electronics

MDPI

Article

An Approach for Matrix Multiplication of 32-Bit Fixed Point Numbers by Means of 16-Bit SIMD Instructions on DSP

Ilia Safonov *, Anton Kornilov and Daria Makienko

Computer Science and Control Systems Department, National Research Nuclear University MEPhI, Kashirskoye Highway, 31, 115409 Moscow, Russia

* Correspondence: ilia.safonov@gmail.com

Abstract: Matrix multiplication is an important operation for many engineering applications. Sometimes new features that include matrix multiplication should be added to existing and even out-of-date embedded platforms. In this paper, an unusual problem is considered: how to implement matrix multiplication of 32-bit signed integers and fixed-point numbers on DSP having SIMD instructions for 16-bit integers only. For examined tasks, matrix size may vary from several tens to two hundred. The proposed mathematical approach for dense rectangular matrix multiplication of 32-bit numbers comprises decomposition of 32-bit matrices to matrices of 16-bit numbers, four matrix multiplications of 16-bit unsigned integers via outer product, and correction of outcome for signed integers and fixed point numbers. Several tricks for performance optimization are analyzed. In addition, ways for block-wise and parallel implementations are described. An implementation of the proposed method by means of 16-bit vector instructions is faster than matrix multiplication using 32-bit scalar instructions and demonstrates performance close to a theoretically achievable limit. The described technique can be generalized for matrix multiplication of n-bit integers and fixed point numbers via handling with matrices of n/2-bit integers. In conclusion, recommendations for practitioners who work on implementation of matrix multiplication for various DSP are presented.

Keywords: GEMM; SIMD instructions; outer product; fixed point; DSP; parallel processing

Citation: Safonov, I.; Kornilov, A.; Makienko, D. An Approach for Matrix Multiplication of 32-Bit Fixed Point Numbers by Means of 16-Bit SIMD Instructions on DSP. *Electronics* **2023**, *12*, 78. https://doi.org/10.3390/electronics12010078

Academic Editors: Juan M. Corchado, Byung-Gyu Kim, Carlos A. Iglesias, In Lee, Fuji Ren and Rashid Mehmood

Received: 11 November 2022
Revised: 21 December 2022
Accepted: 22 December 2022
Published: 25 December 2022

1. Introduction

General matrix to matrix multiplication (GEMM) is the cornerstone for a wide range of algorithms used in practical applications. There are plenty of high-optimized software libraries having GEMM implementation according to the basic linear algebra subprograms (BLAS) [1] specification. Nevertheless, an emergence of new computing devices as well as development for existing and even out-of-date embedded platforms require elaboration on the most appropriate ways for matrix multiplication.

In particular, such tasks appeared during development for the embedded platform of mobile multi-functional printers (MFP). One of the main requirements for mobile electronics is low power consumption, which guarantees a long battery life. On the other hand, the hardware and firmware of such devices should be sufficiently universal to be able to flexibly adjust to various tasks. The given embedded platform has several digital signal processor (DSP) intellectual property (IP) cores for system-on-chip (SoC) and random-access memory (RAM) accessible for all cores. Below this, RAM is called a global memory. The IP core contains a low-power DSP with a frequency of 400 MHz intended for image processing, 512 Mb of tightly coupled memory (TCM), so-called scratchpad, and direct memory access (DMA) controller for access to global memory. Below TCM, it is called a local memory. The given DSP has a very long instruction word (VLIW), single instruction, multiple data (SIMD) architecture, instructions for processing of 32-bit integer scalars, and vector instructions for 16-bit integers. There is no support for floating point data.

A DSP VLIW pipeline allows the execution of four various instructions at the same time in one cycle of the processor. For example, loading data from TCM (for addresses divisible

by 16), multiplication, addition, and storing data to TCM can be executed simultaneously. On the other hand, four multiplications or four instructions for loading data from TCM cannot be performed at the same time. A compiler is fully responsible for the generation of a code, which uses the pipeline effectively, because the given DSP has no scheduler. Therefore, an efficiency of different algorithms and their implementations in C/C++ programming language significantly differ.

Said IP core allows effective implementation of a large number of image processing algorithms for scanning and printing because a majority of such techniques operate with 8-bit and 16-bit integers in row-wise access mode. There are some problems, which require intensive usage of multiplication of dense rectangular matrices containing floating point numbers, where matrix size varies from several tens to about two hundred [2,3]. Due to the absence of floating point operations, calculations in 32-bit fixed point numbers can be used [4]. Of course, an application of fixed point arithmetic has restrictions connected with an accumulation of calculation errors and numerical stability of algorithms [5]. Matrix multiplication via 32-bit scalars has unacceptably long runtime. It is necessary to employ vector instructions, but the given DSP has SIMD instructions for 16-bit integers only.

The main contribution of this paper is an algorithm for dense rectangular matrix multiplication of 32-bit signed fixed point numbers and integers by means of vector instructions for 16-bit integers. Surely, a use of 32-bit vector instructions is preferable, but in the case of absence of SIMD instructions of required bit-length, the proposed approach may be an apposite workaround. The proposed technique can be easily generalized for n-bit numbers and $n/2$-bit SIMD instructions. For instance, our approach can be employed for realization of matrix multiplication of 64-bit fixed point numbers via vector instructions for 32-bit integers. The proposed algorithm satisfies the important requirements for the matrix multiplication algorithms, as it supports an ability to process matrices in block-wise manner and allows parallel processing in several identical IP blocks of SoC. Used for experiments, the embedded platform was a prototype and the considered IP core is not available on the market at present time. Nevertheless, the proposed algorithm does not use any specific features of the given DSP, which has ordinary architecture, and a skilled person is able to generalize our approaches for many modern signal and graphical processors.

Among other findings of this paper, we would like to emphasize the following:

- an application of outer product for matrix multiplication is underestimated by researchers frequently;
- Strassen's and other galactic algorithms enabling a low asymptotic complexity are effective for huge matrices only;
- although all numerical results in this paper were obtained for square matrices, the proposed method smoothly operates with rectangular matrices and with vector to matrix multiplication;
- sometimes manual code optimization can provide significant performance improvement;
- matrix multiplication is easily parallelizable, but overhead should be estimated for each specific computing platform.

This paper is organized in the following manner. Section 2 gives a brief review of the matrix multiplication problem both as it is discussed in theoretical computer science and as it is implemented on various platforms in practice. Section 3 describes approaches of multiplications of n-bit scalars via n/2-bit ones. In Section 4, an applicability for our case of various methods for matrix multiplication is discussed. Multiplication via outer product outperforms the traditional way via dot product. Section 5 is devoted to the description of the proposed matrix multiplication algorithm for fixed point numbers by means of vector instructions. One can find tricks for speeding up processing on DSP in Section 6. Techniques for parallel processing are considered in Section 7. Section 8 presents performance of the proposed algorithm in comparison with two implementations via 32-bit scalar instructions as well as with the theoretical limit. Section 9 summarizes our findings and contains recommendations for practitioners who work on implementation of matrix multiplication for various computing platforms.

2. Related Works

Matrix multiplication is one of the most fundamental problems in theoretical and practical computer science. Let us start from theoretical estimation of asymptotic complexity in Big O notation. The well-known, so-called school method for matrix multiplication employs the dot (or inner) product of rows of the leftmost multiplier and columns of the rightmost matrix. Many tutorials on linear algebra describe another way for GEMM via outer (or tensor) product [6]. The complexity of both approaches for square matrix at size N by N is $O(N^3)$.

Strassen depicts the algorithm having the complexity $O(N^{2.808})$ [7]. Strassen's algorithm minimizes the number of the multiplication instructions sacrificing the number of add instructions and data locality. In practical implementations, an advantage of Strassen's algorithm in processing time is starting from N equal to several hundred or even more. In addition, it is worth noting that data locality is a key factor of significant profit from SIMD instructions usage. This fact is proved by existing examples of an application of vector instructions for implementation of Strassen's algorithm, which demonstrates modest improvement in comparison with scalar ones [8]. Another shortcoming of Strassen's algorithm in comparison with multiplication via inner or outer product is numerical instability for big matrices. Several investigations are devoted to ways to improve its numerical stability [9,10].

There are several improvements of Strassen's algorithm, for instance: Coppersmith and Winograd depicted an algorithm with a complexity of $O(N^{2.376})$ [11]; Alman and Williams described an approach with $O(N^{2.3729})$ [12]; recently, Duan et al. proposed an algorithm with $O(N^{2.3719})$ [13]. However, these improvements are not used in practice yet, because they relate to so-called galactic algorithms, that is, they outperform other alternatives only for very huge matrices.

Further, we review approaches for implementation of matrix multiplications on various processing units: central processing unit (CPU), graphical processing unit (GPU), DSP, field-programmable gate array (FPGA), and specialized circuit. Aberdeen and Baxter [14] demonstrate two times speeding up in matrix multiplication by means of dot product on a CPU Intel Pentium III due to usage of SSE (SIMD streaming extensions) instructions. Kelefouras et al. [15] describe implementation of GEMM via dot product by wise usage of SSE instructions and decreasing the number of the data cache accesses. In addition, a tiling of matrices for parallel processing by multicore CPU is considered. Hemeida et al. [16] consider an application of advanced vector extension (AVX), a multithreading and memory access optimization for matrix multiplication speeding up. Unfortunately, it is hard to estimate numerically a positive impact of SIMD instructions usage among all optimization tricks. Soliman and Ahmed [8] explore an application of multi-processing, multithreading, and SIMD instruction for implementation of GEMM on a cluster of computers with Intel Xeon processors. Strassen's algorithm and traditional approach via dot product were implemented and investigated. It was pointed out that, for both cases, application of SIMD instructions provides insignificant performance improvement only. Pradyumna [17] compares performance of various implementations of matrix multiplication via dot product and Strassen's algorithm. AVX instructions are able to speed up several times GEMM via dot product. For Strassen's algorithm, performance gain is negligible. It is worth noting, a modern CPU has a rich set of SIMD instructions that includes data rearrange, sum reduction, etc. An application of such a wide set of vector instructions allows the realization of matrix multiplication via inner product rather effectively.

Yang et al. [18] emphasize that most of the works are on evaluating and optimizing large-scale matrix multiplication, but how much the small-scale matrix multiplication is underexposed. The authors define small-scale matrix multiplication as multiplication of matrices having small sizes (less than one hundred according to demonstrated results) or in the case where one matrix dimension is significantly smaller than the other. A use of four mainstream open-source BLAS libraries, OpenBLAS [19], BLIS [20], BLASFEO [21], and Eigen [22], on ARMv8-based many-core architecture is considered. Performance

bottlenecks are highlighted and the following directions for optimization are investigated: mitigating the data packing overhead; processing the edge cases properly; selecting a suitable micro-kernel; and adopting an appropriate parallelization technique.

Modern platforms that combine CPU and FPGA allow effective GEMM implementations due to customizable configuring FPGA for the dot product of a single precision floating point and reduced precision workloads [23]. For multiplication of very large matrices (where matrix sizes are greater than 10,000), distributed algorithms intended for the cluster of computers were described recently [24,25].

One of the existing trends is a transfer of intensive computations to GPUs. The company NVIDIA distributes the cuBLAS library [26], which contains GPU-enabled functions realized by means of compute unified device architecture (CUDA). These functions correspond to a BLAS specification including GEMM for different integer and floating point data types. In addition, NVIDIA developed CUTLASS [27], which is a collection of CUDA C++ template abstractions for implementing GEMM and related computations. Matrix multiplication in CUTLASS is realized via outer product. Recently, several publications appeared that describe Strassen's algorithm implementations on CUDA slightly outperforming cuBLAS for rather large matrices. Huang et al. [28] describe an algorithm, which utilizes both the memory and thread hierarchies on GPUs and reuses shared memory. Krishnan and Goswami [29], following the CUTLASS guidelines, eliminate the requirement of additional workspace associated with Strassen's algorithm by organizing and restructuring operations in stages, where multiple operations in the same stage can be executed in parallel.

Ali et al. [30] describe implementation of BLAS functions on multi-core DSP. Performance of GEMM is optimized based on partitioning of both multipliers and its parallel processing. Yin et al. [31] depict an approach for automatic generation of assembly micro-kernels, allowing irregular-shaped matrix multiplication by means of auto-tuning of block sizes, optimal data loading by direct memory access (DMA), and parallelization strategies. Morad et al. [32] investigate new DSP architecture for efficient dense and sparse matrix multiplication due to loading data for SIMD instruction from both rows and columns.

There are several publications about implementation of matrix multiplication algorithms on various specific FPGAs [33–36] and by means of specialized architecture for circuits [37,38]. Kamranfar et al. [37] depict a configurable linear systolic architecture. Shanmugakumar et al. [38] describe architecture, which utilizes a carry-save adder tree multiplier for multiplication and a carry-lookahead adder for performing addition at the final stage.

Summarizing, the majority of existing investigations are devoted to algorithms for multiplication of huge matrices by means of Strassen's and similar algorithms, while handling of matrices having sizes about one hundred is underexposed. Almost all publications consider the school method of matrix multiplication via dot product, and only a few use multiplication via outer product. Undoubtedly, matrix partitioning for block-wise parallel processing on several processing units is a valuable technique for performance improvement. To the best of our knowledge, there is no well-known method for matrix multiplication of n-bit numbers by means of handling with n/2-bit numbers.

3. Decomposition of Multiplication for Scalars

Let us start from considering approaches for multiplication of 32-bit scalar values by means of operations with 16-bit numbers. Anatoly Karatsuba proposed an algorithm for multiplication of two n-bit scalars with asymptotic complexity of $O(n^{log_2 3})$ [39,40]. Karatsuba's approach is described in a monograph by Donald Knuth in detail [41]. n-bit unsigned integers U with bits $(u_n, u_{n-1}, \ldots u_1)$ and V $(v_n, v_{n-1}, \ldots v_1)$ can be defined as:

$$U = 2^{\frac{n}{2}} U_H + U_L \tag{1}$$

$$V = 2^{\frac{n}{2}} V_H + V_L \tag{2}$$

where U_H (u_n, u_{n-1}, ... $u_{n/2+1}$) is the high (or the most significant) half of n-bit U, and U_L ($u_{n/2}$, $u_{n/2-11}$, ... u_1) is the low (or the less significant) half of U; similarly, V_H and V_L are high and low halves of V, respectively. Then, UV multiplication comprises four multiplications for $n/2$-bit unsigned integers as well as several arithmetic shifts and additions:

$$UV = 2^n U_H V_H + 2^{\frac{n}{2}}(U_H V_L + U_L V_H) + U_L V_L \quad (3)$$

After rearranging of the terms, there are three multiplications for $n/2$-bit unsigned integers:

$$UV = \left(2^n + 2^{\frac{n}{2}}\right) U_H V_H + \left(2^{\frac{n}{2}} + 1\right) U_L V_L + 2^{\frac{n}{2}}(U_H - U_L)(V_L - V_H). \quad (4)$$

Depending on the relative cost of processor instructions for each specific calculation platform, an application of Formula (3) or (4) can be preferable. Formula (4) has a larger number of additions, subtractions, and shifts in comparison with Formula (3), but uses three multiplications instead of four.

Warren's book [42] declares three ways for adjustment of multiplication outcome for signed integers after a use of Formula (3) or (4):

1. obtaining the absolute values of operands, then performing unsigned multiplication for low and high halves and then changing the sign of the result depending on the signs of operands;
2. for Formula (3), using multiplication of unsigned by unsigned integers for low halves of operands, multiplication of signed by signed integers for high halves, and multiplications of signed by unsigned integers for high and low halves; this way requires a sign extension instruction;
3. correction of multiplication outcome for unsigned integers by means of subtractions of operands depending on their sign.

Let us see approach #3 in detail. A signed integer U' is considered as an unsigned integer $U + 2^n U_{n-1}$, where U_{n-1} is a one in the case where U is negative, and it is a zero in the case where U is positive. In a similar manner, signed integer V' is equal to $V + 2^n V_{n-1}$. Then, the multiplication $U'V'$ corresponds to the expression:

$$(U + 2^n U_{n-1})(V + 2^n V_{n-1}) = UV + 2^n(VU_{n-1} + UV_{n-1}) + 2^{2n} U_{n-1} V_{n-1}. \quad (5)$$

To obtain an outcome of the product of the signed integers UV, we subtract from the result of the unsigned multiplication the second and third terms of the right-hand side of Formula (5). It is worth noting that, when multiplying n-bit integers, the third term is ignored, since the multiplication outcome is completely in $2n$ bits, so there is no necessity to calculate bits higher than $2n - 1$. However, in a general case it is necessary to take into account the third term in Formula (5) for matrix multiplication because elements of the resulting matrix in matrix product result might be greater than 2^{2n}.

4. Matrix Multiplication via Outer Product

Matrix multiplication via dot (or inner) product of rows of the leftmost multiplier and columns of the rightmost matrix is the most popular method. However, the technique is ill-suited for an implementation by SIMD instructions of DSP, because elements of columns do not lie in the memory sequentially. A transposition of the rightmost matrix or a copying of elements of columns to a separate array is required. Both actions lead to noticeable overhead in spite of the fact that transposition by Eklundh's algorithm [43] operates quite fast on considered DSP.

Sometimes in matrix multiplication the rightmost matrix should be transposed according to the requirements of a linear algebra algorithm being implemented. In this case, it is preferable not to transpose the rightmost matrix explicitly, but to perform matrix multiplication via dot product of rows for both matrices. However, a usage of dot product requires zero-padding of rows and instruction for effective summation of vector register

elements. For example, if the matrix width is 100 elements and the size of the vector register is 32 elements, the vector register is filled 3 times completely, and for the 4th time there is only the last 4 elements from the matrix row, the next 28 elements of the register are required to be filled with zeros. In addition, there should be a sum reduction instruction for summing the elements of the vector register in order to effectively implement dot product. Unfortunately, the given DSP has no similar instruction for the vector register.

In the case of SIMD instructions, it is preferable to implement matrix multiplication via a sum of outer (or vector tensor) products of columns of the leftmost operand and rows of the rightmost matrix. The outer product of column a by row b forms the following matrix:

$$\begin{pmatrix} a_1 \\ a_2 \\ \vdots \\ a_m \end{pmatrix} \otimes \begin{pmatrix} b_1 & b_2 & \cdots & b_n \end{pmatrix} = \begin{pmatrix} a_1b_1 & a_1b_2 & \cdots & a_1b_n \\ a_2b_1 & a_2b_2 & \cdots & a_2b_n \\ \vdots & \vdots & \ddots & \vdots \\ a_mb_1 & a_mb_2 & \cdots & a_mb_n \end{pmatrix}. \tag{6}$$

The leftmost matrix in a product can be considered as a vector consisting of the columns, and the rightmost matrix as a vector consisting of rows. Then, the matrix multiplying is the sum of the results of the outer products of i-th vectors of the multipliers:

$$\begin{pmatrix} \vec{a}_1 & \vec{a}_2 & \cdots & \vec{a}_n \end{pmatrix} \begin{pmatrix} \vec{b}_1 \\ \vec{b}_2 \\ \vdots \\ \vec{b}_n \end{pmatrix} = \sum_{i=1}^{n} \vec{a}_i \otimes \vec{b}_i. \tag{7}$$

One can see an example of matrix multiplication for matrix dimensions 3 by 3 and 3 by 2 by means of outer product:

$$\begin{aligned} &\begin{pmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{pmatrix} \begin{pmatrix} b_{11} & b_{12} \\ b_{21} & b_{22} \\ b_{31} & b_{32} \end{pmatrix} = \\ &\begin{pmatrix} a_{11} \\ a_{21} \\ a_{31} \end{pmatrix} \otimes \begin{pmatrix} b_{11} & b_{12} \end{pmatrix} + \begin{pmatrix} a_{12} \\ a_{22} \\ a_{32} \end{pmatrix} \otimes \begin{pmatrix} b_{21} & b_{22} \end{pmatrix} + \begin{pmatrix} a_{13} \\ a_{23} \\ a_{33} \end{pmatrix} \otimes \begin{pmatrix} b_{31} & b_{32} \end{pmatrix} \\ &\qquad = \begin{pmatrix} a_{11}b_{11} & a_{11}b_{12} \\ a_{21}b_{11} & a_{21}b_{12} \\ a_{31}b_{11} & a_{31}b_{12} \end{pmatrix} + \begin{pmatrix} a_{12}b_{21} & a_{12}b_{22} \\ a_{22}b_{21} & a_{22}b_{22} \\ a_{32}b_{21} & a_{32}b_{22} \end{pmatrix} \\ &\qquad + \begin{pmatrix} a_{13}b_{31} & a_{13}b_{32} \\ a_{23}b_{31} & a_{23}b_{32} \\ a_{33}b_{31} & a_{33}b_{32} \end{pmatrix} \\ &\qquad = \begin{pmatrix} (a_{11}b_{11} + a_{12}b_{21} + a_{13}b_{31}) & (a_{11}b_{12} + a_{12}b_{22} + a_{13}b_{32}) \\ (a_{21}b_{11} + a_{22}b_{21} + a_{23}b_{31}) & (a_{21}b_{12} + a_{22}b_{22} + a_{23}b_{32}) \\ (a_{31}b_{11} + a_{32}b_{21} + a_{33}b_{31}) & (a_{31}b_{12} + a_{32}b_{22} + a_{33}b_{32}) \end{pmatrix} \end{aligned} \tag{8}$$

It is worth noting that, when multiplying matrices of n/2-bit numbers, the elements of the resulting matrix, in general, require more than n bits. A so-called wide vector register should be used for accumulation of outer product sums. The given DSP has such a wide vector register for multiplication of 16-bit vectors.

A usage of outer product has the following advantages:

- access is performed for sequential memory cells;
- there is no necessity for zero-padding;
- there is no necessity for summation of vector register elements.

Let us compare performance of matrix multiplication by means of 16-bit vector instructions via inner products and via outer products for matrices N by N for 16-bit unsigned

integers. For performance evaluation, we used an accurate software simulator that allows us to estimate processing time in a sole IP-core. Figure 1 demonstrates the results of the comparison. A realization via outer products significantly outperforms the alternative solution even without taking into account time required for transposition of the rightmost matrix.

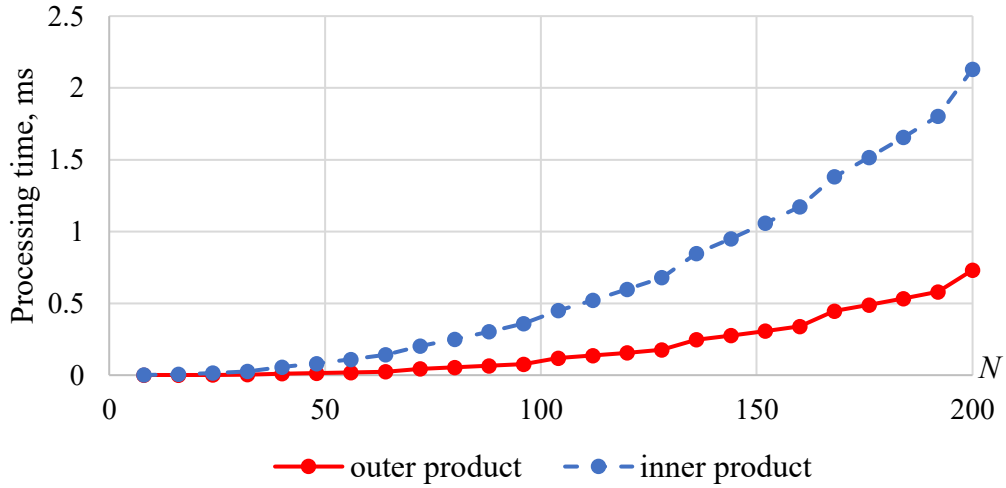

Figure 1. Processing time for matrix multiplication of 16-bit unsigned integers via inner and outer products depending on matrix size.

5. Matrix Multiplication of 32-Bit Numbers via 16-Bit Vector Instructions

In this section, without loss of generality, we describe the method of multiplying rectangular matrices containing 32-bit fixed-point numbers by means of 16-bit vector instructions of DSP. One can find a foundation of the arithmetic of numbers with a fixed point in [4]. In this paper, we handle mostly 16.16 fixed point numbers, where the integer part is in 16 of the most significant bits, and the fractional part is in 16 of the less significant bits. For a product of 16.16 fixed-point numbers, the first step is multiplication of 32-bit multipliers considered as signed integers, and then performing an arithmetic shift to the right by 16, that is, the number of bits of the fractional part. In a general case, the outcome of multiplication of 32-bit integers has 64-bit-length. For the case when the result of multiplication is 16.16 fixed-point numbers, 16 low bits from said 64 bits are omitted due to arithmetic shift to the right, and 16 high bits are clipped. Avoiding a situation where high 16-bits contain important data and cannot be clipped is the obligation of a programmer.

As a start, let us combine it all together: the matrix product and multiplication of 32-bit unsigned integers via 16-bit arithmetic operations. There are two approaches for its implementation by SIMD instructions. The first one is decomposition of 32-bit numbers onto vector registers of 16-bit low and high halves and calculation with those registers according to Formula (3) in the scope of a sole matrix multiplication procedure by means of outer product. We estimated the performance of this approach. For matrices 128×128 it operates only 30% faster in comparison with multiplication by means of 32-bit scalar instructions. This is rather slow. We analyzed our program using a simulator and a profiler in order to explore the causes of the delays. This showed that this approach does not employ the VLIW pipeline effectively; only a few instructions execute simultaneously.

The second approach is decomposition of both matrices A and B of 32-bit integers onto matrices of 16-bit numbers A_H, A_L and B_H, and B_L, where elements of A_L and B_L contain low hales of corresponding elements in matrices A and B and elements of A_H and B_H contain high hales of corresponding elements in matrices A and B, then performing four matrix multiplications for 16-bit numbers between those matrices via outer products. Outcomes of four matrix multiplications are combined to the final matrix of 32-bit integers. Matrix multiplications A_iB_k do not depend on each other's operations and can be implemented in parallel. An application of Formula (3) is better parallelizable for several IP cores, which is

why we prefer Formula (3) rather than (4). In the case of the sole DSP core, and depending on the size of matrices, calculations according to formula (4) may be preferable.

Figure 2 shows a plot of processing time depending on matrix size for the sole matrix multiplication of 32-bit numbers, where 32-bit numbers are multiplied via 16-bit SIMD instructions, and four matrix multiplications were conducted for 16-bit numbers, including decomposition 32-bit matrices to 16-bit ones and composition back. The second way is much faster.

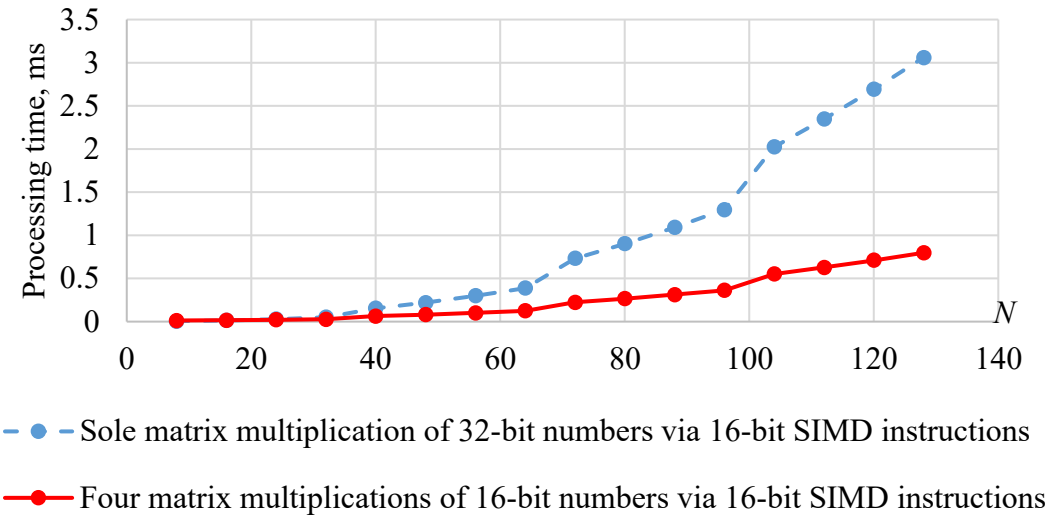

Figure 2. Processing time for matrix multiplication of 32-bit unsigned integers via decomposition on 16-bit vector registers in the scope of a sole matrix multiplication procedure and via four multiplications of 16-bit matrices.

Further, let us modify the second approach for 32-bit fixed point numbers. Multiplication of fixed point numbers can be considered as multiplication of the signed integers and an arithmetic shift to the right on the length (in bits) of a fractional part. Thus, it is necessary to extend the algorithm of matrix multiplication of unsigned integers to a method for matrices of signed integers. Existing approaches for such extensions for scalars were enumerated above in Section 3. Way #1 is unfeasible for matrix multiplication because each element of the final matrix is the sum of products of elements of matrix operands. The given DSP does not have all the necessary vector instructions for way #2. Way #3 is feasible.

Finally, we have the following algorithm for A and B matrix multiplication. Figure 3 shows components of matrix multiplication of 32-bit signed integers via matrix multiplication of 16-bit signed integers. For the considered range of matrix dimensions (less than two hundred), a matrix multiplication of 16-bit numbers produces a matrix of 48-bit numbers. We designate matrices with 16-bit halves of 32-bit numbers from the A and B matrix by indices: L-low, H-high. The first step is decomposition of A on A_L and A_H, and B on B_L and B_H. Four matrix multiplications of A_LB_L, A_HB_L, A_LB_H, and A_HB_H are performed via outer products independently from each other. A_LB_L corresponds to a lower 48 bits of outcome. Products A_LB_H and A_HB_L should be considered as shifted 16 bits to the left; accordingly, they correspond to the range from 63 to 16 bit. A_HB_H should be considered as shifted 32 bits to the left; accordingly, it corresponds to the range from 79 to 32 bit. The maximum length of elements of the resulting matrix is 80 bits. For obtaining of the resulting matrix, it is necessary to perform summation of the shifted outcomes of products A_LB_L, A_HB_L, A_LB_H, and A_HB_H, and to take into account the sign to subtract shifted matrices D_a, D_b, and S, where D_a, D_b, and S are formed according to the following rules:

$$\begin{gathered} \text{if } A(i,j) < 0, \text{ then } Da(i,j) = B(i,j), \text{ else } Da(i,j) = 0; \\ \text{if } B(i,j) < 0, \text{ then } Db(i,j) = A(i,j), \text{ else } Db(i,j) = 0; \\ \text{if } A(i,j) < 0 \text{ and } B(i,j) < 0, \text{ then } S(i,j) = 1, \text{ else } S(i,j) = 0, \end{gathered} \tag{9}$$

where matrices D_a and D_b should be considered as shifted 32 bits to the left, and matrix S as shifted 64 bits to the left.

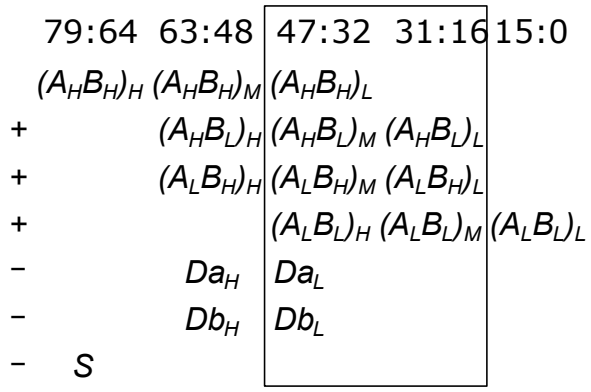

Figure 3. Components of matrix multiplication of 32-bit signed integers via matrix multiplication of 16-bit signed integers.

Depending on the fixed point format, we extract corresponding bits from the 80-bit outcome. For example, for 16.16 fixed point numbers we retrieve bits from 47 to 16 (it is designated by a black rectangle in Figure 3). For faster processing in the case of matrices of 16.16 fixed-point numbers, we can omit calculation of the terms outside the black rectangle in Figure 3. Again, checking the correctness of clipping of high bits is the obligation of a programmer. Nevertheless, for exact integer computations with an 80-bit result, you should not ignore matrix S from Formula (9).

Figure 4 demonstrates a scheme of a matrix multiplication of 32-bit fixed point numbers. The module *Split* makes a decomposition of matrix of 32-bit numbers on two matrices of 16-bit numbers. The module *Mul* performs matrix products of 16-bit numbers. The module *BAdd* is intended for calculation of matrices D_a and D_b as well as their sum, depending on a sign of elements of matrices A and B. Outcomes of multiplications A_HB_L and A_LB_H are summed in the module *Add*. The modules *ShiftAdd* and *ShiftSub* make addition and subtraction of matrices with shifted elements depending on the length of the fractional part of the utilized fixed point format.

The used DSP has no vector instruction for arithmetic shift of 32-bit numbers. An implementation of arithmetic shift by 16-bit vector instructions leads to a big overhead. Therefore, it is preferable to use fixed point number formats with a point position at the byte boundary, that is, 0.32, or 8.24, or 16.16, or 24.8. The approach allows the replacement of the arithmetic shifts by loading of corresponding bytes or 16-bit words.

Figure 5 shows a percentage of time for different stages of the proposed algorithm of 32-bit matrix multiplication depending on matrix size. The time to perform four matrix multiplications of 16-bit numbers exceeds 90% of the total time for a matrix having a size of about one hundred. Therefore, an optimization of matrix multiplication of 16-bit numbers is a key problem for algorithm speeding up.

Depending on the size of matrices in SoC with sole DSP it may be advantageous to perform multiplication according to Formula (4) that allows the use of three multiplications of the matrices of 16-bit numbers instead of four. Nevertheless, Formula (3) has a clear advantage for parallel processing in SoC having several DSPs, because blocks depicted on the same horizontal level in Figure 4 can be executed concurrently, that is, decompositions on matrices of 16-bit numbers and products of the matrices can be done in a parallel manner.

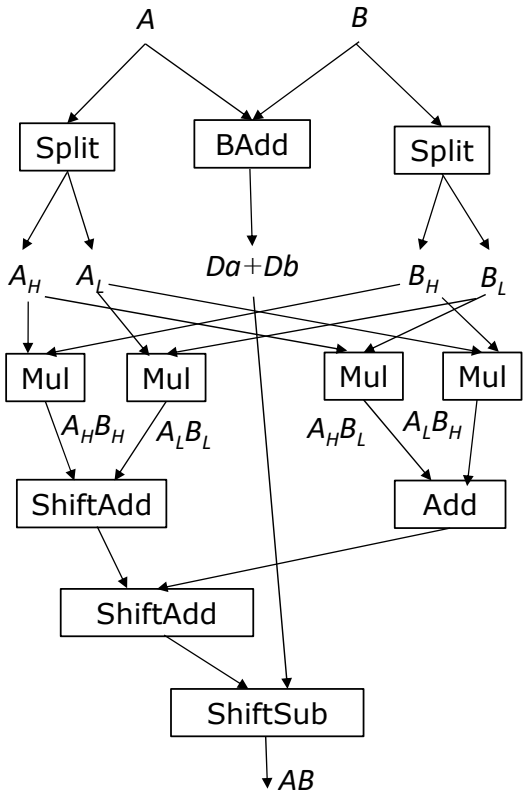

Figure 4. Scheme of matrix multiplication of 32-bit fixed point numbers (16.16) via 16-bit vector instructions.

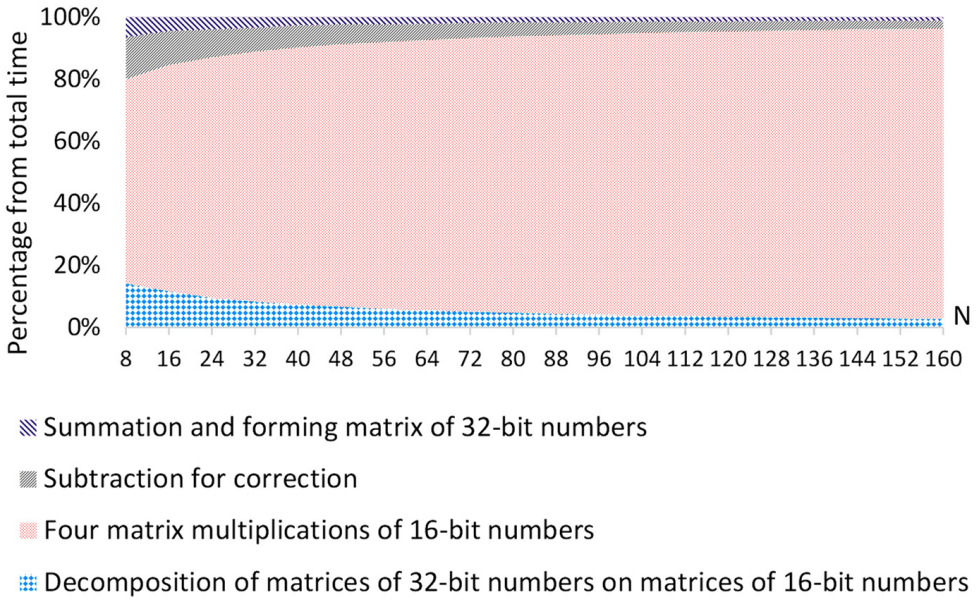

Figure 5. Percentage of time for different stages of the proposed scheme of 32-bit matrix multiplication depending on matrix size.

6. Optimization Tricks

There are many optimization tricks, which are able to decrease processing time significantly [16,44]. Two of them are effective for the considered DSP. The first is aligning the addresses of matrix rows. The address of the row should be divisible by the length (in bytes) of the vector register, thereat loading data from local memory to registers is performed several times faster. Figure 6 illustrates the benefit of rows alignment: in the case

of unaligned addresses of rows, processing time grows significantly. Padding bytes may appear at the end of the row for aligned addresses. There is no necessity for zero-padding in matrix multiplication via outer products. Padding bytes can have arbitrary values.

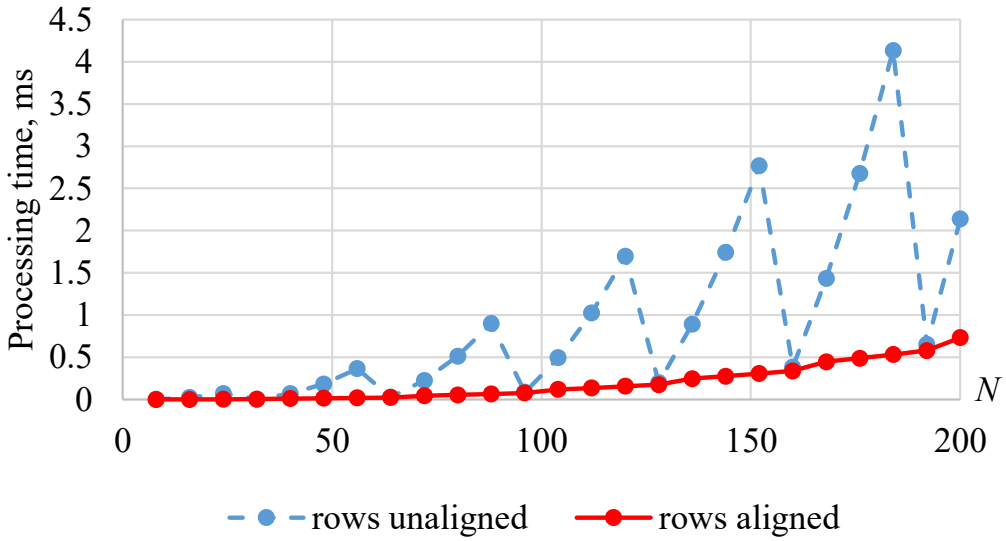

Figure 6. Processing time for matrix multiplication of 16-bit unsigned integers with and without rows alignment depending on matrix size.

The second trick is manual loop unrolling that allows better filling of the VLIW pipeline. The toolset of the used DSP includes a high-performance C/C++ compiler with automatic vectorization to support the VLIW pipeline. In general, modern compilers automatically make effective optimization including loop unrolling. However, in the case of nested loops, such as we have in matrix multiplication, it is possible to find a more apposite way for loop unrolling. In particular, we were able to write approximately 30% faster code for matrix multiplication of 16-bit unsigned integers compared with the optimization by compiler. Figure 7 shows the advantages of our approach for loop unrolling in comparison with automatic vectorization by compiler.

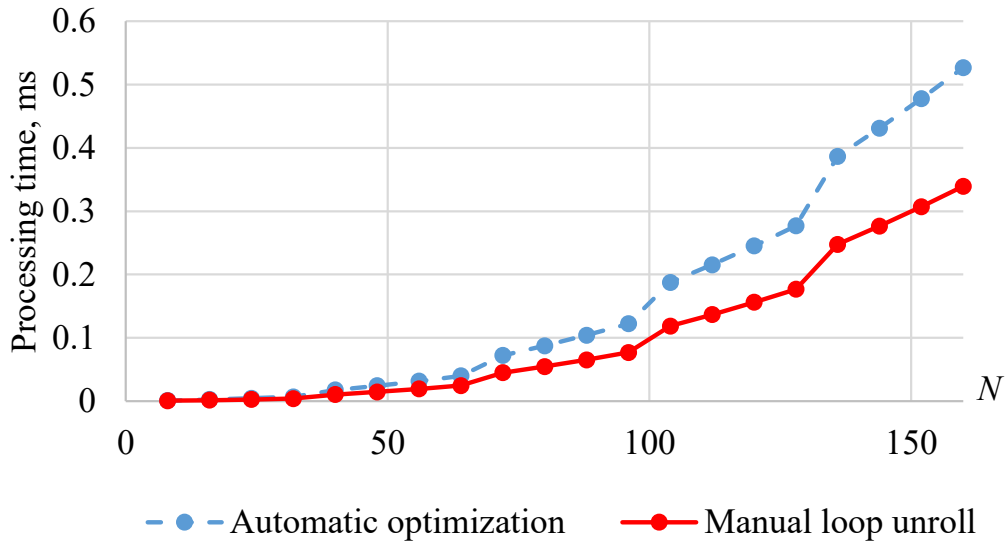

Figure 7. Processing time for matrix multiplication of 16-bit unsigned integers with automatic and manual loop unrolling.

7. Approaches for Parallel Implementation

As was mentioned above, considered SoC can have several identical IP cores with DSP. In that case, it is expedient to implement parallel matrix multiplications. The "natural" approach for parallelization of matrix multiplication by splitting of the rightmost matrix on vertical blocks, multiplication of the leftmost matrix on each block, and concatenation of outcomes (see Figure 8) is valuable [15,16]. The width of the block should be divisible by the bit-width of the vector register. Moreover, the leftmost matrix can be divided into strips. In addition, referencing to the schema in Figure 4, blocks depicted on the same level (decompositions on matrices of 16-bit numbers and products of the matrices) can be executed concurrently.

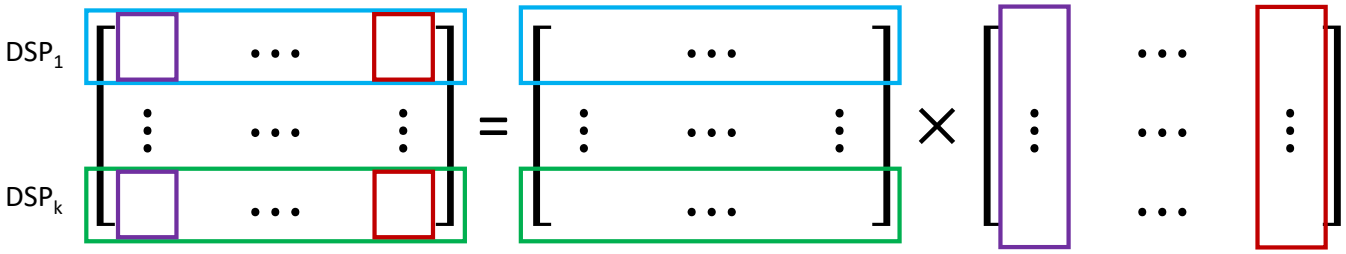

Figure 8. Splitting of the rightmost matrix on vertical blocks and the leftmost matrix on strips for parallel processing.

Undoubtedly, parallel implementation requires additional efforts and has overheads that should be evaluated for each specific computing platform. Unfortunately, we had no tools for estimation of processing time in parallel mode. Nevertheless, we are sure that parallel processing can be implemented effectively, as due to the use of a DMA controller and double buffering, the latency associated with copying from the global to the local memory of the IP block is negligible.

8. Results and Discussion

In this section, we compare processing times of several matrix multiplication algorithms implemented on the sole IP core. The used pipeline-accurate software simulator allows us to estimate processing time precisely. We are interested in multiplication of matrices having sizes of about one hundred.

Figure 9 and Table 1 show an advantage of matrix multiplication by means of 16-bit vector instructions in comparison with solutions that used 32-bit scalar instructions. We implement three algorithms for handling 32-bit scalars: based on dot product; based on outer product, and Strassen's algorithm. Overhead of Strassen's algorithm for the considered range of matrix sizes is significant, because the total number of operations is higher in comparison with matrix multiplication via inner or outer product. Processing times of implementations via dot product and inner product for scalar instructions are close. The realization via outer product slightly better uses the VLIW pipeline. Processing time of the proposed algorithm via SIMD 16-bit instructions for matrix 160 by 160 of 16.16 fixed point numbers is about six times faster in comparison with implementations via inner and outer product by using 32-bit scalar instructions (see Table 1). For matrices of a larger size (however, matrix size is limited by TCM size), the positive effect is higher. Surely, an application of 32-bit vector instructions is capable of achieving higher processing speed, but we consider a case when a processing unit has no such instructions.

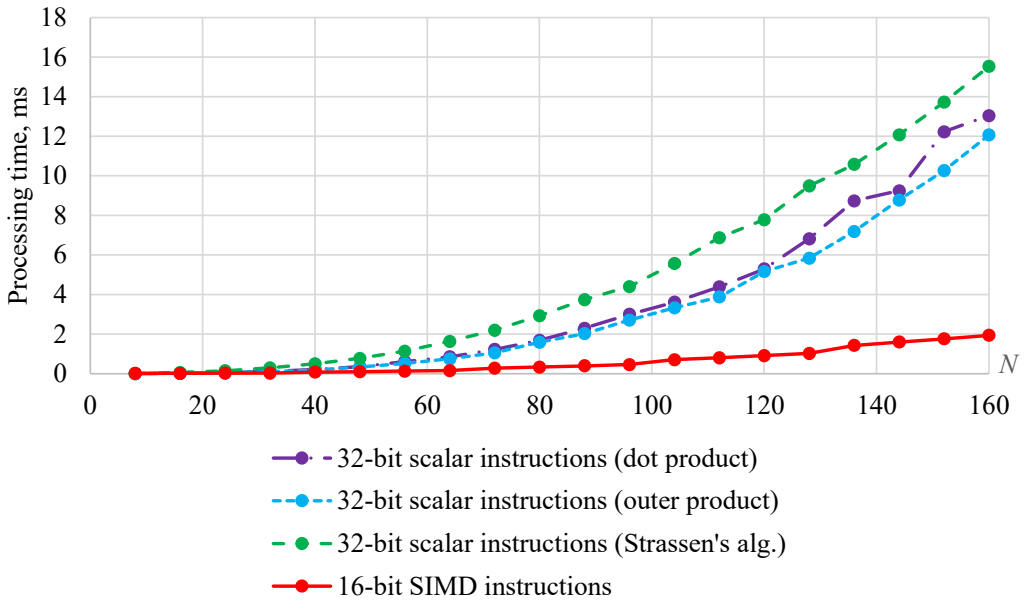

Figure 9. Processing time for matrix multiplication of 16.16 fixed point numbers via 32-bit scalar instructions and by means of 16-bit vector instructions depending on matrix size.

Table 1. Processing time of various matrix multiplication approaches for matrix sizes 80 by 80 and 160 by 160.

An Approach	Processing Time for 80 by 80, ms	Processing Time for 160 by 160, ms
32-bit scalars (dot product)	1.6	12.4
32-bit scalars (outer product)	1.6	11.4
32-bit scalars (Strassen's alg.)	2.9	16.8
Proposed 16-bit SIMD	0.4	1.9

Obviously a more or less well-implemented algorithm that uses vector instructions works faster than an analogous one via scalar instructions. However, there is the following question: what is the theoretical limit of achievable performance? The multiplication of square matrices of size N by N requires N^3 products for scalars. We assume that, in the ideal case, due to the VLIW pipeline, local memory load/store operations and additions/subtractions can be performed simultaneously with the multiplication instruction. A vector register of the used DSP contains 32 16-bit numbers. The proposed algorithm performs four matrix multiplications. Thus, theoretically, the lowest processing time of matrix multiplication of unsigned integers can be estimated as the DSP frequency multiplied by $4N^3/32$.

Figure 10 demonstrates the theoretically achievable lowest processing time for the given DSP for matrix multiplication depending on matrix size, processing time for matrix multiplication of 32-bit unsigned integers, and 16.16 fixed point numbers. Matrix multiplication of 32-bit unsigned integers has a performance of only 8–10% worse in comparison with evaluation for an ideal case. For matrices of fixed point numbers, the generation and subtraction of D_a and D_b matrices lead to 10–15% overhead. Nevertheless, processing speed is quite high.

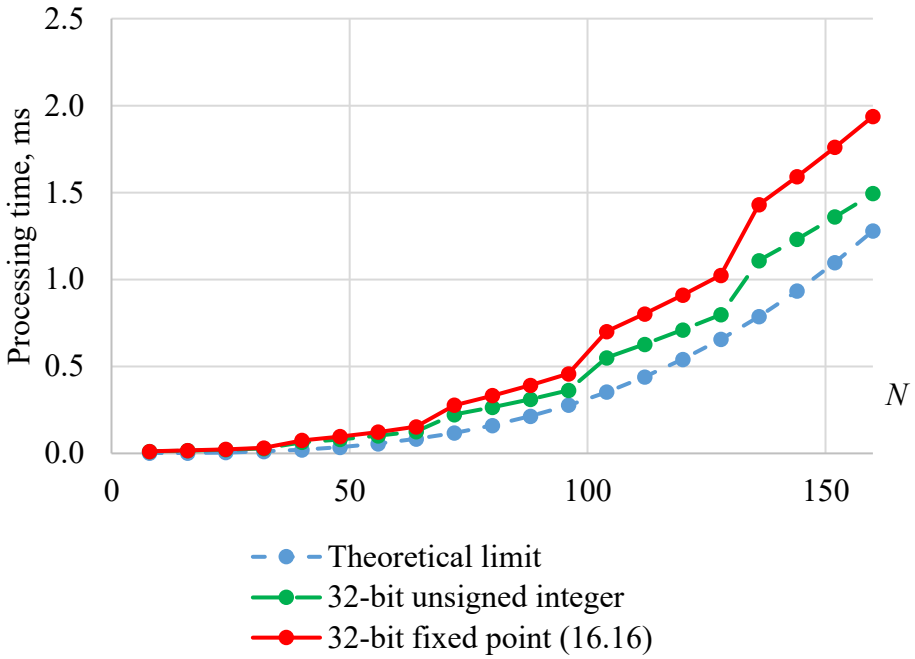

Figure 10. Processing time for matrix multiplication of 32-bit unsigned integers, 16.16 fixed point numbers, and a theoretically achievable limit depending on matrix size.

9. Conclusions

In this paper, an approach for matrix multiplication of 32-bit signed fixed point numbers and integers by means of SIMD instructions for 16-bit integers is proposed. Such problems might appear for development for some low-power DSP and modification of firmware for out-of-date processing units. The proposed approach is much faster than the use of 32-bit scalar instructions, whereas 32-bit scalar instructions can be examined by practitioners as the only way, due to the absence of 32-bit vector instructions. Matrix sizes from several tens to two hundred are considered. For demonstration of the proposed method advantages, a prototype of embedded platform is used. The proposed algorithm does not use any specific features of the given DSP, which has ordinary architecture, and the proposed approach can be generalized for many modern processors and matrix multiplication of n-bit numbers by means of $n/2$-bit SIMD instructions. For example, the proposed technique can be employed for realization of matrix multiplication of 64-bit fixed point numbers and signed integers via vector instructions for 32-bit integers. Of course, processing speed differs for various computing platforms, and the effect from an application of the proposed method should be assessed for each specific hardware and software system.

There are the following limitations of the proposed approach. Latency for loading matrix elements to vector registers should be low. In the considered DSP, both input matrices and the output one have to be stored in local memory. Thus, the size of local memory limits the size of matrices that can be multiplied with high speed. One more limitation on size of matrices is the possible overflow and clipping of high bits. Correct multiplication of matrices of fixed point numbers is a headache for a programmer, because the integer part of each element of matrix product outcome has to be less than the integer part of the used fixed point format. In addition, long handling with fixed point numbers can lead to calculation error accumulation.

As for future work, we are going to extend the proposed approach for matrix multiplication of 64-bit numbers via 16-bit vector instructions. In addition, for DSP having a sign extension instruction, we plan to implement other ways for considering a sign: for Formula (3) using multiplication of unsigned by unsigned integers for low halves of operands, multiplication of signed by signed integers for high halves, and multiplications

of signed by unsigned integers for high and low halves. Such a way allows us to decrease the number of operations in comparison with the used approach, that is, correction of multiplication outcome for unsigned integers by means of subtractions of operands depending on their sign.

Author Contributions: Conceptualization, I.S.; methodology, A.K. and I.S.; software, A.K. and I.S.; validation, D.M.; formal analysis, A.K. and D.M.; investigation, I.S. and A.K.; resources, I.S.; data curation, D.M.; writing, original draft preparation, I.S.; writing, review and editing, A.K. and D.M.; supervision, I.S. All authors have read and agreed to the published version of the manuscript.

Funding: This research received no external funding.

Conflicts of Interest: The authors declare no conflict of interest.

References

1. Dongarra, J.J.; Du Croz, J.; Hammarling, S.; Duff, I.S. A set of level 3 basic linear algebra subprograms. *ACM Trans. Math. Softw.* **1990**, *16*, 1–17. [CrossRef]
2. Safonov, I.V.; Kurilin, I.V.; Rychagov, M.N.; Tolstaya, E.V. *Adaptive Image Processing Algorithms for Printing*; Springer: Singapore, 2018; 304p.
3. Safonov, I.V.; Kurilin, I.V.; Rychagov, M.N.; Tolstaya, E.V. *Document Image Processing for Scanning and Printing*; Springer: Cham, Switzerland, 2019; 305p.
4. Yates, R. *Fixed-Point Arithmetic: An Introduction*; Digital Signal Labs: San Francisco, CA, USA, 2009; 15p.
5. Martel, M.; Najahi, M.A.; Revy, G. Code size and accuracy-aware synthesis of fixed-point programs for matrix multiplication. In Proceedings of the PECCS: Pervasive and Embedded Computing and Communication Systems, Lisbon, Portugal, 7–9 January 2014.
6. Kepner, J.; Gilbert, J. *Graph Algorithms in the Language of Linear Algebra*; SIAM: Philadelphia, PA, USA, 2011; 361p.
7. Strassen, V. Gaussian elimination is not optimal. *Numer. Math.* **1969**, *13*, 354–356. [CrossRef]
8. Soliman, M.I.; Ahmed, F.S. Exploiting ILP, DLP, TLP, and MPI to accelerate matrix multiplication on Xeon processors. In Proceedings of the International Conference on Engineering and Technology (ICET), Cairo, Egypt, 19–20 April 2014.
9. De Silva, H.; Gustafson, J.L.; Wong, W.F. Making strassen matrix multiplication safe. In Proceedings of the 2018 IEEE 25th International Conference on High Performance Computing (HiPC), Bengaluru, India, 17–20 December 2018.
10. Malik, O.A.; Becker, S. Randomization of approximate bilinear computation for matrix multiplication. *Int. J. Comput. Math. Comput. Syst. Theory* **2021**, *6*, 54–93. [CrossRef]
11. Coppersmith, D.; Winograd, S. Matrix multiplication via arithmetic progressions. *J. Symb. Comput.* **1990**, *9*, 251–280. [CrossRef]
12. Alman, J.; Williams, V.V. A refined laser method and faster matrix multiplication. In Proceedings of the 2021 ACM-SIAM Symposium on Discrete Algorithms (SODA), Alexandria, VA, USA, 10–13 January 2020.
13. Duan, R.; Wu, H.; Zhou, R. Faster Matrix Multiplication via Asymmetric Hashing. *arXiv* **2022**, arXiv:2210.10173.
14. Aberdeen, D.; Baxter, J. General matrix-matrix multiplication using SIMD features of the PIII. In Proceedings of the European Conference on Parallel Processing, Munich, Germany, 29 August–1 September 2000.
15. Kelefouras, V.; Kritikakou, A.; Goutis, C. A Matrix–Matrix Multiplication methodology for single/multi-core architectures using SIMD. *J. Supercomput.* **2014**, *68*, 1418–1440. [CrossRef]
16. Hemeida, A.M.; Hassan, S.A.; Alkhalaf, S.; Mahmoud, M.M.M.; Saber, M.A.; Eldin, A.M.B.; Senjyu, T.; Alayed, A.H. Optimizing matrix-matrix multiplication on intel's advanced vector extensions multicore processor. *Ain Shams Eng. J.* **2020**, *11*, 1179–1190. [CrossRef]
17. Pradyumna, S. Performance comparison of matrix multiplication algorithms. In Proceedings of the 2017 International Conference on Innovative Mechanisms for Industry Applications (ICIMIA), Bengaluru, India, 21–23 February 2017.
18. Yang, W.; Fang, J.; Dong, D. Characterizing Small-Scale Matrix Multiplications on ARMv8-based Many-Core Architectures. In Proceedings of the 2021 IEEE International Parallel and Distributed Processing Symposium (IPDPS), St. Petersburg, Florida USA, 15–19 May 2021.
19. Wang, Q.; Zhang, X.; Zhang, Y.; Yi, Q. AUGEM: Automatically generate high performance dense linear algebra kernels on x86 CPUs. In Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis, Denver, CO, USA, 17–21 November 2013.
20. Van Zee, F.G.; van de Geijn, R.A. BLIS: A Framework for Rapidly Instantiating BLAS Functionality. *ACM Trans. Math. Softw.* **2015**, *41*, 1–33. [CrossRef]
21. Frison, G.; Kouzoupis, D.; Sartor, T.; Zanelli, A.; Diehl, M. BLASFEO: Basic linear algebra subroutines for embedded optimization. *ACM Trans. Math. Softw. (TOMS)* **2018**, *44*, 1–30. [CrossRef]
22. Eigen. Available online: https://eigen.tuxfamily.org (accessed on 7 October 2022).

23. Moss, D.J.; Krishnan, S.; Nurvitadhi, E.; Ratuszniak, P.; Johnson, C.; Sim, J.; Mishra, A.; Marr, D.; Subhaschandra, S.; Leong, P.H. A customizable matrix multiplication framework for the intel harpv2 xeon+ fpga platform: A deep learning case study. In Proceedings of the 2018 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays, Monterey, CA, USA, 25–27 February 2018.
24. Foldi, T.; von Csefalvay, C.; Perez, N.A. JAMPI: Efficient matrix multiplication in Spark using Barrier Execution Mode. *Big Data Cogn. Comput.* **2020**, *4*, 32. [CrossRef]
25. Misra, C.; Bhattacharya, S.; Ghosh, S.K. Stark: Fast and Scalable Strassen's Matrix Multiplication using Apache Spark. *IEEE Trans. Big Data* **2022**, *8*, 699–710. [CrossRef]
26. NVIDIA cuBLAS. Available online: https://developer.nvidia.com/cublas (accessed on 7 October 2022).
27. NVIDIA CUTLASS 2.10. Available online: https://github.com/NVIDIA/cutlass (accessed on 7 October 2022).
28. Huang, J.; Yu, C.D.; Geijn, R.A.V.D. Strassen's algorithm reloaded on GPUs. *ACM Trans. Math. Softw. (TOMS)* **2020**, *46*, 1–22. [CrossRef]
29. Krishnan, A.G.; Goswami, D. Multi-Stage Memory Efficient Strassen's Matrix Multiplication on GPU. In Proceedings of the IEEE 28th International Conference on High Performance Computing, Data, and Analytics (HiPC), Online, 17–18 December 2021.
30. Ali, M.; Stotzer, E.; Igual, F.D.; van de Geijn, R.A. Level-3 BLAS on the TI C6678 multi-core DSP. In Proceedings of the IEEE 24th International Symposium on Computer Architecture and High Performance Computing, New York, NY, USA, 24–26 October 2012.
31. Yin, S.; Wang, Q.; Hao, R.; Zhou, T.; Mei, S.; Liu, J. Optimizing Irregular-Shaped Matrix-Matrix Multiplication on Multi-Core DSPs. In Proceedings of the 2022 IEEE International Conference on Cluster Computing (CLUSTER), Heidelberg, Germany, 6–9 September 2022.
32. Morad, A.; Yavits, L.; Ginosar, R. Efficient Dense and Sparse Matrix Multiplication on GP-SIMD. In Proceedings of the 24th International Workshop on Power and Timing Modeling, Optimization and Simulation, Palma de Mallorca, Spain, 29 September–1 October 2014.
33. Ahmad, A.; Pasha, M.A. Optimizing hardware accelerated general matrix-matrix multiplication for CNNs on FPGAs. *IEEE Trans. Circuits Syst. II Express Briefs* **2020**, *67*, 2692–2696. [CrossRef]
34. Bhattacharjee, D.; Chattopadhyay, A.; Liwongan, R.J. Accelerating Binary-Matrix Multiplication on FPGA. In Proceedings of the 32nd IEEE International System-on-Chip Conference (SOCC), Singapore, 3–6 September 2019.
35. Umuroglu, Y.; Rasnayake, L.; Själander, M. Bismo: A scalable bit-serial matrix multiplication overlay for reconfigurable computing. In Proceedings of the 28th International Conference on Field Programmable Logic and Applications (FPL), Dublin, Ireland, 27–31 August 2018.
36. Gorlani, P.; Kenter, T.; Plessl, C. OpenCL implementation of Cannon's matrix multiplication algorithm on Intel Stratix 10 FPGAs. In Proceedings of the International Conference on Field-Programmable Technology (ICFPT), Tianjin, China, 9–13 December 2019.
37. Kamranfar, P.; Shahabi, S.A.; Vazhbakht, G.; Navabi, Z. Configurable systolic matrix multiplication. In Proceedings of the 27th International Conference on VLSI Design, Mumbai, India, 5–9 January 2014.
38. Shanmugakumar, M.; Srinivasavarma, V.S.; Mahammad, S.N. Energy Efficient Hardware Architecture for Matrix Multiplication. In Proceedings of the IEEE 4th Conference on Information & Communication Technology (CICT), Chennai, India, 3–5 December 2020.
39. Karatsuba, A.; Ofman, Y. Multiplication of Many-Digital Numbers by Automatic Computers. *Proc. USSR Acad. Sci.* **1962**, *145*, 293–294.
40. Karatsuba, A.A. The Complexity of Computations. *Proceedings of the Steklov Institute of Mathematics* **1995**, *211*, 169–183.
41. Knuth, D.E. *The Art of Computer Programming: Vol. 2 Seminumerical Algorithms*, 3rd ed.; Addison-Wesley: Boston, MA, USA, 1997; 784p.
42. Warren, H.S. *; Jr. Hacker's Delight*, 2nd ed.; Addison Wesley—Pearson Education, Inc.: Boston, MA, USA, 2013; 817p.
43. Eklundh, J.O. A fast computer method for matrix transposing. *IEEE Trans. Comput.* **1972**, *21*, 801–803. [CrossRef]
44. Yufu, L.; Wenhui, L.; Guangshuai, J. Data Compression and Vectorization of Matrix Multiplication on HXDSP. In Proceedings of the MATEC Web of Conferences, Nanjing, China, 24–26 May 2018.