Implementation of General Matrix Multiplication (GEMM) on x86-64 CPUs

Wayne Anderson

(29/06/2024)

1. Introduction

General matrix multiplication (GEMM) is one of the most used operations in both mathematical and computational context. It's also an essential operation in linear algebra, e.g., performing linear transform between two datasets. The definition is given in equation (1.1). Given two matrices $[\mathbf{A}]$ and $[\mathbf{B}]$, each element in the resultant matric $[\boldsymbol{C}]$ is given by:

$$C_{i,j} = \sum_{k=0}^{W_A-1} A_{i,k} * B_{k,j} \tag{1.1}$$

Regrading one dimension of matrix, GEMM is an operation with $O(n^3)$ complexity, which is extremely computationally expensive. There are many implementations already for the optimization on GEMM algorithm. This article introduces various optimization techniques based on x86 64bit CPUs. For 32-bit (fp32) and 64-bit (fp64) floating point number, the article introduces methods of rearranging the matrix B, processing the matrices block by block, etc., to achieve a peak GFLOPs 86% of the theoretical peak computational capacity of the hardware. For complex data, this article introduces Strassen's algorithm to reduce the complex number multiplications during the accumulation stage.

The full version of implementations of various datatypes are given in the source of library DECX. Please visit https://github.com/param0037/DECX for more information.

2. Method

2.1 Rearrangement of matrix B

The height of matrix B is the most frequently accessed dimension during the GEMM algorithm. In this implementation, the matrix B is rearranged so that it's transposed, and the elements adjacent vertically are now adjacent horizontally. Since DECX stores and processes the matrix in row-major, the elements are adjacent in physical address as well. Hence, rearranging the matrix B helps make better use of CPU caches.

2.1.1 The lane of matrix B

AVX (Advanced vector Extensions) are part of the SIMD (Single Instruction Multiple Data) extensions on Intel x86 CPUs [1]. For x86-64 architectures, there are 16 YMM registers in each core. Each YMM register (YMM0~YMM15) is 256 bits in size. For 32-bit floating point (fp32) number in modern C/C++, one YMM register can carry a vector composed of 8 fp32 numbers. In the implementation, 256 bits of data (8x fp32, 4x fp64) is processed in a group. The group is called lane in rearrangement of matrix B. layout of each lane is represented by Figure 2.1.1.

2.1.2 Rearrangement of real matrix B

However, there are 16 YMM registers in each core of 64bit x86 CPU, if only one lane is calculated each time, the minimum required YMM register number is 3 (values from matrix A, B and result for each), which makes YMM registers significantly under-utilized. Hence, the implementation calculates two lanes each time, resulting a dual-lane of data being packed into a group.

Additionally, packing 2 lanes is necessary for cases where the data type is complex number. Strassen's algorithm [2] on 2x2 matrix multiplication can be adapted easily, which will be introduced later.

In DECX, the width of each matrix is aligned to 256 bits. However, in dual-lane mode, 2 lanes with 256 bits for each are packed into one group, which indicates that for some cases there will be a single lane leftover. To address this problem, a new virtual lane will be introduced, filled with zeros. Figure 2.1.1 illustrates how the matrix B is rearranged for fp32 and fp64 datatypes.

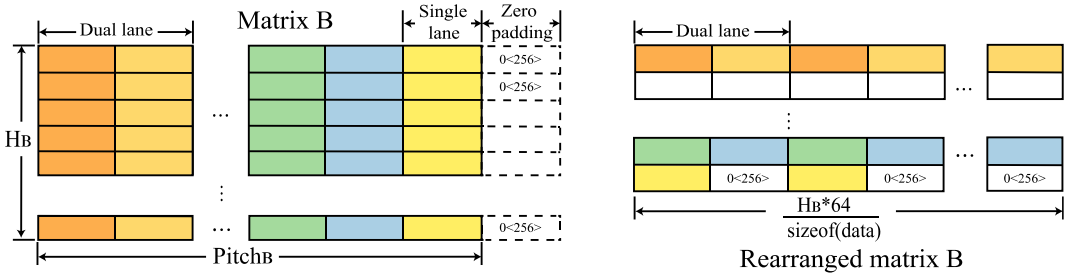

Figure 2.1.1 Arranging real matrix B

Hence, the sizes of rearranged matrix B can be expressed as:

$$W' = H_B * \frac{256}{sizeof(data)} \tag{2.1.1}$$

$$H' = ceil\left(W_B, \frac{256}{sizeof(data)}\right) \tag{2.1.2}$$

Where, $ceil(x, y)$ calculates the value of x divided by y and returns the result rounded towards zero.

2.1.3 Rearrangement of complex matrix B

For complex matrix multiplication, the requirement of computational resource is massive. Starting by observing the multiplication of 2 packs of 4 complex fp32 numbers using AVX2:

```
1.inline _THREAD_CALL_
2.__m256 _cp4_mul_cp4_fp32(const __m256 __x, const __m256 __y) {
3.    __m256 rr_ii = _mm256_mul_ps(__x, __y);
4.    __m256 ri_ir = _mm256_mul_ps(__x, _mm256_permute_ps(__y, 0b10110001));
5.
6.    rr_ii = _mm256_permute_ps(rr_ii, 0b11011000);
7.    ri_ir = _mm256_permute_ps(ri_ir, 0b11011000);
8.    __m256 res = _mm256_unpacklo_ps(rr_ii, ri_ir);
9.    res = _mm256_addsub_ps(res, _mm256_unpackhi_ps(rr_ii, ri_ir));
10.   return res;
11.}
```

The function involves multiple permutations over YMM registers, also involves extra multiplication of two YMMs compared to the real number cases, where only one multiplication and one addition are needed. Strassen's algorithm on 2x2 matrix multiplication reduces one multiplication operation from 8 multiplies to 7. In complex number cases, the reduction significantly saves machine cycles. Hence, it's necessary to adapt Strassen's method in complex number cases. Strassen's algorithm for 2x2 matrix multiply ([C]=[***A***] ∗ [***B***]) can be shown in equations (2.1.3) to (2.1.10):

$$M_1 = (A_{11} + A_{22})(B_{11} + B_{22}) \tag{2.1.3}$$

$$M_2 = (A_{21} + A_{22})B_{11} \tag{2.1.4}$$

$$M_3 = A_{11}(B_{12} - B_{22}) \tag{2.1.5}$$

$$M_4 = A_{22}(B_{21} - B_{11}) \tag{2.1.6}$$

$$M_5 = (A_{11} + A_{12})B_{22} \tag{2.1.7}$$

$$M_6 = (A_{21} - A_{11})(B_{11} + B_{12}) \tag{2.1.8}$$

$$M_7 = (A_{12} - A_{22})(B_{21} + B_{22}) \tag{2.1.9}$$

$$\begin{bmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{bmatrix} = \begin{bmatrix} M_1 + M_4 - M_5 + M_7 & M_3 + M_5 \\ M_2 + M_4 & M_1 - M_2 + M_3 + M_6 \end{bmatrix} \tag{2.1.10}$$

The implementation in DECX adapts Strassen's method in AVX2. However, for 2x2 matrix, the behavior of rearrangement in dual-lane mode merges 2 sub-matrix in one lane for complex fp32 case. Following the method described by Figure2.1.1, the arranged matrix B has following layout:

$$\begin{bmatrix} B_{1,11} & B_{1,12} & B_{2,11} & B_{2,12} & B_{3,11} & B_{3,12} & B_{4,11} & B_{4,12} \\ B_{1,21} & B_{1,22} & B_{2,21} & B_{2,22} & B_{3,21} & B_{3,22} & B_{4,21} & B_{4,22} \end{bmatrix} \tag{2.1.11}$$

Where $X_{k,ij}$ represents the k[th] sub-matrix, i and j represents the row and column of element in each sub-matrix, respectively. To adapt AVX2 in Strassen's method, the matrix B should have the following layout:

$$\begin{bmatrix} B_{1,11} & B_{2,11} & B_{3,11} & B_{4,11} & B_{1,12} & B_{2,12} & B_{3,12} & B_{4,12} \\ B_{1,21} & B_{2,21} & B_{3,21} & B_{4,21} & B_{1,22} & B_{2,22} & B_{3,22} & B_{4,22} \end{bmatrix} \tag{2.1.12}$$

Which can be further represented as:

$$\begin{bmatrix} [B_{1,11}, B_{2,11}, B_{3,11}, B_{4,11}] & [B_{1,12}, B_{2,12}, B_{3,12}, B_{4,12}] \\ [B_{1,21}, B_{2,21}, B_{3,21}, B_{4,21}] & [B_{1,22}, B_{2,22}, B_{3,22}, B_{4,22}] \end{bmatrix} \tag{2.113}$$

Hence, the elements in matrix B during processing are all vectors, e.g., $B_{11} = [B_{1,11}, B_{2,11}, B_{3,11}, B_{4,11}]$. As the result, the packed elements of destinated matrix C are also vectors, which needed to be permuted back to the supposed layout, which will be introduced later.

2.2 Cache friendly control strategy

2.2.1 Blocking of processing region

Each thread calculates its own processing region block by block along the three dimensions, as shown in

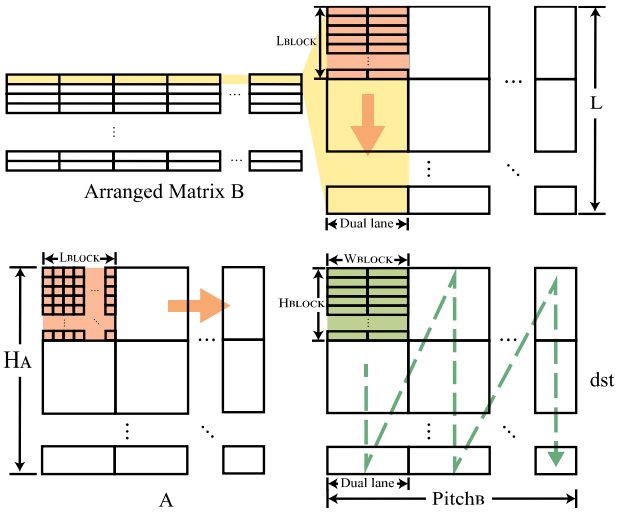

Figure 2.2.1 Block movement

Figure 2.2.1. The green area of dst matrix represents a block. The thread first loop along the height of the processing region, and then the width, as shown by the green dotted arrow in the figure. Each bock only accumulates L_{BLOCK} numbers along dimension L. results of each block are stored to the destinated matrix (dst). After finishing stepping through the entire processing region of destinated matrix, the block step L_{BLOCK} forward on dimension L. The following blocks should all load the previous result from the destinated matrix first before the accumulation, so that the results are accumulated all the way along dimension L instead of replaced by the accumulation of the latest block.

The pseudocodes of the control strategy can be written as:

```
Block_processing() PROC
  for k=0:L-1, step=L_BLOCK
      for i=0:W-1, step=W_BLOCK
          for j=0:H-1, step=H_BLOCK
              If k==0
                  dual_lane_registers = 0<512bits>;
              else
                  dual_lane_registers = dst(j,i);
              end
              Accumulation(dual_lane_registers);
              store(dual_lane_registers, dst(j,i));
          end
      end
  end
ENDP
```

Three sizes: $H_{BLOCK}, W_{BLOCK}, L_{BLOCK}$, are carefully designed, so that during each accumulation stage, blocks from matrix A, B and dst can be fitted into L1 or L2 cache. Assuming that the three blocks are small enough to be fitted into L1 and L2 cache exactly. When looping along the height of the processing region, block from matrix B can be reused. If the order is reversed, that is, looping along width first, block of matrix B will be kicked off the cache when moving rightwards. However, block of matrix A can be preserved since it's still accessing the same rows of matrix A. The first order described, scarifies cache of matrix A instead. The implementation in DECX chooses the first order. As is described above, 2x 256bit data is packed into a group for matrix B during the accumulation. Whereas block from matrix B has the same length on dimension L as the block from matrix A. Hence, the size of block B is several times larger than block A. Thus, scarifying block A introduces less performance loss.

2.2.2 Block sizes preferences

From the analysis of section 2.2.1, it should be guaranteed to be fitted into cache. Given a hardware (CPU), its cache sizes are fixed. Since equation (2.2.1) should always be satisfied:

$$H_{BLOCK} * W_{BLOCK} * L_{BLOCK} = \frac{C}{64} \qquad (2.2.1)$$

Where, C denotes that capacitance of L1 cache (or other hierarchy of memory), in byte; 64 means that there are 64 bytes in each dual-lane. Changing the width and height has little impact on the performance, theoretically. However, W_{BLOCK} should be chosen to be larger than cache line size, in case any waste on the cached data. On the contrast, increasing L_{BLOCK} can improve performance. From section 2.2.1, the thread should spend N times to finish looping through the entire L dimension, where N can be expressed by equation (2.2.2):

$$N = ceil(L, L_{BLOCK}) \qquad (2.2.2)$$

Except for the first time of the loop, the total time of reading the destinated matrix (at the begging of each block), is N-1. The total time of writing the matrix will

be N. Thus, increase L_{BLOCK} decreases N, hence decreases the total read and write time. Moreover, the cache of the previous result on the block region of destinated matrix has already been popped out, since the process has looped through the entire processing region of the destinated matrix, which means that process must request memory access from memory regions that are much slower.

In the implementation, L_{BLOCK} is significantly larger than, at least, H_{BLOCK}, the width might also be large due to the considerable amount of data carried by each dual-lane. In practice, L_{BLOCK} can't be too large, instead, H_{BLOCK}, W_{BLOCK} should be both slightly larger than their minimal values.

3. Threading strategy

Each element of the resultant matrix requires looping through the entire L dimension. In the implementation, the concurrent threads are distributed in 2D form on the destinated matrix, as shown in Figure 3.1.

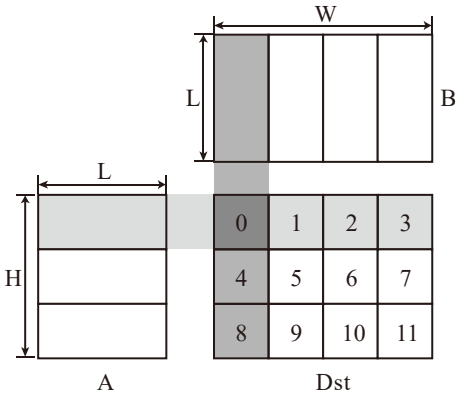

Figure 3.1 2D thread distribution

In this form, each thread only load fragments of matrix A and B, as shown in the shaded areas on matrix A and B. 2D distribution increases the chance of data needed for one thread being fitted into L2 cache.

Figure 3.2 and 3.3 show the two cases for 1D distribution. From the figures, 1D distributions require loading the entire region of either matrix A or B, increasing the size of the necessary data, thus decreasing the possibility of being fitted into any cache.

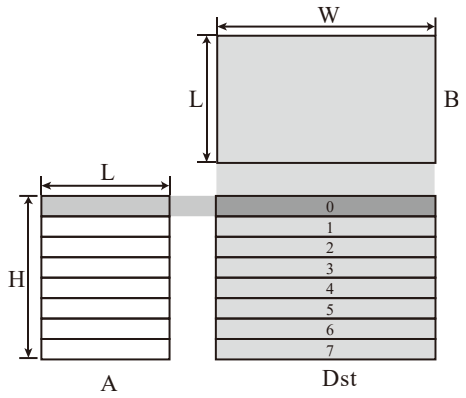

Figure 3.2 Distribute along height

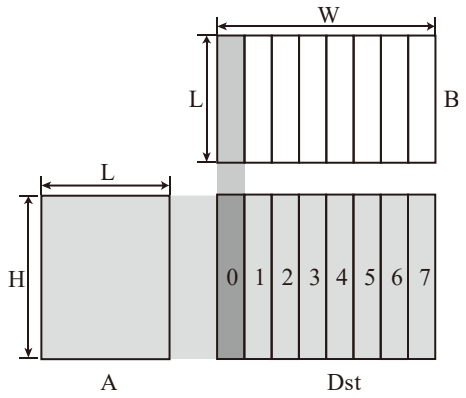

Figure 3.3 Distribute along width

To obtain how many threads are distributed along the both dimensions, DECX calculates the ratio of the width to the height of the destinated matrix. Let k denote such ratio, we wish the thread distribution follows this ratio as well. Hence, the equation (3.1) should be satisfied:

$$T_H T_W = concurrency$$
$$T_H(kT_H) = concurrency \quad (3.1)$$

Where, T_H and T_W denote the number of threads distributes along the height and width, respectively; concurrency denotes the total concurrency of the hardware, e.g., concurrency=12 for Intel Core I7 9750H. Thus, T_H can be obtained:

$$T_H = round\_nearest\left(\sqrt{\frac{concurrency}{k}}\right) \quad (3.2)$$

$$T_W = concurrency/k \quad (3.3)$$

The function $round\_nearest()$ rounds the floating-point number to the nearest integer that can divide $concurrency$ into an integer, without any residual.

4. Profiling

4.1 GFlops and efficiency

The implemented GEMM program can be found in GitHub repository of DECX. The testing environment and configuration are described in Table 4.1.1.

Operating System	Windows 10, 64bit
CPU	Intel Core I7 9750H
Base Frequency	2.59 GHz
# of Logical Processors	12
Matrix A sizes ([W, H])	[1024, 1024]
Matrix B sizes ([W, H])	[1024, 1024]
Matrix dst sizes ([W, H])	[1024, 1024]
Execution times	500

Table 4.1.1 Testing env. and conf.

The executing period is recorded by ctime library. The best performance gives 9.98msec per execution. The peak flops can be calculated by equation (4.1.1) [3]:

$$Flops_{max} = (\#processors) \times (\#cores\_per\_processor) \times \left(clock\ speed\left(\frac{1}{s}\right)\right) \times (2 \times \#FMA_{units}) \times \left(\frac{vecotor\_size(bits)}{32}\right) \quad (4.1.1)$$

Since the peak GFLOPs for the testing machine is:

$$GFlops_{max} = \frac{1 \times 12 \times 2.59 \times 10^9 \times 2 \times 2 \times \left(\frac{256}{32}\right)}{1024^3} = 231.564$$

The actual performance is:

$$GFlops_{actual} = \frac{1024 \times 1024 \times 1024 \times 2}{1024^3 \times 9.98 \times 10^{-2}} = 200.4$$

Hence the implementation reaches $\eta = \frac{200.4}{231.564} = 86.542\%$ of the theoretical peak, which is significantly efficient. The testing result also proofs the validity of the methods proposed by this article.

4.2 Profiling results

In the best case among the testing result, which is described in section 4.1, has the following parameter selections:

$$L_{BLOCK} = 128$$
$$W_{BLOCK} = 2 \times 16 = 32$$
$$H_{BLOCK} = 8$$

Where the values represent the number of fp32. The further profiling results are given by Intel VTune Profiler, as shown in Figure 4.2.1.

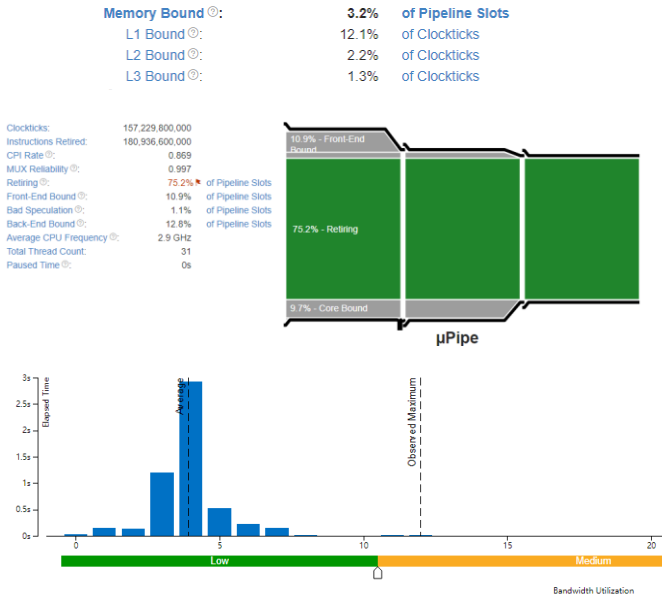

Figure 4.2.1 Intel VTune profiling results

The memory bound is only 3.2% per pipeline slots. The average DRAM bandwidth utilization is only 4GB/sec, approximately, which implies that the program can make good use of the caches.

5. References

[1] Wikipedia contributors. (2024, June 29). Advanced Vector Extensions. In Wikipedia, The Free Encyclopedia. Retrieved June 29, 2024, from https://en.wikipedia.org/wiki/Advanced_Vector_Extensions

[2] Huss-Lederman, S., Jacobson, E. M., Tsao, A., Turnbull, T., & Johnson, J. R. (1996, November). Implementation of Strassen's algorithm for matrix multiplication. In Proceedings of the 1996 ACM/IEEE Conference on Supercomputing (pp. 32-es).

[3] Nakano, A. (n.d.). Theoretical Peak Performance of a Computer. Retrieved June 29, 2024, from https://aiichironakano.github.io/cs596/PeakFlops.pdf