DOI: 10.20535/kpi-sn.2020.2.205115
UDC 004.231.2(045)

I.A. Dychka*, D.A. Vinnyk, Yu.V. Bukhtiyarov, V.Ya. Yurchyshyn
Igor Sikorsky Kyiv Polytechnic Institute, Kyiv, Ukraine
corresponding author: dychka@pzks.fpm.kpi.ua

METHOD OF FAST MATRIX MULTIPLICATION UNDER ARM ARCHITECTURE USING SIMD INSTRUCTIONS

Background. Matrix multiplication is a rather complicated algorithm with a large number of operations. An additional problem is the nonlinear memory traversal of matrices. Matrix multiplication is widely used in various fields, such as neural networks, solutions of linear equation systems, matrix transformations, and so on. Therefore, it is important to develop a method of matrix multiplication, which will take into account the problems of the location of the matrices in memory, and will effectively manage the data when reused.
Objective. The purpose of the paper is to develop a method of fast matrix multiplication of two matrices, as well as multiplying the matrix by the transposed matrix and by a list of vectors (including special case for only one vector), as well as to implement it as a function with optimization for ARM architecture processors. The function must be able to handle different types of data and submatrices. The integer result can be scaled.
Methods. The main ideas of the developed method are simultaneous work with several rows/columns of input matrices and their splitting into blocks, which will allow the algorithm to run on the same memory for a while. The C programming language was chosen for implementation. SIMD instructions were used to increase productivity. We also need to properly organize the memory preloading for effective implementation under the ARM architecture.
Results. A function that performs matrix multiplication by the developed method with the necessary parameters was implemented as a result of the study. Tests on various sizes and types have shown that the implemented function is faster than analogues from the OpenCV2 and Eigen 3 libraries. Testing was done using the vipmed utility for running and measuring features developed for enterprise use at VIT.
Conclusions. The proposed matrix multiplication method gives the expected acceleration of matrix multiplication operations, has passed evaluation test for use and meets the target requirements. For further work, it is necessary to study in more detail the influence of the cache at different levels and compare with other existing libraries.
Keywords: matrix multiplication; ARM architecture; vector operations; matrix transposition.

Introduction

In the everyday life, the matrices are used much wider than people are apt to think. In fact, we face them every day.

Graphics software, such as Adobe Photoshop, uses image matrices for image processing. A square matrix can represent a linear transformation of a geometric object. Matrices and inverse matrices are used in programming to encode and encrypt messages. The message is generated as a sequence of numbers in binary format for communication, and the code theory should be used to solve it.

Many IT companies also use matrices as data structures to track user information, perform search queries, and manage databases. In terms of information security, many systems have been designed to manage matrices. Matrix multiplication is widely used when working with neural networks. Neurons values matrices are multiplied at transition between the network layers [1].

Matrices are broadly used in physics, electrodynamics, electronics, radio engineering. Even a cursory survey of the bibliography on this subject reveals its huge volume. The theory of matrix methods is sufficiently developed, but the practical implementation of these methods has not exhausted its potential.

The aim of this research is to explain what factors affect matrix multiplication and how to use them for improving performance. At first, we describe main problems of effective matrix multiplication implementation. Then we show some of the existing solutions and describe own one. Finally, we compare our implementation with the existing described above.

Leaving aside the application of the method described below in solving practical problems for the future, we will now turn to a detailed description of the method itself.

Problem statement

The aim of our research is to implement effective matrix multiplication method.

Overview of the existing solutions

The problem of using matrix multiplication is that it has the burden of performing a great deal of operations. In example, we define two matrices $\boldsymbol{A}$ and $\boldsymbol{B}$:

$$A = \begin{pmatrix} a_{11} & \cdots & a_{1m} \\ \vdots & \ddots & \vdots \\ a_{n1} & \cdots & a_{nm} \end{pmatrix}, B = \begin{pmatrix} b_{11} & \cdots & b_{1p} \\ \vdots & \ddots & \vdots \\ b_{m1} & \cdots & b_{mp} \end{pmatrix}.$$

Assume that the result is $\boldsymbol{C}$:

$$C = \begin{pmatrix} c_{11} & \cdots & c_{1p} \\ \vdots & \ddots & \vdots \\ c_{n1} & \cdots & c_{np} \end{pmatrix}.$$

Therefore, matrix multiplication formula is:

$$c_{ij} = a_{i1}b_{1j} + \cdots + a_{im}b_{mj} = \sum_{k=1}^{m} a_{ik}b_{kj},$$

for $i = 1, ..., n$ and $j = 1, ..., p$.

The complexity of this elementary algorithm is $O(nmp)$ or $O(n^3)$ if matrices are square($n = m = p$) [2]. It is not necessary to go into the details that the cubic complexity is a bad property.

The existing solutions having open source code are not effective enough (this will be proven in the results).

There are many algorithms for rapid matrix multiplication that reduce the complexity of the operation. The best-known and used in practice is the Strassen algorithm, which reduces the complexity to $O(n^{log27})$, which is approximately equal to $O(n^{2.81})$ [3]. All the other algorithms are only theoretical and approximate, so they are practically not used [4].

These algorithms are purely mathematical and do not take into account such an important point as placement the matrices in memory. They only reduce the number of multiplications, so in practical programming this is not enough.

Considering the chosen processor architecture, memory management is very important. To speed up this work, we use memory preload in cache. The auto-preloader of X86 processors, unlike ARM, works quite well, so it makes little sense to do this job manually. In ARM architecture, the well-timed use of the memory preloader can result in speeding up operation many-fold. However, if one makes a mistake, there is a significant drop in performance.

Therefore, we propose a new method of the matrix multiplication considering above information.

There are many libraries, including open source code, that implement matrix multiplication. In the specification of basic linear algebra subroutines (BLAS), this operation has a more enhanced interface and is called gemm. There are many implementations of this specification and most of them implement matrix multiplication using SIMD technology [5].

OpenCV is a library of software functions primarily focused on real-time computer vision algorithms. This library is a very powerful tool: it has many useful features, is cross-platform and implemented in several programming languages. It is distributed as BSD-licensed open source code software. OpenCV is not BLAS compatible, but implements a similar gemm function [6].

Eigen is a template library, it provides a simple and very common C++ 98 template interface for matrix/vector operations and related algorithms. This library, like OpenCV, contains implementations that leverage vector operations for optimization. Important thing is that there are some (more optimal) implementations for some fixed sizes. The main feature of this library is that it is fully implemented in the headers, so one only needs to download these files to be used [7].

These libraries stated that they have optimized algorithms for matrix operations, so we choose them for comparing with ours.

It is pointless to describe in detail how the matrix multiplication algorithm is implemented in the above libraries. The result of their work and comparison with the proposed method will be shown below. Their main disadvantage is the lack of performance, so the purpose of this article is to develop a faster method of implementing matrix multiplication.

Description of the proposed method

The matrix multiplication of the $M \times K$ matrix $\boldsymbol{A}$ and $K \times N$ matrix $\boldsymbol{B}$ results in the creation of $M \times N$ matrix $\boldsymbol{C}$. Each element in matrix $\boldsymbol{C}$ can be considered as a scalar product of the corresponding row of matrix $\boldsymbol{A}$ and column of matrix $\boldsymbol{B}$.

It is possible to implement all matrix multiplications by using a primitive scalar product, but such implementation would be far from effective. In a scalar product, we load two elements for each multiplication-add operation, and on modern processors, this implementation will be limited by memory or cache bandwidth instead of the computing power of multiplication-add units. Neverthe-

less, a minor modification – calculating point products from several rows in ***A*** and several columns in ***B*** at a time – improves performance significantly.

The modified primitive takes *MR* elements of elements in ***A*** and *NR* of ***B*** elements and performs multiplication operations with *MR*x*NR* accumulation. The number of registers and other details of the processor architecture limit maximum *MR* and *NR* values. But in most modern systems, they are large enough to make the operation limited, and all high-performance implementations of matrix-matrix multiplication are built on this primitive microkernel commonly called PDOT (panel dot product). The $M = N = 4$ workaround was selected in the method stated below. Such a small number causes the limitations of the chosen architecture.

This paper considers the matrix multiplication method for the ARM v7 architecture. It is 32-bit and has some limitations on the number of registers. With the exception of Armv6-M, Armv7-M, Armv8-M.baseline and Armv8-M.mainline based processors, there are 33 32-bit general-purpose registers, including redundant SP and LR registers. Fifteen general-purpose registers are visible at any time, depending on the current processor mode. These are R0-R12, SP and LR. PC (R15) is not considered a general-purpose registry [8].

SP (or R13) is a stack pointer. C and C++ compilers always use SP as a stack pointer. Arm devalues most SP applications as a general-purpose registry. In T32 state, SP is strictly defined as a stack pointer. ARM official documentation describes when SP and PC can be used.

In a user mode, LR (or R14) is used as a register of links to store a return address when a subroutine is called. It can also be used as a general-purpose register, if the return address is stored in the stack.

In exception handling modes, the LR stores the return address for the exception or the return address of the subroutine, if subroutine calls are made within the exception. The LR can be used as a general-purpose register if the return address is stored in the stack.

From the above it is clear that only 13 regular registers are available for general use without any restrictions.

The algorithm requires much more regular registers to move with the four rows of the first to the second matrix, as noted above. Matrix row pointers (4 first + 4 second) also need pointers to the resulting matrix rows, registers to store sizes and iterators for each of them. Since matrices can actually only be parts of larger matrices, the notion of a step between rows is introduced. This step is defined in bytes and is equal to the actual width of the entire image, multiplied by the size of one element. The user with an understanding of the storage area in which one operates should transmit this data. These steps, for each of the three matrices, accordingly, require registers. So, even not taking into account the extra registers that may be needed to calculate, the required number equals already to twenty. Therefore, to save all the necessary values, one needs to allocate additional temporary memory. To use this data, they should be loaded in free registers and stored back in time, in case of necessity, the register should be freed up (saving constant values, such as matrix sizes and line spacing, each time is not required).

The main practical problem in calculating the product of matrices is the inefficient bypass of the second matrix, because the result of a particular element of the result matrix is the product of the row of the first matrix and the second matrix column. Column matrix bypass is quite inefficient in Row-Major mode (when a row in memory is in sequence). To solve this problem, the accumulation of results in a temporary buffer was chosen, while moving linearly on the second matrix. With this approach, the number of bypasses of the second matrix increases. In fact, with each row of the first matrix, the second one is completely read-out. However, gradual read-out of the data slows down the program operation to such an extent, that multiple linear reading of the matrix still faster than just one inconsistent read. Considering that the bypass will be performed at once by four rows of the first matrix, the number of bypasses of the second one is reduced by four times (it is fully read-out once per every four rows of the first matrix).

Of course, the disadvantage of this approach is the considerable amount of additional dedicated memory (4* the width of the second matrix). Thus, the resulting calculation is divided into two parts: first, there is an accumulation in the temporary buffer, and then, only at the last iteration, the entry in the resulting matrix. The last iteration is the calculation on the last rows of the second matrix. The magnitude of the so-called matrix tail will be equal to the remainder of the division of the second matrix height (or the width of the first, since they are equal) by the number of rows that are bypassed within one iteration (in this case by four). If there is no remainder, then one iteration less is performed in the total cycle and when processing the last four rows, the result is immediately written to the resulting matrix.

As a result, we have the following general algorithm for matrices $A(M\times K)$ and $B(K\times N)$ multiplied into the matrix $C(M\times N)$ by the blocks $m\times R$ and $R\times n$ with the tail having t value:

1. Allocation of necessary additional memory (for variables and accumulation), initial data initialization (including memory reset into which the result will be accumulated) and their storage.

2. Reading-out R elements from m rows of the first matrix.

3. Reading-out n elements from the R rows of the second matrix.

4. Read n elements from m lines of temporary buffer with intermediate results.

5. Scalar multiplication of blocks $m\times R$ and $R\times n$ and their accumulation to data read-out from a temporary buffer.

6. Record of intermediate results back to the temporary buffer.

7. Implementation of items 3–6 N/n times.

8. Upon bypass of the entire width of the second matrix, the transition to next R rows of that matrix and the R elements of the first one is performed. Temporary buffer pointers are moved to the beginning and accumulation will further occur in it.

9. Implementation of items 2–8 (K/R-t) times.

10. At the tail iteration, we read-out the last t elements from the m rows of the first matrix.

11. We read n elements from the last t rows of the second matrix.

12. Same as item 4.

13. Scalar multiplication of blocks m x t and t x n and their accumulation to read-out data from the temporary buffer.

14. Resetting the temporary buffer.

15. Writing m rows of n elements in the resulting matrix.

16. Implementation of items 11–15 N/n times.

17. Transition to the next m rows of the first and the resulting matrices.

18. Implementation of items 2–17 M/m times.

Implementation using SIMD instructions

In practical application, this algorithm is good enough. It is worth to note that aliquant values are not taken into account here, that is, it is necessary to additionally process the remainders, but it is decided to describe the algorithm without such, quite clear, details.

Of course, fast calculation requires more than just an efficient algorithm. One has to additionally look for optimization methods (both algorithmic and architecture related). Using algorithmic optimization, one can specify items related to tail processing. In a simple algorithm, individual points are not dedicated to this: first, the result is completely obtained in the temporary buffer (i.e., points 2–8 are performed K/R times), then the result is rewritten into matrix C and the last step is the resetting of the temporary buffer. In this algorithm, all these operations occur in one pass.

As to lower-level optimization, one should start with the vector instructions. Such operations allow executing operation with several values written in vector registers.

SIMD is a class of parallel programming, which is based on such operations. Most modern processors are designed to support SIMD instructions to enhance performance. This class is particularly popular in signal processing, where, as a rule, a large number of identical data is processed with similar operations. SIMD also allows processing several similar data types with the same instruction (as indicated in the name – Single instruction, multiple data; which is rendered as: one instruction for lots of data).

In ARM architecture processors, SIMD is represented as NEON (Advanced SIMD) extension. The Registry Bank of this extension is a collection of registers that can be accessed both as 64-bit and 128-bit vector registers. Advanced SIMD and VFP (floating-point values operations) use the same registers and differ from the main ARM register bank.

128-bit registers are called Q-registers, and 64-bit – D-registers. Each Q-register corresponds to 2 D-registers, they are overlapping. The mapping between the registers is as follows:

D ⟨2n⟩ maps the least significant half of Q ⟨n⟩;

D ⟨2n + 1⟩ maps the most significant half of Q ⟨n⟩.

For example, one can access the least significant half of the vector elements in Q6 by referring to D12 and the most significant half of the elements – by referring to D13.

Therefore, in general, the registry bank can be represented by:

– Sixteen 128-bit registers Q0-Q15;
– Thirty-two 64-bit registers D0-D31;
– A combination of D and Q registers.

The SIMD extension treats each register as containing 1, 2, 4, 8, or 16 elements of the same size and type (the number depends on the register size and the element, respectively). Individual elements can also be accessed as scalars.

Let us consider using this technique in the proposed algorithm.

We omit the moments with reading and writing, we assume that the matrix elements have already been read into the vector registers and will be written from them.

Let us have a more detailed look at item 5. We will consider the example of a 32-bit float type. In the developed method, for $m = 4$ and $n = 4$, R is also assumed as four, due to the limitation of the registers number. Thus, 4×4 elements of the first matrix have been read in item 2. This corresponds to 64 bytes, so four Q registers are required. In items 3 and 4 the same number of Q registers of the second matrix and time buffer was read out, respectively.

According to the matrix multiplication algorithm, it is necessary to multiply the first row of the second matrix by the first element of each row of the first one, the second row – by every second element, and so on. Moreover, the products obtained from the i-th row of the first matrix and the j-th column of the second matrix correspond to the element (i,j) of the temporary buffer.

Given the specifics of the actions described, a vector-to-vector operation is not appropriate, and as stated above, NEON allows getting access to an individual element. Therefore, some instructions allow performing vector-scalar operations. VMLA is one such instruction [9].

VMLA (Vector Multiple and Accumulate) – multiplies the corresponding elements of two vectors and adds the product to the corresponding elements of the result register. In the scalar-vector case, each element of the vector is multiplied by a scalar. The general syntax of the instruction is as follows:

VMLA {cond} .datatype {Qd}, Qn, Qm.

And the vector scalar option looks like:

VMLA {cond} .datatype {Qd}, Qn, Dm [x].

Let us consider each element of the structure:

– {cond} – an optional parameter, NEON allows conditional execution of instructions, cond box is clicked for the condition under which the instruction will be executed;

– datatype – a vector register element type, in our case F32, denoting a 32-bit floating point number;

– Qd, Qn, Qm – input registers for the operation, the following operation is conditionally performed: Qd + = Qn * Qm;

– Dm [x] – a scalar, for the vector-scalar operation it is important that the Q-registers are used as vectors, but the scalar gets from the D-register; x – the index of the desired element from the vector Dm.

Therefore, we assume that vectors Q4–Q7 were read from the first matrix, from the second matrix – Q8–Q11 vectors, and from the temporary buffer – Q0–Q3. Then the set of commands required to get the result will look as follows:

Multiplying the 1st row of the second matrix by the first element of each row of the first matrix with accumulation in the temporary buffer.

VMLA.F32 Q0, Q8, D8[0]

VMLA.F32 Q1, Q8, D10[0]

VMLA.F32 Q2, Q8, D12[0]

VMLA.F32 Q3, Q8, D14[0]

Multiplication of the 1st row of the second matrix by the first element of each row of the first matrix with accumulation in the temporary buffer, etc.

VMLA.F32 Q0, Q9, D8[1]

VMLA.F32 Q1, Q9, D10[1]

VMLA.F32 Q2, Q9, D12[1]

VMLA.F32 Q3, Q9, D14[1]

VMLA.F32 Q0, Q10, D9[0]

VMLA.F32 Q1, Q10, D11[0]

VMLA.F32 Q2, Q10, D13[0]

VMLA.F32 Q3, Q10, D15[0]

VMLA.F32 Q0, Q11, D9[1]

VMLA.F32 Q1, Q11, D11[1]

VMLA.F32 Q2, Q11, D13[1]

VMLA.F32 Q3, Q11, D15[1]

So, we obtained 16 vector VMLA operations. A similar solution by the conventional method would produce 64 multiplication and the same number of addition operations.

Another way to optimize is linearly reading and writing to the temporary buffer. In a simple way, for convenience, the temporary buffer corresponds to the four rows of the resulting matrix. Given that this buffer is only a temporary one, it is possible to read and write it linearly. At first glance, it is just an elementary change, when it comes to vector registers and if we consider the further processing of remainders, but, in fact, it leads to serious confusion. These

details will not be described in this publication. The main thing is to correctly understand at what point the corresponding elements for accumulation are located and what registers are to be written in sums and where. It should be noted here that NEON allows to effectively read/write vector registers with one instruction. However, there are two limitations: one instruction can write at most two Q-registers at a time; the registers should be serial. That is, reading/writing of Q1, Q3 registers with one instruction is impossible. These limitations are one of the reasons for the problems with the transition from reading the temporary buffer by 4 lines to serial.

Given that the bypass of the second matrix is performed a large number of times, the overall execution time is greatly affected by the padding size. This is a value equal to the width of the matrix subtracted from the step between the rows. Briefly, this is the size of the region of the entire matrix that does not take part in the multiplication (if the multiplication is performed on the part of the larger matrix). By reducing the ratio between the width at which the multiplication is performed and the width at which it is not performed, the rate of execution decreases. Sometimes the deterioration is such that it is quicker to make a copy of the sub-matrix into a new memory where paddings will be removed and to perform the operation without them. It is due to these reasons that one more optimization occurred: from the above data, as well as the height of the first matrix (the number of the second matrix full readouts depends on this), the conversion factor is calculated, which makes it advantageous to first make a copy to the extra memory and only then to multiply the matrices on that memory.

For the ARM architecture the proper placement of the memory reboot is very important. As practice shows, ×86 architecture types have a good auto-preloader, unlike ARM. In this processor family, correct and timely memory rebooting can result in a huge acceleration. On the contrary, if programmer makes a mistake, the performance can drop significantly. The official documentation does not give any flexible advice. It only recommends preloading with 128-byte indentation unit.

The PLD instruction performs a 64-byte preload of the transmitted pointer memory with some preset indentation unit. As mentioned above, the official documentation recommends that 128 bytes indentation should be made in advance. However, practical use shows that this is only a minimum of the real capabilities of this instruction. In different situations accelerations produce single preloads before the start of the general cycle, or in the cycle itself (not always with 128 byte indentation unit, and sometimes their number can be more than one). The PLD instruction preloads 64 bytes at once. The above design only makes 16-bytes pass, so this operation is redundant at every iteration. The simplest solution to this problem is to spin a 64-byte cycle. In fact, one such iteration will simply contain 4 blocks of operations of 16 bytes each, but this approach makes it possible to perform preloading much more efficiently.

Finally, it can be noted that due to the different behavior when bypassing the matrices (first, second and temporary), the schemes of their preload also differ.

Similarly, not only matrix multiplication on floating-point numbers is implemented, but integer 8-bit scaling as well. The option of the matrix multiplication by the transposed one and the list of vectors is also implemented (the actual implementation is one, the transposed goes to the vectors list). This option allows to read both matrices sequentially without additional memory buffer and to use a vector-vector type VMLA instruction.

The results of the comparison of the proposed method with others

The OpenCV and Eigen libraries described above were selected for the purpose of comparison. Matrix multiplication functions in all libraries (including the developed one) return the same result, so we assume that it is accurate. The data was verified by a vipmed application designed specifically to test and verify image and matrix management functions. This software allows running various mathematical functions of different libraries with the transfer of the necessary parameters, compare their results, and measure the execution time within the accuracy of a microsecond using C library function “clock”.

All measurements were made on a NVIDIA Tegra K1 chip with ARM Cortex-A15 processors that support ARMv7 and NEON.

Testing was performed on square matrices with dimensions 10×10, 100×100, 200×200, 500×500, 1,000×1,000, 2,000×2,000 and 4,000×4,000 elements with padding of 0 and 4,000 bytes on 8-bit unsigned integers (u8) and 32-bit floating point numbers (f32). Variants of ordinary matrix multiplication and matrix multiplication by the transposed one were tested.

For illustration purposes, the tables show the percentage of run time of existing OpenCV and Eigen methods from the one developed for Vipm library.

Table 1 represents multiplying the $N \times N$ matrix by the $N \times N$ matrix using 8-bit unsigned integers without a padding.

***Table* 1.** Matrix multiplication on type u8 with 0 byte padding

N	Vipm (ms)	OpenCV (ms)	Eigen (ms)	cv (%)	ei (%)
10	0.001	0.005	0.002	500	200
100	0.203	1.799	1.167	886	575
200	1.658	16.574	8.733	1000	527
500	26.487	281.808	134.164	1064	507
1000	216	2326	1064	1078	493
2000	2259	18706	8457	828	374
4000	20956	151309	67598	722	323

Table 2 represents multiplying the $N \times N$ matrix by the $N \times N$ matrix using 8-bit unsigned integers with 4,000 bytes padding.

***Table* 2.** Matrix multiplication on type u8 with 4000 bytes padding

N	Vipm (ms)	OpenCV (ms)	Eigen (ms)	cv (%)	ei (%)
10	0.002	0.005	0.003	250	150
100	0.208	1.799	1.175	865	565
200	1.672	16.721	8.748	1000	523
500	28.331	281.932	134.221	995	474
1000	216	2341	1067	1085	494
2000	2336	18709	8470	801	363
4000	20957	151314	67651	722	323

Table 3 represents multiplying the $N \times N$ matrix by the $N \times N$ matrix using 32-bit floating-point numbers without padding.

***Table* 3.** Matrix multiplication on type f32 with 0 byte padding

N	Vipm (ms)	OpenCV (ms)	Eigen (ms)	cv (%)	ei (%)
10	0.001	0.003	0.002	300	200
100	0.223	1.727	0.309	774	139
200	1.708	15.725	2.301	921	135
500	26.170	264.676	39.118	1011	149
1000	248	2302	318	926	128
2000	2160	18563	2475	860	115
4000	17938	150348	19131	838	107

Table 4 represents multiplying the $N \times N$ matrix by the $N \times N$ matrix using 32-bit floating-point numbers with 4,000 bytes padding.

***Table* 4.** Matrix multiplication on type f32 with 4000 bytes padding

N	Vipm (ms)	OpenCV (ms)	Eigen (ms)	cv (%)	ei (%)
10	0.002	0.003	0.002	150	100
100	0.231	1.933	0.327	837	142
200	1.746	15.725	2.361	901	135
500	26.728	265.320	39.345	993	147
1000	250	2303	319	920	127
2000	2210	18576	2477	840	112
4000	18630	150349	19140	807	103

Table 5 represents multiplication of the $N \times N$ matrix by the transposed $N \times N$ matrix using 8-bit unsigned integers without padding.

***Table* 5.** Matrix multiplication (transposed multiplicand) on type u8 with 0 byte padding

N	Vipm (ms)	OpenCV (ms)	Eigen (ms)	cv (%)	ei (%)
10	0.001	0.007	0.002	700	200
100	0.264	2.243	1.171	850	444
200	2.184	16.969	8.745	777	400
500	33.740	272.955	135.278	809	401
1000	246	2154	1069	877	435
2000	2035	17105	8473	841	416
4000	16002	136829	67651	855	423

Table 6 represents multiplying the $N \times N$ matrix by the transposed $N \times N$ matrix using 8-bit unsigned integers with 4,000 bytes padding.

***Table* 6.** Matrix multiplication (transposed multiplicand) on type u8 with 4000 bytes padding

N	Vipm (ms)	OpenCV (ms)	Eigen (ms)	cv (%)	ei (%)
10	0.002	0.007	0.003	350	150
100	0.265	2.246	1.172	848	442
200	2.187	17.007	8.746	778	400
500	33.815	272.958	135.370	807	400
1000	247	2181	1069	885	434
2000	2044	17106	8475	837	415
4000	16405	136841	67651	834	412

Table 7 represents multiplying the $N \times N$ matrix by the transposed $N \times N$ matrix using 32-bit floating-point numbers without padding.

***Table* 7.** Matrix multiplication (transposed multiplicand) on type f32 with 0 byte padding

N	Vipm (ms)	OpenCV (ms)	Eigen (ms)	cv (%)	ei (%)
10	0.001	0.004	0.002	400	200
100	0.245	2.141	0.313	874	128
200	1.786	17.022	2.298	953	129
500	26.884	269.551	38.846	1003	144
1000	275	2166	317	789	115
2000	2425	17275	2466	712	102
4000	18909	138233	19104	731	101

Table 8 represents multiplying the $N \times N$ matrix by the transposed $N \times N$ matrix using 32-bit floating-point numbers with 4,000 bytes padding.

***Table* 8.** Matrix multiplication (transposed multiplicand) on type f32 with 4000 bytes padding

N	Vipm (ms)	OpenCV (ms)	Eigen (ms)	cv (%)	ei (%)
10	0.002	0.004	0.002	200	100
100	0.245	2.142	0.315	874	129
200	1.789	17.042	2.301	953	129
500	27.042	269.964	38.867	998	144
1000	290	2167	317	749	110
2000	2426	17277	2466	712	102
4000	18996	138337	19114	728	101

The results show that in all cases OpenCV works about in the same way (time only increases with image size, which is quite logical), and options of the floating point are much more efficiently implemented in Eigen. The algorithm developed is also executed at about the same rate in all conditions, but generally, runs much faster than OpenCV – sometimes ten-fold. Eigen is also much more efficient than OpenCV, but the 8-bit unsigned numbers are still significantly inferior to the developed algorithm. It can be assumed that this library does not have a direct implementation for this type of values, so it is executed by additional conversions to floating point numbers, which causes a delay. When using floating-point numbers, the algorithm for the given values is still faster. For higher values, the efficiency of the Eigen library approaches the developed algorithm.

Testing was performed on various sizes and types using vipmed software (designed for in-house use by VIT).

Conclusions

The proposed matrix multiplication method gives the expected speed of matrix multiplication operations (not slower than existing analogues) and has passed evaluation test for use.

The efficiency of the method is tested by practical application.

The method is used in the corporate library of one of the leading companies in Ukraine.

The article can help to understand how preloaders and caches are work and how to use them in operations such as matrix multiplication.

This algorithm do not consider that block sizes are depend on cache size. Therefore, speed on machines with a different cache size can be unexpected.

For future work, we need to explore more existed libraries with matrix multiplication. It will help understand in what way we can move next.

References

[1] *Application of Matrices in Real-Life* [Online]. Available: https://www.ukessays.com/essays/mathematics/application-of-matrices-in-real-life-problems.php

[2] Le Gall, François, "Powers of tensors and fast matrix multiplication", in *Proc. 39th Int. Symp. Symbolic and Algebraic Computation*, Kobe, Japan, 2014, pp. 157–164.

[3] V. Strassen, "Gaussian elimination is not optimal", *Numer. Math.*, vol. 13, no. 4, pp. 354–356, 1969.

[4] S. Robinson, "Toward an optimal algorithm for matrix multiplication", *SIAM News*, vol. 38, no. 9, 2005.

[5] C.L. Lawson *et al.*, "Basic linear algebra subprograms for FORTRAN usage", in *ACM Trans. Math. Software*, 1979, pp. 308–323.

[6] OpenCV official site [Online]. Available: https://opencv.org/

[7] *Eigen Official Site* [Online]. Available: http://eigen.tuxfamily.org/

[8] *ARM Official Site* [Online]. Available: https://developer.arm.com/

[9] *ARM Information Center* [Online]. Available: https://infocenter.arm.com/

І.А. Дичка, Д.А. Вінник, Ю.В. Бухтіяров, В.Я. Юрчишин

МЕТОД РЕАЛІЗАЦІЇ ШВИДКОГО МАТРИЧНОГО МНОЖЕННЯ ПІД АРХІТЕКТУРУ ARM ІЗ ВИКОРИСТАННЯМ SIMD-ІНСТРУКЦІЇ

Проблематика. Матричне множення є досить складним алгоритмом із великою кількістю операцій. Додатковою проблемою також є нелінійний обхід матриць по пам'яті. Операція матричного множення широко використовується в різних сферах, таких як нейронні мережі, розв'язки систем лінійних рівнянь, матричні перетворення тощо. Тож важливо розробити метод матричного множення, що враховуватиме проблеми з розташуванням матриць у пам'яті, а також ефективно розпоряджатиметься даними при їх повторному використанні.

Мета дослідження. Розробити метод швидкого матричного множення двох матриць, множення матриці на транспоновану та на список векторів (у т.ч. окремий випадок для одного вектора); реалізувати його у вигляді функції з оптимізацією для процесорів архітектури ARM. Функція має вміти працювати з різними типами даних та з підматрицями. Цілочисловий результат може бути масштабований.

Методика реалізації. Головними ідеями розробленого методу є одночасних прохід декількома рядками/стовпчиками вхідних матриць та їх розбиття на блоки, що дасть алгоритму змогу деякий час працювати на одній і тій самій пам'яті. Для реалізації було вибрано мову програмування C. Для збільшення продуктивності використано SIMD-інструкції. Для ефективної реалізації під архітектуру ARM також необхідно правильно організувати роботу з попереднім завантаженням пам'яті.

Результати дослідження. Реалізовано функцію, що виконує матричне множення за розробленим методом із необхідними параметрами. Перевірки на різних розмірах і типах показали, що реалізована функція є швидшою за аналоги з бібліотек OpenCV2 та Eigen 3. Тестування відбувалося за допомогою утиліти vipmed для запусків і замірів характеристик, розробленої для корпоративного користування у компанії VIT.

Висновки. Запропонований метод множення матриць дає очікуване прискорення операції множення матриць, пройшов оціночний тест на використання та відповідає заданим у меті вимогам. Для подальшої роботи необхідно детальніше дослідити вплив кеша різного рівня та порівняти з іншими існуючими бібліотеками.

Ключові слова: матричне множення; архітектура ARM; векторні операції; транспонування матриці.

И.А. Дычка, Д.А. Винник , Ю.В. Бухтияров, В.Я. Юрчишин

МЕТОД РЕАЛИЗАЦИИ БЫСТРОГО МАТРИЧНОГО УМНОЖЕНИЯ ПОД АРХИТЕКТУРУ ARM С ИСПОЛЬЗОВАНИЕМ SIMD-ИНСТРУКЦИИ

Проблематика. Матричное умножение является достаточно сложным алгоритмом с большим количеством операций. Дополнительной проблемой также является нелинейный обход матриц по памяти. Операция матричного умножения широко используется в различных сферах, таких как нейронные сети, решения систем линейных уравнений, матричные преобразования и т.п. Поэтому важно разработать метод матричного умножения, который будет учитывать проблемы расположения матриц в памяти, а также эффективно будет распоряжаться данными при их повторном использовании.

Цель исследования. Разработать метод быстрого матричного умножения двух матриц, умножения матрицы на транспонированную и на список векторов (в т.ч. частный случай для одного вектора); реализовать его в виде функции с оптимизацией для процессоров архитектуры ARM. Функция должна уметь работать с различными типами данных и с подматрицамы. Целочисленный результат может быть отмасштабирован.

Методика реализации. Главными идеями разработанного метода является одновременный проход несколькими строками/столбцами входных матриц и их разбиение на блоки, что позволит алгоритму некоторое время работать на одной и той же памяти. Для реализации был выбран язык программирования C. Для увеличения производительности использованы SIMD-инструкции. Для эффективной реализации под архитектуру ARM также необходимо правильно организовать работу с предварительной загрузкой памяти.

Результаты исследования. Реализована функция, которая выполняет матричное умножение по разработанному методу с необходимыми параметрами. Проверки на разных размерах и типах показали, что реализованная функция быстрее аналогов из библиотек OpenCV2 и Eigen 3. Тестирование проходило с помощью утилиты vipmed для запусков и замеров характеристик, разработанной для корпоративного пользования в компании VIT.

Выводы. Предложенный метод умножения матриц дает ожидаемое ускорение операции умножения матриц, прошел оценочный тест на использование и соответствует заданным в цели требованиям. Для дальнейшей работы необходимо подробнее исследовать влияние кэша разного уровня и сравнить с другими существующими библиотеками.

Ключевые слова: матричное умножение; архитектура ARM; векторные операции; транспонирование матрицы.

Рекомендована Радою
факультету прикладної математики
КПІ ім. Ігоря Сікорського

Надійшла до редакції
03 лютого 2020 року

Прийнята до публікації
05 червня 2020 року