ISSN 2079-3316 **PROGRAM SYSTEMS: THEORY AND APPLICATIONS** **vol. 10**, No 3(42), pp. 41–58

CSCSTI 50.07.05,50.33.04
UDC 519.681.5:004.272.25

Alexey A. Rybakov, Sergey S. Shumilin

Vectorization of the Riemann solver using the AVX-512 instruction set

Abstract. Numerical methods based on solving the Riemann problem of the decay of an arbitrary discontinuity are extremely demanding of computational resources. Applying the data of numerical methods to modern computational grids requires the use of a supercomputer. Among the various tools for improving the performance of supercomputer applications, we can emphasize the vectorization of program code. The AVX-512 instruction set has a number of unique features allowing to apply vectorization to the Riemann solver software context, which results in a significant acceleration of the solver. Using the exact Riemann solver as an example, the article discusses practical approach to vectorizing a various program contexts, including simple linear blocks, regions with complex control, and nested loops. The basis of the approach under consideration is the possibility of simultaneously executing several instances of some pure function on the same processor core. This feature is achieved by translating the program code into the predicate form and using AVX-512 vector instructions. In this case the number of simultaneously running instances is equal to the width of the vector. It is shown that using the features of the AVX-512 instruction set allows to successfully vectorize the considered program context. The proposed approach can be applied to vectorize a wide range of applications.

Key words and phrases: Riemann problem of the decay of an arbitrary discontinuity, Riemann solver, AVX-512, KNL, vectorization, intrinsic functions.

2010 *Mathematics Subject Classification*: 68W10; 65P99,68M07,

Introduction

The solution of the Riemann problem is used in numerical methods for non-stationary problems with large discontinuities [1]. At the same time, this approach can be applied both to ordinary single-component gas-dynamic equations and to the equations of multicomponent gas dynamics [2]. The Godunov method for solving systems of unsteady equations of gas dynamics [3] is based on an exact or approximate solution of the Riemann problem. In this method, at each iteration of the calculations, the Riemann problem is

The work was done at the JSCC RAS as part of the state assignment on the topic 0065-2019-0016 with the support of RFBR grant 18-07-00638_a.

10.25209/2079-3316-2019-10-3-41-58

solved on each face of each cell of the computational grid to determine the flows through these faces. Due to the fact that the sizes of computational grids, which are used today for carrying out calculations, amount to tens, hundreds of millions of cells and more, it is necessary to use supercomputers and the entire spectrum of approaches to parallel computing. To increase the efficiency of supercomputer computing, tools such as MPI, OpenMP, OpenACC and others are used.

The most low-level approach used to improve the performance of supercomputer applications is code vectorization. The use of a special instruction set AVX-512 has unique features that make it possible to create an effective parallel code, which leads to a multiple acceleration of supercomputer applications.

Improving the performance of Riemann solvers on parallel architectures has been the subject of much research, which continues to this day, as new architectures constantly emerge with new features. The [4] demonstrated an algorithm for parallelizing numerical methods using an approximate Riemann solver for problems of magnetohydrodynamics, leading to parallel operation on a multiprocessor machine with parallelization efficiency close to unity. The work [5] provides a comprehensive study on the parallelization of computational methods based on the Riemann solver using MPI, OpenMP and graphics accelerators. As a result, good performance in terms of scalability was demonstrated, in particular, a strong scalability indicator at around 85%. The [6] work highlights the use of the PetClaw tool, which allows to apply numerical methods based on the Riemann solver on several thousand computational processes with a parallelization efficiency close to unity. The works [7], [8] highlighted the issues of parallel implementation of the AstroPhi program for modeling astrophysical flows. It is shown that the current implementation allows achieving almost 50% of the scalar peak performance of the Intel Xeon Phi KNC coprocessors and provides the estimate of the expected effect of future vectorization (more than 80% of the total peak performance). As one of the most detailed studies on the vectorization of Riemann solvers, we can mention the work [9], which deals with the optimization of an approximate Riemann solver for solving shallow water equations. The approaches used made it possible to achieve acceleration of 6-7 times on the Intel Xeon Phi KNC coprocessors on real operations with single precision. Later this study was continued and the work [10] describes methods by which acceleration was obtained in the 2.4-6.5 times range for double precision operations (calculations were performed on Intel Xeon Phi KNL microprocessors).

This article discusses the vectorization of the exact Riemann solver containing complex structures, including solving a nonlinear equation and complex control. The article shows that using the features of the AVX-512 instruction set allows you to successfully vectorize this software context.

The main contribution of this article is the development of approaches to vectorization of complex program context using a unique set of instructions AVX-512. The exact Riemann solver is very suitable for this purpose, as it has a compact implementation and at the same time contains a number of features of the program context that require their own techniques during vectorization.

The first section of the article provides a brief description of the features of the AVX-512 instruction set, which allow vectorization of a complex program context using a predicate code representation. The second section presents the general scheme of the exact Riemann solver, and sections 3-5 reveal the features of the vectorization of its individual parts (simple linear context, sections with complex controls and a nested loop). Section 6 presents the results of numerical experiments on the Intel Xeon Phi KNL microprocessor, and section 7 presents a comparison with related works.

1. Features of the AVX-512 instruction set

The AVX-512 instruction set is an extension of the 256-bit AVX instruction of the Intel x86 architecture. This instruction set is supported in the second-generation Intel Xeon Phi microprocessor (Knight Landing, KNL) and Intel Xeon Skylake families.

AVX-512 instructions work with 512-bit vector registers (`zmm`), which may contain integer or real data. Each `zmm` register is capable of holding, for example, 8 real double precision values (`double`) or 16 real single precision values (`float`). The AVX-512 instruction set implements a variety of operations with vector arguments, including arithmetic operations, comparison operations, memory read and write operations, transcendental operations, combined operations like $\pm a \cdot b \pm c$, elements permutation operations vectors and others.

To support selectively applying packed data operations to specific vector elements, most of the AVX-512 instructions use special mask registers as arguments. There are 8 such registers (`k0-k7`). When performing a vector operation, the element of the result vector will be calculated only if the bit of the mask with the corresponding number is set to one, otherwise the operation for these elements will be ignored. This unique feature of the AVX-512 instruction set provides the implementation of the predicate execution mode [11], which is supported in such architectures as ARM or «Elbrus» [12]. The presence of the predicate execution mode allows you to apply merger optimization of execution branches and, thus, get rid of unnecessary control transfer operations, which helps to create highly efficient parallel code.

Other important features of the AVX-512 instruction set include multiple reads of vector elements located in memory with arbitrary offsets from the

base address, as well as similar operations of writing vector elements into memory with arbitrary offsets (`gather/scatter` operations). Although these operations are extremely slow, they in some cases help to significantly simplify the logic of vectorized code. It should also be noted a large variety of different operations of permutation, mixing, duplication, transfer of vector elements, which allows arbitrary change the order of data processing. Also, combined operations that combine the multiplication and addition operations into one operation can bring significant acceleration.

To simplify the use of vector instructions for optimizing the program code for the icc compiler, special intrinsic functions have been developed (they are defined in the header file `immintrin.h`) [13]. These functions do not cover the entire set of AVX-512 instructions, however they eliminate the need to manually write assembly code. Instead, it is possible to operate the built-in data types for 512-bit vectors and use them when working with the intrinsics functions as normal base types (the `zmm` registers will be used when building the executable code by the compiler). Some intrinsic functions correspond not to one separate command, but to an entire sequence, such as the group of `reduce` functions, while others simply expand into a call of library function (for example, trigonometric functions or the `hypot` function).

In this article, we consider the features of the Riemann solver in the classical implementation of E. F. Toro [14] and describe approaches that allow to vectorize this solver to simultaneously solve 16 copies of the Riemann problem of decay of an arbitrary discontinuity at once.

2. Description of the Riemann Solver

The implementation of the Riemann solver considered in this article is in the public domain on the Internet as part of the NUMERICA. 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, the result of the function depends only on the values of the input parameters), which, by the density, velocity and gas pressure values to the left and right of the discontinuity, finds the values of the same quantities at the discontinuity at zero time after removal of the septum.

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

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

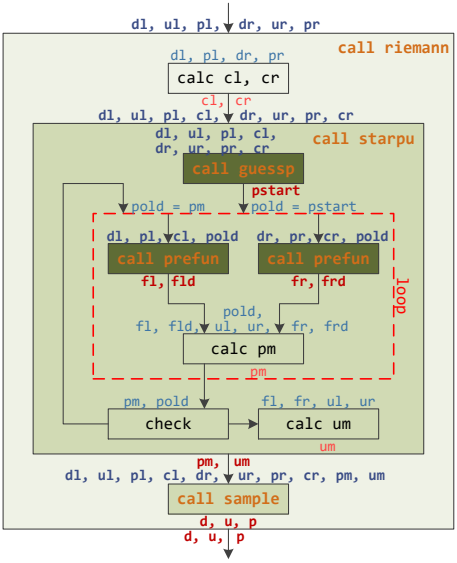


FIGURE 1. Data flow diagram in Riemann solver

denote the density, velocity and pressure of the gas obtained by solving the Riemann problem.

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.

Figure 1 shows a diagram of the work of the Riemann solver with the indicated data streams and calls of all the functions involved in the implementation. The `riemann` function calculates the speed of sound on the right and left, performs a vacuum test, and subsequently 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 non-linear equation by the iterative Newton 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 gap configuration by calculating a set of conditions. This function contains a very branched control, the nesting of conditions in it reaches four, which makes it difficult to use vectorization.

In the counting process, a number of calls to the `riemann` function with different sets of input data are made using numerical methods based

```
01 void prefun(float &f, float &fd, float &p,
02              float &dk, float &pk, float &ck)
03 {
04     float ak, bk, pratio, qrt;
05
06     if (p <= pk)
07     {
08         pratio = p / pk;
09         f = G4 * ck * (pow(pratio, G1) - 1.0);
10         fd = (1.0 / (dk * ck)) * pow(pratio, -G2);
11     }
12     else
13     {
14         ak = G5 / dk;
15         bk = G6 * pk;
16         qrt = sqrt(ak / (bk + p));
17         f = (p - pk) * qrt;
18         fd = (1.0 - 0.5 * (p - pk) / (bk + p)) * qrt;
19     }
20 }
```

Figure 2. Original version of the function `prefun`

on the Riemann solver (during each counting iteration, one call is made for each face of each cell of the computational grid). Since the riemann function is pure, calls for different input data sets (`dl`, `ul`, `pl`, `dr`, `ur`, `pr`) are independent and there is a desire to merge calls for the purpose of effective use of vector (element-wise) instructions. As such a combined call, we will consider a function in which, instead of atomic data of the type `float`, the corresponding vectors containing 16 elements each are passed:

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

In the 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 values of gas density, obtained by solving 16 Riemann problems combined into one challenge. This also applies to other variables.

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

3. Simple context vectorization

The simplest context for computing vectorization when combining calls is the `prefun` function (the `guessp` function has similar properties and is not considered). A non-vectorized version of the function is presented in Figure 2.

This code contains only simple arithmetic operations, square root

```
01  void prefun_16(__m512 *f, __m512 *fd, __m512 p,
02                 __m512 dk, __m512 pk, __m512 ck,
03                 __mmask16 m)
04  {
05      __m512 pratio, ak, bkp, ppk, qrt;
06      __mmask16 cond, ncond;
07
08      // Conditions.
09      cond = _mm512_mask_cmp_ps_mask(m, p, pk, _MM_CMPINT_LE);
10      ncond = m & ~cond;
11
12      // The first branch.
13      if (cond != 0x0)
14      {
15          pratio = _mm512_mask_div_ps(z, cond, p, pk);
16          *f = _mm512_mask_mul_ps(*f, cond, MUL(g4, ck),
17                  SUB(_mm512_mask_pow_ps(z, cond, pratio, g1), one));
18          *fd = _mm512_mask_div_ps(*fd, cond,
19                   _mm512_mask_pow_ps(z, cond, pratio, SUB(z, g2)),
20                  MUL(dk, ck));
21      }
22
23      // The second branch.
24      if (ncond != 0x0)
25      {
26          ak = _mm512_mask_div_ps(z, ncond, g5, dk);
27          bkp = FMADD(g6, pk, p);
28          ppk = SUB(p, pk);
29          qrt = _mm512_mask_sqrt_ps(z, ncond,
30                                    _mm512_mask_div_ps(z, ncond, ak, bkp));
31          *f = _mm512_mask_mul_ps(*f, ncond, ppk, qrt);
32          *fd = _mm512_mask_mul_ps(*fd, ncond, qrt,
33                  FNMADD(_mm512_mask_div_ps(z, ncond, ppk, bkp),
34                         SET1(0.5), one));
35      }
36  }
```

FIGURE 3. Vectorized version of the function `prefun`

extraction, exponentiation and comparison. For all these actions, vector equivalents are provided in the AVX-512 command set. The function code is vectorized by replacing arithmetic operations with vector analogs. The calculation of values under the condition (`if-else`) is vectorized by using vector predicates obtained using vector comparison (see Figure 3). Three points are worth noting in the vectorization of the `prefun` function.

Since the vector division operation is slow, the following identities were used to reduce the number of divisions:

$$\frac{a}{\frac{b}{c}} = \frac{ac}{b}, \quad \frac{\frac{a}{b}}{c} = \frac{a}{bc}. \tag{3}$$

Since the `prefun` function is called inside the loop in the `starpu` function, an additional mask argument must be added to its implementation, by which the elements processed in this call are selected (Figure 3 line 03). This will be described in more detail below, when the vectorization of the `starpu` function will be discussed.

One more important point of vectorization is optimization of work with conditions. In the process of performing vectorization, we have formed two groups of vector instructions executed under the predicates `cond` (Figure 3, lines 15-20) and `ncond` (Figure 3, lines 26-34). Since physical tasks are characterized by continuous changes in physical quantities, it can be argued that the data sets processed in a vectorized call are similar. In other words,

```
001 void sample(float dl, float ul, float pl, float cl,
002              float dr, float ur, float pr, float cr,
003              const float pm, const float um,
004              float &d, float &u, float &p)
005 {
006     float c, cml, cmr, pml, pmr, shl, shr, sl, sr, stl, str;
007
008     if (0.0 <= um)
009     {
010         if (pm <= pl)
011         {
012             shl = ul - cl;
013
014             if (0.0 <= shl)
015             {
                    < d, u, p = dl, ul, pl >
019             }
020             else
021             {
022                 cml = cl * pow(pm / pl, G1);
023                 stl = um - cml;
024
025                 if (0.0 > stl)
026                 {
027                     d = dl * pow(pm / pl, 1.0 / GAMA);
028                     u = um;
029                     p = pm;
030                 }
031                 else
032                 {
                        < high-density code, low prob >
037                 }
038             }
039         }
040         else
041         {
042             pml = pm / pl;
043             sl = ul - cl * sqrt(G2 * pml + G1);
044
045             if (0.0 <= sl)
046             {
                    < d, u, p = dl, ul, pl >
050             }
051             else
052             {
053                 d = dl * (pml + G6) / (pml * G6 + 1.0);
054                 u = um;
055                 p = pm;
056             }
057         }
058     }
059     else
060     {
            < symmetrical branch >
109     }
110 }
```

Figure 4. Original version of the function `sample`

all the vectors involved in the calculations contain similar values. The same statement applies to predicate vectors. Indeed, the execution statistics shows that for physical tasks the most frequent values of the predicate vectors (in this case, `cond` an `ncond`) are the values `0x0` and `0xFFFF`. Thus, checks of predicate vector registers for emptiness (Figure 3, lines 13 and 24) immediately cut off entire blocks of unnecessary operations performed with a zero mask.

4. Vectorization of highly branched conditions

The `sample` function contains a highly branched control with a nesting level of 4 (see Figure 4). The condition tree built for this function contains 10 leaf nodes, in each the value of the gas-dynamic values `d`, `u`, `p` is determined. Direct computation of the vector predicates of all leaf nodes and the

execution of their code under these predicates slows down the resulting code, so 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 removing the assignment operations upward through the function code. Thus, 4 linear sections were deleted. This initialization of the parameters `d`, `u`, `p` does not contain arithmetic operations (the parameters are initialized by the arguments of the function `dl`, `ul`, `pl` or `dr`, `ur`, `pr`), which means it can be performed using vector merge operations `blend`.

It was further noted that the `sample` function handles the right and left profiles of the breakup decay in the same way with minor changes. Using a simple replacement of variables, which consists in changing the sign of the velocity value, we managed to merge the code for two subtrees, based on the condition `pm` $\leq$ `pl`, which reduced the amount of the calculation code by half and expectedly reduced the time execution 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 caused by the unlikely part of the code, heavy operations, among which there are calls to the function of raising to a power (Figure 4, lines 033-036). The vector predicate of this code segment in more than 95 percent of cases has the value `0x0`, therefore, before executing this code segment, it is advisable to check this predicate for emptiness (which corresponds to the removal of 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 [**15**].

For the rest of the code, the merge can be done with the remarks described in the previous section. As a result, the vectorized sample function was accelerated more than 10 times. The final vector code is presented in Figure 5. In this code, lines 11–14 correspond to initialization, lines 16, 17, and 50 are responsible for replacing the variables for merging symmetric sections of the code, and the unlikely branch is separated in the block located in lines 41–47.

5. Loop nests vectorization

The most complex context for code vectorization is the `starpu` function, which contains a loop with an unknown number of iterations (Figure 6). The loop located in this function in lines 15-33, 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

```
01 void sample_16(__m512 dl, __m512 ul, __m512 pl, __m512 cl,
02                __m512 dr, __m512 ur, __m512 pr, __m512 cr,
03                __m512 pm, __m512 um,
04                __m512 *d, __m512 *u, __m512 *p)
05 {
06     __m512 c, ums, pms, sh, st, s, uc;
07     __mmask16 cond_um, cond_pm, cond_sh, cond_st, cond_s, cond_sh_st;
08
09     // d/u/p/c/ums
10     cond_um = _mm512_cmp_ps_mask(um, z, _MM_CMPINT_LT);
11     *d = _mm512_mask_blend_ps(cond_um, dl, dr);
12     *u = _mm512_mask_blend_ps(cond_um, ul, ur);
13     *p = _mm512_mask_blend_ps(cond_um, pl, pr);
14     c = _mm512_mask_blend_ps(cond_um, cl, cr);
15     ums = um;
16     *u = _mm512_mask_sub_ps(*u, cond_um, z, *u);
17     ums = _mm512_mask_sub_ps(ums, cond_um, z, ums);
18
19     // Calculate main values.
20     pms = DIV(pm, *p);
21     sh = SUB(*u, c);
22     st = FNMADD(POW(pms, g1), c, ums);
23     s = FNMADD(c, SQRT(FMADD(g2, pms, g1)), *u);
24
25     // Conditions.
26     cond_pm = _mm512_cmp_ps_mask(pm, *p, _MM_CMPINT_LE);
27     cond_sh = _mm512_mask_cmp_ps_mask(cond_pm, sh, z, _MM_CMPINT_LT);
28     cond_st = _mm512_mask_cmp_ps_mask(cond_sh, st, z, _MM_CMPINT_LT);
29     cond_s = _mm512_mask_cmp_ps_mask(~cond_pm, s, z, _MM_CMPINT_LT);
30
31     // Store.
32     *d = _mm512_mask_mov_ps(*d, cond_st,
33                             MUL(*d, POW(pms, SET1(1.0 / GAMA))));
34     *d = _mm512_mask_mov_ps(*d, cond_s MUL(*d, DIV(ADD(pms, g6),
35                                                    FMADD(pms, g6, one))));
36     *u = _mm512_mask_mov_ps(*u, cond_st | cond_s, ums);
37     *p = _mm512_mask_mov_ps(*p, cond_st | cond_s, pm);
38
39     // Low prob - ignore it.
40     cond_sh_st = cond_sh & ~cond_st;
41     if (cond_sh_st != 0x0)
42     {
43         *u = _mm512_mask_mov_ps(*u, cond_sh_st, MUL(g5, FMADD(g7, *u, c)));
44         uc = DIV(*u, c);
45         *d = _mm512_mask_mov_ps(*d, cond_sh_st, MUL(*d, POW(uc, g4)));
46         *p = _mm512_mask_mov_ps(*p, cond_sh_st, MUL(*p, POW(uc, g3)));
47     }
48
49     // Final store.
50     *u = _mm512_mask_sub_ps(*u, cond_um, z, *u);
51 }
```

Figure 5. Vectorized version of the function `sample`

performing vectorization, this cycle must be transformed into a predicate form, in which the body should not contain transition operations. All cycle instructions are executed under their predicates, and the execution of the cycle is interrupted if all predicates are zeroed. This mechanism is described in [16] in relation to the vectoring of Shell sort and in [17] in relation to the Mandelbrot set construction. At the same time, it is worth noting that calls to prefun functions must also have corresponding predicates. After the body of the cycle is transformed into a predicate form, it can be vectorized, after which the predicates of the instructions are replaced by vector mask registers (this is where the additional parameter of the vectorized function `prefun` appears in the form of a mask). The result of the `starpu` function vectorization is presented in Figure 7. Line 18 shows the initial initialization of the full mask for performing vectorized loop iterations. As the cycle works, the mask is exhausted (line 32), and when it is fully zeroed, the cycle ends.

It should be noted that the vectorization of a loop with an unknown

```
01 void starpu(float dl, float ul, float pl, float cl,
02              float dr, float ur, float pr, float cr,
03              float &p, float &u)
04 {
05     const int nriter = 20;
06     const float tolpre = 1.0e-6;
07     float change, fl, fld, fr, frd, pold, pstart, udiff;
08
09     guessp(dl, ul, pl, cl, dr, ur, pr, cr, pstart);
10     pold = pstart;
11     udiff = ur - ul;
12
13     int i = 1;
14
15     for ( ; i <= nriter; i++)
16     {
17         prefun(fl, fld, pold, dl, pl, cl);
18         prefun(fr, frd, pold, dr, pr, cr);
19         p = pold - (fl + fr + udiff) / (fld + frd);
20         change = 2.0 * abs((p - pold) / (p + pold));
21
22         if (change <= tolpre)
23         {
24             break;
25         }
26
27         if (p < 0.0)
28         {
29             p = tolpre;
30         }
31
32         pold = p;
33     }
34
35     if (i > nriter)
36     {
37         cout << "divergence in Newton-Raphson iteration" << endl;
38         exit(1);
39     }
40
41     u = 0.5 * (ul + ur + fr - fl);
42 }
```

FIGURE 6. Original version of the function `starpu`

number of iterations can be quite dangerous, since the number of iterations of a vectorized cycle is equal to the maximum of the iterations number of the cycles with 16 combined calls of the original non-vectorized function. If there is a big difference in the number of iterations of the original code, there is a drop in efficiency due to the low density of the masks of the executable instructions, as shown in [**16**].

6. Results analysis

Before starting the optimization of the program code of the Riemann solver, an execution profile was collected, which showed that the execution time was distributed between individual functions according to the diagram presented in Figure 8.

To collect the execution profile, the source program was compiled with the prohibition of optimizing the substitution of the function body into the call point (inline). Thus, the diagram shows the net execution time of functions without taking into account nested calls. It can be seen from

```
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)));
}
```

Figure 7. Vectorized version of the function `starpu`

the diagram that the largest share of the execution time falls on the `prefun` function (36%), which contains a simple program context with one condition. Also, a significant part of the execution time falls on the `starpu` function (29%), which contains a loop nest with an unknown number of iterations. The remaining time is divided between the three other functions `guessp` (18%), `sample` (11%), `riemann` (6%).

The approaches to the vectorization of the Riemann solver functions described in the article were implemented in the C programming language using the intrinsics functions and tested on the Intel Xeon Phi 7290 microprocessors that are part of the computing segment knl of the MVS-10P supercomputer located in JSCC RAS.

Performance testing was carried out on the input data arrays collected in solving standard test problems: the Soda problem, the Lax problem, the weak shock wave problem, the Einfeldt problem, the Woodward-Colella problem, the Schu-Osher problem and others [18].

The diagram Figure 9 shows the effect of applying various optimizations to each of the functions in question, as well as the total acceleration obtained

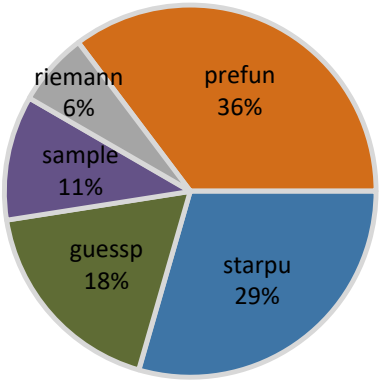


FIGURE 8. The distribution of the execution time of the Riemann solver between individual functions

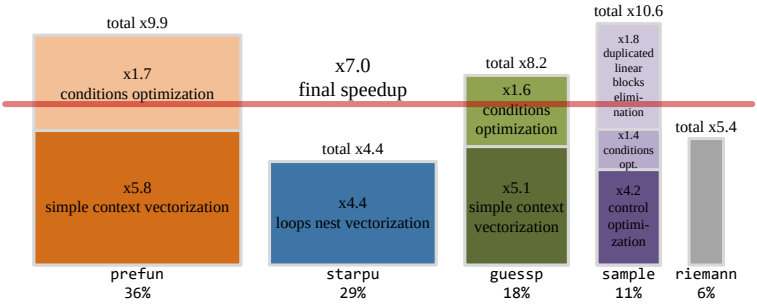


FIGURE 9. The acceleration diagram of individual functions and the total acceleration of the Riemann solver

as a result of vectorization.

It can be noted that the effect of the vectorization of a simple context varies from 5.1 to 5.8 times (for the functions `guessp` and `prefun`). It should also be noted a significant effect of the optimization of conditions (check on the emptiness of the mask of predicates, under which the execution of the block of operations is located). This is a fairly simple conversion, which accelerates the code from 1.4 to 1.7 times (for the functions `sample` and `prefun`) depending on how close the conditions are from adjacent iterations of the vectorized loop.

Separately, the diagram emphasizes the effect of applying variable optimization, which allowed the `sample` function to be accelerated 1.8 times by merging two subtrees of the control flow graph (that is, duplicate linear sections were removed).

As a result of applying all the described optimizations, it was possible to achieve acceleration of individual sections of the program execution 10 or more times, and the total acceleration of the entire Riemann solver was 7 times (marked with a red line in Figure 9).

7. Related works

Here is a brief comparison of the results obtained in this article with the results of other works aimed at improving the efficiency of the Riemann solvers on parallel architectures. At the same time, we note that although the work did not consider the parallelization of numerical methods based on the Riemann solver using MPI, OpenMP, as well as using graphics accelerators, it would be wrong not to mention the works concerning these aspects of parallelization.

The work [4] considers an algorithm for numerically solving the equations of magnetohydrodynamics based on the approximate Riemann solver, developed by Roe. In this case, the calculations are carried out on a block-structured computational grid, and for parallelization using MPI, the blocks of the grid are decomposed (cutting along one or several directions). The results of measuring the indicator of weak scalability when using up to 16 processors are presented. This indicator is close to unity and even in some cases exceeds it. Such a phenomenon, called superlinear scalability, can also be found in other works devoted to the parallelization of numerical methods of gas dynamics, for example, in [19].

The work [5] is devoted to parallelizing numerical methods based on various approximate Riemann solvers for execution on NVIDIA Tesla C2050 graphics accelerators. In addition to the Roe solver, the HLLE and HLLC [20] solvers are also used. Methods are demonstrated that allow achieving acceleration on a graphics accelerator compared to the Intel Xeon E5530 processor by 101 times using a uniform grid. The indicators of weak and strong scalability of a parallelized application using 32 GPUs are also given, they amounted to 98.8% and 85.0%, respectively.

The work [6] presents the results of using the PetClaw tool (a Python shell integrating the Clawpack computing core written in Fortran and the PETSc library into a single parallel application) for parallelizing hydrodynamic calculations using MPI to a large number of cores, up to 16 thousand. At the same time, a parallelization efficiency indicator close to unity was demonstrated.

The works [7], [8] describe approaches to parallelizing computations for modeling astrophysical flows on hybrid supercomputers equipped with Intel Xeon Phi accelerators. In particular, 134-fold acceleration of computational codes on one accelerator was demonstrated using multi-threaded parallelization, as well as 75% scalability when using up to 224

accelerators. The article notes that to further accelerate the application, a vectorization of the computing core is needed and a forecast is given to achieve 80% peak performance provided that vectorization is applied.

Closest to this work are [9], [10], in which special attention is paid to the vectorization of the Riemann solver for the numerical solution of shallow water equations. The work [9] presents results on acceleration of 6-7 times when vectorizing a Riemann solver for a given single precision on Intel Xeon Phi KNC accelerators. In [10], the vectorization method was improved and the effect was already 2.4-6.5 in double-precision operations on Intel Xeon Phi KNL microprocessors. However, these works use a specially developed approximate Riemann solver for shallow water equations [21]. The implementation of this solver contains a simpler computational context compared to the exact solver, whose vectorization is considered in the current work. The approximate solver from [21] contains only arithmetic operations and can be automatically vectorized after special preparation of input parameters (grouping several solver calls into one call with vector parameters).

Based on a comparison of current work with related work in this area, it can be noted that the acceleration rate of the exact Riemann solver, equal to 7 and achieved by vectorizing the program code using the AVX-512 instruction set, can be considered acceptable.

Conclusion

The paper examined approaches to vectorizing complex program contexts using the AVX-512 instruction set. This instruction set appeared in Intel microprocessors starting with Intel Xeon Phi KNC accelerators, and then entered Intel Xeon Phi KNL microprocessors, Intel Xeon Skylake and beyond. The AVX-512 instructions support predicate processing of data elements, which allows you vectorize complex, branched program code with them.

The exact Riemann solver was used as the target task, since it has a compact implementation but at the same time contains features that prevent automatic vectorization by means of the compiler (function calls, complex control, nested loops). For vectorization, an approach was used in which several consecutive calls to the solver function were replaced with one call with vector parameters (and with corresponding changes to the function body). This allowed several instances of the task to be performed simultaneously on the same processor core (the number of instances equals the width of the vector, in this case 16).

The components of the Riemann solver were analyzed, their features were highlighted, and a method of effective vectorization was proposed for each of them. As a result of vectorization, acceleration was achieved more

than 10 times in individual sections of the solver, and the final acceleration was 7 times on the data with single precision.

The developed methods of vectorization of a complex program context can be used to optimize other computational problems. In particular, at present, a team of authors of this article is developing a library aimed at vectorization of arbitrary flat cycles, that is, cycles in which there are no inter-iteration dependencies. In this case, the body of the loop can include elements such as complex control, loop sockets, calls to pure functions, goto statements, and others. The implementation of such a tool will significantly improve the performance of calculation codes, the automatic vectorization of which by the compiler is impossible.

The work was done at the JSCC RAS as part of the state assignment on the topic 0065-2019-0016 (registration number AAAA-A19-119011590098-8).

References

[1] A. G. Kulikovskiy , N. V. Pogorelov, A. Y. Semenov. *Mathematical aspects of numerical solution of hyperbolic systems*, Fizmatlit, M., 2001 (in Russian), 608 pp. ↑41

[2] V. E. Borisov, Yu. G. Rykov. "An exact Riemann solver in the algorithms for multicomponent gas dynamics", *Keldysh Institute preprints*, 2018, 096 (in Russian), 28 pp. ↑41

[3] S. K. Godunov, A. V. Zabrodin, M. Y. Ivanov, A. N. Krayko, G. P. Prokopov. *Numerical solution of multidimensional problems of gas dynamics*, Nauka, M., 1976 (in Russian), 400 pp. ↑41

[4] U. Shumlak, B. Udrea. *An approximate Riemann solver for MHD computations on parallel architectures*, AIAA-2001-2591, 15th AIAA Computational Fluid Dynamics Conference (11 June 2001–14 June 2001, Anaheim, CA, USA), 2001, 8 pp. ↑42,54

[5] H.-Y. Schive, U.-H. Zhang, T. Chiueh. "Directionally unsplit hydrodynamic schemes with hybrid MPI/OpenMP/GPU parallelization in AMR", *International Journal of High Performance Computing Applications*, **26**:4 (2011), pp. 367–377. ↑42,54

[6] K. T. Mandly, A. Alghamdi, A. Ahmadia, D. I. Ketcheson, W. Scullin. "Using Python to construct a scalable parallel nonlinear wave solver", *Proceedings of the 10th Python in Science Conference*, SCIPY 2011 (11–16 Jule 2011, Austin, Texas, USA), 2011, pp. 61–66. ↑42,54

[7] I.M. Kulikov, I.G. Chernykh, E.I. Vorobyev, A.V. Snytnikov, D.V. Vins and others. "Numerical hydrodynamics simulation of astrophysical flows at Intel Xeon Phi supercomputers", *Vestn. YuUrGU. Ser. Vych. Matem. Inform.*, **5**:4 (2016), pp. 77–97 (in Russian). ↑42,54

[8] I. Kulikov, I. Chernykh, V. Vshivkov, V. Prigarin, V. Mironov, A. Tatukov. "The parallel hydrodynamic code for astrophysical flow with stellar equation

of state", *RuSCDays 2018: Supercomputing*, Communications in Computer and Information Science, vol. **965**, Springer, Cham, 2018, pp. 414–426. ↑42,54

[9] M. Bader, A. Breuer, W. Höltz, S. Rettenberger. "Vectorization of an augmented Riemann solver for the shallow water equations", *Proceedings of the 2014 International Conference on High Performance Computing and Simulation*, HPCS 2014 (21–25 July 2014, Bologna, Italy), 2014, pp. 193–201. ↑42,55

[10] C. R. Ferreira, K. T. Mandli, M. Bader. "Vectorization of Riemann solvers for the single- ans multi-layer shallow water equations", *Proceedings of the 2018 International Conference on High Performance Computing and Simulation*, HPCS 2018 (16–20 July 2018, Orleans, France), 2018, pp. 415-422. ↑42,55

[11] V. Y. Volkonskiy, S. K. Okunev. "Predicate representation as the basis of program optimization for architectures with pronounced parallelism", *Informatsionnyye tekhnologii*, 2003, no.4, pp. 36–45 (in Russian). ↑43

[12] A. K. Kim, V. I. Perekatov, S. G. Yermakov. *Microprocessors and computing complexes of the «Elbrus» family*, Piter, Saint Petersburg, 2013 (in Russian), 273 pp. ↑43

[13] Intel Intrinsics Guide, `https://software.intel.com/sites/landingpage/IntrinsicsGuide/`. ↑44

[14] E. F. Toro. *Riemann Solvers and Numerical Methods for Fluid Dynamics: A Practical Introduction*, 2nd Edition, Springer, Berlin–Heidelberg, 1999, 645 pp. ↑44

[15] A. A. Rybakov. "Optimization of the problem of conflict detection with dangerous aircraft movement areas to execute on Intel Xeon Phi", *Programmnyye produkty i sistemy*, **30**:3 (2017), pp. 524–528 (in Russian). ↑49

[16] A. A. Rybakov, P. N. Telegin, B. M. Shabanov. "Cycle socket vectoring issues using AVX-512iInstructions", *Programmnyye produkty, sistemy i algoritmy*, 2018, no.3 (in Russian), 11 pp. ↑50,51

[17] O. Krzikalla, F. Wende, M. Hohnerbach. "Dynamic SIMD vector lane scheduling", *ISC High Performance 2016: High Performance Computing*, Lect. Notes Comput. Sci., vol. **9945**, Springer, Cham, 2016, pp. 354–365. ↑50

[18] P. V. Bulat, K. N. Volkov. "One-dimensional gas dynamics problems and their solution based on high-resolutinon finite difference schemes", *Nauchno-tekhnicheskiy vestnik informatsionnykh tekhnologiy, mekhaniki i optiki*, **15**:4 (2015), pp. 731–740 (in Russian). ↑52

[19] L.A. Benderskiy, D.A. Lyubimov, A.A. Rybakov. "Scaling of fluid dynamic calculations using the RANS/ILES method on supercomputer", *Trudy NIISI RAN*, **7**:4 (2017), pp. 32–40 (in Russian). ↑54

[20] C. Kong. *Comparison of Approximate Riemann Solvers*, A dissertation of the degree of Master of Science in Mathematical and Numerical Modeling of the Atmoshere and Oceans, Department of Mathematics, University of Reading, 2011. ↑54

[21] D. L. George. "Augmented Riemann solvers for the shallow water equations over variable topography with steady states and inundation", *Journal of Computational Physics*, **227**:6 (2008), pp. 3089–3113. ↑55

Received	19.02.2019
Revised	10.09.2019
Published	30.09.2019

Recommended by *prof. S. M. Abramov*

Sample citation of this publication:

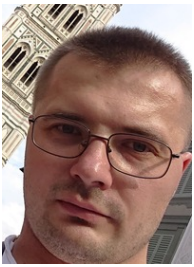
Alexey A. Rybakov, Sergey S. Shumilin. "Vectorization of the Riemann solver using the AVX-512 instruction set". *Program Systems: Theory and Applications*, 2019, **10**:3(42), pp. 41–58.

10.25209/2079-3316-2019-10-3-41-58

http://psta.psiras.ru/read/psta2019_3_41-58.pdf

The same article in Russian: 10.25209/2079-3316-2019-10-3-59-80

About the authors:



Alexey Anatoljevich Rybakov

Rybakov Alexey Anatolyevich - candidate of Physico-Mathematical Sciences, leader researcher of JSCC RAS - Branch of the Federal State Institution NIISI RAS. Research interests include mathematical modelling of gas dynamics problems using supercomputers, methods for constructing and managing computational grids, discrete mathematics, graph theory, random graph models, parallel programming, and functional programming.

0000-0002-9755-8830

e-mail: rybakov@jscc.ru

Sergey Sergeevich Shumilin

Shumilin Sergey Sergeevich - senior engineer of JSCC RAS - Branch of the Federal State Institution NIISI RAS. Areas of scientific interest are machine learning, data analysis, algorithms, parallel programming.

0000-0002-3953-7054

e-mail: shumilin@jscc.ru

Эта же статья по-русски: 10.25209/2079-3316-2019-10-3-59-80