required, as shown in Figure 5.

Step 1: Three new matrices of size $n \times (n+P_{SZ})$ are allocated, producing A_{padded}, B_{padded}, and C_{padded}.
Step 2: The data in the matrices A, B, and C are copied to A_{padded}, B_{padded}, and C_{padded}.
Step 3: The AVX512 algorithm is applied to the padded matrices, with the multiplication result being stored in C_{padded}.
Step 4: C_{padded} is copied back to C matrix.
Step 5: The A_{padded}, B_{padded}, and C_{padded} matrices are deallocated.

Figure 5. Five steps for applying zero padding to our 2D-tiling matrix-matrix multiplication.

Experimental Results

Performance was measured on a Core i9-7900X (Skylake microarchitecture) with ten processing cores, using 64-bit Microsoft Windows 10 with hyper-threading turned on, and the Microsoft Visual Studio 2019 C++ compiler. Compiler optimization was set to /O2 with parallel code generation turned on.

We determined the best parameters for our algorithm by testing three *UF* values (4, 8, 16), combined with four T_2 values (32, 64, 128, 256), and four T_1 values (32, 64, 128), for a total of 36 combinations. The optimal combination of $\{UF, T_2, T_1\}$ was found to be {8, 128, 64}, and all subsequent tests used these settings.

We examined how zero padding affected multiplication efficiency by repeats our tests with and without zero padding. With padding, P_{SZ} for each matrix dimension was calculated using Equation (3), and matrices of size $n \times n$ with ten different n values were examined, starting from n == 1024. The size was incremented in steps of 1024 until n == 10240. The results are shown in Figure 6.

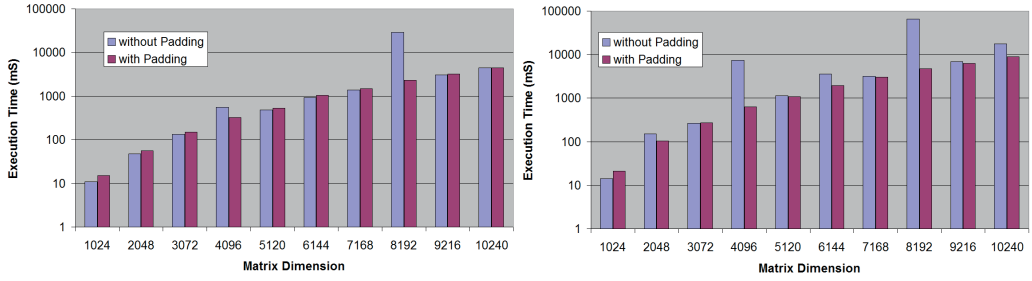


Figure 6. (a) Performance of our single precision AVX512 multiplication with and without zero padding. (b) Performance of our double precision AVX512 multiplication with and without zero padding.

For single precision multiplication, the padded version outperformed the unpadded version only when the matrix dimension were 4096 and 8192. For double precision, the padded version was faster than the unpadded version almost all dimensions, except for 1024 and 3072.

We also compared the performance of our algorithm with the open-source libraries OpenBLAS (version 0.3.10) and Eigen (version 3.3.7). For those tests, zero padding was only applied to single precision multiplication when the matrix dimension were 4096 and 8192. For double precision, zero padding was only utilized when the matrix dimensions were not 1024 or 3072. For single precision matrix multiplication (see Figure 7(a)), our algorithm is on average 2.12 and 2.13 times faster than OpenBLAS and Eigen. For double precision matrix multiplication (see Figure 7(b)), our algorithm is on average 2.06 and 2.17 times faster than OpenBLAS and Eigen.

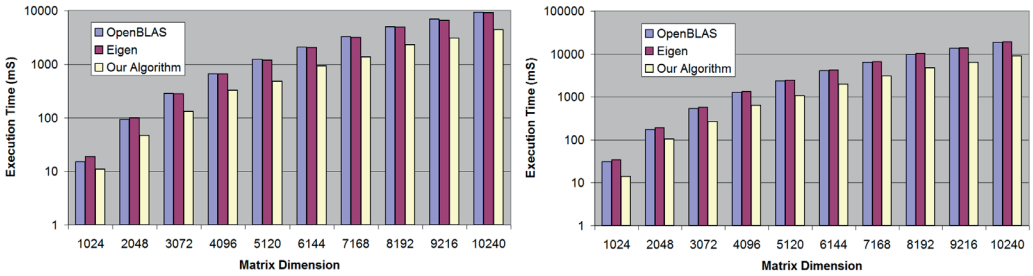


Figure 7. (a) Performance comparison of our single precision AVX512 multiplication versus OpenBLAS and Eigen. (b) Performance comparison of our double precision AVX512 multiplication versus OpenBLAS and Eigen.

Discussions

The results in Figure 6 raise two questions. Why does the padded version of our multiplication outperform unpadded data when the matrix dimension is 4092 or 8192? Secondly, why does the padded version of our double precision multiplication outperform unpadded data for almost every matrix dimension?

To answer these questions, the cache mechanism of the CPU must be analyzed. The L1 cache of the Core i9-7900X is separated into data and instruction caches, which occupy 32 KB per core, and use an eight-way set-associative mapping. A *W*-way set-associative mapping means that each cache set consists of *W* blocks. The chip's L2 cache offers 1024 KB per core, and uses a 16-way set-associative mapping. The unified L3 cache is of size 14080 KB, is shared between 10 processing cores, and uses 11-way set-associative mapping. The line size of these three cache levels is 64 bytes.

When zero padding was applied to matrices of size 4092 and 8192, the first 32 lines of the *B* tile were mapped to different sets of the L1 cache, and the next 32 lines were mapped to the same pattern. For example, line numbers {0, 32}, {1, 33}, {2, 34}, and {3, 35} were mapped to cache set numbers 0, 1, 2, and 3 respectively. Since the L1 cache of the Core i9-7900X consists of eight ways, all 64 lines of the *B* tiles could be put into L1 simultaneously, allowing all of them to be reused in the next iteration of the *I*-loop. Retaining all the *B* tile data in the L1 cache meant that padding improved performance when the matrix size was 4096 or 8192 for both single and double precision floating-point matrices.

When the single precision matrix dimension was not 4092 or 8192, all of the unpadded data lines in the *B* tile could be mapped to different set numbers of the L2 cache. For example, when the matrix size was 3072, the data line numbers {0, 16}, {1, 17}, {2, 18}, ..., {15, 31} could be mapped to the L2 cache set numbers 0, 1, 2, ..., 15. Therefore, all of the *B* tile of the unpadded version could be placed in the L2 cache simultaneously, and be reused in the next iteration of the *I*-loop. Although the padding mechanism moved all of this *B* tile data up to the L1 cache which was faster, it was offset by the additional cost of memory allocation and deallocation for the A_{padded}, B_{padded}, and C_{padded} matrices, combined with the cost of data copying between the original *A*, *B*, *C* and the padded matrices.

For double precision matrices with sizes other than 4096 or 8192, the unpadded data in the *B* tile could be mapped to some L2 cache sets, but at a lower level than for single precision data. For example, when the matrix size was 3072, the data line numbers {0, 8}, {1, 9}, {2, 10}, ..., {7, 15} could be mapped to the L2 cache set numbers 0, 1, 2, ..., 7. Since unpadded multiplication needed to read the *B* tile from the L3 cache more often, padded multiplication (which moves *B* up to L1) could run slightly faster even though it incurred an additional cost, as explained above.

Conclusions

We have proposed an effective implementation for matrix-matrix multiplication using 2-D tiling and AVX512 intrinsics. For single precision matrix multiplication, our algorithm is on average 2.12 and 2.13 times faster than OpenBLAS and Eigen. For double precision matrix multiplication, our algorithm is on average 2.06 and 2.17 times faster than OpenBLAS and Eigen respectively. There are three main reasons for our algorithm's higher performance: 1) the utilization of SIMD instructions, unrolling, and OpenMP directives, increases instruction level parallelism and efficiently distributes the workload among the AVX512 engines in the processing cores; 2) dividing data into tiles allows each one to be small enough to be kept in the CPU's cache, which greatly reduces memory bottlenecks; and 3) zero padding allows tiles to be reused in upper cache levels which is faster, and reduces the wait state of the CPU's AVX512 engines.

We also present an equation for calculating the optimum padding size, which permits our algorithm to be applied to different sizes of matrix on CPUs with different cache mappings.

Acknowledgements

This work was supported in part by the Higher Education Research Promotion and Thailand's Education Hub for Southern Region of ASEAN Countries Project Office of the Higher Education Commission under Grant TEH-AC 048/2016. The authors are grateful to Dr. Andrew Davison for his kind help in polishing the language of this paper.

References

[1] Ahmad, A., & Pasha, M. A. (2020). Optimizing hardware accelerated general matrix-matrix multiplication for CNNs on FPGAs, *IEEE Transactions on Circuits and Systems II: Express Briefs, 67*(11), 2692–2696.

[2] Choi, Y. R., Nikolskiy, V., & Stegailov, V. (2020). Matrix-matrix multiplication using multiple GPUS connected by Nvlink. In *The 2020 Global Smart Industry Conference: GloSIC 2020*. 354-361. November 17-19, 2020, Chelyabinsk, Russian Federation: IEEE.

[3] Rasouli, M., Kirby, R. M., & Sundar, H. (2021). A compressed, divide and conquer algorithm for scalable distributed matrix-matrix multiplication. In *The 2021 International Conference on High Performance Computing in Asia-Pacific Region, HPC Asia 2021*. 110-119. January 20-22, 2021. Virtual Event, Republic of Korea. New York: Association for Computing Machinery.

[4] Jacob, B., & Guennebaud, G. (2020). *Eigen is a C++ template library for linear algebra: matrices, vectors, numerical solvers, and related algorithms* (Online). Retrieved Jan 15, 2020, from https://eigen.tuxfamily.org/index.php?title=Main_Page.

[5] Kroeker, M. (2020). *Open BLAS 0.3.10 version* (Online). Retrieved June 24, 2020, from https://github.com/xianyi/OpenBLAS/releases/tag/v0.3.10.

[6] WikiChip. (2020). *Intel Skylake (client) Microarchitectures* (Online). Retrieved March 18, 2021, from https://en.wikichip.org/wiki/intel/microarchitectures/skylake_(client).

[7] Intel Corporation. (2020). *Intel Core i3-1000G1 processor* (Online). Retrieved September 2, 2020, from https://ark.intel.com/content/www/us/en/ark/products/197122/intel-core-i3-1000g1-processor-4m-cache-up-to-3-20-ghz.html.

[8] ARM Limited. (2020). *Neon Intrinsics: Getting started on Android user guide* (Online). Retrieved March 20, 2021, from https://developer.arm.com/solutions/os/android/developer-guides/neonintrinsics getting-started-on-android.

[9] IBM Corporation. (2018). *Power ISA: Version 2.07B* (Online). Retrieved March 23, 2021, from https://ibm.ent.box.com/s/jd5w15gz301s5b5dt375mshpq9c3lh4u.

[10] Guimaraes, A., Aranha, D. F., & Borin, E. (2019). Optimized implementation of QC-MDPC codebased cryptography, *Concurrency and Computation-Practice & Experience*, *31*(18), e5089. DOI: 10.1002/cpe.5089.

[11] Ginting, B. M., & Mundani, R. P. (2019). Comparison of shallow water solvers: Applications for Dam-Break and Tsunami cases with reordering strategy for efficient vectorization on modern hardware, *Water*, *11*(4), 639. DOI: 10.3390/w11040639.

[12] Ahmad, N., & Bakar, R. (2019). An analysis for the performance of reservoir simulations on a multicore CPU. In *The International Field Exploration and Development Conference: IFEDC 2019*. 3514–3530. October 16–18, 2019, Chengdu, China: Springer.

[13] Lasch, R., Oukid, I., Dementiev, R., May, N., Demirsoy, S. S., & Sattler, K. U. (2020). Faster & strong: string dictionary compression using sampling and fast vectorized decompression, *The VLDB Journal*, *29*, 1263–1285. DOI: 10.1007/s00778-020-00620-x.

[14] Hassana, S. A., Hemeida, A. M., & Mahmoud, M. M. M. (2016). Performance evaluation of matrixmatrix multiplications using Intel's Advanced Vector Extensions (AVX). *Microprocessors and Microsystems*, *47*(SI), 369–374, DOI:10.1016/j.micpro.2016.10. 002.

[15] Hassan, S. A., Mahmoud, M. M. M., Hemeida, A. M., & Saber, M. A. (2018). Effective implementation of matrix-vector multiplication on Intel's AVX multicore processor. *Computer Languages Systems & Structures*, *51*, 158–175.