ISSN 1995-0802, Lobachevskii Journal of Mathematics, 2020, Vol. 41, No. 12, pp. 2575–2592. © Pleiades Publishing, Ltd., 2020.

Vectorization of Flat Loops of Arbitrary Structure Using Instructions AVX-512

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

(Submitted by A. M. Elizarov)

[1] *Joint Supercomputer Center, Scientific Research Institute for System Analysis of the Russian Academy of Sciences, Moscow, 119334 Russia*

Received May 6, 2020; revised May 31, 2020; accepted June 6, 2020

Abstract—Widespread application of supercomputer technologies in various spheres of life, as well as the need of high-performance calculations allows us to speak about the relevance of the problem of increasing the performance of computer codes on supercomputers of modern architectures. Vectorization of program code is a low-level optimization that can, with a relatively local and compact application, increase the productivity of computational codes by several times. Modern Intel microprocessors have support for a unique set of instructions AVX-512, which, due to its features, allows you to vectorize almost any kind of code written in a predicate form. A set of simple restrictions when developing programs along with vectorization tools to enable the use of the AVX-512 instruction set can significantly speed up the resulting program. The article discusses approaches to vectorization of flat loops—a special-purpose program context, the successful vectorization of which allows to increase the productivity of supercomputer applications even for such program code for which optimizing compilers are powerless.

DOI: 10.1134/S1995080220120331

Keywords and phrases: *supercomputers, vectorization, AVX-512, flat loop, predicated execution, intrinsic function.*

1. INTRODUCTION

Simulations using supercomputers currently play an important role in conducting research in various fields of science, industry, business and society [1–3]. Supercomputer modeling allows you to perform analysis and multi-criteria optimization of the processes of object interaction and extract information that is not available without using these tools. Conducting high-performance calculations implies using models which use computational grids consisting of tens and hundreds of millions of individual nodes or cells. Such mesh sizes are already commonplace, and over time, the requirements for accuracy and complexity of models only become more stringent. In such conditions, improving the performance of supercomputer calculations is an urgent task. Along with increasing the peak performance of the computing systems themselves, studies are underway to improve the efficiency of data exchange systems between computing nodes [4]. There are studies of technologies for managing computational grids [5] and uniform distribution of computations on a cluster [6]. We also see an active development of programming language tools aimed at facilitating the creation of high-performance parallel code [7].

The lowest-level direction for creating high-performance parallel executable code is the vectorization of calculations, which allows you to directly use the hardware capabilities of the computers. The AVX-512 instruction set is a foremost tool of improving the performance of modern Intel microprocessors. In microprocessors with support for the AVX-512 instruction set, the peak performance is calculated

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

taking into account 512-bit vector registers and dual FMA (fused multiply-add) operations. This means that without vectorization of code when using a single-precision real operations performance higher than 3.125% of the peak can not be achieved even theoretically. In view of this, the search for methods and solutions for applying vectorization in a complex program context is an extremely urgent task [8].

Despite the diversity of the possible software context of supercomputer applications, there are types of software constructs whose effective optimization can significantly increase the efficiency of the entire application. In this article, only one type of vectorized program context will be considered—a flat loop, for which vectorization methods using AVX-512 will be proposed.

2. FLAT LOOPS

In the framework of this article, we will restrict ourselves to considering only computational problems working with real numbers of single precision. Let's give the definition of a flat loop for such problems. In general case a flat loop is a program construct with the form shown in Listing 1.

Listing 1. General view of a flat loop

```
1 for (int w = 0; w < WIDTH; w++)
2 {
3     block(w);
4 }
```

We will follow the next agreements. First, we assume that the number of iterations of the flat loop $WIDTH$ is exactly equal to the number of elements in the vector. In our case, we consider 512-bit vectors and real data with single precision (each element takes 32 bits), so $WIDTH = 16$. This restriction does not really violate generality, so we can always divide a loop with any number of iterations into several loops of $WIDTH$ iterations and, possibly, another loop with a smaller number of iterations (loop epilogue).

Second, we assume that inside the code block $block(w)$ there are arrays with only float values, access to which is possible only in the form $x[w]$, in addition, all arrays are 512-bit aligned in memory. Note that there may be program code that does not meet these requirements, but in most cases it is more advantageous to adjust the program architecture and achieve the data alignment requirement than to deal with negative effects in the absence of alignment (unaligned memory access operations take significantly more time, than aligned).

Thirdly, there are no inter-iteration dependences inside the loop. Requirements for calls of the form $x[w]$ to arrays exclude the possibility of inter-iteration dependencies in the loop when accessing arrays, but theoretically there may be conflicts over access to other variables. For example, the loop shown in Listing 2 is not a flat one, because there is dependency in access to s.

Listing 2. Simple example of a non-flat loop

```
1 for (int w = 0; w < WIDTH; w++)
2 {
3     s += x[w];
4 }
```

Loops satisfying the described conventions are extremely convenient in terms of vectorization. The absence of inter-iteration dependencies in such a loop allows iterations of the loop to be performed independently of each other, in any order, which means that they can be performed in parallel. The requirement for the number of iterations in the loop, the alignment in memory, as well as the type of accesses to the elements of arrays ($x[w]$) ensures that inside this loop, when working with some array x, we refer only to its part limited by items from 0 to $WIDTH - 1$. Thus, to access arrays inside a flat loop, we can use vector commands to read aligned sections of memory.

Of course, if the body of a flat loop is limited only by memory accesses and simple arithmetic operations, then to optimize such a loop, an optimizing compiler is enough to easily replace scalar instructions with vector analogues inside the loop. However, we are interested in those loops, whose

bodies have features that impede automatic vectorization. In such loops, the main problems are complex branched control, the presence of nested loops, and function calls inside the loop. All of these issues for flat loops can be resolved using the AVX-512 instruction set. This requires translating the body of the loop into the predicate representation, and then replacing scalar predicate operations with vector predicate instructions.

Since we assumed that a flat loop is always controlled by an inductive variable, varying from 0 to $WIDTH - 1$, later we will characterize a flat loop only by its body. For example for the loop from Listing 1 we say that we consider a flat loop $block(w)$.

Before we proceed to a discussion of specific approaches for vectorizing flat loops using the predicate representation of the loop body, it is necessary to briefly describe the AVX-512 instruction set, and especially instructions suitable for vectorizing flat loops.

3. AVX-512 INSTRUCTION SET

The AVX-512 instruction set [9] is an extension of the AVX and AVX2 sets. It is supported on Intel microprocessors, starting with Intel Xeon Phi Knights Landing, Intel Xeon Skylake, Intel Xeon Cascade Lake. Instructions AVX-512 in their work operate with vector registers zmm. There are 32 such registers (zmm0–zmm31), and they are an extension of the ymm registers (ymm0–ymm31, respectively). Each 512-bit vector register zmm can operate with integer and real data types. For example, a single zmm register may contain 64 integer elements of size 8 bits (byte), 32 integer elements of size 16 bits (word), 16 integer elements of size 32 bits (double), or 8 integer elements of size 64 bits (quadro). Similarly, the zmm register can contain 16 real values of 32 bits (real numbers with single precision), or 8 real values of 64 bits (real numbers of double precision). In addition, the instructions from the AVX-512 set use 8 special mask registers (k0–k7) in their work, which help to selectively apply operations to individual elements of vectors.

The AVX-512 instruction set is constantly updated with the release of new lines of microprocessors. The main subsets of instructions can be distinguished: AVX-512 F—Foundation—Basic instructions working with 32-bit and 64-bit data. This includes element-wise operations on vectors, operations with masks, merging vectors by mask, comparison of vectors, operations of permutation of elements of vectors, conversion operations, and others. AVX-512 CD—Conflict Detection—A set of operations designed to resolve conflicts during loop vectorization. The basic instruction from this set VPCONFLICT allows you to compare each element of the first vector with each element of the second vector. This instruction is used to dynamically check address ranges for conflicts. AVX-512 ER—Exponential and Reciprocal—A set of instructions for element-wise calculation with increased accuracy of the functions 2^x, $1/x$, $1/\sqrt{x}$. AVX-512 PF —Prefetch—Instructions for pre-paging data for reading and writing to addresses with arbitrary offsets (VGATHER / VSCATTER). AVX-512 VL—Vector Length—Extension of many instructions on data elements of size 128 and 256 bits. AVX-512 DQ—Doubleword and Quadword—Contains additional instructions for working with 32 and 64 bit data elements. AVX-512 BW—Byte and Word—Extension of the set of instructions for working with data sizes of 8 and 16 bits. AVX-512 VNNI—Vector Neural Network Instructions—Additional instructions introduced to optimize machine learning algorithms.

The most important distinguishing feature of the AVX-512 instruction set is the presence in them of support for vector predicates (or masks), which allow you to apply the operation to individual elements of vectors.

Let's describe the semantics of the AVX-512 vector instructions suitable for vectorizing flat loops. We denote in small latin letters the data elements we operate with in the process of counting (in this case, these are real data elements of single precision). We write arithmetic operations in their natural form, for example, $r = a + b$ means the calculation of the sum of two data elements. Vectors made up of individual elements will be written using capital latin letters. Thus we say that A is a vector consisting of $WIDTH$ elements $A[w]$. By $R = A + B$ we mean the element-wise sum of the vectors A and B and copying the result into the vector R. Replacing the addition operation with the arbitrary operation op (not necessarily the operation of two arguments), we obtain the semantics of the vector element-wise operation in the form $R = op\ A, B$.

To consider the semantics of vector operations that work with masks, we need a representation of vector predicates. Predicates will be denoted by latin letters with a checkmark at the top. Using the

Table 1. AVX-512 vector single precision float instructions for flat loops vectorization and their semantic

Instructions	Using examples	Instrucition semantic
VMOVA, VMOVU, VSQRT, VGETEXP, VGETMANT, VRCP14, VREDUCE, VRNDSCALE, VRSQRT14, VSCALEF	$R = \text{VMOVA}(A)$	$R = op\ A$
	$R = \text{VMOVA}(\check{P}, A)$	$R = \check{P}\ ?\ (op\ A) : R$
	$R = \text{VMOVAz}(\check{P}, A)$	$R = \check{P}\ ?\ (op\ A) : 0$
VADD, VAND, VANDN, VDIV, VMAX, VMIN, VMUL, VOR, VSUB, VRANGE	$R = \text{VADD}(A, B)$	$R = op\ A, B$
	$R = \text{VADD}(\check{P}, A, B)$	$R = \check{P}\ ?\ (op\ A, B) : R$
	$R = \text{VADDz}(\check{P}, A, B)$	$R = \check{P}\ ?\ (op\ A, B) : 0$
VFMADD132, VFMADD213, VFMADD231, VFMSUB132, VFMSUB213, VFMSUB231, VFNMADD132, VFNMADD213, VFNMADD231, VFNMSUB132, VFNMSUB213, VFNMSUB231	$R = \text{VFMADD132}(R, A, B)$	$R = op\ R, A, B$
	$R = \text{VFMADD132}(\check{P}, R, A, B)$	$R = \check{P}\ ?\ (op\ R, A, B) : R$
	$R = \text{VFMADD132z}(\check{P}, R, A, B)$	$R = \check{P}\ ?\ (op\ R, A, B) : 0$
VCMP (all variety of compare instructions)	$\check{P} = \text{VCMP}(A, B)$	$\check{Q} = op\ A, B$
	$\check{Q} = \text{VCMP}(\check{P}, A, B)$	$\check{Q} = \check{P}\ ?\ (op\ A, B) : 0$
VBLENDM	$R = VBLENDM(\check{P}, A, B)$	$R = \check{P}\ ?\ A : B$

predicate when choosing one of the two arguments will be written using the ternary operator. Thus, in $r = \check{p}\ ?\ a : b$ the element r takes the value of a when the predicate $\check{p}$ is true, otherwise it takes the value of b. The vector analogue of previous notation is $R = \check{P}\ ?\ A : B$. This operation is executed element-wise for elements in positions $w\ (0 \le w < WIDTH)$.

Now we consider the main classes of instructions suitable for vectorization of flat loops (examples of operations and their semantics are given in Table 1).

The first type of operations in question are vector operations with one argument. In this case, the same operation is applied to each element of the vector (for example, obtaining the reciprocal of the value or calculating the square root), after which the results are written into the resulting vector. With the help of an additional predicate argument, you can select subset of processed elements of the vector. If an operation should not be applied to the vector element, then the corresponding element of the resulting vector can either be left unchanged or zeroed (this is regulated by a special flag in the instruction).

The semantics of arithmetic vector instructions with two and three arguments are written in the same way. Operations with two arguments are the usual operations of addition, subtraction, multiplication, obtaining the maximum of two numbers and others. Arithmetic operations with three arguments are the so-called doubled, or combined FMA operations, which allow one to calculate $\pm ab \pm c$ by one operation.

The next large class of operations is comparison operations. In the table, this class is represented by a single VCMP operation, however, this operation hides many different comparison operations (VCMPEQ, VCMPLE, VCMPNE and others). These operations perform element-wise comparison of two vectors and write the results into a vector predicate.

The last class of vector instructions considered is represented by one VBLENDM instruction, which is an implementation of the vector ternary operator $R = \check{P}\ ?\ A : B$.

In fact, looking at Table 1, one can notice that the descriptions of the semantics of all the vector operations given in it are simply flat loops in explicit form. The converse is also true—if a flat loop can be written in the form of the semantics of one or more vector instructions, then it can be implemented using these instructions.

In general the purpose of the article is the methods of flat loops description using semantics of the vector instructions shown in Table 1.

4. FLAT LOOPS VECTORIZATION TECHNIQUES

4.1. Branched Control Vectorization

Branched control in the body of a flat loop takes place in the presence of control transfer operators, for example, $if{-}else$, *case*, *goto* and loop organization operators for, $while$, $do{-}while$, $break$, $continue$. Consider simple example when the body of a flat loop consists of one structure $if{-}else$, as shown in Listing 3.

Listing 3. The body of a flat loop, consisting of $if{-}else$ structure

```
for (int w = 0; w < WIDTH; w++)
{
    if (cond(w))
    {
        block1(w); // t1, p1
    }
    else
    {
        block2(w); // t2, p2
    }
}
```

First of all, we note that in this structure the condition $cond(w)$ depends on w. Otherwise, under the constant condition, it is possible to split the loop according to the condition so that two separate flat loops are obtained whose bodies do not contain control. Further, when considering the conditions in the body of a flat loop, we will assume that they are not constant. Thus, the body of a flat loop consists of two blocks $block1(w)$ and $block2(w)$, one of which is executed depending on $cond(w)$.

Each block can be associated with its length, defined as the number of operations in this block. We will call this characteristic block execution time (blocks from Listing 3 have t_1 and t_2 execution times, respectively). Also, each of these two blocks can be characterized by the probability of its execution. This probability is the ratio of the number of block executions to the total number of iterations of the flat loop (the blocks from Listing 3 have the execution probabilities p_1 and p_2, respectively). Then we can calculate the total execution time of one iteration of a flat loop as $t_1p_1 + t_2p_2$. The total execution time of a flat loop is $T_{orig} = W(t_1p_1 + t_2p_2)$, where W is the total number of iterations of the loop, which is indicated in the code examples by $WIDTH$.

To vectorize this flat loop, it is necessary to vectorize both blocks and execute them under opposite vector predicates $COND$ and $\tilde{}COND$, as it is shown in Listing 4.

Listing 4. Vectorized construction $if{-}else$

```
BLOCK1 ? COND;
BLOCK2 ? ~COND;
```

If we assume that the blocks $BLOCK1$ and $BLOCK2$ are ideally vectorized, that is, their length (the number of vector operations) is equal to the length of the corresponding original blocks $block1(w)$ and $block2(w)$ (the number of scalar operations), then the run time of the vectorized loop is $T_{vec} = t_1 + t_2$. The acceleration from the use of vectorization can be calculated by the formula

$$\frac{T_{orig}}{T_{vec}} = \frac{W(t_1p_1 + t_2p_2)}{t_1 + t_2}.$$

If there is only one block (for example, with $t_2 = 0$, $p_1 = 1$) we get the expected acceleration from vectorization exactly W times. If we consider two identical blocks ($t_1 = t_2$, $p_1 = p_2 = 1/2$), then the expected acceleration from vectorization will be $W/2$. This is due to the fact that in the vectorized code there will appear commands executed under vector predicates, and some of the elements of these predicate are zero. Thus, vector instructions will process fewer scalar elements than their length allows. Note that these calculations were made under the assumption that the blocks $BLOCK1$ and $BLOCK2$

are ideally vectorized. However, they themselves may contain control transfer operators, which will further reduce the effectiveness of vectorization.

Thus, the complete merging of all execution branches is not an effective way of vectorizing flat loops. In the case when one of the executed branches has low probability and also a large length, then in the absence of side effects, we can apply removal of an unlikely branch from the loop.

Listing 5. Vectorization of $if{-}else$ with unlikely branch

```
BLOCK1 ? COND;

if (~COND)
{
    BLOCK2 ? ~COND;
}
```

Listing 5 shows a modification of a vectorized version of a flat loop with the construction $if{-}else$, provided that the second branch has low probability. Now the unlikely branch is executed under the condition $if(\sim COND)$. Note that this is not a vector condition, but a regular check of the fact that the mask $\sim COND$ is non-zero, and it makes sense to execute vector instructions. In addition, with a low probability of execution of the second block, there is not even a need to vectorize it, this will not affect performance. In real applications, such unlikely branches can handle some very rare and sophisticated situations, and the code in these blocks can be much more complicated than the code that processes regular cases [10].

Another important technique for vectorizing branched conditions is the use of vector instructions containing logic and the use of identities based on them. Instructions containing logic include operations such as VBLENDM, VABS (which are implemented through the VAND instruction), VMIN, VMAX. In fact, all these operations contain hidden controls of the type $if{-}else$. In real applications, there are often sections of code whose logic can be rewritten using these operations, thereby reducing the number of different branches of execution. As an example, consider the limitation of the value of a variable using a given lower and upper bounds, as shown in Listing 6.

Listing 6. Limiting the value using the lower and upper bounds

```
for (int w = 0; w < WIDTH; w++)
{
    if (x[w] < a[w])
    {
        x[w] = a[w];
    }
    else if (x[w] > b[w])
    {
        x[w] = b[w];
    }
}
```

Merging such a section of code will not be efficient enough (in combination with other operations of a flat loop). However, using simple identities using the VMIN, VMAX commands, you can get an equivalent and simpler record of the same section in vector form, as shown in Listing 7.

Listing 7. Vectorized form of limiting a value using the lower and upper bounds

```
X = VMAX X, A
X = VMIN X, B
```

It is worth noting that to ensure equivalence of the transformation, the condition $a[w] \leq b[w]$ must be satisfied for all values of w. Therefore, the optimizing compiler will not be able to apply this template, and it must be performed manually by the programmer. This also applies to many other transformations that are not equivalent in the strict sense, but are admissible from the point of view of program logic.

4.2. Vectorization of Nested Loops

The flat loop body itself may contain loop nests. If the conditions for exiting these nested loops are constant, then no problems with vectorization arise (in this case, the same analogy applies as in the case of constant conditions in a branched control). Therefore, we assume that the nested loop has an exit condition depending on w. In other words, the number of iterations of such a loop is not constant. For clarity, consider a simple construction that meets these requirements (Listing 8).

Listing 8. Flat loop containing a loop with a non-constant number of iterations

```
for (int w = 0; w < WIDTH; w++)
{
    while (true)
    {
        if (sqrt(x[w]) < 1.0)
        {
            x[w] -= 1.0;

            break;
        }

        x[w] /= 2.0;
    }
}
```

The main problem in vectorizing this fragment is that for different values of w, a different number of iterations of the inner loop can be performed (for some program contexts, you cannot even guarantee that this number of iterations will be finite). To ensure vectorization, you need to create a special vector predicate that will signal for which w it is necessary to continue the internal loop, and for which it should be interrupted. A complete termination of a loop will occur only when the entire vector predicate of the loop continuation becomes zero. To vectorize the indicated nest, it must first be transformed into a predicate form. It is important to distinguish two predicates: predicate of need for iteration execution and predicate of logic expression $sqrt(x[w]) < 1.0$. The peculiarity of this fragment is that in the case of a false predicate for the next iteration of the loop, we no longer have the right to check the condition $sqrt(x[w]) < 1.0$ (since the argument of the square root extraction function will become negative), that is, checking of this condition should occur under the predicate of the loop iteration (for scalar code this is already fulfilled, and for the vector analogue this requirement must be provided explicitly). Listing 9 shows a predicate representation of this nest.

Listing 9. The predicate form of a flat loop containing a loop with a non-constant number of iterations.

```
for (int w = 0; w < WIDTH; w++)
{
    bool is_iter = true;

    while (is_iter)
    {
        bool is_pos = is_iter ? (sqrt(x[w]) < 1.0) : false;
        bool is_neg = is_iter && !is_pos;

        x[w] = is_pos ? (x[w] - 1.0) : x[w];
        x[w] = is_neg ? (x[w] / 2.0) : s[w];
        is_iter = is_neg;
    }
}
```

After reducing the nest to the predicate form, we note that all control structures have disappeared from the internal loop, and each instruction is executed under the control of predicates. All elements except the $while(is\_iter)$ operator have vector analogues. But for the condition for the continuation of

the loop, the vector analogue is not needed, since the vectorized loop should continue until the mask composed of the individual conditions is_iter is not zero. Thus, we have a fully vectorized flat loop, the record of which is shown in Listing 10.

Listing 10. The vector form of a flat loop containing a loop with a non-constant number of iterations

```
IS_ITER = 0xFFFF;

while (IS_ITER != 0x0)
{
    IS_POS = VCMPLTz(IS_ITER, VSQRTz(IS_ITER, X), V1)
    IS_NEG = IS_ITER & ~IS_POS

    X = VSUB(IS_POS, X, V1)
    X = VDIV(IS_NEG, X, V2)
    IS_ITER = IS_NEG
}
```

This listing shows $V1$—vector consisting of values 1.0, $V2$ —vector consisting of values 2.0.

When vectorizing a flat loop, the body of which is a nest of loops, there is no restriction on the depth of the nest. However, it is necessary for each loop from the nest to maintain its own predicate, which allows iteration of the loop so that data elements that were not processed in the scalar code are not accidentally processed when executing vector instructions (otherwise this can lead to undesirable situations, such as exceptions due to work with garbage data). The need to keep track of these predicates, of course, introduces difficulties in vectorization. In some architectures (for example, in the "Elbrus" architecture), to facilitate the execution of operations whose results may not be needed (and whose arguments may contain incorrect data), a special speculative execution mode is provided that ignores exceptional situations and helps to efficiently merge different code branches.

In [11] the use of this technique for the implementation of construction of the Mandelbrot set is demonstrated.

4.3. Vectorization of Function Calls Under the Condition

In flat loops, function calls may occur. Of course, we assume that all called functions are pure, that is, the result of the function depends only on its arguments. The problem arises when the function should not be called for all iterations of a flat loop, but only for some. Consider a simple example in which such a call is present; it is presented in Listing 11.

Listing 11. Example of a flat loop with a function call under the condition

```
for (int w = 0; w < WIDTH; w++)
{
    if (cond(w))
    {
        x[w] = f(a[w]);
    }
    else
    {
        x[w] = b[w];
    }
}
```

In this example, we cannot directly vectorize the function $f()$ replacing the scalar argument in it with a vector one. The problem is that, as in the case of vectorization of loop nests, it is necessary to keep track of the predicate that signals which data elements the function should be applied to. The only way to do this is to pass the given predicate directly inside the called function. Thus, the scalar function $f(a)$ must be expanded to the vector function $F(P, A)$ into which, in addition to the vector argument, the mask is also passed.

After adjusting the function being called, the proposed loop can be easily vectorized by merging two execution branches using VBLENDM, as shown in the Listing 12.

Listing 12. Vectorization of a flat loop with a function call under the condition

```
1 X = VBLENDM(COND, F(COND, A), B)
```

Note that in this example, there is also some redundancy in the form of repeated application of the vector predicate $COND$ (first in the function call, and then in the merge operation). This redundancy is present in order to avoid exceptions when processing those data elements that should not be processed in scalar code. In this case, the problem may not even be an exception in the arithmetic operation (for example, division by zero), but the potential opportunity to go along a different path of execution, which was absent in the scalar code. And this, in turn, can lead to more serious consequences, such as incorrect writing to memory or the occurrence of an infinite loop in a function.

5. FLAT LOOPS VECTORIZATION PRACTICAL EXAMPLES

This section briefly describes two practical examples of the application of flat loops vectorization that the authors encountered as part of their work. When optimizing these fragments, the techniques described in the previous section were applied. Using flat loops vectorization methods allowed to achieve significant acceleration of the executable code in all cases. When vectorizing program code written in C, a special library of intrinsic functions was used, which allows to directly use the AVX-512 instructions.

5.1. Using Intrinsic Functions To Implement Parallel Code

There is a library of special intrinsic functions that facilitates the use of the AVX-512 set. A complete list and description of these functions is available on the website [12].

To use intrinsic functions in the program, you need to include a header file $immintrin.h$.

Intrinsic functions do not cover all AVX-512 instructions, however, they eliminate the need to manually write assembler code and allow the use of built-in data types for 512-bit vectors (_ _$m512$, _ _$m512i$, _ _$m512d$). Some intrinsics do not correspond to a separate command, but to a whole sequence, such as for the operation of adding all the elements of a vector. Of the many intrinsics, the following groups of functions, similar in structure, can be distinguished. The *swizzle*, *shuffle*, *permute* and *permutevar* functions rearrange the elements of a vector and expand into a sequence of operations in which VSHUF and mask forwarding are present. For a larger number of operations, the AVX-512 implements intrinsics that expand into one specific operation. Among them are arithmetic operations, bitwise operations, operations of reading from memory and writing to memory, conversion operations, merging of two vectors, finding inverse values, obtaining the minimum and maximum of two values, comparison operations, operations with masks, combined operations and others. Some intrinsics, especially those designed to perform packed transcendental operations, are expanded simply into a library function call (for example $\_mm512\_log\_ps$, trigonometrical functions).

When considering vector codes of practical problems, the intrinsics functions of vector commands without using masks will be written in short form. For example, instead of the intrinsic function $\_mm512\_add\_ps$ we will simply write ADD. For operations with masks, the intrinsics functions will be specified in full.

5.2. Riemann Solver Vectorization

The implementation of the Riemann solver considered in this section is publicly available on the Internet as part of the NUMERICA library [13]. In this case, we will be interested in the one-dimensional case for a one-component medium, implemented as a pure function (functions without side effects, the result of the function depends only on the values of the input parameters). This function from the values of density, velocity and pressure of the gas to the left and right of discontinuity, finds the values of these quantities at the discontinuity at zero time after the removal of the septum.

$$U_l = \begin{pmatrix} d_l \\ u_l \\ p_l \end{pmatrix}, \quad U_r = \begin{pmatrix} d_r \\ u_r \\ p_r \end{pmatrix}, \quad U = \begin{pmatrix} d \\ u \\ p \end{pmatrix} = riem(U_l, U_r). \tag{1}$$

In formula (1) gas density, velocity and pressure to the left of the gap (they are combined in the structure U_l —state of the gas to the left of discontinuity) denoted by d_l, u_l, p_l. Similarly, d_r, u_r, p_r denote gas density, velocity, and pressure to the right of discontinuity, combined into the gas state U_r. The variables d, u, p denote the density, velocity and pressure of the gas obtained as a result of solving the Riemann problem.

The NUMERICA library is implemented in the FORTRAN programming language, therefore direct vectorization of this code using intrinsics functions is impossible, so, the version of the code ported to the C programming language was used.

During the calculation, using numerical methods based on the Riemann solver, many calls to the $riemann$ function are made with different sets of input data (at each iteration of the account, one call is made for each face of each cell of the computational grid). Since the $riemann$ function is pure, calls for different sets of input data (dl, ul, pl, dr, ur, pr) are independent and it becomes possible to combine calls in order to effectively use vector (element-wise) instructions. As such a combined call, we will consider a function in which instead of atomic data of type float corresponding vectors containing $WIDTH$ elements will be passed:

$$\overline{U_l} = \begin{pmatrix} \overline{d_l} \\ \overline{u_l} \\ \overline{p_l} \end{pmatrix}, \quad \overline{U_r} = \begin{pmatrix} \overline{d_r} \\ \overline{u_r} \\ \overline{p_r} \end{pmatrix}, \quad \overline{U} = \begin{pmatrix} \overline{d} \\ \overline{u} \\ \overline{p} \end{pmatrix} = riem(\overline{U_l}, \overline{U_r}) \tag{2}$$

In formula (2) all variables $\overline{d_l}$, $\overline{u_l}$, $\overline{p_l}$, $\overline{d_r}$, $\overline{u_r}$, $\overline{p_r}$, $\overline{d}$, $\overline{u}$, $\overline{p}$ are vectors of length 16. For example, the vector $\overline{d}$ contains 16 gas density values obtained by solving 16 Riemann problems combined into one call. Similarly with other variables.

At the same time, with vector data, you can perform the same actions as with the basic types—perform calculations, pass to functions, return as a result.

The Riemann solver has a complex program structure and consists of several functions, each of which can be vectorized using special approaches. However, we will consider only one of these functions $starpu$, which has the most interesting feature—when combining several of its calls into one function, it generates a flat loop whose body is a loop with an unknown number of iterations.

Listing 13. Original version of *starpu*

```
void starpu(float dl, float ul, float pl, float cl,
            float dr, float ur, float pr, float cr,
            float &p, float &u)
{
    const int nriter = 20;
    const float tolpre = 1.0e-6;
    float change, fl, fld, fr, frd, pold, pstart, udiff;

    guessp(dl, ul, pl, cl, dr, ur, pr, cr, pstart);
    pold = pstart;
    udiff = ur - ul;

    int i = 1;

    for ( ; i <= nriter; i++)
    {
        prefun(fl, fld, pold, dl, pl, cl);
        prefun(fr, frd, pold, dr, pr, cr);
        p = pold - (fl + fr + udiff) / (fld + frd);
        change = 2.0 * abs((p - pold) / (p + pold));

        if (change <= tolpre)
        {
            break;
        }

        if (p < 0.0)
        {
            p = tolpre;
        }

        pold = p;
    }

    if (i > nriter)
    {
        cout << "divergence in Newton-Raphson iteration" << endl;
        exit(1);
    }

    u = 0.5 * (ul + ur + fr - fl);
}
```

The loop located in this function, in addition to an unknown number of iterations, also contains conditional transitions (if, $break$) and calls of $prefun$ functions, which also complicates its vectorization. Before vectorizing, this loop must be converted to a predicate form in which the body should not contain transition operations. All loop instructions are executed under their own predicates, and loop execution is interrupted if all predicates are zeroed. It should be noted that calls to $prefun$ functions must also have corresponding predicates. After converting the body of the loop to the predicate form, it can be vectorized. After that, the predicates of the instructions are replaced by vector register-masks (in this place an additional parameter of the vectorized function $prefun$ appears in the form of a mask). The result of vectorizing the *starpu* function is shown in the Listing 14.

Listing 14. Vectorized version of *starpu*

```
void starpu_16(__m512 dl, __m512 ul, __m512 pl, __m512 cl,
               __m512 dr, __m512 ur, __m512 pr, __m512 cr,
               __m512 *p, __m512 *u)
{
    __m512 two, tolpre, tolpre2, udiff, pold, fl, fld, fr, frd, change;
    __mmask16 cond_break, cond_neg, m;
    const int nriter = 20;
    int iter = 1;

    two = SET1(2.0);
    tolpre = SET1(1.0e-6);
    tolpre2 = SET1(5.0e-7);
    udiff = SUB(ur, ul);

    guessp_16(dl, ul, pl, cl, dr, ur, pr, cr, &pold);

    // Start with full mask.
    m = 0xFFFF;

    for (; (iter <= nriter) && (m != 0x0); iter++)
    {
        prefun_16(&fl, &fld, pold, dl, pl, cl, m);
        prefun_16(&fr, &frd, pold, dr, pr, cr, m);
        *p = _mm512_mask_sub_ps(*p, m, pold,
                                _mm512_mask_div_ps(z, m,
                                                   ADD(ADD(fl, fr),
                                                       udiff),
                                                   ADD(fld, frd)));
        change = ABS(_mm512_mask_div_ps(z, m, SUB(*p, pold),
                                        ADD(*p, pold)));
        cond_break = _mm512_mask_cmp_ps_mask(m, change,
                                             tolpre2, _MM_CMPINT_LE);
        m &= ~cond_break;
        cond_neg = _mm512_mask_cmp_ps_mask(m, *p, z, _MM_CMPINT_LT);
        *p = _mm512_mask_mov_ps(*p, cond_neg, tolpre);
        pold = _mm512_mask_mov_ps(pold, m, *p);
    }

    // Check for divergence.
    if (iter > nriter)
    {
        cout << "divergence in Newton-Raphson iteration" << endl;
        exit(1);
    }

    *u = MUL(SET1(0.5), ADD(ADD(ul, ur), SUB(fr, fl)));
}
```

The initialization of the full mask for performing vectorized iterations of the loop is visible in the Listing 14. As the loop runs, the mask depletes, and when it is completely zeroed, the loop ends.

It is worth noting that vectorizing a loop with an unknown number of iterations can be quite dangerous, since the number of iterations of a vectorized loop is equal to the maximum of the number of loop iterations from 16 combined calls of the original non-vectorized function. With a large difference in the number of iterations of the original code, a drop in efficiency occurs.

As a result of applying vectorization for all the functions of the Riemann solver, total 7 times acceleration was achieved. You can also note the works [14, 15] in this direction.

5.3. Vectorization of Cells Classification in the Implementation of the Immersed Boundary Method

When solving gas dynamics problems numerically, bodies with complex geometry are a frequent case. For such bodies, building a consistent computational grid can be an extremely complex task. An alternative in this case is the use of the immersed boundary method [16, 17]. This method allows you to use an inconsistent grid for calculations and even a simple cartesian grid, which greatly simplifies the calculation. As part of the implementation of the immersed boundary method, it is required to solve a particular problem of determining the intersection of a triangle and a rectangular parallelepiped in space for a large number of these geometric primitives.

Let a triangle be given by three points: $A(x_A, y_A, z_A)$, $B(x_B, y_B, z_B)$, $C(x_C, y_C, z_C)$. Then the coordinates of any point $P(x, y, z)$, located inside a triangular can be represented as follows:

$$\begin{cases} x = x_A + (x_B - x_A)\alpha + (x_C - x_A)\beta, \\ y = y_A + (y_B - y_A)\alpha + (y_C - y_A)\beta, \\ z = z_A + (z_B - z_A)\alpha + (z_C - z_A)\beta, \end{cases}$$

where $\alpha \geq 0$, $\beta \geq 0$, $\alpha + \beta \geq 1$.

The geometric locus of the points of a rectangular parallelepiped is the set of points $P(x, y, z)$, the coordinates of which satisfy the following system of inequalities:

$$\begin{cases} x_l \leq x \leq x_h, \\ y_l \leq y \leq y_h, \\ z_l \leq z \leq z_h. \end{cases}$$

To establish the fact of intersection between triangle and rectangular parallelepiped, it is necessary to determine whether there is a solution of the system of inequalities below with respect to α and β:

$$\begin{cases} x_l \leq x_A + (x_B - x_A)\alpha + (x_C - x_A)\beta \leq x_h, \\ y_l \leq y_A + (y_B - y_A)\alpha + (y_C - y_A)\beta \leq y_h, \\ z_l \leq z_A + (z_B - z_A)\alpha + (z_C - z_A)\beta \leq z_h, \\ \alpha \geq 0, \\ \beta \geq 0, \\ \alpha + \beta \leq 1. \end{cases}$$

For such systems of inequalities, there are many different solutions. In our case, the system is quite simple, contains two variables, and the method of convolution of finite systems of linear inequalities can be applied to it.

To solve by the convolution method, we transform the system of inequalities so that it contains inequalities of only the form $k_\alpha\alpha + k_\beta\beta + k \leq 0$. After this, one step of folding (or deformation of the system) is performed and it turns into a system of inequalities with respect to one variable. This system is easy to verify for the existence of a solution. We will perform the deformation of the system in order to exclude the variable α from it. To do this, we compose a new system that includes all the inequalities of the original system of the form $k_\beta\beta + k \leq 0$, and each pair of inequalities

$$\begin{cases} k^1_\alpha\alpha + k^1_\beta + k^1 \leq 0, \\ k^2_\alpha\alpha + k^2_\beta + k^2 \leq 0, \end{cases}$$

where $k^1_\alpha < 0$, and $k^2_\alpha > 0$ will enter the deformed system in the form

$$(k^1_\beta k^2_\alpha - k^2_\beta k^1_\alpha)\beta + (k^1 k^2_\alpha - k^2 k^1_\alpha) \leq 0.$$

Since the original system contains 9 equations, at least one of which has a zero coefficient for the variable α. Of the remaining eight, half of the coefficients for the variable α are non-negative, and half are non-positive, then the deformed system will contain no more than 17 equations.

Consider the implementation of the function

$$tri\_box\_intersect(xa, ya, za, xb, yb, zb, xc, yc, zc, xl, xh, yl, yh, zl, zh) \to int,$$

analyzing the presence of intersection between triangle and a rectangular parallelepiped. The function returns 1, if there is an intersection, and 0, if not. The logic of the function is as follows. First, the coefficients of the system of inequalities are put in a two-dimensional array of coefficients $b[bec][3]$, where bec (basic equations count)—the number of initial inequalities of the system (in our case, 9). Then, one step of convolution of the system is performed. At the same time, a set of solutions is searched for β. Before convolution, the set of valid values for the variable β is taken as a range $[0, 1]$ ($lo = 0$, $hi = 1$). During the convolution of the system of inequalities, this set of solutions decreases. If at some stage of convolution the set of solutions turns to empty ($lo > hi$), then the function finishes and returns 0. If, after performing all the folding operations, the set of solutions remains nonzero, this means that there is an intersection, and the function returns 1. The convolution implementation of the system of equations is presented in the listing below:

Listing 15. The initial implementation of convolution of system of linear inequalities aimed to determine the intersection of a triangle and a rectangular parallelepiped

```
for (i = 0; i < bec; i++)
{
    bi0 = b[i][0];

    if (bi0 == 0.0)
    {
        if (!upgrade(b[i][1], b[i][2], &lo, &hi))
        {
            return 0;
        }
    }
    else
    {
        for (j = i + 1; j < bec; j++)
        {
            if (bi0 * b[j][0] < 0.0)
            {
                f0 = bi0 * b[j][1] - b[j][0] * b[i][1];
                f1 = bi0 * b[j][2] - b[j][0] * b[i][2];

                if (bi0 < 0.0)
                {
                    f0 = -f0;
                    f1 = -f1;
                }

                if (!upgrade(f0, f1, &lo, &hi))
                {
                    return 0;
                }
            }
        }
    }
}

return 1;
```

The *upgrade* function, which is called from the code, is designed to update the current set of valid values for β taking into account the newly obtained constraint of the form $k_\beta\beta + k \le 0$, the coefficients of which are passed in the first and second parameters. The current set of solutions is a segment with boundaries stored in variables lo and hi, and depending on the sign of the coefficient k_β one of these boundaries inside a call of *upgrade* can change (either lo increases, or hi decreases). If after updating the set of solutions it turns out to be empty (lower boundary exceeds upper), then *upgrade* returns 0, otherwise it returns 1.

After considering the implementation of *tri_box_intersect* we can pass to vectorization of computations. To analyze the intersection of two grids, *tri_box_intersect* should be called multiple times with different sets of input parameters. To optimize the process let's implement *tri_box_intersect_w* function, uniting processing of $WIDTH$ calls of *tri_box_intersect*.

The following is an implementation of the *tri_box_intersect_w* as a first approximation.

Listing 16. Original implementation of a function combining $WIDTH$ function calls *tri_box_intersect*

```
void tri_box_intersect_w(float *xa, float *ya, float *za,
                         float *xb, float *yb, float *zb,
                         float *xc, float *yc, float *zc,
                         float *xl, float *xh,
                         float *yl, float *yh,
                         float *zl, float *zh,
                         int *r)
{
    for (int w = 0; w < WIDTH; w++)
    {
        r[w] = tri_box_intersect(xa[w], ya[w], za[w],
                                 xb[w], yb[w], zb[w],
                                 xc[w], yc[w], zc[w],
                                 xl[w], xh[w],
                                 yl[w], yh[w],
                                 zl[w], zh[w]);
    }
}
```

Of course, the real task requires processing millions of calls, but they can be divided into groups of $WIDTH$ and processed using the function *tri_box_intersect_w*, so we will consider vectorization of this function.

To vectorize the function *tri_box_intersect_w*, you should substitute the body of the function *tri_box_intersect* at the place of its call. After performing this transformation, we obtain a flat loop whose body is a nest of loops with complex control. The presence of a large number of conditional statements in the source code generates many instructions under zero predicates in the resulting code in the case of code merging, which negatively affects performance. To reduce the number of conditional operators, we can use mathematical identities that replace conditional constructions with instructions that have vector analogues in the AVX-512 instruction set. For example, in the code under consideration, the calculation of the values of $f0$ and $f1$ taking into account the condition $b[i][0] * b[j][0] < 0$ can be replaced by the following:

Listing 17. Using identities to implement vector instructions

```
f0 = fabs(bi0) * b[j][1]
     + fabs(b[j][0]) * b[i][1]
f1 = fabs(bi0) * b[j][2]
     + fabs(b[j][0]) * b[i][2]
```

Similar difficulties are caused by an external condition if on the Listing 15. This condition has an alternative branch containing a loop. Merging the data of the two branches reduces the performance of the resulting code, therefore, in this case, it is advantageous to split the outer loop according to the $if{-}else$ construct. In this case, two nests of loops are formed, each of which can be vectorized independently. Note that the described transformation in general sense is not equivalent, since both

branches of the condition $if-else$, and therefore the bodies of the resulting loops, contain outputs from the function. Performing a splitting of loop may change the condition that provokes the exit from the function. For this reason, the compiler is not able to perform this conversion automatically. However, from the point of view of the result of the function, this transformation is correct, therefore we use it.

There is one more extremely positive moment in the considered program context. The condition for exiting nested loops is to achieve the inductive variable value bec. This value is a constant, which means that it does not depend on the iteration number, therefore this condition can be transferred to the vector code without changes. If the condition for exiting the loop depends on the iteration number, the condition itself must also be vectorized, which can lead to performance losses. However, in this code there are no problems with loop vectorization with an irregular number of iterations. On the listings below, we present the resulting code for the function $tri\_box\_intersect\_w$, as well as its schematic transformation into a vector analogue.

Listing 18. Scheme of program code before conversion to vector form

```
float b[bec][3][WIDTH];

<init b>;

for (int w = 0; w < WIDTH; w++) r[w] = 1;

for (int w = 0; w < WIDTH; w++)
{
    lo = 0.0;
    hi = 1.0;

    for (i = 0; i < bec; i++)
    {
        upgrade(b[i][0][w] == 0.0,
                b[i][1][w], b[i][2][w], &lo, &hi);
        if (lo > hi) break;
    }

    for (i = 0; i < bec; i++)
    {
        bi0 = b[i][0][w];
        abi0 = fabs(bi0);

        for (j = i + 1; j < bec; j++)
        {
            bj0 = b[j][0][w];
            abj0 = fabs(bj0);

            upgrade(bi0 * bj0 < 0.0,
                    abi0 * b[j][1][w] + abj0 * b[i][1][w],
                    abi0 * b[j][2][w] + abj0 * b[i][2][w],
                    &lo, &hi);
            if (lo > hi) break;
        }

        if (lo > hi) break;
    }

    if (lo > hi) r[w] = 0;
}
```

Listing 19. Vector form of the resulting code from Listing 18

```
__m512 b[bec][3];

<init b>;

_mm512_store_epi32(r, _mm512_set1_epi32(1));



__m512 lo = z0;
__m512 hi = z1;

for (i = 0; i < bec; i++)
{
    upgrade(_mm512_cmpeq_ps_mask(b[i][0], z0),
            b[i][1], b[i][2], &lo, &hi);
    if (!_mm512_cmplt_ps_mask(lo, hi)) break;
}

for (i = 0; i < bec; i++)
{
    bi0 = b[i][0];
    abi0 = ABS(bi0);

    for (j = i + 1; j < bec; j++)
    {
        bj0 = b[j][0];
        abj0 = ABS(bj0);

        upgrade(_mm512_cmplt_ps_mask(MUL(bi0, bj0), z0),
                FMADD(abi0, b[j][1], MUL(abj0, b[i][1])),
                FMADD(abi0, b[j][2], MUL(abj0, b[i][2])),
                &lo, &hi);
        if (!_mm512_cmplt_ps_mask(lo, hi)) break;
    }

    if (!_mm512_cmplt_ps_mask(lo, hi)) break;
}

_mm512_mask_store_epi32(r,
                        _mm512_cmplt_ps_mask(hi, lo),
                        _mm512_set1_epi32(0));
```

From the Listings 18 and 19 it is obvious, that correctly composed predicate code can be quite simply translated into a vector analogue. To achieve this, the following conditions were satisfied. First, all *else* branches in conditional statements have been removed. Second, the call to the *upgrade* function has been removed from the predicate. Instead, in our case, the scalar condition for calling the *upgrade* function was transformed into the argument of this function, and after that the code can be easily vectorized. In the rest, all real scalar instructions were simply replaced with vector analogues.

The transformations made it possible to accelerate the function *tri_box_intersect_w* by 6.7 times.

6. CONCLUSION

Programming constructs called flat loops were considered and their importance for improving the performance of supercomputer computations was explained. Methods for vectorizing flat loops using the AVX-512 instruction set were demonstrated. Namely, using the features of these instructions (mainly the presence of vector instructions that use masks), methods for vectorizing bodies of flat loops containing complex branched control, loop nests, and function calls under the condition were discussed. The possibility of vectorizing such a non-trivial software context opens up great opportunities for accelerating supercomputer applications.

In addition to the theoretical part, two practical examples of the application of flat loops vectorization methods were considered. In both cases, a complicated program context was vectorized, and the code acceleration as a result of the vectorization was approximately 7 times.

The obtained practical results indicate the good potential of the proposed methods, and in the future, the authors intend to support this area of research, paying particular attention to increasing the automation of program code vectorization (for the considered practical examples, manual rewriting of the code using intrinsics functions was used).

ACKNOWLEDGMENTS

The supercomputer MVS-10P, located at the JSCC RAS, was used for calculations during the research.

FUNDING

The work has been done at the JSCC RAS as part of the state assignment for the topic 0065-2019-0016 (reg. no. AAAA-A19-119011590098-8).

REFERENCES

1. R. Fadeev, K. Ushakov, M. Tolstykh, R. Ibrayev, V. Shashkin, and G. Goyman, "Supercomputing the seasonal weather prediction," in *Supercomputing. RuSCDays 2019,* Ed. by V. Voevodin and S. Sobolev, Commun. Comput. Inform. Sci. **1129** (2019).
2. Y. Hu, H. Yang, Z. Luan, L. Gan, G. Yang, and D. Qian, "Massively scaling seismic processing on sunway TaihuLight supercomputer," IEEE Trans. Parallel Distrib. Syst. **31**, 1194–1208 (2020).
3. K. E. Jones, "Supercomputing improves predictions of fluid flow in rock," Comput. Sci. Eng. **21** (6), 74–76 (2019).
4. V. Kalantzis, "Data analytics, accelerators, and supercomputing: The challenges and future of MPI," XRDS **23**, 50–52 (2017).
5. A. A. Rybakov, "Inner respresentation and crossprocess exchange mechanism for block-structured grid for supercomputer calculations," Program Syst.: Theory Appl. **32** (8:1), 121–134 (2017).
6. A. V. Baranov, G. I. Savin, B. M. Shabanov, et al., "Methods of jobs containerization for supercomputer workload managers," Lobachevskii J. Math. **40** (5), 525–534 (2019).
7. J. Doerfert and H. Finkel, "Compiler optimizations for parallel programs," in *Languages and Compilers for Parallel Computing LCPC 2018,* Ed. by M. Hall and H. Sundar, Lect. Notes Comput. Sci. **11882** (2019).
8. B. M. Shabanov, A. A. Rybakov, and S. S. Shumilin, "Vectorization of high-performance scientific calculations using AVX-512 intruction set," Lobachevskii J. Math. **40** (5), 580–598 (2019).
9. *Intel 64 and IA-32 Architectures Software Developer's Manual* (Intel Corp., 2019), Combined Vols.: 1, 2A, 2B, 2C, 2D, 3A, 3B, 3C, 3D and 4.
10. 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).
11. O. Krzikalla, F. Wende, and M. Höhnerbach, "Dynamic SIMD vector lane scheduling," Lect. Notes Comput. Sci. **9945**, 354–365 (2016).
12. Intel Intrinsics Guide. https://software.intel.com/sites/landingpage/IntrinsicsGuide/. Accessed 2020.
13. E. F. Toro, NUMERICA, A Library of Sources for Teaching, Research and Applications. https://github.com/dasikasunder/NUMERICA. Accessed 2018.
14. M. Bader, A. Breuer, W. Höltz, S. Rettenberger, "Vectorization of an augmented Riemann solver for the shallow water equations," in *Proceedings of the 2014 International Conference on High Performance Computing and Simulation HPCS 2014* (2014), pp. 193–201.
15. C. R. Ferreira, K. T. Mandli, and M. Bader, "Vectorization of Riemann solvers for the single- ans multi-layer shallow water equations," in *Proceedings of the 2018 International Conference on High Performance Computing and Simulation, HPCS 2018* (2018), pp. 415–422.
16. R. Mittal and G. Iaccarino, "Immersed boundary methods," Ann. Rev. Fluid Mech. **37**, 239–261 (2005).
17. Y.-H. Tseng and J. H. Ferziger, "A ghost-cell immersed boundary method for flow in complex geometry," J. Comput. Phys. **192**, 593–623 (2003).