Proceedings of the 2013 Federated Conference on
Computer Science and Information Systems pp. 447–454

Application of AVX (Advanced Vector Extensions) for Improved Performance of the PARFES – Finite Element Parallel Direct Solver

Sergiy Fialko
Tadeusz Kościuszko Cracow University of Technology
ul. Warszawska 24 St., 31-155 Kraków, Poland
Email: sfialko@poczta.onet.pl

***Abstract*—The paper considers application of the AVX (Advanced Vector Extensions) technique to improve the performance of the PARFES parallel finite element solver, intended for finite element analysis of large-scale problems of structural and solid mechanics using multi-core computers. The basis for this paper was the fact that the *dgemm* matrix multiplication procedure implemented in the Intel MKL (Math Kernel Library) and ACML (AMD Core Math Library) libraries, which lays down the foundations for achieving high performance of direct methods for sparse matrices, does not provide for satisfactory performance with the AMD Opteron 6276 processor, Bulldozer architecture, when used with the algorithm required for PARFES. The procedure presented herein significantly improves the performance of PARFES on computers with processors of the above architecture, while maintaining the competitiveness of PARFES with the Intel MKL *dgemm* procedure on computers with Intel processors.**

I. Introduction

THE PARFES (Parallel Finite Element Solver) is a sparse direct method for solving linear equation sets with sparse symmetric matrices, which arise when the finite element method is applied to structural and solid mechanics problems, is presented in [7], [8]. The method is developed to be used in FEA software focused on multi-core shared memory computers. PARFES supports core mode (CM) as well as two out of core modes – OOC and OOC1. In the core mode, the solver only utilizes random access memory (RAM), demonstrating good performance and speed up when the number of threads increases. If the dimension of the problem exceeds the RAM capacity, the method switches to the OOC mode, in which disk storage is used, and the amount of I/O operations is minimal. Performance and speed up deteriorate slightly compared to the CM. If the amount of RAM is not sufficient for the OOC mode, PARFES switches to OOC1. In this mode, the number of I/O operations is greatly increased; however, the RAM amount requirements are low. The performance and speed up degrade significantly, but this method allows solving problems of several million equations using desktop and laptop computers.

The option to use disk memory is the advantage of PARFES compared to PARDISO (Parallel Direct Solver), which is described in [16] and presented in the Intel MKL library [11]. Although PARDISO formally supports the OOC mode, practice showed that in this mode, this method is considerably inferior both to PARFES, and the multifrontal method where small tasks are concerned [1], [5], [10], and simply crashes when used for larger problems [7], [15].

In contrast to the multifrontal method, PARFES demonstrates significantly higher performance and speed up, and smaller RAM requirements (in OOC1 mode) [7], [8].

This paper describes further development of PARFES for the use with Intel AVX instructions [14] that implement computation vectorization elements with 256-bit registers, allowing to perform four multiplications or four additions of double type values in one CPU cycle.

It was discovered that the *dgemm* matrix multiplication procedure as implemented in Intel MKL 11.0 [12] does not provide for satisfactory performance of PARFES on a computer with a 16-core AMD Opteron 6276 CPU 2.3/3.2 GHz processor, Bulldozer architecture. For test 1: $\mathbf{C} = \mathbf{C} - \mathbf{A}\cdot\mathbf{B}$, where $\mathbf{A}$, $\mathbf{B}$, $\mathbf{C}$ are 8 000 × 8 000 square matrices, the performance of this procedure is 3 958 MFLOPS with a single thread and 35 013 MFLOPS with 16 threads. The performance of the same procedure as implemented in ACML 15.2.0 (AMD Core Math Library) [2] is 14 203 MFLOPS and 94 852 MFLOPS respectively.

However, when solving test 2 (Fig. 1, 2) it was found that the performance of this algorithm degrades (see Table 1), and the threads run in the OS kernel mode for a considerable amount of time.

```
#pragma omp parallel for
for(ib=0; ib<Nb; ++ib)
{
    ip = omp_get_thread_numb();
    Cib = Cib − Aib·B;
}
```

Fig.1 Algorithm for test 2

Matrices $\mathbf{C}$ and $\mathbf{A}$ have a block structure (Fig. 2), *ip* is the thread number, and *ib* is the block number. Inside the loop, the single-treaded version of the *dgemm* procedure (ACML [2]) is used. The arrows indicate the packing of data in the respective matrices.

This work was supported by Narodowy Centrum Nauki on the basis of decision DEC-2011/01/B/ST6/00674.

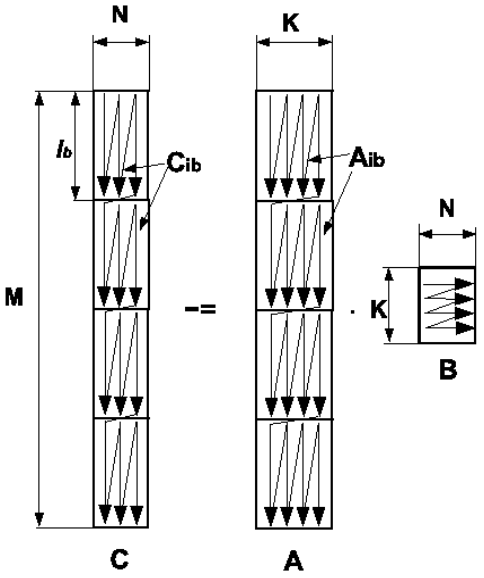

Fig. 2 Structure of **A**, **B**, **C** matrices in test 2

Test 2 is a good simulation of the PARFES correction procedure [7], [8], when the *jb* block-column (matrix **C**) is updated by the *kb* block-column (matrix **A**) located to the left of the former.

Thus, it was decided to develop a new procedure, *microkern_8x4_AVX*, which would allow achieving high performance with processors that support AVX instructions on the × 64 platform.

II. Factorization Stage

A.Problem definition

Let us consider the direct method for solving linear equation sets.

$$\mathbf{KX}=B,\quad K=K^T,\quad X=\{x_i\},\quad B=\{b_i\},\quad i\in[1,\,nrhs]\ , \quad (1)$$

where **K** is the symmetric sparse stiffness matrix; **X** and **B** are solution vectors and right-hand parts for multiple load cases; and *nrhs* is the number of right-hand parts. The decomposition is sought in the form of

$$K=L\cdot S\cdot L^T\ , \quad (2)$$

where **L** is the lower triangular matrix and **S** is the sign diagonal that summarizes the Cholesky decomposition method into a class of indefinite matrices. After factorization (2), forward substitution, diagonal scaling and back substitution are carried out:

$$\begin{aligned} L\cdot Y&=B \rightarrow Y\\ S\cdot Z&=Y \rightarrow Z\\ L^T\cdot X&=Z \rightarrow X \end{aligned}\ . \quad (3)$$

B.Sparse matrix analysis

First of all the adjacency graph for nodes of the finite element model is reordered to reduce the number of non-zero entries in the factorized stiffness matrix. The number of non-zero entries and the non-zero structure of the sparse lower triangular matrix **L** depend on the reordering method used [3].

Each node of FE model, which has *dof* degrees of freedom, produces a dense submatrix with the dimensions *dof* × *dof*. Therefore, the physical formulation of the problem leads to the division of the original sparse matrix into dense submatrices of relatively small dimensions. To achieve high performance, we should enlarge the dimension of these blocks, and do so in a way that provides for the minimal number of zero entries appearing as the result of such procedure. To this end, we use the algorithm presented in [8]. As a result, matrix **L** is divided into dense rectangular blocks, and the blocks located on the main diagonal are filled completely. The blocks located below the main diagonal may be filled either completely or partially. Memory is not allocated to empty blocks, and for partially filled blocks, only non-zero rows are taken into consideration (Fig. 3).

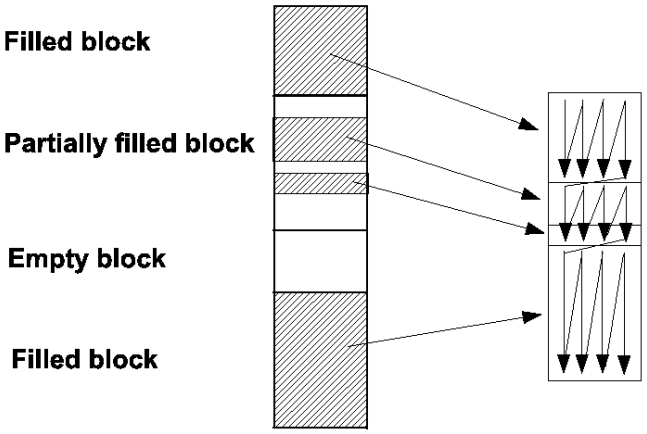

Fig.3. Block-column consisting of empty, partially and completely filled blocks. The packing of data in column major storage is shown to the right.

A more detailed description of the method is provided in [7], [8].

C.Numerical factorization

The algorithm used in this method of left-looking block factorization for the CM mode is shown in Fig. 4, 5.

Factorization is performed in a loop going over *jb* block-columns, the current block column *jb* is corrected by the fully factored block columns located to the left (p.2). Nb is the number of block-columns (p. 1, Fig. 5).

To avoid a situation when two or more threads attempt to modify the same block $\mathbf{A}_{ib,jb}$ in a *jb* block-column, all blocks of current block row are mapped to the same thread. To evenly distribute the processor load, the weight of each block row is calculated (the number of non-zero elements in this block-row), the block rows are sorted in the descending weight order, and then mapped to the threads alternately; with that, the current block row is assigned to the thread with the currently-minimal amount of computation.

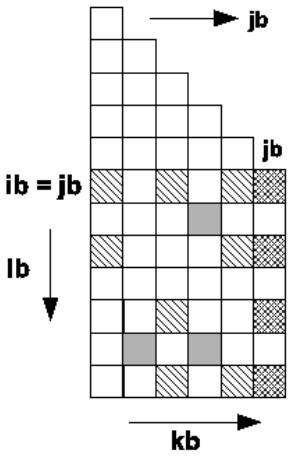

Fig. 4. Left-looking factorization of jb block-column. All block-columns located to the left of jb (kb < jb) are fully factorized.

1. **do** *jb=1,Nb*
2. update of block-column *jb*
prepare parallel tasks Q[*ip*] for update of block-column *jb*
#pragma omp parallel
while(Q[*ip*])
$\{\mathbf{L}_{jb,kb}, \mathbf{L}_{ib,kb}, kb\} = (Q[ip]/\{\mathbf{L}_{jb,kb}, \mathbf{L}_{ib,kb}, kb\})$
$\mathbf{A}_{ib,jb} = \mathbf{A}_{ib,jb} - \mathbf{L}_{ib,kb} \cdot \mathbf{S}_{kb} \cdot \mathbf{L}_{jb,kb}^T$
$(kb \in List_k[jb];\ ib \in L_{kb})$
end while
end of parallel region
end of update
3. factoring of block-column *jb*
$\mathbf{A}_{jb,jb} = \mathbf{L}_{jb,jb} \cdot \mathbf{S}_{jb} \cdot \mathbf{L}_{jb,jb}^T$
#pragma omp parallel for $(ib \in L_{jb})$
$\mathbf{L}_{jb,jb} \cdot \mathbf{S}_{jb} \cdot \mathbf{L}_{ib,jb}^T = \mathbf{A}_{ib,jb}^T \rightarrow \mathbf{L}_{ib,jb}^T$
end of factoring
4. prepare *List_k* for block-columns, which are located
to the right of block-column *jb* and will be updated by it
end **do**

Fig. 5. Looking-left block factorization algorithm

As a result, a queue of tasks Q[*ip*] is created for each *ip* thread, *ip* = 0, 1, …, *np*-1, *np* is a number of threads. Each queue element $\{\mathbf{L}_{jb,kb}, \mathbf{L}_{ib,kb}, kb\}$ contains pointers to the factorized matrix blocks $\mathbf{L}_{jb,kb}$, $\mathbf{L}_{ib,kb}$ and the *kb* index of the sign diagonal block $\mathbf{S}_{kb}$. The *jb* block-column is corrected only by those block-columns that have non-zero blocks $\mathbf{L}_{jb,kb}^T$ in the block row *ib* = *jb* (*kb* ∈ *List_k*[*jb*]).

In the parallel region each thread runs a *while* loop going over its own queue of tasks Q[*ip*] until it is exhausted.

The nearest element is popped from the queue and immediately deleted: $\{\mathbf{L}_{jb,kb}, \mathbf{L}_{ib,kb}, kb\} = (Q[ip]/\{\mathbf{L}_{jb,kb}, \mathbf{L}_{ib,kb}, kb\})$. Then, the task $\mathbf{A}_{ib,jb} = \mathbf{A}_{ib,jb} - \mathbf{L}_{ib,kb} \cdot \mathbf{S}_{kb} \cdot \mathbf{L}_{jb,kb}^T$ is performed. Conditions *ib* ∈ $\mathbf{L}_{kb}$ and *ib* = ∈ $\mathbf{L}_{jb}$ mean that the *ib* index accepts only those values that correspond to the non-zero blocks in the *kb* and *jb* block-columns respectively.

The details of this algorithm are presented in [7].

To ensure high performance, we can use the *dgemm* matrix multiplication procedure, or the microkern_8x4_AVX procedure presented in this paper.

When the *jb* block-column is completely corrected, it is factorized (p.3), and then the *jb* index is pushed to the *List_k* block-column list, which will be updated by the *jb* block-column at the next factorization steps (p. 4).

Therefore, performance at the numerical factorization stage is mainly determined by the performance of the matrix multiplication procedure. Test 2 (see Fig. 1, 2) simulates correction of the *jb* block-column by block-columns located to the left of it. Therefore, this was the test mainly used to test out the microkern_8x4_AVX procedure. Following [6], we will refer to the procedure microkern_8x4_AVX, whose code is written based on the AVX instructions, as microkernel.

D.Microkernel microkern_8x4_AVX

The proposed approach uses the same idea as in the development of the microkernel, based on SSE2 [6], [9], [10]. To achieve high performance, it is necessary to use cache blocking, register blocking, computing vectorization, data repacking in order to reduce the number of cache misses, and to unroll the inner loop to maximize the use of the processor pipelines. Since in the PARFES method, the maximum dimension of block l_b is 120, the l_b, K, N dimensions do not exceed this value. Therefore, division of matrices $\mathbf{A}_{ib}$, **B** and $\mathbf{C}_{ib}$ into blocks (cache blocking) is not required, and the dimension of TLB (translation look aside buffer) will not be exceeded [6].

Modern processors supporting AVX have 256-bit YMM registers, and 16 registers are available on the ×64 platform. Four floating-point double precision words can be loaded into each register, and four additions or four multiplications are carried out per each clock of processor. The register blocking diagram for the ×64 platform is shown in Fig. 6.

The result is stored in 8 registers intended for the elements of matrix $\mathbf{C}_{ib}$. Two registers are used for elements of matrix $\mathbf{A}_{ib}$, one register – for elements of matrix **B** and one register is required to store the intermediate multiplication results. The block dimension is $m_r \times n_r = 8 \times 4$. When the inner loop runs, the elements of matrices $\mathbf{A}_{ib}$ and **B** are repacked to ensure their locations in neighboring RAM addresses. This reduces the number of cache misses and allocates the data in the cache extremely densely. We denote: **AA = repack($\mathbf{A}_{ib}$), BB = repack(B),** where **Dest = repack(Source)** means repacking from array **Source** to array **Dest** (Fig. 6, bottom). The elements of matrices $\mathbf{A}_{ib}$, **B** and $\mathbf{C}_{ib}$ are located in the RAM column-major storage. Prefetch instructions are applied to hide memory system latency.

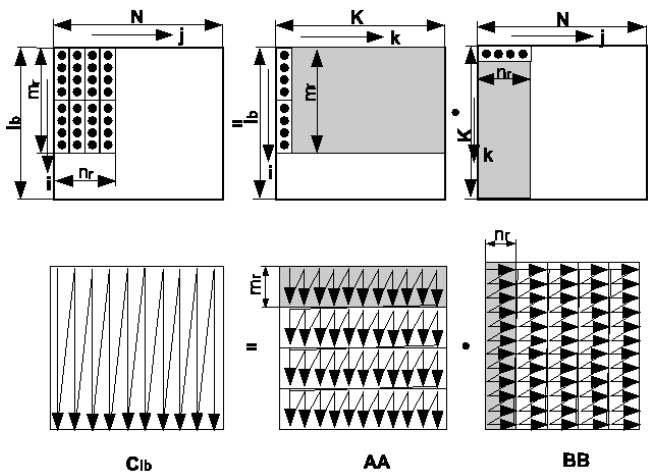

Fig. 6. Diagram of register blocking (top) and data repacking (bottom)

The AVX is used to accelerate the transmission of data when matrices $\mathbf{A}_{ib}$, **B** are repacked into arrays **AA** and **BB** respectively. The pseudo code presenting the microkern_8x4_AVX procedure is shown in Fig 7.

1. **Procedure Pack_BB: B = repack(BB)** (fig. 6, bottom).

2. **Procedure microkern_8x4_AVX:**

$\mathbf{C}_{ib} = beta \cdot \mathbf{C}_{ib} + alpha \cdot \mathbf{A}_{ib} \cdot \mathbf{B}$ (in the future index *ib* is omitted)

AA = repack(A)

```
for(j=0; j<N; j+=nr)
{
   //pBB0 = BB+K*j;  //point to current
   //vertical pane of BB
    for(i=0; i<lb; j+=mr)
    {
      pAA=AA+i*K; //point to current
                  //horizontal pane of AA
      pC=C+ldc*j+i; //point to Cij, ldc =
                    //lb.
      pBB = pBB0;

      //move Cij, Ci+1,j, … , Ci+7, j to cache
      //untill CPU run internal loop
      //move Ci,j+1, Ci+1,j+1, … , Ci+7, j+1 to cache
      //untill CPU run internal loop
      _mm_prefetch((const char *)(pC+ldc),
                   _MM_HINT_T0);
      _mm_prefetch((const char *)(pC+2*ldc),
                   _MM_HINT_T0);
      c1 = _mm256_setzero_pd();  //c1 ← 0
      ..................................
      c8 = _mm256_setzero_pd();  //c8 ← 0
      for(k=0; k<K; k+=16)
      {
        _mm_prefetch((const char *)(pAA+mr),
                     _MM_HINT_T0);
        _mm_prefetch((const char *)(pBB+2*nr),
                     _MM_HINT_T0);
        a0 = _mm256_load_pd(pAA);
        a1 = _mm256_load_pd(pAA+4);
        b0 = _mm256_broadcast_sd(pBB);
        b1 = _mm256_broadcast_sd(pBB+1);
        b2 = _mm256_broadcast_sd(pBB+2);
        b3 = _mm256_broadcast_sd(pBB+3);

        mul = _mm256_mul_pd(a0, b0);
        c1  = _mm256_add_pd(c1, mul);
        mul = _mm256_mul_pd(a1, b0);
        c2  = _mm256_add_pd(c2, mul);

        mul = _mm256_mul_pd(a0, b1);
        c3  = _mm256_add_pd(c3, mul);
        mul = _mm256_mul_pd(a1, b1);
        c4  = _mm256_add_pd(c4, mul);

        mul = _mm256_mul_pd(a0, b2);
        c5  = _mm256_add_pd(c5, mul);
        mul = _mm256_mul_pd(a1, b2);
        c6  = _mm256_add_pd(c6, mul);

        mul = _mm256_mul_pd(a0, b3);
        c7  = _mm256_add_pd(c7, mul);
        mul = _mm256_mul_pd(a1, b3);
        c8  = _mm256_add_pd(c8, mul);
        //and so on 15 times

        pAA += 16*mr;
        pBB += 16*nr;
      }//end k loop

// put alpha*A*B to c1 – c8
      mul =_mm256_set_pd(alpha,alpha,alpha,alpha);
      c1  = _mm256_mul_pd(c1, mul);
      c2  = _mm256_mul_pd(c2, mul);
      c3  = _mm256_mul_pd(c3, mul);
      c4  = _mm256_mul_pd(c4, mul);
      c5  = _mm256_mul_pd(c5, mul);
      c6  = _mm256_mul_pd(c6, mul);
      c7  = _mm256_mul_pd(c7, mul);
      c8  = _mm256_mul_pd(c8, mul);

      if(beta)
      {
        //put alpha*AAi*BBj + beta*CCij to c1 – c8
        b0  = _mm256_set_pd(beta,beta,beta,beta);
        a0  = _mm256_loadu_pd(pC);
        a1  = _mm256_loadu_pd(pC+4);
        mul = _mm256_mul_pd(b0, a0);
        c1  = _mm256_add_pd(c1, mul);
        mul = _mm256_mul_pd(b0, a1);
        c2  = _mm256_add_pd(c2, mul);

        a0  = _mm256_loadu_pd(pC+ldc);
        a1  = _mm256_loadu_pd(pC+ldc+4);
        mul = _mm256_mul_pd(b0, a0);
        c3  = _mm256_add_pd(c3, mul);
        mul = _mm256_mul_pd(b0, a1);
        c4  = _mm256_add_pd(c4, mul);

        a0  = _mm256_loadu_pd(pC+2*ldc);
        a1  = _mm256_loadu_pd(pC+2*ldc+4);
        mul = _mm256_mul_pd(b0, a0);
        c5  = _mm256_add_pd(c5, mul);
        mul = _mm256_mul_pd(b0, a1);
        c6  = _mm256_add_pd(c6, mul);

        a0  = _mm256_loadu_pd(pC+3*ldc);
        a1  = _mm256_loadu_pd(pC+3*ldc+4);
        mul = _mm256_mul_pd(b0, a0);
        c7  = _mm256_add_pd(c7, mul);
        mul = _mm256_mul_pd(b0, a1);
        c8 = _mm256_add_pd(c8, mul);
      }//end if(beta)

      //unload c1 – c8 to matrix C
```

```
    _mm256_storeu_pd(pC,    c1);
    _mm256_storeu_pd(pC+4,  c2);
    pC += ldc;
    _mm256_storeu_pd(pC,    c3);
    _mm256_storeu_pd(pC+4,  c4);
    pC += ldc;
    _mm256_storeu_pd(pC,    c5);
    _mm256_storeu_pd(pC+4,  c6);
    pC += ldc;
    _mm256_storeu_pd(pC,    c7);
    _mm256_storeu_pd(pC+4,  c8);
  }//end i loop
}//end j loop
```

Fig. 7. microkern_8x4_AVX

Here, for ease of understanding the basic idea of the method, we consider only the case when M is a multiple of m_r, N multiple of n_r, and K is a multiple of 16. In the real microkernel, submatrices of dimension $M_1 \times K$ and $K \times N_1$ are extracted from matrices **A** and **B**, where M_1 and N_1 assume the greatest value with the following limitations: ($M_1 \le M$) ^ (M_1 % m_r) = 0, ($N_1 \le N$) ^ (N_1 % n_r) = 0, where $a\%b$ means that the remainder after the division of a by b is zero. Therefore, matrices $\mathbf{A}_{ib}$, **B** are divided into blocks, in which the largest submatrices are a multiple of m_r and n_r respectively. The YMM register blocking scheme (Fig. 6) is applied specifically for these submatrices. For the remaining small submatrices, simpler multiplication methods are used.

Matrix B is repacked in a separate procedure, allowing us to use it only once for each *kb* block-column. To produce register blocking, indexes i, j are increased by increments of m_r, n_r correspondingly. The pointers pAA and pBB are set to the beginning of the horizontal strip of matrix AA and the vertical strip of BB (Fig. 6, 8), before loops with indexes i, j are initiated.

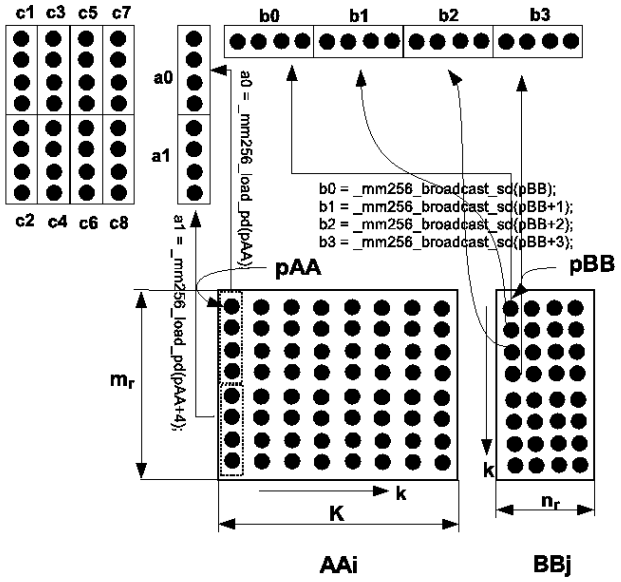

Fig. 8. Map of YMM register's loading

Registers c1 through c8 are zeroed before the loop with index *k* is started. The inner loop with index k is unrolled 16 times. The eight consecutive elements of array **AA** (the entire column of horizontal strip – Fig. 6) are loaded into registers a0, a1 by means of the instruction _mm256_load_pd(...). Each of the four consecutive elements of array **BB** (the entire vertical strip row) is sent to registers b0, b1, b2, b3 correspondingly, by means of the instruction _mm256_broadcast_sd(...) (Fig. 7, 8). As a result, the first element from the vertical strip row is found four times in register b0, the second – four times in register b1, etc.

The contents of register a0 are multiplied by the contents of register b0, and the result is placed into register mul – instruction mul = _mm256_mul_pd(a0, b0). Instruction c1 = _mm256_add_pd (c1, mul) adds up the contents of registers c1 and mul, and sends the result into register c1. Then, the contents of register a1 are multiplied by the contents of register b0, and the result is added to the contents of register c2. The contents of registers a0, a1 are multiplied by b1, and the results are added to the contents of registers c3, c4, etc. respectively. At the end of the loop with index *k*, registers c1 through c8 contain the fully computed elements of the $m_r \times n_r$ block of matrix $\mathbf{C}_{ib}$, which constitute the result of multiplying the horizontal strip $m_r \times K$ of matrix $\mathbf{A}_{ib}$, repacked into array **AA**, by the vertical strip $K \times n_r$ of matrix **B**, repacked into array **BB**.

This result is multiplied by scalar factor *alpha*. If the *beta* coefficient is non-zero, the 8 elements c_{ij}, $c_{i+1,j}$, ..., $c_{i+7,j}$ are loaded into registers a0, a1 by using _mm256_loadu_pd(...). The elements of matrix $\mathbf{A}_{ib}$ are loaded from array **AA** by using instruction _mm256_load_pd (...), because memory for array **AA** is allocated with a 32 byte alignment. Memory for matrix **C** is allocated without the 32 byte alignment, so here we use the _mm256_loadu_pd (...). The elements of matrix **C** held in registers a0, a1 are multiplied by factor *beta* and added to the contents of registers c1 and c2. Then, eight elements from the next column of matrix $\mathbf{C}_{ib}$ – $c_{i,j+1}$, $c_{i+1,j+1}$, ..., $c_{i+7,j+1}$ are loaded into registers a0 and a1, multiplied by factor *beta*, and added to the contents of registers c3, c4, etc. Transition to the next column of matrix $\mathbf{C}_{ib}$ is made by offsetting $ldc = l_b$ of pC pointer. While the current iteration is running, the prefetch instruction is applied to transmit the elements of matrix $\mathbf{C}_{ib}$ from RAM to the cache, as required for the next iteration of the loop with index *i*.

As a result, registers c1 through c8 hold the accumulated result of *alpha*·**AA**i***BB**j+*beta*·**CC**ij, where **AA**i, **BB**j are, respectively, the horizontal strip of matrix $\mathbf{A}_{ib}$, determined by the value of index *i*, and the vertical strip of matrix **B**, defined by the value of index *j*, and **CC**ij – the corresponding block of matrix $\mathbf{C}_{ib}$. Instructions _mm256_storeu_pd (...) unload data from registers c1 through c8 to the corresponding elements of matrix $\mathbf{C}_{ib}$.

III. Numerical results

A. Test 2

The results of test 2, described in the introduction (Fig. 1, 2), have been obtained on two computers and are shown in Table 1.

The first computer has a 16-core AMD Opteron 6276 CPU 2.3/3.2 GHz processor, 64 GB DDR3 RAM, and runs Windows Server 2008 R2 Enterprise SP1, 64 bit. The second computer has a 4-core Intel i7 2760QM CPU 2.4/3.5 GHz processor, 8 GB DDR3 RAM, and runs Windows 7 Professional SP1, 64 bit.

For the ACML 15.2.0 procedure, column for computing on 16 threads (the computer with AMD processor) contains two values: the first (41 469 MFLOPS) corresponds to solving the problem by parallelizing only within the *dgemm* procedure, using its multi-threaded version; while the second (10 061 MFLOPS) – to the use of the single-threaded version of *dgemm* in a parallel OpenMP loop. The first value is used to estimate the top performance of the AMD Opteron 6276 CPU for this test, since the ACML library is best adapted to AMD processors. The second value confirms that the *dgemm* procedure from the ACML library does not work properly in the mode required by PARFES.

A comparison of the results (Table 1) showed that the proposed microkern_8x4_AVX procedure successfully solved this problem on the computer with AMD Opteron 6276 processor, as well as on the computer with Intel i7 2760QM processor.

Next, we consider two real-life problems, taken from the collection of SCAD Soft – a Software Company (www.scadsoft.com) developing software for civil engineering. SCAD is FEA software, which is widely used in the CIS region and has a certificate of compliance to local regulations.

I. Problem 1

A design model of multistorey building contains 2 546 400 equations, consists of triangular, quadrilateral shell finite elements, as well as spatial frame ones (Fig. 9).

The original stiffness matrix contains 27 927 845 nonzero entries, and lower triangular factorized matrix – 1 124 085 204 nonzero entries. METIS reordering method [13] has been used.

The duration and performance of the factorization stage is presented in Table 2. As in the preceding example, the *dgemm* procedure from the Intel MKL 11.0 library does not achieve the desired performance on the computer with AMD Opteron 6276 processor. The *dgemm* procedure from the ACML library works well on a single thread, but when PARFES implements multithreading, the procedure is not performing its task. The microkern_8x4_AVX procedure proposed in this paper demonstrates good results with a single thread as well as during multi-threading. It is interesting to note that on a computer with Intel i7 2760QM processor, this procedure was not inferior to the *dgemm* procedure from the Intel MKL 11.0 library.

B. Problem 2

A design model of soil-structure interaction problem contains 2 989 476 equations (Fig. 10, 11) and consists of triangular, quadrilateral shell finite elements, as well as spatial frame and volumetric finite elements simulating the behavior of the ground.

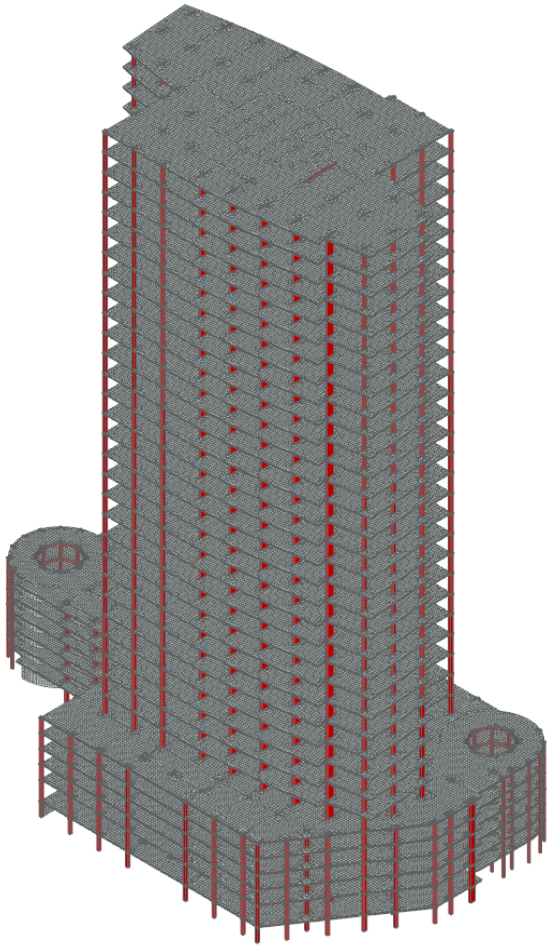

Fig. 9. Problem 1. Design model of multistorey building (2 546 400 equations).

This problem is very challenging for direct methods, because the soil prism, simulated by volumetric finite elements, generates a relatively dense part of a sparse matrix. Of all the methods available for reordering – the minimum degree algorithm MMD [4], the nested dissection method ND [3], the parallel section method [3] in conjunction with the MMD – the most efficient method for this task is METIS [13]. The number of nonzero elements in original matrix is 68 196 176 and in the lower triangular matrix – 4 966 055 936 (37 GB).

The duration of the numerical factorization phase and the performance obtained on the computer with AMD Opteron 6276 processor is shown in Table 3. The solution of this problem on the computer with Intel i7 2760QM processor and 8 GB of RAM was not effective due to the small amount of core memory. PARFES was run in the OOC1 mode, performing a large number of I/O operations. For this reason, performance analysis of the matrix multiplication procedure is not applicable.

The suggested microkernel procedure is slightly inferior to the *dgemm* procedure from the ACML 15.2.0 library on a single thread, but greatly outperforms it on 16 threads. In all cases, the proposed procedure is faster than the *dgemm* procedure from Intel MKL.

TABLE 1.
PERFORMANCE (MFLOPS) OF ALGORITHM **C = C – A·B** FOR MATRICES M × N × K = 2 000 000 × 120 × 120 (A – MATRIX M × K, B – K × N, C – M × N)

Procedure	AMD Opteron 6276 CPU 2.3/3.2 GHz		Intel i7 2760QM CPU 2.4/3.5 GHz	
	Single thread	16 threads	Single thread	4 threads
dgemm MKL 11.0	2 377	24 945	17 921	40 563
dgemm ACML 15.2.0	6 837	41 469 / 10 061	–	–
microkern_8x4_AVX	6 373	50 571	17 582	41 025

TABLE 2.
PROBLEM 1. DURATION (S) AND PERFORMANCE (MFLOPS) OF PARFES ON THE NUMERICAL FACTORIZATION STAGE (PROBLEM 1)

Procedure	AMD Opteron 6276 CPU 2.3/3.2 GHz				Intel i7 2760QM CPU 2.4/3.5 GHz			
	Single thread		16 threads		Single thread		4 threads	
	Duration, s	Perform.	Duration, s	Perform.	Duration, s	Perform.	Duration, s	Perform.
dgemm MKL 11.0	1 139	3 619	160	25 789	330	12 554	191	21 787
dgemm ACML 15.2.0	718	5 743	542	7 628	–	–	–	–
microkern_8x4_AVX	753	5 477	118	34 843	294	14 196	157	27 649

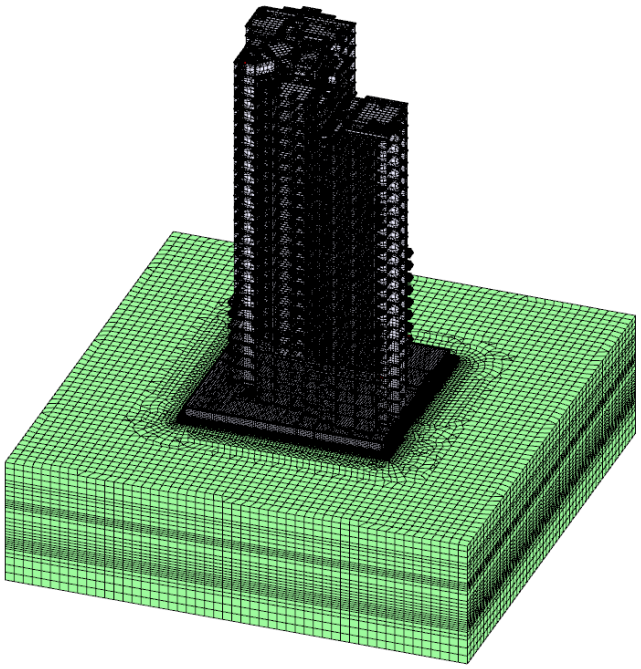

Fig. 10. Problem 2. Design model of soil-structure interaction problem (2,989,476 equations).

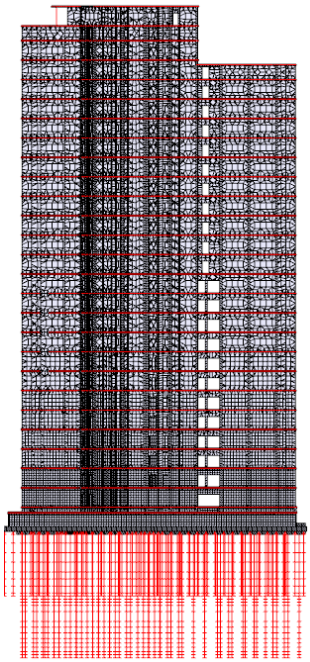

Fig. 11. The pile foundation (ground is hidden)

TABLE 3.
PROBLEM 2. DURATION (S) AND PERFORMANCE (MFLOPS) OF PARFES ON THE NUMERICAL FACTORIZATION STAGE. COMPUTER WITH AMD OPTERON 6276 PROCESSOR (PROBLEM 2)

Procedure	Single thread		16 threads	
	Duration, s	Performance, MFLOPS	Duration, s	Performance, MFLOPS
dgemm MKL 11.0	18 992	3 138	2 123	28 068
dgemm ACML 15.2.0	12 871	4 630	10 897	5 476
microkern_8x4_AVX	13 541	4 400	1 481	40 216

The speed up with the increase in the number of processors is depicted in Fig. 12.

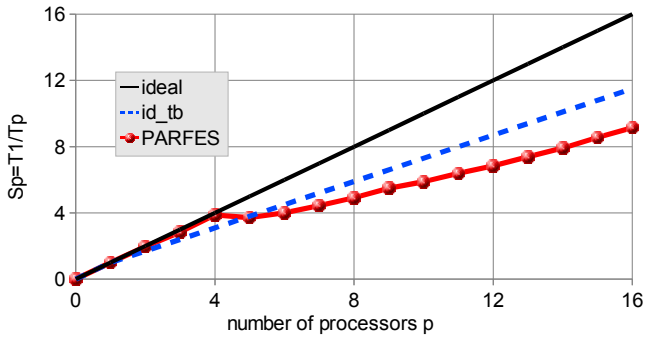

Fig. 12. Speed up with the increase in the number of threads. Ideal – the ideal speed up, id_tb – the ideal speed up on processors with Turbo Core support, PARFES – the real speed up.

The straight line of the "ideal" speed up passes through the points {0, 0}, {1, 1}, {2, 2}, … . This means that if the problem is solved using p threads, we would like to solve it p times faster than when using one thread. The id_tb curve approximates the ideal speed up for processors that support the Turbo Core mode – when a small number of cores is loaded, the processor increases the clock frequency, and when the number of loaded cores increases, reduces the frequency to the nominal value of 2.3 GHz. This curve is represented by a square parabola passing through the points {0, 0}, {1, 1}, {16, 11.5}. The ordinate of the last point was obtained as 16 × (minimum clock frequency of the processor) / (maximum clock frequency of the processor) = 16·3.2/2.3 = 11.5.

When using up to 4 threads, the speed up of PARFES is almost perfect. We explain the anomaly at p = 5 by the features of the Turbo Core control on this processor, because testing of PARFES on computers with different processors [7], [8] does not produce such behavior. The speed up of the method when p > 4 is stable up to p = 16, although lower than for the id_tb curve.

IV. Conclusion

Developing the microkernel procedure, based on AVX, in the parallel direct solver PARFES designed to solve problems of structural and solid mechanics that arise as a result of applying the finite element method, significantly accelerates matrix factorization on computers with AMD Opteron 6276 processor, Bulldozer architecture, while maintaining high performance and competitiveness with the Intel MKL *dgemm* procedure on computers with Intel processors.

References

[1] P. R Amestoy, I. S Duff, and J. Y L'Excellent, "Multifrontal parallel distributed symmetric and unsymmetric solvers," *Comput. Meth. Appl. Mech. Eng*., vol. 184, pp 501–520, 2000.

[2] ACML 15.2.0. URL: http://developer.amd.com/tools/cpu-development/amd-core-math-library-acml/ (accessed 17.11.2012).

[3] A. George and J. W. H. Liu, *Computer solution of sparse positive definite systems*. New Jersey : Prentice-Hall, Inc. Englewood Cliffs, 1981.

[4] A. George and J. W. H. Liu, "The Evolution of the Minimum Degree Ordering Algorithm," SIAM Rev, vol. 31, pp. 1–19, March, 1989.

[5] N. I. M Gould, Y. Hu and J. A. Scott, "A numerical evaluation of sparse direct solvers for the solution of large sparse, symmetric linear systems of equations," Technical report RAL-TR-2005-005, Rutherford Appleton Laboratory, 2005.

[6] K. Goto and R. A. Van De Geijn, "Anatomy of High-Performance Matrix Multiplication," *ACM Transactions on Mathematical Software*, vol. 34 (3), pp. 1–25, 2008.

[7] S. Fialko, "PARFES: A method for solving finite element linear equations on multi-core computers," *Advances in Engineering software,* vol. 40, 12, pp. 1256 – 1265, 2010.

[8] S. Fialko, "Parallel Finite Element Solver for Multi-Core Computers", *Federated Conference on Computer Science and Information Systems*, September 9–12, 2012, Wrocław, Poland. IEEE Xplore Digital Library, 978-83-60810-51-4, IEEE Catalog Number CFP1285N-USB, pp. 1 – 8. URL: http://proceedings.fedcsis.org/2012/pliks/101.pdf .

[9] S. Fialko, "The block substructure multifrontal method for solution of large finite element equation sets," *Technical Transactions*, 1-NP, issue 8, pp. 175 – 188, 2009.

[10] S. Fialko, *The direct methods for solution of the linear equation sets in modern FEM software*. Moscow: SCAD SOFT, 2009 (in Russian).

[11] Intel® Math Kernel Library Reference Manual. Document Number: 630813-029US. URL: http://www.intel.com/software/products/mkl/docs/WebHelp/mkl.htm.

[12] Intel MKL 11.0 release notes. URL: http://software.intel.com/en-us/articles/intel-mkl-110-release-notes/ (accessed 17.11.2012).

[13] G. Karypis and V. Kumar, "METIS: Unstructured Graph Partitioning and Sparse Matrix Ordering System,". Technical report, Department of Computer Science, University of Minnesota, Minneapolis, 1995.

[14] Optimize for Intel® AVX Using Intel® Math Kernel Library's Basic Linear Algebra Subprograms (BLAS) with DGEMM Routine. URL: http://software.intel.com/en-us/articles/optimize-for-intel-avx-using-intel-math-kernel-librarys-basic-linear-algebra-subprograms-blas-with-dgemm-routine/ (accessed 19.11.2011).

[15] D. Pardo, Myung Jin Nam, Carlos Torres-Verdín, Michael G. Hoversten and Iñaki Garay, "Simulation of marine controlled source electromagnetic measurements using a parallel Fourier hp-finite element method," *Comput. Geosci*., vol. 15, pp. 53–67, 2011.

[16] O. Schenk, K. Gartner, "Two-level dynamic scheduling in PARDISO: Improved scalability on shared memory multiprocessing systems," *Parallel Computing,* vol. 28, pp. 187–197, 2002.