ISSN 1995-0802, Lobachevskii Journal of Mathematics, 2019, Vol. 40, No. 5, pp. 580–598. © Pleiades Publishing, Ltd., 2019.

Vectorization of High-performance Scientific Calculations Using AVX-512 Intruction Set

B. M. Shabanov[1*], **A. A. Rybakov**[1**], **and S. S. Shumilin**[1***]

(Submitted by A. M. Elizarov)

[1]*Joint Supercomputer Center, Russian Academy of Sciences, Moscow, 119334 Russia*

Received February 1, 2019; revised February 22, 2019; accepted February 22, 2019

Abstract—Modern calculation codes used in supercomputing are very demanding of computing resources. For their effective appliance requires the use of parallelization at all levels, starting with the use of multiprocess and multi-threaded programming, and ending with vectorization. The AVX-512 instruction set, first introduced in Intel Xeon Phi Knights Landing and Intel Xeon Skylake microprocessors, opens up broad possibilities for vectorizing code and allows to speed up the execution of applications in several times. This article discusses some aspects of the application of vectorization in the program code of some kinds, which is found in high-performance scientific computing.

DOI: 10.1134/S1995080219050196

Keywords and phrases: *optimization, vectorization, AVX-512, predicated execution, intrinsics, application performance.*

1. INTRODUCTION

Today, supercomputer technologies are increasingly used in various fields of science, industry and business. The use of simulation modeling using supercomputer calculations allows analyzing various situations and interaction scenarios of objects of the surrounding world and obtaining results that are unavailable without the use of these tools. At the same time, the amount of data involved in supercomputer calculations is constantly increasing. The size of the calculated grids of hundreds of millions cells is already common for launches on the supercomputers of the petaflops performance range [1, 2], and gradually there is a need to use grids containing billions of cells. Today, the most powerful supercomputer is the IBM's Summit supercomputer, equipped with Power9 processor and NVidia graphics processors with Volta architecture.

According to some optimistic forecasts, the first exaflops supercomputer capable of performing 10^{18} floating point operations per second is possible by 2024 [3]. Along with the increasing of supercomputers computing power, questions about the effectiveness of their use arise. In particular, works aimed to improve the efficiency of data exchange systems between computing nodes [4, 5], to develop computational grid management technologies and to evenly distribute computations on a cluster are carried out [6–8]. Developing tools of programming languages aimed to facilitate the creation of high-performance parallel code are actively developed [9, 10].

[*]E-mail: shabanov@jscc.ru
[**]E-mail: rybakov.aax@gmail.com
[***]E-mail: shumilin@jscc.ru

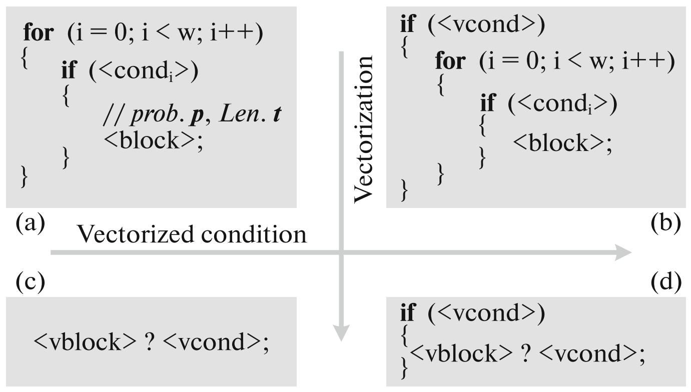
1.1. Review of Research Papers

The Intel Xeon Phi ×200 Knights Landing processor family [11] is the second generation of the Intel Xeon Phi line and the first generation of solvers that act as a standalone processor (the first generation of the Knights Corner was co-processor [12, 13]). The first KNL processors were introduced in 2016. Each processor contains up to 36 active tiles, each consists of two cores and a L2 cache of 1 MB in size, which is common to the data of the two cores. Each core contains a VPU (Vector Processing Unit) that supports 512-bit vector instructions and a 64 KB L1 cache divided into equal by-size instruction cache and data cache of 32 KB each. Each core supports 4 threads, which gives a total of 288 logical processors per socket [14]. Although the frequency of each KNL core is lower than that of Intel Xeon server processors, such a number of execution threads and the presence of 512-bit vector instructions provide impressive peak performance, exceeding 6 TFLOPS on single-precision operations. The emergence of such a powerful hardware has opened up new possibilities for optimizing software used in supercomputer calculations. Approaches described in [15] allowed the usage of vectorization for KNL to speed up the computational cores of the LAMMPS program code up to 12 times compared with the non-vectorized version, which led to a general acceleration of solver runs by 2–3 times. You can note the successful application of vectorization to accelerate operations with sparse matrices of high dimensionality, which resulted in a 5-fold acceleration on these operations [16]. Special features of KNL processors associated with the use of masked vector operations are used to vectorize cycles even with an unknown number of iterations and complex controls, as shown by the example of the vectorization of the Mandelbrot set construction code [17]. The work [18] describes the achievement of 6-fold acceleration achieved by low-level optimization of computational codes for problems of nuclear physics. In [19], an example of the application of code vectorization which is performed by removing the unlikely branch of execution from a hot inner loop. However, despite the high potential of KNL, studies show that the acceleration achieved compared to launches on Intel Xeon processors of the Haswell and Broadwell generations rarely exceeds 1.5–2 times [20, 21]. Partially it is related to imbalance between peak bandwidth when working with memory and the intensity of arithmetic operations [22]. Active studies are underway on the use of various memory modes and various clustering modes when launching applications on KNL [23].

2. FLAT LOOPS VECTORIZATION

The simplest context for vectorization is a flat loop. We will call a loop flat if it has the following properties. First, it should not contain inter-iteration dependencies, that is, all loop iterations can be executed in parallel. Secondly, within the iteration of the loop with number i there can only be such memory access operations that read or write elements of arrays with indices i (all references to memory can be reduced to the form $a[i]$). A trivial example of a flat loop is the addition of two arrays. In the most cases, flat loops containing only arithmetic operations are automatically compiled by the compiler and demonstrate multiple acceleration, since the AVX-512 instruction set contains vector analogues of all arithmetic operations [24]. The situation is more complicated with vectorization of flat loops containing peculiarities. Conditional operators or control transfer commands can be given as examples of the flat loops peculiarities. Such loops are also easily vectorized, but in practice the compiler often refuses to build a vector code due to the theoretical estimate of the possible acceleration. The presence of a strongly branched control in a flat loop usually leads to the rejection of vectorization, in such cases the vector code must be built manually. Presence of nested loops or function calls in a flat loop is the more serious property that prevents the construction of effective parallel code. In this case, the compiler is not able to perform vectorization, although using the intrinsic functions [25, 26], optimization can also be performed in these cases.

Let us analyze the question of vectorization of flat loops more thorough. Suppose that the vectorization uses vectors capable of containing w primitive data elements, in this case we will call w the vectorization width (AVX-512 vectors can contain 8 double elements or 16 float elements). We will not consider the option in which the loop body contains only arithmetic operations due to its triviality. Let the considered flat loop contain n iterations. Then all iterations of this loop can be divided into n/w groups by w iterations, and each group can be considered separately (a residual group of iterations containing less than w iterations is also possible; the presence of this group does not affect the optimization efficiency, it can be completed in the original form or vectorized using masks). Thus, for simplicity, we can restrict ourselves to the consideration of flat cycles containing exactly w iterations.



```
for (i = 0; i < w; i++)
{   if (<cond_i>)
    {
        // prob. p, Len. t
        <block>;
    }
}
```

```
if (<vcond>)
{   for (i = 0; i < w; i++)
    {
        if (<cond_i>)
        {   <block>;
        }
    }
}
```

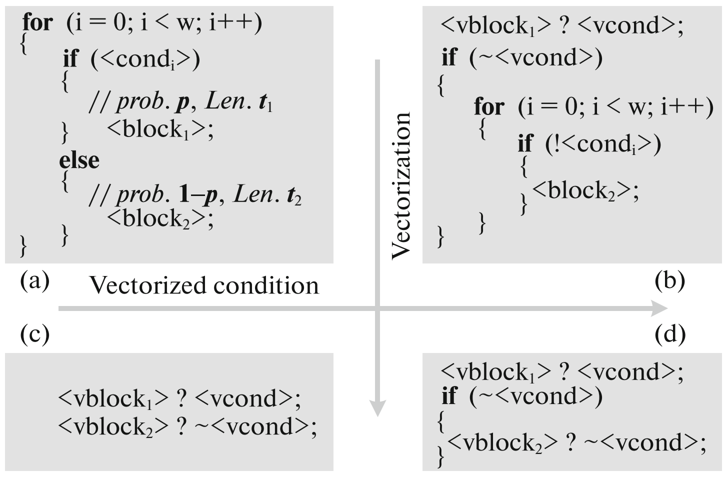
```
<vblock> ? <vcond>;
```

```
if (<vcond>)
{
<vblock> ? <vcond>;
}
```

Fig. 1. Example of flat loop with single linear section $block$ under condition $cond_i$.

As a first example, we consider a flat loop whose body consists of a single linear section $block$, which is executed under the condition $cond_i$ with probability p and has a length t (Fig. 1a). In this case, the length of a linear section is understood to be a characteristic proportional to the execution time of a given linear section (the number of operators or machine instructions). The total number of operations contained in the loop is ptw (in this case, we assume that the auxiliary instructions related to the calculation of the condition and the provision of control can be neglected). If the probability that the condition $cond_i$ is satisfied is low enough, then we can only stop at the vectorization of this condition and perform the whole loop under vectorized condition $vcond$. This transformation will be called the vectorization of the condition (Fig. 1b). Moreover, the probability that the condition $vcond$ turns out to be instinctive is $1-(1-p)^w$ (corresponds to the fact that at least one of the w elementary conditions is true). The total number of operations in the loop from this conversion will not change (the $cond_i$ and $vcond$ conditions are not independent) and will remain equal to ptw, but the number of auxiliary operations and control operations will be reduced. To vectorize the entire loop, it is necessary to convert the program code into a predicate code [27, 28], in which each instruction of the linear section $block$ is executed under the predicate $cond_i$, then all the instructions of the linear section can be replaced by vector analogs executed under the vector predicate $vcond$ (Fig. 1c). The number of operations of a fully vectorized loop is simply t. At the same time, it is also not forbidden to use $vcond$ checking for voidness before executing vectorized code in order to avoid executing code blocks with empty masks (Fig. 1d). In this case, the number of loop operations will be reduced by $1-(1-p)^w$ times, since the addition of a condition will cut off the execution of vector operations with an empty mask.

As another example, consider a flat loop whose body is an if-else statement. Without loss of generality, we assume that the branch executed by the if-condition ($block_1$) is more likely than its alternative ($block_2$), that is $p \geq 1-p$, or $p \geq 1/2$. The lengths of the linear sections in this example are t_1 and t_2, respectively (Fig. 2a). In addition, we will assume that the $block_1$ linear section is suitable for vectorization in any case, and a vector code is constructed for it (otherwise, the optimization of this code loses its meaning). The total number of operations performed during the loop operation is calculated as the mathematical expectation of the number of operations per iteration multiplied by the number of iterations of the loop, that is $(pt_1+(1-p)t_2)w$. In real computational problems, it often happens that an unlikely branch in such cycles is not appropriate for vectorization or this requires some effort. Such unlikely branches contain processing of extremely rare exceptions or unexpected situations. In such cases, it is reasonably to vectorize only the probable branch, and leave the alternative under the vectorized condition (Fig. 2b). Such a half vectorized loop will contain $t_1+(1-p)t_2w$ operations. In the case of a complete vectorization of the loop body, both linear sections fall under opposite predicates ($vcond$, $\sim vcond$) and the loop body is completely freed from the control transfer commands (Fig. 2c). The number of executed instructions in a loop for a fully vectorized code is t_1+t_2. As a last action, we apply the addition of checking the condition $vcond$ for an unlikely branch to exclude the execution of operations with empty masks (Fig. 2d). Since the probability of executing an alternative is $1-p^w$, the final number of operations performed for a fully vectorized cycle with checking the vectorized condition for an alternative is $t_1+(1-p^w)t_2$.



```
for (i = 0; i < w; i++)
{
    if (<cond_i>)
    {
        // prob. p, Len. t1
        <block1>;
    }
    else
    {
        // prob. 1–p, Len. t2
        <block2>;
    }
}
```

```
<vblock1> ? <vcond>;
if (~<vcond>)
{
    for (i = 0; i < w; i++)
    {
        if (!<cond_i>)
        {
            <block2>;
        }
    }
}
```

```
<vblock1> ? <vcond>;
<vblock2> ? ~<vcond>;
```

```
<vblock1> ? <vcond>;
if (~<vcond>)
{
    <vblock2> ? ~<vcond>;
}
```

Fig. 2. Example of flat loop with if-else statement.

The Table 1 contains data on the number of operations performed for the two considered examples of flat loops, in which the loop body consists of one if-condition and one if-else-construction.

We will analyze only the second example with the if-else statement (the first example is treated similarly). A fully vectorized loop is more profitable than a loop with a vectorized condition, if it contains fewer operations, that is, if the condition $t_1 + t_2 \leq t_1 + (1-p)t_2w$, or $(1-p)w \geq 1$ is satisfied. Thus, for the benefit of vectorization, either a high vectorization width, or a high probability of an alternative. If we consider the application of checking vectorized conditions for a fully vectorized loop body, then we obtain the condition $t_1 + (1-p^w)t_2 \leq t_1 + (1-p)t_2w$, which translates into $1 - p^w \leq (1-p)w$. This condition is equivalent to the following:

$$\sum_{i=0}^{w-1} p^i \leq w. \tag{1}$$

This condition is obviously satisfied, since $p \leq 1$, and the total terms count is exactly w. Thus, we have obtained that using a fully vectorized loop body with a preliminary check of a vectorized condition for an unlikely branch is always more beneficial than using the original version of the code or condition vectoring. Note that this conclusion is true only under the assumption that the $block_1$ linear segment is probable, and a vector code is always built for it.

We can consider the general case of vectorization of a flat loop containing an arbitrary number of alternatives. Let the loop body contain n alternatives (with numbers from 0 to $n-1$), each of which is executed with probability p_i. In this case, we can neglect the operations of calculating conditions. It is also known that each alternative has a length t_i. It is required to understand the conditions under which the use of vectorization is beneficial, as well as the application of checking vectorized conditions for a vector code. As in the previous examples, you can calculate the total number of running operations

Table 1. Operations count for if and if-else statements vectorization in flat loops

If statement	Operations count	If-else statement	Operations count
a)	ptw	a)	$(pt_1 + (1-p)t_2)\,w$
b)	ptw	b)	$t_1 + (1-p)t_2w$
c)	t	c)	$t_1 + t_2$
d)	$(1-(1-p)^w)\,t$	d)	$t_1 + (1-p^w)t_2$

for a non-vectorized code; it is equal to the mathematical expectation of the number of operations per iteration of the loop multiplied by the number of iterations.

$$T = \left(\sum_{i=0}^{n-1} p_i t_i \right) w. \tag{2}$$

With a complete loop vectorization, the number of vector operations is, of course, equal to the sum of the lengths of all linear sections t_i. Thus, the condition of the advantage of vectorization is the following equation:

$$\left[\sum_{i=0}^{n-1} p_i t_i w \right] / \left[\sum_{i=0}^{n-1} t_i \right] \geq 1. \tag{3}$$

In particular, it follows from this equation that if the loop body contains a large number of alternative execution branches with probabilities less than $1/w$, then vectoring of this loop cannot be beneficial. When adding checks of vectorized conditions for each of the cycle alternatives, we obtain the following equation for the advantage of vectorization:

$$w \left[\sum_{i=0}^{n-1} p_i t_i \right] / \left[\sum_{i=0}^{n-1} t_i (1 - (1 - p_i)^w) \right] \geq 1. \tag{4}$$

Consider this formula more closely. With a large number of alternatives, their probabilities are close to zero, so the term $(1 - p_i)^w$ is a positive value not much less than one. From this we can conclude that the use of vectorized checks before performing vector versions of alternatives in the body of a flat loop significantly increase the effect of vectorization. At least the theoretical estimates made in our assumptions indicate the benefits of applying such transformations. However, one should not forget that vector commands generally execute more slowly than their scalar counterparts. Moreover, the derivation of theoretical estimates did not take into account the influence of management operations and preparation of conditions, did not take into account the fact of the presence of possible dependencies between the calculation of conditions for various alternatives.

If in (4) we assume that the probabilities of all branches of execution are equal to $p_i = 1/n$, and the lengths of all linear sections are the same, then we obtain the following formula for the theoretical acceleration from vectorization:

$$S(w, n) = w \left(\sum_{i=1}^{w} (-1)^{i+1} C_w^i n^{1-i} \right)^{-1}. \tag{5}$$

From this formula, it is clear that $S(w, 1) = w$ and $\lim_{n \to \infty} S(w, n) = 1$, which is consistent with the mathematical meaning of vectorization.

3. MATRIX OPERATIONS VECTORIZATION

In modern numerical methods used in high-performance computing, special importance is given to operations with vectors and matrices [29, 30]. Such operations include calculating the scalar product of two vectors, multiplying the matrix by the vector, finding the inverse matrix, decomposing the matrix, and other operations. In particular, the most frequently used operation is the multiplication of two matrices. Obtaining the product of two matrices, in which the scalar product of each row of the first matrix and each column of the second matrix is calculated, is a rather difficult operation that can take a considerable part of the total program execution time. The presence of effective approaches to the implementation of this operation is necessary to ensure the effective operation of program codes using matrix calculations.

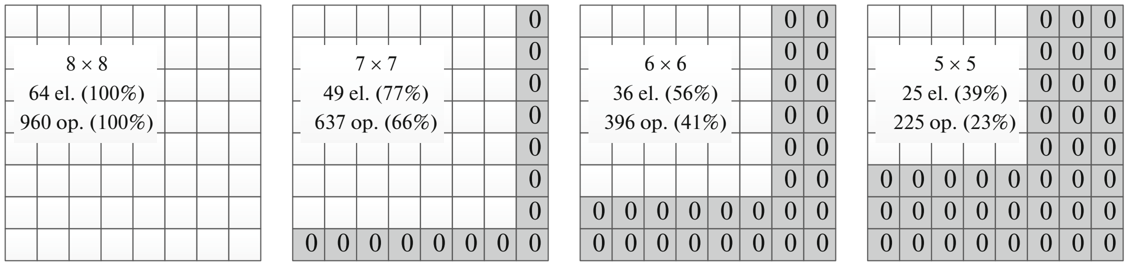


Fig. 3. Matrices of size 8 × 8, 7 × 7, 6 × 6, 5 × 5 located inside the matrix of size 8 × 8. The number of elements, and the number of addition and multiplication operations required to perform the multiplication of two matrices of a given size.

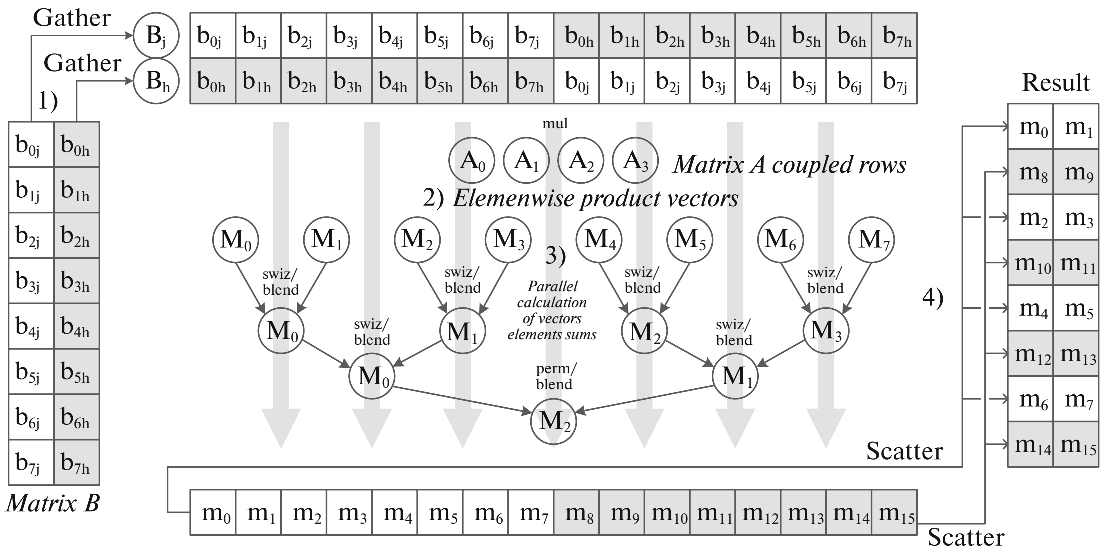


Fig. 4. Scheme of two matrices multiplication with direct reading of rows and columns of matrices and parallel computation of their pairwise scalar products.

The capabilities of the AVX-512 instruction set allow to create efficient code for implementations of matrix operations providing significant acceleration of applications, which use them. In this section, we will consider the operations of multiplying small-size matrices (having sizes 8 × 8, 7 × 7, 6 × 6, and 5 × 5 elements). Operations with such matrices take, for example, up to 40% of the total operating time of the indigenous Russian RANS/ILES calculation codes used in the modeling of unsteady turbulent flows [31, 32]. For reasons of alignment in memory, these matrices are considered as submatrices of a matrix of size 8 × 8 elements, as shown in Fig. 3.

We will compare the two approaches to implementing the multiplication of matrices of size 8 × 8, extend these approaches to smaller matrices and obtain the results of comparison of the considered approaches performance. The multiplication logic of two matrices is that the results of the pairwise scalar product of all rows of the first matrix and all columns of the second matrix form the elements of the resulting matrix.

First, we consider a direct approach to matrix multiplication, in which we will directly operate with the rows of the first matrix and the columns of the second matrix. The matrices in programming language C, represented as multidimentional arrays, are located in memory row by row, so for reading matrix row it is needed to use the operation of consequential loading from memory (`load`), and for reading matrix column it is needed to use operation of loading data elements with arbitrary offsets from base address (`gather`). We consider matrices whose elements are real single-precision values, therefore one `zmm` register with which commands from the instruction set AVX-512 operate, contains 16 elements of such matrix. Thus, the AVX-512 allows to load two rows or two columns of a matrix using one command.

Figure 4 illustrates the implementation of this approach. First of all, we note that the left matrix (matrix A) is completely loaded into 4 vectors A_1, A_2, A_3, A_4 (each vector contains two adjacent rows of the matrix). This can be done using the usual instructions for reading sequentially located data from the memory (`load`). Next, we present a scheme for calculating two adjacent columns of the resulting matrix (columns numbered j and $h = j + 1$). Two adjacent columns of the right matrix (matrix B) are loaded in the forward and reverse order into the vectors B_j and B_h (using the instructions `gather` for reading data from the memory and `permute` for placing `zmm` vector halves in reverse order, Fig. 4, 1). Next, the operations of the elementwise multiplication of the vectors B_j and B_h by all the vectors, in which the matrix A is loaded, are performed. The results of these multiplications are 8 vectors M_0–M_7 (Fig. 4, 2). Each half of the $M_i, i \in [0, 7]$ vector (lower or high) contains elements, the sum of which forms one of the elements of the resulting matrix. The calculation of the sum of the halves of all 8 vectors can be performed in parallel, with the result that all 16 required resulting values will be located in one vector (Fig. 4, 3). Finally, the resulting 16 values are written into the columns with the j and h numbers of the resulting matrix, using command `scatter` (Fig. 4, 4).

Let us draw special attention to the parallel calculation of the sums of the `zmm` vectors halves. AVX-512 instruction set does not contain horizontal operations of vectors elements addition, therefore, these operations have to be emulated by adding a vector with its copy, in which the elements are rearranged as needed. For these purposes, we will use specially implemented macros consisting of `swizzle`, `permute`, `blend` and `add` instructions and allowing to add adjacent vector elements, pairs of adjacent vector elements and fours of adjacent vector elements, depending on the masks submitted. Below is a listing of the implementation of these macros (Listing 1).

Listing 1. Macros that implement bundles from operations swizzle, permute, blend

```
#define SWIZ_2_ADD_2_BLEND_1(X, Y, SWIZ_TYPE, BLEND_MASK)          \
    _mm512_mask_blend_ps(BLEND_MASK,                                \
                         ADD(X, _mm512_swizzle_ps(X, SWIZ_TYPE)), \
                         ADD(Y, _mm512_swizzle_ps(Y, SWIZ_TYPE)))
#define PERM_2_ADD_2_BLEND_1(X, Y, PERM_TYPE, BLEND_MASK)                \
    _mm512_mask_blend_ps(BLEND_MASK,                                      \
                         ADD(X, _mm512_permute4f128_ps(X, PERM_TYPE)), \
                         ADD(Y, _mm512_permute4f128_ps(Y, PERM_TYPE)))
```

The `_mm512_swizzle_ps` intrinsic is designed to rearrange the vector elements inside 128-bit quarters (composed of 4 single-precision floating-point elements), the `_mm512_permute4f128_ps` command shuffles the 128-bit quarters themselves, and the `_mm512_mask_blend_ps` intrinsic performs the merging of two vectors using a mask. To calculate the elements sums for 16 halves of 8 registers `zmm`, the use of 7 such macros is required (4 operations for the first level of merging, 2 operations for the second level of merging and the last operation for obtaining the resulting vector M_2, as shown on the Fig. 4, 3). Since the scheme shown in Fig. 4 provides the calculation of two adjacent columns of the resulting matrix, four such iterations are required to obtain the entire matrix. Below is a complete source code listing of the function `mul_8x8`, implementating matrices multiplication in accordance with the described approach (Listing 2).

Listing 2. Direct approach to mul_8x8 implementation

```
void mul_8x8(float * __restrict a, float * __restrict b,
             float * __restrict r)
{
    .......................................
    __m512 bj, bj2,
           m0, m1, m2, m3, m4, m5, m6, m7;

    // Indices for matrix columns.
    __m512i ind_cc = _mm512_set_epi32(7*V8+1, 6*V8+1, 5*V8+1, 4*V8+1,
                                      3*V8+1, 2*V8+1,   V8+1,      1,
                                      7*V8  , 6*V8  , 5*V8  , 4*V8  ,
                                      3*V8  , 2*V8  ,   V8  ,      0);
    __m512i ind_st = _mm512_set_epi32(7*V8  , 7*V8+1, 5*V8  , 5*V8+1,
                                      3*V8  , 3*V8+1,   V8  ,   V8+1,
                                      6*V8+1, 6*V8  , 4*V8+1, 4*V8  ,
                                      2*V8+1, 2*V8  ,      1,      0);

    // Load the rows of matrix "a".
    __m512 a0 = LD(&a[0]);
    .......................................
    __m512 a3 = LD(&a[6 * V8]);

    // Matrix "b" columns loop.
    for (int j = 0; j < V8; j += 2)
    {
        bj = _mm512_i32gather_ps(ind_cc, &b[j], _MM_SCALE_4);
        bj2 = _mm512_permute4f128_ps(bj, _MM_PERM_BADC);

        // Matrix "a" rows on matrix "b" columns multiplication.
        m0 = MUL(a0, bj);
        m1 = MUL(a0, bj2);
        .......................................
        m6 = MUL(a3, bj);
        m7 = MUL(a3, bj2);

        // Parallel calculation of vectors sums of elements.
        m0 = SWIZ_2_ADD_2_BLEND_1(m0, m1, _MM_SWIZ_REG_CDAB, 0xAAAA);
        m1 = SWIZ_2_ADD_2_BLEND_1(m2, m3, _MM_SWIZ_REG_CDAB, 0xAAAA);
        m2 = SWIZ_2_ADD_2_BLEND_1(m4, m5, _MM_SWIZ_REG_CDAB, 0xAAAA);
        m3 = SWIZ_2_ADD_2_BLEND_1(m6, m7, _MM_SWIZ_REG_CDAB, 0xAAAA);
        m0 = SWIZ_2_ADD_2_BLEND_1(m0, m1, _MM_SWIZ_REG_BADC, 0xCCCC);
        m1 = SWIZ_2_ADD_2_BLEND_1(m2, m3, _MM_SWIZ_REG_BADC, 0xCCCC);
        m2 = PERM_2_ADD_2_BLEND_1(m0, m1, _MM_PERM_CDAB, 0xF0F0);

        // Store the result.
        _mm512_i32scatter_ps(&r[j], ind_st, m2, _MM_SCALE_4);
    }
```

The experiments carried out demonstrate the effectiveness of this approach in relation to matrices of size 16×16. However, using smaller matrices, a decrease in performance was observed. The main reason for this is the use of slow multiple memory access operations with arbitrary offsets [33]. To eliminate this drawback, we consider another approach, in which calls to memory are performed only by sequential addresses. At the same time, we note that this approach is applicable only for matrices of size no more than 8×8, since it requires a large number of `zmm` registers. In case of shortage of `zmm` registers, these registers are immediately dumped into memory, which causes extra store and load commands (so-called spill/fill operations appear) and leads to a catastrophic performance decrease.

So, when forming another approach to the multiplication of small-format matrices, we will consider the removal of multiple memory access operations as well as the use of special combined operations, which allow the expression $\pm a \cdot b \pm c$ to be calculated in one operation, as the main goals. We write the formulas for the i-th column of the resulting matrix:

$$\begin{cases} r_{i0} = a_{i0}b_{00} + a_{i1}b_{10} + \ldots + a_{i7}b_{70}, \\ \ldots, \\ r_{i7} = a_{i0}b_{07} + a_{i1}b_{17} + \ldots + a_{i7}b_{77} \end{cases} \tag{6}$$

or in vector form

$$\overline{r_i} = a_{i0}\overline{b_0} + a_{i1}\overline{b_1} + \ldots + a_{i7}\overline{b_7}. \tag{7}$$

Similar expressions can be written for the row with the number $i+1$:

$$\begin{cases} r_{i+1,0} = a_{i+1,0}b_{00} + a_{i+1,1}b_{10} + \ldots + a_{i+1,7}b_{70}, \\ \ldots \\ r_{i+1,7} = a_{i+1,0}b_{07} + a_{i+1,1}b_{17} + \ldots + a_{i+1,7}b_{77} \end{cases} \tag{8}$$

or in vector form

$$\overline{r_{i+1}} = a_{i+1,0}\overline{b_0} + a_{i+1,1}\overline{b_1} + \ldots + a_{i+1,7}\overline{b_7}. \tag{9}$$

Considering that the `zmm` registers contain 16 float elements, it is advisable to combine the above formulas into one, written in vector form as follows:

$$\begin{pmatrix} \overline{r_i} \\ \overline{r_{i+1}} \end{pmatrix} = \left(\begin{pmatrix} \overline{a_{i0}} \\ \overline{a_{i+1,0}} \end{pmatrix} \circ \begin{pmatrix} \overline{b_0} \\ \overline{b_0} \end{pmatrix}\right) + \left(\begin{pmatrix} \overline{a_{i1}} \\ \overline{a_{i+1,1}} \end{pmatrix} \circ \begin{pmatrix} \overline{b_1} \\ \overline{b_1} \end{pmatrix}\right) + \ldots + \left(\begin{pmatrix} \overline{a_{i7}} \\ \overline{a_{i+1,7}} \end{pmatrix} \circ \begin{pmatrix} \overline{b_7} \\ \overline{b_7} \end{pmatrix}\right), \tag{10}$$

where $\begin{pmatrix} \overline{r_i} \\ \overline{r_{i+1}} \end{pmatrix}$ denotes the combined vector consisting of the vectors $\overline{r_i}$ and $\overline{r_{i+1}}$, $\begin{pmatrix} \overline{b_j} \\ \overline{b_j} \end{pmatrix}$ denotes the combined vector consisting of two copies of the vector $\overline{b_j}$, and the expression $\begin{pmatrix} \overline{a_{ij}} \\ \overline{a_{i+1,j}} \end{pmatrix}$ denotes a vector, the first 8 elements of which are equal to a_{ij}, and the remaining 8 elements are equal to $a_{i+1,j}$ ("$\circ$" is Hadamard product, or elementwise product of vectors). Note that the combined vector obtained according to this formula $\begin{pmatrix} \overline{r_i} \\ \overline{r_{i+1}} \end{pmatrix}$ is located in the memory sequentially, and it can be recorded into memory using the `store` instruction. In this case, we assume that the value of i is even, that is, the vector $\begin{pmatrix} \overline{r_i} \\ \overline{r_{i+1}} \end{pmatrix}$ is properly aligned in the memory. Other combined vectors in this expression are obtained using the `permute` instruction (intrinsic function `_mm512_permutexvar_ps`) applied to the corresponding loaded adjacent rows of the A and B matrices. Thus, the implementation of the above formula does not require the use of slow `gather`/`scatter` instructions, since we don't read or write the columns of the matrices (calculations are performes only with rows). After the required vectors are formed, we need to perform their pairwise elementwise multiplication, and then add them into one vector (8 operations of elementwise multiplication, 7 addition operations). These actions can be performed using combined operations `fmadd`. To calculate the value $\begin{pmatrix} \overline{r_i} \\ \overline{r_{i+1}} \end{pmatrix}$, we need 8 vector operations (1 operation `mul` and 7 operations `fmadd`).

Below is a complete source code listing of the function `mul_8x8`, implementating matrices multiplication in accordance with the described approach (Listing 3).

Listing 3. Approach to mul_8x8 implementation with line-by-line reading of matrixes from memory

```
void mul_8x8(float * __restrict a, float * __restrict b,
             float * __restrict r)
{
    ........................................
    // Vector duplication indices.
    __m512i ind_df = _mm512_set_epi32( 7,  6,  5,  4,  3,  2, 1, 0,
                                       7,  6,  5,  4,  3,  2, 1, 0);
    __m512i ind_ds = _mm512_set_epi32(15, 14, 13, 12, 11, 10, 9, 8,
                                      15, 14, 13, 12, 11, 10, 9, 8);

    // Load and duplicate all rows of the matrix "b".
    __m512 b0 = LD(&b[0]);
    __m512 b1 = _mm512_permutexvar_ps(ind_ds, b0);
    b0 = _mm512_permutexvar_ps(ind_df, b0);
    ........................................
    __m512 b6 = LD(&b[6 * V8]);
    __m512 b7 = _mm512_permutexvar_ps(ind_ds, b6);
    b6 = _mm512_permutexvar_ps(ind_df, b6);

    // Load all rows of the matrix "a".
    __m512 a0 = LD(&a[0]);
    ........................................
    __m512 a6 = LD(&a[6 * V8]);

    // 8 indices for select elements from matrix "a".
    __m512i ind_0 = _mm512_set_epi32( 8,  8,  8,  8,  8,  8,  8,  8,
                                      0,  0,  0,  0,  0,  0,  0,  0);
    ........................................
    __m512i ind_7 = _mm512_set_epi32(15, 15, 15, 15, 15, 15, 15, 15,
                                      7,  7,  7,  7,  7,  7,  7,  7);

// Main block of operations.
#define BLOCK(N, A)                           \
    ST(&r[N * V8],                            \
      FMADD(PERMXV(ind_0, A), b0,             \
    ........................................
                  FMADD(PERMXV(ind_6, A), b6, \
                    MUL(PERMXV(ind_7, A), b7))))))));

    // Calculate and store the result (4 blocks).
    BLOCK(0, a0);
    ........................................
    BLOCK(6, a6);

#undef BLOCK

}
```

To implement the matrices multiplication in this function first we perform a full load of both matrices A and B. This requires 8 `load` operations, since two adjacent rows of the matrix are loaded in one operation. Next, we need to form 8 vectors of the form $\begin{pmatrix}\overline{b_j}\\ \overline{b_j}\end{pmatrix}, j \in [0,7]$, for which we will need 8 more `permute` operations (for each pair of loaded rows of matrix B, we need to duplicate the first row and duplicate the second row). The formation of index vectors for `permute` operations does not require computational time, since the indices are static and are calculated at the compilation stage. After preparing all the necessary data, the elements of the resulting matrix are calculated. The block of operations `BLOCK` performs the calculation of two adjacent rows of the resulting matrix. The block implementation consists of 8 `permute` operations, 1 `mul` operation and 7 `fmadd` operations, besides,

Table 2. Speedup of `mul_8x8`, `mul_7x7`, `mul_6x6`, `mul_5x5` functions for different approaches to optimization

Matrices size	*Vect Old*	*Full Unroll*	*Vect New*
8×8	2.69	3.69	5.86
7×7	2.04	2.74	4.66
6×6	1.38	2.29	3.68
5×5	0.80	1.44	2.50

one `store` operation of writing data to the memory. A total of four such blocks are performed, which, in sum and taking into account data preparation operations, leads to the following result: 8 simple `load` operations from memory, 40 `permute` operations, 4 `mul` operations, 28 `fmadd` operations, 4 `store` operations to memory.

Note that 28 vector operations `fmadd` and 4 vector operations `mul` correspond to $(28 \cdot 2 + 4) \times 16 = 960$ scalar operations, which exactly coincides with the number of scalar operations required to perform the multiplication of two 8×8 matrices. Thus, in the proposed implementation there are no unnecessary arithmetic operations, and the result of each performed operation affects the final result. When implementing multiplication of 7×7, 6×6, 5×5 matrices, superfluous vector operations are removed (for example, multiplication by a vector, all elements of which are zero), but still there are elements of vectors whose processing is excessive, which leads to a decrease in the efficiency of vectorization in these cases.

The approaches to the vectorization of matrix multiplication described in this secton were tested on the MVS-10P supercomputer placed in JSCC RAS, on its computational segment containing Intel Xeon Phi 7290 KNL microprocessors. Four functions were considered: `mul_8x8`, `mul_7x7`, `mul_6x6`, `mul_5x5`, performing the multiplication of matrices of the corresponding sizes.

For each function, 3 implementation options were considered. As the first option, the old vectorization method with parallel calculation of the sums of vector elements (denoted by *Vect Old*) was used. As the second option, direct manual calculation of each element of the resulting matrix in which all cycles were deleted and for each element multiple copies were written scalar code (denoted by *Full Unroll*) was taken. As the third option, the considered approach based on referring only to matrix rows and using combined fmadd operations (denoted *Vect New*) was used. All speedup results are collected in Table 2.

As can be seen from Table 2, *Vect Old* turned out to be the least efficient approach, its use is even less profitable than the direct writing of a scalar code. For matrices of size 5×5, this optimization method completely slows down the original non-optimized version of the function. The *Vect New* method demonstrates the best results from the described approaches, the vectorized code for 8×8 matrices multiplication is almost 6 times faster than the original code. As the matrix dimensions decrease, the vectorization efficiency decreases too, but even for 5×5 matrices, an acceleration of about 2.5 times is observed, which makes it possible to implement this method in an industrial code.

4. VECTORIZATION OF LOOPS WITH IRREGULAR NUMBER OF ITERATIONS

Loop nests with an irregular number of iterations are the most difficult context for code vectorization. Such specifics of the program code creates more difficulties for vectorization than even the presence of function calls inside loops (of course, if these are calls without side effects). At the same time, cycles with an irregular number of iterations are often encountered in industrial program codes. As a matter of fact, implementation of practically any iterative numerical method belongs to such a context. It is worth noting that the irregular number of iterations and the non-constant number of iterations are slightly different things. In the case of a non-constant number of iterations, we can observe a picture when the number of iterations of a particular cycle changes in time, but the changes themselves are quite small. To a greater extent this is typical for physical tasks. When we talk about a loop with an irregular number of iterations, we mean that this number can vary in time arbitrarily sharply and unpredictably. Such situations can often be observed in problems of discrete mathematics (for example in sorting algorithms [34–36]). As an example, we will consider the classical implementation of Shell sorting [37], the number of iterations of the inner loop for which is fundamentally irregular.

4.1. Shell Sorting Description

Shell sorting algorithm, presented in Listing 4, is an improved version of insertion sort, proposed in 1959 by Donald L. Shell. Unlike the insertion sort, in which the element of the array is moved only by one position each time, which leads to the quadratic complexity of the algorithm, Shell sorting allows you to compare and swap elements that are on a large distance from each other, performing the so-called coarse sort passes. At the beginning of the algorithm, the starting step of k_0 is selected, after which the subarrays with indexes $j|j = j_0 + ik_0$ for all $j_0 < k_0$ are sorted. Further, a similar procedure is performed with the following increments $k_1 < k_0$. The number of preliminary rough passes can be arbitrary. In the end, the final pass is performed in increments equal to one, which is the trivial insertion sort. The average running time of the algorithm depends on the choice of pass steps. Listing 4 presents an implementation of Shell sorting in the C with a simple sequence of steps with an initial step equal to half the size of the array with a decrease in the width of the next pass in half (such a sequence was originally used by Shell).

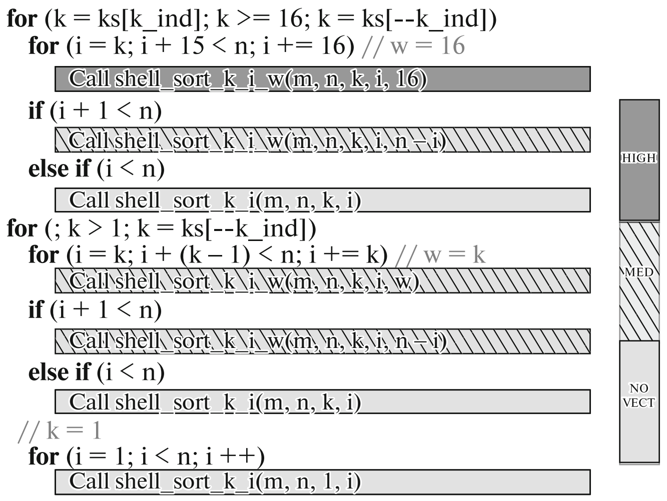
Listing 4. Shell sorting

```

void shell_sort(float *m, int n)
{
    int i, j, k;

    for (k = n / 2; k > 0; k /= 2)      // outer loop
    {
        for (i = k; i < n; i++)         // intermediate loop
        {
            float t = m[i];

            for (j = i; j >= k; j -= k) // inner loop
            {
                if (t < m[j - k])
                {
                    m[j] = m[j - k];
                }
                else
                {
                    break;
                }
            }

            m[j] = t;
        }
    }
}
```

4.2. Vectorization of Shell Sorting with AVX-512 Instructions

Consider the possibilities for vectorization of Shell sorting for an array of float type real values (the AVX-512 vector contains 16 such values). The most nested loop (the loop with the counter j in Fig. 5, we will call it simply internal) performs the sorting of one slice consisting of array of elements, with the distance k between adjacent elements. The internal loop cannot be vectorized without making additional code modifications, since there is an inter-iteration dependence between writing the $m[j]$ element and reading the $m[j-k]$ element. However, it can be noted that two iterations of the middle nesting cycle (the cycle with the inductive variable i, we will call it intermediate) with the numbers i_1 and i_2 do not intersect the data and can be executed in parallel if the condition $|i_1 - i_2| < k$ is fulfilled. Perform the Shell sorting decomposition so that you can explicitly highlight a kernel that can be vectorized.

In Fig. 5 the decomposition scheme of Shell sort algorithm is presented, also sections with different widths of vectorization are shown. First, the block for steps $k \geq 16$ is depicted. For these values of steps, it is possible to perform 16 adjacent iterations of the intermediate loop in parallel, while achieving the maximum vectorization density (shown in dark grey in the figure). All iterations of the intermediate loop are divided into groups of 16 adjacent iterations and the remainder, which is vectorized with a



```
for (k = ks[k_ind]; k >= 16; k = ks[--k_ind])
  for (i = k; i + 15 < n; i += 16) // w = 16
    Call shell_sort_k_i_w(m, n, k, i, 16)
  if (i + 1 < n)
    Call shell_sort_k_i_w(m, n, k, i, n – i)
  else if (i < n)
    Call shell_sort_k_i(m, n, k, i)
for (; k > 1; k = ks[--k_ind])
  for (i = k; i + (k – 1) < n; i += k) // w = k
    Call shell_sort_k_i_w(m, n, k, i, w)
  if (i + 1 < n)
    Call shell_sort_k_i_w(m, n, k, i, n – i)
  else if (i < n)
    Call shell_sort_k_i(m, n, k, i)
// k = 1
  for (i = 1; i < n; i ++)
    Call shell_sort_k_i(m, n, 1, i)
```

Fig. 5. Shell sorting decomposition with highlighted segments of vectorized code.

width of less than 16 (shown in shaded grey, and if the remainder consists of just one iteration, then no vectorization is required, which is shown in light grey in the figure). Next, the block of step values $1 < k < 16$ is considered. With these values, the width of the vectorization is always less than 16, moreover, as in the previous block, a non-vectorizable remainder may appear. The last to be considered is the non-vectorizable final insertion sort for $k = 1$. The presence of code sections with a vectorization width of less than 16 leads to a non-optimal result code, but there is a more dangerous reason for the low efficiency of vectorization.

The `shell_sort_k_i_w` function, which appeared after decomposing the Shell sorting algorithm, contains the implementation of sorting w adjacent array slices, taken with step k. At the same time, the number of iterations of the inner loop of this function is unknown. Moreover, the number of iterations of the inner loop when sorting one slice is in no way connected with the number of iterations of the inner loop when sorting the next slice. This is a significant problem when attempting to merge the code for sorting adjacent slices using vector instructions. For such a union, it is necessary to rewrite the slice sorting code in predicate form, then replace all instructions with vector analogs, and predicates with vector masks. Moreover, if, prior to vectorization, the internal loop terminated when the predicate is false, then, after vectorization, the internal loop will terminate only if all elements of the corresponding vector mask are reset. Thus, the number of iterations of the vectorized inner loop is equal to the maximum number of iterations of all w combined cycles. If the values of the number of iterations of adjacent merged cycles differ greatly (and for Shell sorting this statement is true), then we get a loss of vectorization efficiency due to the low density of vector instruction masks (that is, a small percentage of vector elements is actually processed when performing vector operation).

Vectorized version of Shell sorting kernel presented in Listing 5. Also worth paying attention to the scatter operation, which appeared in the vector code, writing multiple data into memory with arbitrary offsets relative to the base address. This command appeared as a vector analogue of a write operation to memory from the original code due to the same reason—the irregularity of the number of iterations of the internal loop. It is worth noting that the scatter commands are extremely slow, which also causes a decrease in the efficiency of vectorization.

In addition to the designated scatter command, there are other memory reference commands in the vectorized code (Listing 5, lines 10, 16, 19). These are the usual load and store commands that, generally speaking, must be addressed by aligned addresses in memory. In the general case, the addresses given to them are certainly not aligned, however, the use of non-aligned calls instead of them significantly slows down the code, so it was decided to leave these instructions.

Listing 5. Vectorized version of Shell sorting kernel

```
void shell_sort_k_i_w(float *m, int n, int k, int i, int w)
{
    int j = i;
    __mmask16 ini_mask = ((unsigned int)0xFFFF) >> (16 - w);
    __mmask16 mask = ini_mask;
    __m512i ind_j = _mm512_add_epi32(_mm512_set1+epi32(i),
                                     ind_straight);
    __m512 t, q;

    t = _mm512_mask_load_ps(t, mask, &m[j]);

    do
    {
        mask = mask & _mm512_mask_cmp_epi32_mask(mask, ind_j, ind_k,
                                                  __MM_CMPINT_GE);
        q = _mm512_mask_load_ps(q, mask, &m[j - k]);
        mask = mask & _mm512_mask_cmp_ps_mask(mask, t, q,
                                               __MM_CMPINT_LT);
        _mm512_mask_store_ps(&m[j], mask, q);
        ind_j = _mm512_mask_sub_epi32(ind_j, mask, ind_j, ind_k);
        j -= k;
    }
    while (mask != 0x0);

    _mm512_mask_i32scatter_ps(m, ini_mask, ind_j, t, _MM_SCALE_4);
}
```

4.3. Acceleration Measurements

By theoretical acceleration, we will simply mean the ratio of the number of iterations of the internal loop of the non-vectorized version to the one in the optimized vectorized version of the code. Define this acceleration more formally.

First consider the non-vectorized code. Denote by $I(k,i)$ the number of iterations of the inner loop with fixed k and i. Then it is not difficult to calculate the total number of iterations of the inner loop when performing sorting (we denote this value as T)

$$T = \sum_{k\in ks}\sum_{i=k}^{n-1} I(k,i). \qquad (11)$$

Now consider the vectorized version of the code. We also denote by $I_v(k,i')$ the number of iterations of the inner loop with fixed k and i'. We know that the vectorization width cannot exceed k (due to dependencies on accessing the array), and 16 (vector size), that is, $w(k) = \min\{k,16\}$. Moreover, the entire range of i values from k to $n-1$, whose length is $n-k$ is divided into $\lfloor \frac{n-k}{w(k)} \rfloor$ groups in w elements in each, and the number of iterations in the vectorized cycle for each group is equal to the maximum value of the iterations of the non-vectorized cycles combined into this vectorized loop

$$I_v(k,i') = \max_{i=i'}^{\min(i'+w(k)-1,n-1)} I(k,i). \qquad (12)$$

Taking into account the vectorization of the tail part of the loop, we obtain the following formula for the total number of iterations of a vectorized inner loop

$$T_v = \sum_{k\in ks}\left(\left(\sum_{g=0}^{G(k)-1} \max_{i=k+w(k)g}^{k+w(k)(g+1)-1} I(k,i)\right) + \max_{i=k+w(k)G(k)}^{n-1} I(k,i)\right), \qquad (13)$$

where $w(k) = \min\{k,16\}$, $G(k) = \lfloor \frac{n-k}{w(k)} \rfloor$. The values of $T = T(n)$ and $T_v = T_v(n)$ were calculated when sorting pseudo-random arrays with the number of elements from 10 thousand to 2 million

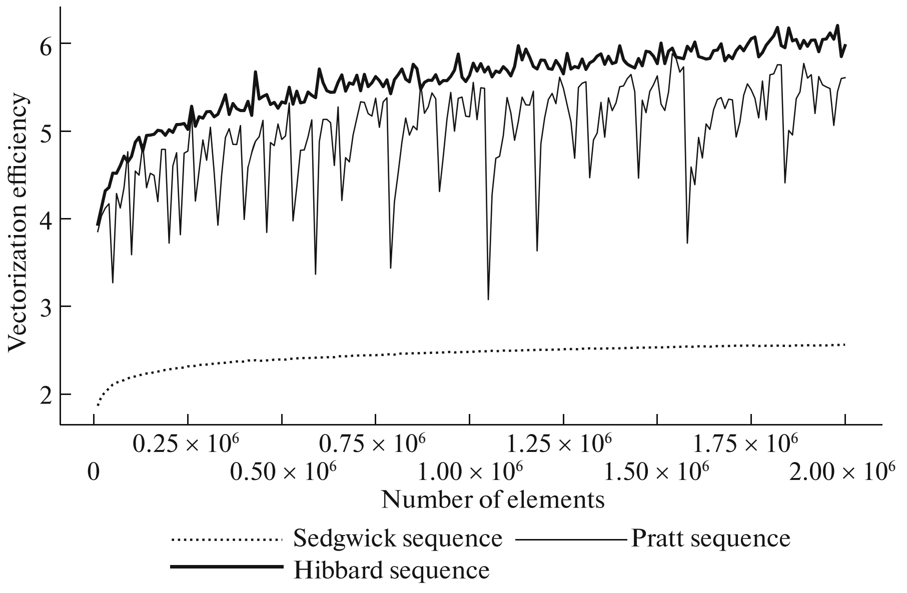


Fig. 6. Comparison of the theoretical acceleration of a vectorized version of Shell sorting for different sequences of steps.

for each of the sequences of steps of Shell, Hibbard, Pratt and Sedgewick. Based on this, the theoretical acceleration $s(n) = T(n)/T_v(n)$ was calculated. In addition, similar characteristics were calculated without taking into account the step $k = 1$ (these values are denoted $T'(n)$, $T_v'(n)$ and $s'(n)$, respectively). The results were compared with the results of experimental launches on Intel Xeon Phi 7290 KNL microprocessor.

On the Fig. 6 and Fig. 7 the results of theoretical estimates and experimental launches are presented. The figures show the dependence of vectorization efficiency on number of elements for Sedgwick, Pratt and Hibbard step sequences. It can be seen from the figures that even theoretical acceleration is far from ideal upper limit (which is equal to 16) of the acceleration achieved when vectorizing nested loops with a regular number of iterations. The experimental results, as was expected, are much lower and, finally, acceleration of Shell sorting barely exceeds the 2 mark.

5. PHYSICAL CALCULATIONS VECTORIZATION

In this section, we will consider the application of the described approaches for vectorization of program code to the implementation of physical calculations. The object of the study was the Riemann problem of the decay of an arbitrary discontinuity, on the exact or approximate solution of which many numerical methods for modeling gas-dynamic processes are based [38]. The core of the program code in which the solution of the Riemann problem is implemented is called the Riemann solver. The implementation of the Riemann solver considered in this article is publicly available on the Internet as part of the NUMERICA library [39]. In this case, we will be interested in the one-dimensional case for a single-component medium, implemented as a pure function (function without side effects), which, by the density, velocity and gas pressure values to the left and right of the discontinuity, finds the values of the same quantities on the discontinuity itself at zero time after removal of the septum.

$$U_l = \begin{pmatrix} d_l \\ u_l \\ p_l \end{pmatrix}, \quad U_r = \begin{pmatrix} d_r \\ u_r \\ p_r \end{pmatrix}, \quad U = \begin{pmatrix} d \\ u \\ p \end{pmatrix} = riemann(U_l, U_r). \tag{14}$$

The NUMERICA library is implemented in the FORTRAN programming language, therefore the vectorization of this code using intrinsic functions is not directly possible, so the version of the code ported to the C programming language was used.

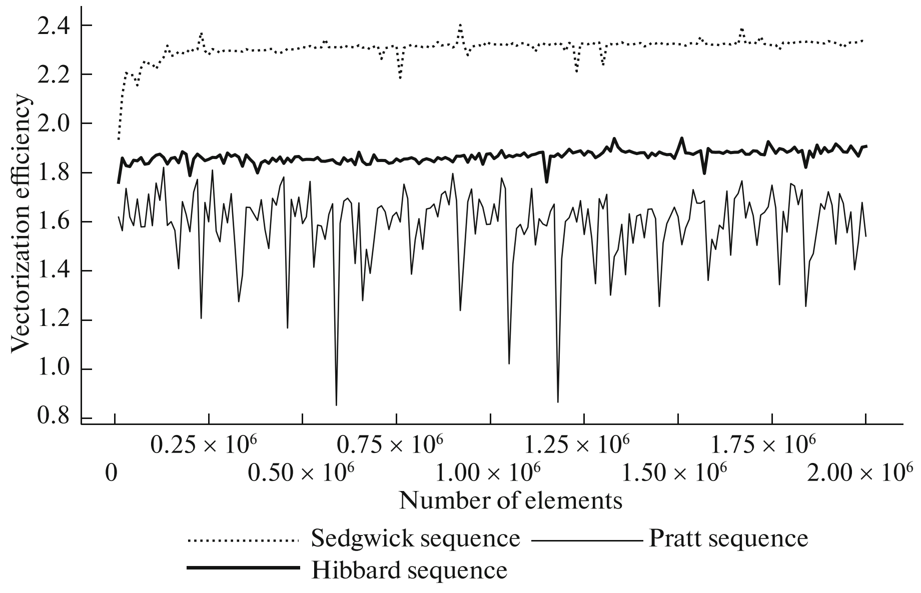


Fig. 7. Comparison of experimental acceleration of a vectorized version of Shell sorting for different sequences of steps.

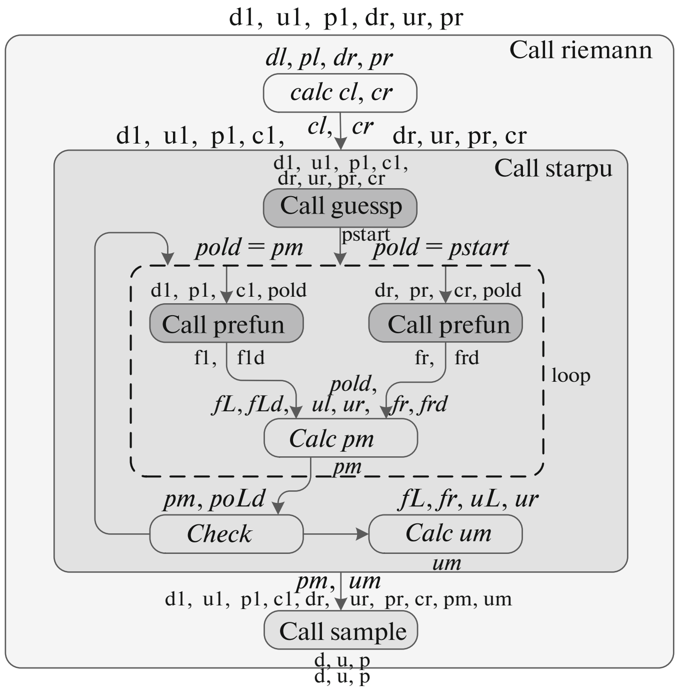


Fig. 8. Riemann solver scheme.

In Fig. 8 the scheme of work of the Riemann solver with the indicated data streams and calls of all functions included in the implementation is shown. The `riemann` function calculates the speed of sound on the right and left, performs a vacuum test and successively calls the `starpu` and `sample` functions. The `starpu` function calculates the velocity and pressure values in the middle region between the left and right waves (star region), and the function contains a cycle with an unknown number of iterations to solve a nonlinear equation using the Newton iterative method, inside which there are calls to other functions (`prefun`). The `guessp` and `prefun` functions contain only arithmetic calculations and simple conditions and are the simplest from the point of view of vectorization. Finally, the last `sample` function determines the final configuration of the discontinuity by calculating a set of conditions. This function contains a very extensive control, the nesting of conditions in it reaches four, which makes it difficult to use vectorization.

In the counting process, numerous `riemann` function calls are made with different input data sets using numerical methods based on the Riemann solver (each call iteration makes one call for each face of each cell of the computational grid). Since the riemann function is pure, calls for different input data sets $(d_l, u_l, p_l, d_r, u_r, p_r)$ are independent and there is a desire to combine calls with the goal of effectively using vector (element) instructions. As such a combined call, we will consider a function into which, instead of atomic data of the float type, the corresponding vectors containing 16 elements each are fed

$$\overline{U_l} = \begin{pmatrix} U_l^1 \\ \dots \\ U_l^{16} \end{pmatrix}, \quad \overline{U_r} = \begin{pmatrix} U_r^1 \\ \dots \\ U_r^{16} \end{pmatrix}, \quad \overline{U} = \begin{pmatrix} U^1 \\ \dots \\ U^{16} \end{pmatrix} = riemann^{16}(\overline{U_l}, \overline{U_r}). \tag{15}$$

At the same time, it is possible to perform the same actions with vector data as with basic types—perform calculations, transfer to functions, and return as a result. In the process of optimization, for clarity, it will avoid substitution of the function body into the call point. Thus, as a result of vectorization, our goal is to obtain the vector analogs of all the functions described above used in the Riemann solver.

The `prefun` and `guesssp` functions contain the usual arithmetic calculations and are vectorised by direct replacement of all instructions with vector analogs. At the same time, the functions do not contain complex control, therefore all calculations can be directly translated from the predicate code and combined into a single branch of execution.

The `sample` function contains a highly branched control with a nesting level of 4. The condition tree constructed for this function contains 10 leaf nodes, each contains the determined value of the gas-dynamic quantities d, u, p. Direct computation of the vector predicates of all leaf nodes and the execution of their code under these predicates leads to a slowdown of the resulting code, therefore, the following actions were performed when vectoring this function. First, it was noted that the 4 linear sections contain the definition of the gas-dynamic parameters d, u, and p, which could be changed to initialization by moving the assignment operations up through the function code. Thus, 4 linear sections were deleted. At the same time, this initialization of the d, u, p parameters does not contain arithmetic operations (the parameters are initialized by the arguments of the function d_l, u_l, p_l or d_r, u_r, p_r), which means it can be performed using the vector merge `blend` operations. It was further noted that the `sample` function processes the right and left profiles of the breakup decay in the same way with minor changes. Using a simple change of variables, which consists in changing the sign of the velocity value, we merged the code for two subtrees based on the condition $p_m \leq p_l$, which reduced the amount of the calculation code by half and expectedly reduced the execution time by 45%. After the reduction of the code, the number of leaf nodes in the conditions tree was reduced to three. However, even in this case, direct merging of the code using vector predicates turned out to be ineffective. This was due to the unlikely part of the code, heavy operations, among which there are calls to the function of raising to a power. The vector predicate of this code segment in more than 95 percent of cases has a value of 0x0, therefore, before executing this code section, it is advisable to check this predicate for emptiness (which corresponds to removing the unlikely branch of execution from the function body). The removal of an unlikely branch of execution from the main program context can significantly speed up the executable code, since the presence of a large number of such rare computations can serve as a reason for refusing vectorization. As a result, the vectorized `sample` function was accelerated more than 10 times.

The `starpu` function is the most complex because it contains a loop with a variable number of iterations and other functions calls. The approach described in the previous section is applied to this function. In addition, although the loop in this function contains an unknown number of iterations, this value changes slowly, which leads to a high density of masks for executable commands in vectorized code. In addition, it can be noted that the average number of loop iterations is equal to 3, which also reduces the number of unnecessary operations in the resulting code. As a result, the vectorized version of the function `starpu` function turns out to be faster than the original version more than 5 times.

The approaches to the vectorization of the Riemann solver functions described in the section were implemented in the C programming language using intrinsic functions and tested on microprocessors Intel Xeon Phi 7290, which are part of the computational segment KNL of supercomputer MVS-10P, located in the JSCC RAS.

Performance testing was performed on the input data arrays collected when solving standard test problems: the Sod problem, the Lax problem, the weak shock wave problem, the Einfeldt problem, the Woodward–Colella problem, the Schu–Osher problem and others. As a result of the executed transformations of the source code, the Riemann solver accelerated by 7 times compared to the non-optimized versions.

6. CONCLUSION

The approaches to vectorization of program code considered in the article allow us to conclude about the applicability of the AVX-512 instruction set to loops of almost any kind, used in scientific calculations. The features of the AVX-512 instruction set, including the use of vector masks, allow to vectorize any program code written in predicate form. Thus, loops can be vectorized, containing a complex settlement, function calls and loop nests with an unknown number of iterations. All aspects of the program code vectorization considered in the article were implemented in the C programming language and proved to be effective, as demonstrated by the practical results obtained.

FUNDING

The work has been done at the Joint Supercomputer Center of the Russian Academy of Sciences as part of the state assignment for the topic 0065-2019-0016 (reg. no. AAAA-A19-119011590098-8). The supercomputer MVS-10P, located at the JSCC RAS, was used for calculations during the research.

REFERENCES

1. C. Rettinger, C. Godenschwager, S. Eibl, et al., “Fully resolved simulations of dune formation in riverbeds,” Lect. Notes Comput. Sci. **10266**, 3–21 (2017).
2. T. Krappel and S. Riedelbauch, “Scale resolving flow simulations of a Francis turbine using highly parallel CFD simulations,” in *Proceedings of the Conference on High Performance Computing in Science and Engineering'16* (2016), pp. 499–510.
3. S. Markidis, I. B. Peng, J. L. Träff, et al., “The EPiGRAM project: preparing parallel programming models for exascale,” Lect. Notes Comput. Sci. **9945**, 56–68 (2016).
4. B. Klenk and H. Fröning, “An overview of MPI characteristics of exascale proxy applications,” Lect. Notes Comput. Sci. **10266**, 217–236 (2016).
5. M. Abduljabbar, G. S. Markomanolis, H. Ibeid, et al., “An overview of MPI characteristics of exascale proxy applications,” Lect. Notes Comput. Sci. **10266**, 79–96 (2017).
6. A. A. Rybakov, “Inner respresentation and crossprocess exchange mechanism for block-structured grid for supercomputer calculations,” Program. Sist.: Teor. Prilozh. **32** (8:1), 121–134 (2017).
7. R. F. van der Wijngaart, E. Georganas, T. G. Mattson, et al., “A new parallel research Kernel to expand research on dynamic load-balancing capabilities,” Lect. Notes Comput. Sci. **10266**, 256–274 (2017).
8. L. A. Benderskiy, D. A. Lyubimov, and A. A. Rybakov, “Analysis of scaling efficiency in high-speed turbulent flow calculations on a RANS/ILES supercomputer using the high resolution method,” Tr. SRISA RAS **7** (4), 32–40 (2017).
9. T. Heller, H. Kaiser, P. Diehl, et al., “Closing the performance gap with modern C++,” Lect. Notes Comput. Sci. **9945**, 18–31 (2016).
10. V. Roganov, V. Osipov, and G. Matveev, “Solving the 2D Poisson PDE by Gauss-Seidel method with parallel programming system,” Program. Sist.: Teor. Prilozh. **30** (7:3), 99–107 (2016).
11. J. Jeffers, J. Reinders, and A. Sodani, *Intel Xeon Phi Processor High Performance Programming, Knights Landing Edition* (Morgan Kaufmann, 2016).
12. J. Jeffers and J. Reinders, *Intel Xeon Phi Coprocessor Processor High Performance Programming* (Morgan Kaufmann, 2013).
13. J. Dorris, J. Kurzak, and P. Luszczek, “Task-based Cholesky decomposition on knights corner using OpenMP,” Lect. Notes Comput. Sci. **9945**, 544–562 (2016).
14. J. Tobin, A. Breuer, A. Heinecke, et al., “Accelerating seismic simulations using the Intel Xeon Phi Knights landing processor,” Lect. Notes Comput. Sci. **10266**, 139–157 (2017).
15. W. McDoniel, M. Hohnerbach, R. Canales, et al., “LAMMPS' PPPM long-range solver for the second generation Xeon Phi,” Lect. Notes Comput. Sci. **10266**, 61–78 (2017).
16. T. Malas, T. Kurth, and J. Deslippe, “Optimization of the sparse matrix-vector products of an IDR Krylov iterative solver in EMGeo for the Intel KNL manycore processor,” Lect. Notes Comput. Sci. **9945**, 378–389 (2016).

17. O. Krzikalla, F. Wende, and M. Höhnerbach, "Dynamic SIMD vector lane scheduling," Lect. Notes Comput. Sci. **9945**, 354–365 (2016).
18. B. Cook, P. Maris, and M. Shao, "High performance optimizations for nuclear physics code MFDn on KNL," Lect. Notes Comput. Sci. **9945**, 366–377 (2016).
19. A. A. Rybakov, "Optimization of the problem of conflict detection with dangerous aircraft movement areas to execute on Intel Xeon Phi," Program. Produkty Sist. **30**, 524–528 (2017).
20. D. Sengupta, Y. Wang, N. Sundaram, et al., "Performance incremental SVM learning on Intel Xeon Phi processors," Lect. Notes Comput. Sci. **10266**, 120–138 (2017).
21. M. Kronbichler, K. Kormann, and I. Pasichnyk, "Fast matrix-free discontinuous Galerkin kernels on modern computer architectures," Lect. Notes Comput. Sci. **10266**, 237–255 (2017).
22. D. Doerfler, J. Deslippe, and S. Williams, "Applying the roofline performance model to the Intel Xeon Phi Knights landing processor," Lect. Notes Comput. Sci. **9945**, 339–353 (2016).
23. C. Rosales, J. Cazes, and K. Milfeld, "Comparative study of application performance and scalability on the Intel Knights landing processor," Lect. Notes Comput. Sci. **9945**, 307–318 (2016).
24. *Intel 64 and IA-32 Architectures Software Developer's Manual,* (Intel Corp., 2017), Combined Volumes: 1, 2A, 2B, 2C, 2D, 3A, 3B, 3C, 3D and 4.
25. *Intel C++ Compiler 16.0 User and Reference Guide* (Intel Corp., 2016).
26. Intel Intrinsics Guide. https://software.intel.com/sites/landingpage/IntrinsicsGuide/. Accessed 2018.
27. S. A. Mahlke, D. C. Lin, W. Y. Chen, and R. E. Hank, "Effective compiler support for predicated execution using the hyperblock," in *Proceedings of the 25th International Symposium on Microarchitecture, 1992,* pp. 45–54.
28. W. W. Hwu, "The superblock: an effective technique for VLIW and superscalar compilation," J. Supercomput. **7**, 229–248 (1993).
29. G. H. Golub and C. F. van Loan, *Matrix Computations* (John Hopkins Univ. Press, 1989).
30. H. Zhang, R. T. Mills, K. Rupp, and B. F. Smith, "Vectorized parallel sparse matrix-vector multiplication in PETSc Using AVX-512," in *Proceedings of the 47th International Conference on Parallel Processing ICPP 2018* (ACM, 2018), No. 55.
31. D. A. Lyubimov, "Development and application of a high-resolution technique for jet flow computation using large eddy simulation," High Temp. **50**, 420–436 (2012).
32. L. A. Benderskii, D. A. Lyubimov, A. O. Chestnykh, B. M. Shabanov, and A. A. Rybakov, "The use of the RANS/ILES method to study the influence of coflow wind on the flow in a hot, nonisobaric, supersonic airdrome jet during its interaction with the jet blast deflector," High Temp. **56**, 247–254 (2018).
33. F. Aleen, V. P. Zakharin, R. Krishnaiyer, G. Gupta, D. Kreitzer, and C.-S. Lin, "Automated compiler optimization of multiple vector loads/stores," Int. J. Parallel Program. **46**, 471–503 (2018).
34. B. Bramas, "Fast sorting algorithms using AVX-512 on Intel Knights landing," arXiv: 1704.08579 (2018).
35. S. Gueron and V. Krasnov, "Fast quicksort implementation using AVX instructions," Comput. J. **59**, 83–90 (2016).
36. B. Bramas, "A novel hybrid quicksort algorithm vectorized using AVX-512 on Intel Skylake," Int. J. Adv. Comput. Sci. Appl. **8** (10) (2017).
37. D. E. Knuth, *The Art of Computer Programming,* Vol. 3: *Sorting and Searching,* 2nd ed. (Addison-Wesley Professional, Reading, MA, 1998).
38. E. F. Toro, *Riemann Solvers and Numerical Methods for Fluid Dynamics: A Practical Introduction*, 2nd ed. (Springer, Berlin, Heidelberg, 1999).
39. E. F. Toro, NUMERICA, A Library of Sources for Teaching, Research and Applications. https://github.com/dasikasunder/NUMERICA. Accessed 2018.