Computer Assisted Methods in Engineering and Science
30(4): 427–459, 2023, doi: 10.24423/cames.578

Porting of Finite Element Integration Algorithm to Xeon Phi Coprocessor-based HPC Architectures

Filip KRUŻEL[1)*], Krzysztof BANAŚ[2)], Mauro IACONO[3)]

[1)] *Cracow University of Technology, Warszawska 24, 31-155 Kraków, Poland*

[2)] *AGH University of Science and Technology, al. Mickiewicza 30, 30-059 Kraków, Poland; e-mail: pobanas@cyf-kr.edu.pl*

[3)] *The University of Campania Luigi Vanvitelli, Viale Abramo Lincoln n. 5, 81100 Caserta, Italy; e-mail: mauro.iacono@unina2.it*

[*] *Corresponding Author e-mail: filip.kruzel@pk.edu.pl*

In the present article, we describe the implementation of the finite element numerical integration algorithm for the Xeon Phi coprocessor. The coprocessor was an extension of the many-core specialized unit for calculations, and its performance was comparable with the corresponding GPUs. Its main advantages were the built-in 512-bit vector registers and the ease of transferring existing codes from traditional x86 architectures. In the article, we move the code developed for a standard CPU to the coprocessor. We compare its performance with our OpenCL implementation of the numerical integration algorithm, previously developed for GPUs. The GPU code is tuned to fit into a coprocessor by our auto-tuning mechanism. Tests included two types of tasks to solve, using two types of approximation and two types of elements. The obtained timing results allow comparing the performance of highly optimized CPU and GPU codes with a Xeon Phi coprocessor performance. This article answers whether such massively parallel architectures perform better using the CPU or GPU programming method. Furthermore, we have compared the Xeon Phi architecture and the latest available Intel's i9 13900K CPU when writing this article. This comparison determines if the old Xeon Phi architecture remains competitive in today's computing landscape. Our findings provide valuable insights for selecting the most suitable hardware for numerical computations and the appropriate algorithmic design.

Keywords: CPU, optimization, parallelization, vectorization, Intel Xeon Phi.

Copyright © 2023 The Author(s).
Published by IPPT PAN. This work is licensed under the Creative Commons Attribution License CC BY 4.0 (https://creativecommons.org/licenses/by/4.0/).

1. Introduction

Today's trends in developing computing architectures focus on two main aspects. The first is connected with adding the computing cores, and the second concerns extending vector units in the individual core. In the development of

modern computing accelerators, the first architectures were based on GPUs, which allow processing computationally demanding fragments of calculations in a SIMD manner with thousands of threads operating simultaneously. The second type of accelerator combines relatively large amount of computing cores with the extensive registers. The CELL Broadband Engine developed by IBM can be chosen as an example of such an architecture. This architecture comprises one general processing core and eight smaller specialized cores (Synergistic Processor Elements) with wide 128-bit vector registers and AL units. These two solutions, developed by Nvidia and IBM, inspired Intel, their main competitor, to search for a solution in computing accelerators.

The wide vector registers characterising the Cell/BE architecture inspired Intel to create Larabee's graphics card architecture. At the same time, the company attempted to overcome the main barrier that hindered the greater spread of accelerator programming techniques, which was the complex model and programming method. The main advantages of the designed architecture were extensive (512-bit) vector registers, specialised texture units, coherent memory hierarchy and compatibility with x86 architecture [47]. At the same time, Intel was conducting Single Chip Computer and Teraflops Research Chip projects characterised by a vast multi-core structure. Based on these projects, the Intel MIC (Many Integrated Core) architecture was developed, which was used in the Intel Xeon Phi coprocessors codenamed Knights Corner (KNC) [19]. The MIC architecture was advertised as an architecture that combines GPU accelerators' power with the ease of programming that characterises CPUs.

The next generation of the MIC architecture – Knights Landing, was offered as a separate peripheral component interconnect-express (PCI-express) card and the standalone CPU unit. Intel Xeon Phi was part of Tianhe-2, the fastest supercomputer globally from 2013 to 2016, and allowed for achieving 33.86 petaFLOPS [48]. Even though Intel Xeon Phi architecture was officially discontinued [25], its evolution led to the Intel Xe graphics cards, announced in November 2019 [24]. The new architecture was declared to avoid repeating the Xeon Phi's mistakes and provide a unified programming model (oneAPI), and outstanding performance [13]. Although the architecture used in this study may seem outdated, it will give researchers essential tips on how the hardware and software evolution draws from its predecessors' mistakes.

In this paper, the authors try to port the numerical integration algorithm on Xeon Phi coprocessor and provide a detailed performance analysis. The results will be compared with the modern Intel i9-13900K CPU, characterized by 32 available threads and a five times higher core frequency than the tested coprocessors. This can lead to interesting conclusions. Our previous study includes the development of the algorithm for modern GPUs [4, 5, 34], CPUs [29], hybrid systems [33] as well as Intel Xeon Phi [2, 3, 32] and Intel Xe [35]. This article

sums up the obtained knowledge and shows the final version of the developed numerical integration algorithm.

2. Numerical integration

One of the most demanding engineering tasks connected with accelerators is the procedures of the finite element methods (FEMs). Numerical integration, used to prepare elementary stiffness matrices for a system solver, is one of the essential parts of FEM. Most studies about using the computing power of accelerators focused on using the GPUs to accelerate solving the final system of linear equations [7, 17, 18]. The solver procedure is often optimised first because it is the most time-consuming part of the FEM. However, after optimising the abovementioned procedure, the earlier calculation steps, such as, e.g., numerical integration and assembling, also significantly affect execution time [37].

Previous research using Xeon Phi in the FEM procedures includes solving a differential equation using numerical integration in total variation diminishing (TVD) methods [8], elastodynamic finite integration technique [46], use the shifted boundary method for solving the FEM problem [1] or solving partial differential equations using hybridised discontinuous Galerkin discretisation [40]. As in the case of GPU, all this work focused on the final solution to the FEM problem. Our approach is to provide a complex study of the numerical integration in the FEM on various architectures, including some that may be considered outdated, such as PowerXCell [30, 31], and Intel Xeon Phi. The authors find these architectures significant predecessors of today's hardware, and their internal architecture will still be present in today's and future technologies.

2.1. Finite element method

The FEM is a numerical technique used to solve partial differential equations (PDEs) in complex geometries, often in 3D, with given boundary conditions. The method involves dividing the computation area, denoted as Ω, into more minor elements with simple geometry, such as tetrahedrons, cubes, or prisms. The general form of the weak formulation for the problems analysed in this paper is as follows [6, 27].

Find the unknown function $\mathbf{u}$, belonging to a certain space of partially polynomial functions, for which equation:

$$\int_{\Omega}\left(\sum_i\sum_j \mathbf{C}^{i;j}\frac{\partial \mathbf{w}}{\partial \mathbf{x}_i}\frac{\partial \mathbf{u}}{\partial \mathbf{x}_j}+\sum_i \mathbf{C}^{i;0}\frac{\partial \mathbf{w}}{\partial \mathbf{x}_i}\mathbf{u}+\sum_i \mathbf{C}^{0;i}\mathbf{w}\frac{\partial \mathbf{u}}{\partial \mathbf{x}_i}+\mathbf{C}^{0;0}\mathbf{w}\mathbf{u}\right)\mathrm{d}\Omega+\mathrm{BCL} = \int_{\Omega}\left(\sum_i \mathbf{D}^i\frac{\partial \mathbf{w}}{\partial \mathbf{x}_i}+\mathbf{D}^0\mathbf{w}\right)\mathrm{d}\Omega+\mathrm{BCR}, \quad (1)$$

is satisfied for each test function $\mathbf{w}$ defined in the same (or slightly modified) function space.

In the above formula, $\mathbf{C}^{i;j}$ and $\mathbf{D}^{i}$, $i,j = 0,...,N_D$ denote the coefficients depending on the solved problem (N_D – the number of space dimensions), and BCR and BCL denote the right and left expressions, respectively, related to the boundary conditions and the boundary integrals $\partial\Omega$. In this work, the authors focus on calculating the integral over the Ω region since the boundary integrals are usually less computationally demanding. From the algorithmic point of view, they repeat a similar integration scheme. The part of the computation responsible for the boundary conditions in the authors' code is always performed by the CPU cores using standard FEM methods.

2.2. Numerical integration algorithm

Numerical integration in the FEM is correlated with the geometry used and an approximation type by the given elemental shape functions. Therefore, the suitable geometric transformation in mesh geometry used for computing should be applied. Denoting physical coordinates in the mesh as $\mathbf{x}$, a transformation from reference element with coordinates $\boldsymbol{\xi}$ is denoted as $\mathbf{x}(\boldsymbol{\xi})$. Usually, it is obtained through the general form of linear, multilinear, square, cubic, or other transformations of basic geometry functions and a set of degrees of freedom.

The use of the Jacobian matrix $\mathbf{J} = \frac{\partial \mathbf{x}}{\partial \boldsymbol{\xi}}$ is required to transform the coordinates from the reference to a real element, and the whole process is a distinctive part of numerical integration in the FEM. It significantly differentiates this algorithm from other integration and matrix multiplication algorithms.

With the application of the variable change to the reference element $\widehat{\Omega}$ for the selected integral example, we get the formula:

$$\int_{\Omega_e} \sum_i \sum_j \mathbf{C}^{i;j} \frac{\partial \boldsymbol{\phi}^r}{\partial \mathbf{x}_i} \frac{\partial \boldsymbol{\phi}^s}{\partial \mathbf{x}_j} \,\mathrm{d}\Omega = \int_{\widehat{\Omega}} \sum_i \sum_j \mathbf{C}^{i;j} \sum_k \frac{\partial \widehat{\boldsymbol{\phi}}^r}{\partial \boldsymbol{\xi}_k} \frac{\partial \boldsymbol{\xi}_k}{\partial \mathbf{x}_i} \sum_z \frac{\partial \widehat{\boldsymbol{\phi}}^s}{\partial \boldsymbol{\xi}_z} \frac{\partial \boldsymbol{\xi}_z}{\partial \mathbf{x}_j} \det \mathbf{J} \,\mathrm{d}\Omega, \quad (2)$$

where $\widehat{\boldsymbol{\phi}}^r$ denotes the shape functions for the reference element. This formula uses the determinant of the Jacobian matrix $\det \mathbf{J} = \det(\frac{\partial \mathbf{x}}{\partial \boldsymbol{\xi}})$ and components of the Jacobian inverse transformation matrix $\boldsymbol{\xi}(\mathbf{x})$, from real elements to reference elements.

The numerical quadrature transforms the analytical integral into a sum over integration points within the reference domain. From different possible quadra-

tures, we will concentrate on the most popular Gaussian quadratures [36]. Coordinates in the reference element are marked as $\boldsymbol{\xi}^Q$ and weights as $\mathbf{w}^Q$ where $Q = 1, ..., N_Q$ (N_Q – the number of Gauss points that depends on the type of element and the degree of approximation used). For integral (2), this leads to the formula:

$$\int\limits_{\Omega_e} \sum_i \sum_j \mathbf{C}^{i;j} \frac{\partial \boldsymbol{\phi}^r}{\partial \mathbf{x}_i} \frac{\partial \boldsymbol{\phi}^s}{\partial \mathbf{x}_j} \,\mathrm{d}\Omega \approx \sum_{Q=1}^{N_Q} \left(\sum_i \sum_j \mathbf{C}^{i;j} \frac{\partial \boldsymbol{\phi}^r}{\partial \mathbf{x}_i} \frac{\partial \boldsymbol{\phi}^s}{\partial \mathbf{x}_j} \det \mathbf{J} \right) \Bigg|_{\boldsymbol{\xi}^Q} \mathbf{w}^Q. \quad (3)$$

The final form of Eq. (3) may vary depending on the approximation and element types selected. For example, some values are constant for the entire element for geometrically linear elements like tetrahedrons.

To standardise and describe the algorithm from the point of view of mathematical computing for the most efficient implementation on the hardware, some modifications of the above formulas have been made with the introduction of the following indices:

- $\boldsymbol{\xi}^Q[i_Q]$, $\mathbf{w}^Q[i_Q]$ – tables with local coordinates of integration points (Gauss points) and weights assigned to them, $i_Q = 1, 2, ..., N_Q$, where N_Q – the number of Gauss points that depends on the geometry and the type and degree of approximation chosen,
- $\mathbf{G}^e$ – table with element geometry data (related to the transformation from reference element to real element),
- $\mathbf{vol}^Q[i_Q]$ – table with volumetric elements $\mathbf{vol}^Q[i_Q] = \det\left(\frac{\partial \mathbf{x}}{\partial \boldsymbol{\xi}}\right) \times \mathbf{w}^Q[i_Q]$,
- $\boldsymbol{\phi}[i_Q][i_S][i_D]$, $\boldsymbol{\phi}[i_Q][j_S][j_D]$ – tables with values of subsequent local shape functions, and their derivatives relative to a global $\left(\frac{\partial \phi^{i_S}}{\partial x_{i_D}}\right)$ and local coordinates $\left(\frac{\partial \widehat{\phi}^{i_S}}{\partial \xi_{i_D}}\right)$ at subsequent integration points i_Q,
  - $i_S, j_S = 1, 2, ..., N_S$, where N_S – the number of shape functions that depend on the geometry and the degree of approximation chosen,
  - $i_D, j_D = 0, 1, ..., N_D$, where N_D – dimension of space. For i_D, j_D different from zero, the tables refer to the derivatives relative to the coordinate at index i_D, and for $i_D = 0$ to the shape function, so $i_D, j_D = 0, 1, 2, 3$;
- $\mathbf{C}[i_Q][i_D][j_D][i_E][j_E]$ – table with values of the problem coefficients (material data, values of degrees of freedom in previous nonlinear iterations and time steps, etc.) at subsequent Gauss points, $i_E, j_E = 1, 2, ..., N_E$, where N_E – number of vector components in the solution – e.g. $N_E = 3$ for linear elasticity theory where the coefficients depend on a 3×3 stress tensor,
- $\mathbf{D}[i_Q][i_D][i_E]$ – table with d^i coefficient values in subsequent Gauss points,

- $\mathbf{A}^e[i_S][j_S][i_E][j_E]$ – an array storing the local, elementary stiffness matrix,
- $\mathbf{b}^e[i_S][i_E]$ – an array storing the local, elementary right-hand side vector.

With the use of the presented notation, general formula for the elemental stiffness matrix was created:

$$\mathbf{A}^e[i_S][j_S][i_E][j_E] = \sum_{i_Q}^{N_Q} \sum_{i_D,j_D}^{N_D} \mathbf{C}[i_Q][i_D][j_D][i_E][j_E] \times \boldsymbol{\phi}[i_Q][i_S][i_D] \times \boldsymbol{\phi}[i_Q][j_S][j_D] \times \mathbf{vol}^Q[i_Q]. \quad (4)$$

Analogically, right-hand side vector formula was created:

$$\mathbf{b}^e[i_S][i_E] = \sum_{i_Q}^{N_Q} \sum_{i_D}^{N_D} \mathbf{D}[i_Q][i_D][i_E] \times \boldsymbol{\phi}[i_Q][i_S][i_D] \times \mathbf{vol}^Q[i_Q]. \quad (5)$$

Through the introduced notation, we create a general numerical integration algorithm for finite elements of the same type and degree of approximation (Algorithm 1).

The algorithm's optimal structure must include the hardware's capabilities for which the algorithm should be developed. For external accelerators, such as Xeon Phi, the cost of sending data to and from the accelerator can be very high and should be hidden by a sufficiently large number of calculations. The designed form of Algorithm 1 in which the outer loop is a loop over all elements favours such a situation. Also, the general form of Algorithm 1, which does not consider the location of data at different levels of memory, allows us to treat each internal loop as independent and change its order for optimal performance. We can also achieve this because all necessary data can be calculated in advance and used when needed. This allows the creation of different algorithm variants depending on the hardware used.

3. Problems solved

The Poisson and convection–diffusion problems were chosen to test the algorithm's performance for low- and high-intensity tasks. The Poisson problem is ideal for fine-grained algorithm testing as its exact solution is known, and it requires relatively few resources. On the other hand, the resource-intensive convection–diffusion–reaction problem is ideal for testing coarse-grained implementations with fewer large elements to calculate. This allows in-depth hardware testing for tasks commonly found in FEM and scientific and technical calculations.

Algorithm 1: Generalised numerical integration algorithm for elements of the same type and degree of approximation.

```
1  - determine the algorithm parameters – N_EL, N_Q, N_E, N_S;
2  - load tables ξ^Q and w^Q with numerical integration data;
3  - load the values of all shape functions and their derivatives relative to local
     coordinates at all integration points in the reference element;
4  for e = 1 to N_EL do
5      - load problem coefficients common for all integration points (Array C^e);
6      - load the necessary data about the element geometry (Array G^e);
7      - initialize element stiffness matrix A^e and element right-hand side
         vector b^e;
8      for i_Q = 1 to N_Q do
9          - calculate the data needed for Jacobian transformations (∂x/∂ξ, ∂ξ/∂x, vol);
10         - calculate the derivatives of the shape function relative to global
             coordinates using the Jacobian matrix;
11         - calculate the coefficients C[i_Q] and D[i_Q] at the integration point;
12         for i_S = 1 to N_S do
13             for j_S = 1 to N_S do
14                 for i_D = 0 to N_D do
15                     for j_D = 0 to N_D do
16                         for i_E = 1 to N_E do
17                             for j_E = 1 to N_E do
18                                 A^e[i_S][j_S][i_E][j_E]+ = C[i_Q[[i_D][j_D][i_E][j_E]×
                                     ×φ[i_Q][i_S][i_D]×φ[i_Q][j_S][j_D]×vol;
19                             end
20                         end
21                         if i_S = j_S and i_D = j_D then
22                             for i_E = 0 to N_E do
23                                 b^e[i_S][i_E]+ = D[i_Q][i_D][i_E]×φ[i_Q][i_S][i_D]×vol;
24                             end
25                         end
26                     end
27                 end
28             end
29         end
30     end
31     - write the entire matrix A^e and vector b^e;
32 end
```

3.1. Poisson's equation

One of the tasks studied was a simple Poisson equation describing, e.g. stationary temperature distribution:

$$\nabla^2 u = f. \tag{6}$$

The result for this type of task is scalar. Therefore, the number of solution vector elements $N_E = 1$. For Poisson's task, convection–diffusion–reaction coefficients matrices $\mathbf{C}[i_Q]$ have the form (7) for all integration points:

$$\mathbf{C}[i_Q] = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix}. \tag{7}$$

This allows obtaining the individual words of the stiffness matrix by the simplified formula:

$$(A^e)^{rs} = \int_{\Omega_e} \sum_i \sum_j \frac{\partial \boldsymbol{\phi}^r}{\partial \mathbf{x}_i} \frac{\partial \boldsymbol{\phi}^s}{\partial \mathbf{x}_j} \, \mathrm{d}\Omega. \tag{8}$$

The $\mathbf{D}[i_Q]$ coefficients vector has the form:

$$\mathbf{D}[i_Q] = \begin{bmatrix} 0 & 0 & 0 & S_v \end{bmatrix}, \tag{9}$$

where S_v is a right-hand side (RHS) coefficient, different for each integration point.

3.2. Generalized convection–diffusion–reaction problem

Another of the tasks examined was the generalised convection–diffusion–reaction problem. To maximise the use of resources, it was assumed that the $\mathbf{C}[i_Q]$ matrix and the $\mathbf{D}[i_Q]$ vector coefficients will be filled (formula (10)). Arrays of this type appear, for example, in convective heat transfer problems, after applying SUPG stabilisation:

$$\mathbf{C}[i_Q] = \begin{bmatrix} A_{xx} & A_{xy} & A_{xz} & T_x \\ A_{yx} & A_{yy} & A_{yz} & T_y \\ A_{zx} & A_{zy} & A_{zz} & T_z \\ B_x & B_y & B_z & C_R \end{bmatrix}, \tag{10}$$

where A_{ij} – diffusion coefficients, B_i, T_j – convections coefficients, C_R – reaction coefficient.

The same situation appears in the RHS vector:

$$\mathbf{D}[i_Q] = \begin{bmatrix} Q_x & Q_y & Q_z & S_v \end{bmatrix}, \tag{11}$$

where Q_i – right side integration coefficient S_v derivatives with respect to x, y, z.

In the case of the tested convection–diffusion–reaction problem, the $\mathbf{C}[i_Q]$ and $\mathbf{D}[i_Q]$ coefficients are constant for the whole element. This allows generalising the solved problem by freeing it from the details of specific applications while maintaining the increased computational intensity of the algorithm.

4. Approximation

4.1. Finite element discretization and element types

The FEM discretises the considered continuous area into several elements. These are usually simple geometry elements such as tetrahedrons, cubes or prisms. In the cases considered in this work, prismatic and tetrahedral elements were used (Fig. 1).

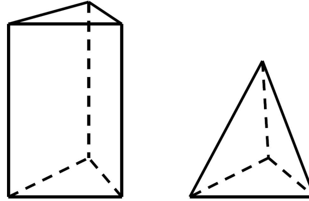

Fig. 1. Reference elements used.

Combining these elements makes it possible to reproduce even the most complex geometry of the computational area Ω [28]. To determine the quantities sought for the entire calculation area (and not just at the vertices of the elements), the correct shape functions for each element type should be determined depending on the type and degree of approximation selected.

4.2. Linear approximation

For a standard first-order linear approximation, the degrees of freedom of an element are related to its vertices. The basic shape functions for the prismatic reference element are in a form:

$$
\begin{aligned}
\Phi_0 &= \frac{-z+1}{2}(1-x-y),\\
\Phi_1 &= \frac{-z+1}{2}x,\\
\Phi_2 &= \frac{-z+1}{2}y,\\
\Phi_3 &= \frac{z+1}{2}(1-x-y),\\
\Phi_4 &= \frac{z+1}{2}x,\\
\Phi_5 &= \frac{z+1}{2}y,
\end{aligned}
\tag{12}
$$

where x, y and z are coordinates of the given point. They are the combination of 2D function that depends on x and y, with the function dependent on z. Similarly,

for the tetrahedron reference element, the form of the base shape functions can be represented as:

$$\begin{aligned} \Phi_0 &= 1 - x - y - z, \\ \Phi_1 &= x, \\ \Phi_2 &= y, \\ \Phi_3 &= z. \end{aligned} \tag{13}$$

The geometry of elements can be described similarly to their solution using shape functions. Tetrahedral elements have linear geometry that simplifies Jacobian transformations, whereas prismatic elements have more complicated polynomial geometry. Linear approximation is often used for less precise problems due to its simplicity and speed, while more resource-intensive problems can be tested using the discontinuous Galerkin approximation [10].

4.3. Discontinuous Galerkin approximation

The discontinuous Galerkin approximation was chosen as an example of the use of higher-order approximation and thus increased demand for resources. This approximation relates an element's degrees of freedom to its interior, and its accuracy depends on the number of degrees of freedom. The convergence of the discontinuous approximate solution to the exact solution is obtained by introducing additional terms in a weak formulation, including integrals on the sides of the elements inside the computational area, where the solution pitch between two elements is integrated [10–12]. In this paper's approach, these integrals are treated similarly to integrals related to boundary conditions in standard formulations because they repeat the same integration scheme and are irrelevant for testing the hardware. The integrals of the elements discussed in the paper are identical for the case of continuous and discrete discretisation.

For the tested prismatic reference element, the number of shape functions is given by the formula $\frac{1}{2}(p+1)^2(p+2)$. The number of Gauss points needed increases as the degree of approximation increases to maintain the convergence of the solution [9], and for a three-dimensional (3D) prism is of the order $O(p^3)$ as for the number of shape functions. The basic parameters of Algorithm 1 depending on the degree of approximation are presented in Table 1.

TABLE 1. The number of Gauss points and shape functions in relation to the degree of approximation.

Parameter	Degree of approximation p						
	1	2	3	4	5	6	7
N_Q	6	18	48	80	150	231	336
N_S	6	18	40	75	126	196	288

5. Numerical integration algorithms

The stiffness matrix components are scalar numbers for both Poisson and convection–diffusion–reaction cases, so the N_E loop in Algorithm 1 is skipped. The linear approximation uses two types of reference elements – prismatic and tetrahedral – and the convection–diffusion–reaction problem with discontinuous Galerkin approximation is tested for complex prismatic elements. Parallelising the algorithm depends on hardware resources and the type of element used. The algorithm can be modified by moving calculations for geometrically linear elements outside the integration point loops and changing the order of loops over integration points and shape functions. Six variants of Algorithm 1 were developed and named based on the element type and the order of the loops (QSS, SQS and SSQ *general* and *linear*). To optimise the algorithm for specific hardware, estimating the number of operations and memory accesses is necessary.

When estimating the number of operations in individual versions of the numerical integration algorithm, we can specify the following calculation stages:

1. Jacobian calculations (line 9 in Algorithm 1).
   - Calculation of derivatives of the basic shape functions for prismatic elements (Eq. (12)) – due to repeated calculations, we can assume that each compiler will optimise the whole process, reducing the number of operations to 14.
   - The matrix **J** and its inverse, determinant and volumetric element (**vol**) – 18 operations for each of the geometric degrees of freedom for prisms and nine operations for tetrahedrons when creating a Jacobian matrix + 43 operations related to its inverse and determinant for both element types.
2. Calculation of derivatives of shape functions relative to global coordinates (line 10 of Algorithm 1) – usually performed before the double loop over the shape functions to avoid calculation redundancy (although the version of the algorithm with derivatives calculated inside the loop over the i_S index can be considered) – 15 operations for each shape function.
3. For the convection–diffusion problem – calculating the product of the matrix $C[i_Q]$ coefficients with shape functions dependent on the first loop over the shape functions (i_S) – optimisation reducing the repetition of calculations by taking j_S expressions that are independent of the loop index – 28 operations for linear approximation and 22 operations for discontinuous Galerkin approximation.
4. Calculation of the right-hand side vector (line 24 of Algorithm 1) – 3 operations for Poisson's and 9 operations for the convection–diffusion problem.
5. Calculation of the stiffness matrix – 9 operations for convection–diffusion and 7 for Poisson's task.

The number of operations for each algorithm variant depends on both manual and automatic optimizations made by the compiler. Some compilers can further reduce operations by exploiting constant coefficients and symmetry of matrices. Unrolling loops may also reduce the number of operations, but it may be more difficult for prismatic elements due to separate calculations for each integration point. For linear approximation, coefficients in Table 2 should be used to calculate the final number of operations for each numerical integration stage.

TABLE 2. Multiplication factors for each of the stages and individual variants of the numerical integration algorithm with standard linear approximation.

Variant		Poisson		Convection–diffusion	
		tetra	prism	tetra	prism
QSS	1	1	N_Q	1	N_Q
	2	N_S	N_S	N_S	N_S
	3	0	0	$N_Q * N_S$	$N_Q * N_S$
	4	$N_Q * N_S$	$N_Q * N_S$	$N_Q * N_S$	$N_Q * N_S$
	5	$N_Q * {N_S}^2$	$N_Q * {N_S}^2$	$N_Q * {N_S}^2$	$N_Q * {N_S}^2$
SQS	1	1	$N_Q * N_S$	1	$N_Q * N_S$
	2	N_S	$N_Q * N_S$	N_S	$N_Q * N_S$
	3	0	0	$N_Q * N_S$	$N_Q * N_S$
	4	$N_Q * N_S$	$N_Q * N_S$	$N_Q * N_S$	$N_Q * N_S$
	5	$N_Q * {N_S}^2$	$N_Q * {N_S}^2$	$N_Q * {N_S}^2$	$N_Q * {N_S}^2$
SSQ	1	1	$N_Q * {N_S}^2$	1	$N_Q * {N_S}^2$
	2	N_S	$N_Q * {N_S}^2$	N_S	$N_Q * {N_S}^2$
	3	0	0	$N_Q * {N_S}^2$	$N_Q * {N_S}^2$
	4	$N_Q * {N_S}^2$	$N_Q * {N_S}^2$	$N_Q * {N_S}^2$	$N_Q * {N_S}^2$
	5	$N_Q * {N_S}^2$	$N_Q * {N_S}^2$	$N_Q * {N_S}^2$	$N_Q * {N_S}^2$

In the case of the standard linear approximation, N_Q and N_S are equal to 6 for prisms and 4 for tetrahedrons, respectively. A summary of the abovementioned estimates for a standard linear approximation is presented in Table 3.

As can be seen from in table, SQS and SSQ algorithms for prisms cause many redundant calculations, which can affect performance. However, it should be noted that in the case of accelerator programming, a large number of operations can be a significant advantage affecting the saturation of the hardware with the appropriate number of calculations and a favourable ratio of the number of calculations to the number of accesses to memory.

An algorithm with the QSS loops organisation and prismatic elements was used in the tasks with discontinuous Galerkin approximation. As mentioned earlier, load vector creation was omitted in this approximation type. Using the N_Q

TABLE 3. The number of operations for individual variants of numerical integration for standard linear approximation.

Variant		Poisson		Convection–diffusion	
		tetra	prism	tetra	prism
QSS	1	52	990	52	990
	2	60	540	60	540
	3	0	0	448	1008
	4	48	108	144	324
	5	448	1512	576	1944
	$\sum$	**608**	**3150**	**1280**	**4806**
SQS	1	52	5940	52	5940
	2	60	3240	60	3240
	3	0	0	448	1008
	4	48	108	144	324
	5	448	1512	576	1944
	$\sum$	**608**	**10 800**	**1280**	**12 456**
SSQ	1	52	35 640	52	35 640
	2	60	19 440	60	19 440
	3	0	0	1792	6048
	4	192	648	576	864
	5	448	1512	576	1944
	$\sum$	**752**	**57 240**	**3056**	**63 936**

and N_S parameters presented in Table 1 and the coefficients from the corresponding columns of Table 2, the number of operations was calculated for each degree of approximation. The result is presented in Table 4.

TABLE 4. Number of operations for convection–diffusion task with discontinuous Galerkin approximation.

Stage	Degree of approximation p						
	1	2	3	4	5	6	7
1	990	2970	7920	13 200	24 750	38 115	55 440
2	540	4860	28 800	90 000	283 500	679 140	1 451 520
3	792	7128	42 240	132 000	415 800	996 072	2 128 896
5	1944	52 488	691 200	4 050 000	21 432 600	79 866 864	250 822 656
$\sum$	**4266**	**67 446**	**770 160**	**4 285 200**	**22 156 650**	**81 580 191**	**254 458 512**

Another aspect that requires analysis is the number of accesses to RAM and the size of data needed to store current calculations. For linear approximation, the size of the data needed is shown in Table 5.

TABLE 5. Basic parameters of the numerical integration algorithm, along with memory requirements for different versions of the procedure

Basic parameters	Poisson		Convection–diffusion	
	tetra	prism	tetra	prism
Integration data (reference element)	0 (constant)	$3 * N_Q = 18$	0 (constant)	$3 * N_Q = 18$
Each finite element I/O data				
Geometrical data (input)	12	18	12	18
Problem data (input)	$1 * N_Q = 4$	$1 * N_Q = 6$	20	20
Stiffness matrix $\mathbf{A}^e$ (output)	$N_S * N_S = 16$	$N_S * N_S = 36$	$N_S * N_S = 16$	$N_S * N_S = 36$
Load vector $\mathbf{b}^e$ (output)	$N_S = 4$	$N_S = 6$	$N_S = 4$	$N_S = 6$
For each integration point – QSS version				
Problem coefficients **C** and **D**	1	1	20	20
Shape functions and their derivatives $\boldsymbol{\phi}$	$4 * N_S = 16$	$4 * N_S = 24$	$4 * N_S = 16$	$4 * N_S = 24$
Sum (including $\mathbf{A}^e$ and $\mathbf{b}^e$)	**37**	**67**	**56**	**86**
For all integration points – SQS version				
Problem coefficients **C** and **D**	$1 * N_Q = 4$	$1 * N_Q = 6$	20	20
Shape functions and their derivatives $\boldsymbol{\phi}$	$3 * N_S + 1 * N_Q * N_S = 28$	$4 * N_S * N_Q = 144$	$3 * N_S + 1 * N_Q * N_S = 28$	$4 * N_S * N_Q = 144$
Sum (including $\mathbf{A}^e$ and $\mathbf{b}^e$)	**52**	**192**	**68**	**206**
For all integration points – SSQ version				
Problem coefficients **C** i **D**	$1 * N_Q = 4$	$1 * N_Q = 6$	20	20
Shape functions and their derivatives $\boldsymbol{\phi}$	4	4	8	8
Sum (including one element from $\mathbf{A}^e$ and $\mathbf{b}^e$)	**10**	**12**	**30**	**30**

As can be seen, the number of values used ranges from several to several hundred numbers. This can be a crucial criterion for choosing a suitable algorithm for architecture. The SSQ version has the lowest resource requirements. However, it should be considered that, e.g. for prismatic elements, the number of operations needed increases rapidly due to repeated Jacobian calculations. On the other hand, the QSS version, in which we avoid repeated calculations, has very high resource requirements. The SQS version, considered a version between the previous ones, is characterised by the fact that it requires storing only one row of the matrix and repeating the Jacobian calculations only for it. By multiplying the number of data needed from Table 5 by the number of repetitions of external loops and adding the number of integration and geometric data needed, we obtain the estimated number of memory accesses for each version of the algorithm. At the same time, it should be noted that access to the stiffness matrix and the right-hand side vector for each algorithm takes place twice – during reading and writing. The calculations of the memory accesses are presented in Table 6.

Table 6. Number of memory accesses for different versions of the numerical integration algorithm.

Variant	Poisson		Convection–diffusion	
	tetra	prism	tetra	prism
QSS	240	690	256	704
SQS	288	1410	304	1424
SSQ	336	1050	416	1208

Analogically, the analysis was performed for the discontinuous Galerkin approximation, using data in Table 5 for prisms and the QSS algorithm, which was then expanded to include a higher approximation degree. In this case, the memory accesses for the right-hand side vector were omitted, and the number of problem coefficients in the convection–diffusion task was reduced to 16. Also, it should be noted that the weights associated with Gauss points are not constant for this type of approximation. Thus, the number of integration data needed increases to $4 * N_Q$. Table 7 presents the updated number of accesses.

As can be seen in the table, the number of memory accesses increases significantly with the degree of approximation. To illustrate the number of accesses to the number of operations ratio for the tested algorithms, their arithmetic intensity was calculated. It is defined as the number of operations to the number of memory accesses ratio. The results of calculations for the task with linear approximation are presented in Table 8.

Similar calculations for the discontinuous Galerkin approximation are presented in Table 9.

Table 7. Number of memory accesses for the numerical integration in convection–diffusion and discontinuous Galerkin approximation.

Parameter	Degree of approximation p						
	1	2	3	4	5	6	7
Integration data	24	72	192	320	600	924	1344
Problem coefficients $\mathbf{C}$	16	16	16	16	16	16	16
Geometrical data	18	18	18	18	18	18	18
$\boldsymbol{\phi}$	144	1296	7680	24 000	75 600	181 104	387 072
$\mathbf{A}^e$ matrix	432	11 664	153 600	90 000	4 762 800	17 748 192	55 738 368
$\sum$	**634**	**13 066**	**161 506**	**924 354**	**4 839 034**	**17 930 254**	**56 126 818**

Table 8. Arithmetic intensity of the algorithm with standard linear approximation.

Variant	Poisson		Convection–diffusion	
	tetra	prism	tetra	prism
QSS	2.53	4.57	5.00	6.83
SQS	2.11	7.66	4.21	8.75
SSQ	2.24	54.51	7.35	52.93

Table 9. Arithmetic intensity of the algorithm with discontinuous Galerkin approximation, convection–diffusion problem with prismatic elements.

Degree of approximation p						
1	2	3	4	5	6	7
6.73	5.16	4.77	4.64	4.58	4.55	4.53

As can be seen in the tables, besides the SSQ algorithm for prismatic elements, the numerical integration algorithm is characterised by a low arithmetic intensity, indicating the significance of the problem of adequate memory bandwidth and need for the optimal data organisation.

When designing the algorithm for individual computing machines, it is necessary to consider available resources and memory organization when designing algorithms for computing machines. Vector memory access can reduce downloads, but increasing data can be problematic for accelerators with separate memory systems. Memory access is critical when developing computing accelerator architectures to increase available resources, including memory sizes of individual types.

6. Intel Xeon Phi

Intel Xeon Phi coprocessors are expansion cards connected to the host system using the PCI-Express interface. They have up to 16 memory channels and

a separate system management controller, which, in addition to managing data transmission, also controls the operation of the fan (optional) and monitors the card's temperature. These accelerators have a proprietary Linux-based operating system loaded into internal flash memory. Depending on the version, 57–61 cores are available to programmers with multi-thread hardware support (4 threads per core).

In this study, we utilised two coprocessors, 5110P and 7120P, which vary in their core count and RAM capacity. Table 10 provides an overview of the characteristics of the coprocessors we selected. It is worth noting that these coprocessors come with an operating system that reserves one core and a portion of the memory. As a result, programmers have access to 236 and 240 threads and 5.6 GB and 12 GB of RAM, respectively [44].

Table 10. Tested Intel Xeon Phi coprocessors [23].

Parameter	5110P	7120P
Frequency	1.05 GHz	1.24 GHz
Cores	60	61
RAM	8 GB	16 GB
Threads	240	244

The architecture of KNC cores is based mainly on the Pentium 4 microarchitecture but is expanded with 512-bit vector registers. The compatibility with x86 architecture should allow for easy transfer of existing codes to improve performance. Each core has hardware support for four threads and two processing pipelines. The core has 32 kB of L1 cache for data and the same amount of cache memory for code. Additionally, it has a 512 kB second-level cache (L2). The main feature of Intel Xeon Phi accelerator cores is a specialised vector processing unit. Each VPU can process eight double or sixteen single-precision numbers and has a built-in extended math unit to support transcendent operations, similar to GPUs. The manufacturer has named the extended instruction set for the VPU unit IMCI. Unfortunately, it is incompatible with the AVX instructions, making transferring previously compiled vectorised codes impossible.

Each of the cores is connected by a high-speed interface, which, together with the coherent cache memory, causes almost instant data exchange between them. The coherence of cache memory is made possible by the tag directory for each core directly connected to L2 memory. GDDR memory is located between the cores, allowing more even distribution of work and promoting efficient data division between the cores.

Both coprocessors used in this study allow for calculations in several ways. One is the native mode, for which the whole program is run on the accelerator.

The second is offload mode, where the coprocessor calculates the code fragments specified by the programmer. A similar mode is used with the OpenCL framework, where some code is run on the coprocessor as a separate procedure (the so-called kernel).

Intel Xeon Phi coprocessors were thought to be the next step for creating a high-performance architecture to support scientific and technical calculations. Some of the mechanisms first presented in them are used in the modern processors. The authors found it reasonable to check the usability of this architecture with the numerical integration algorithm and, thus, its general suitability for FEM calculations.

7. Development tools

7.1. Finite Element Method computational framework

As the primary tool used in research, a modular programming framework for engineering calculations using the FEM – ModFEM was used [39]. Its modular structure allows for the modification of individual FEM calculation fragments. The framework includes the problem, mesh, approximation, and solver modules. Other modules are used for work division on different hardware types. ModFEM is suitable for distributed computing, including on accelerators and individual computers, clusters, and supercomputers. During the research, an extension of the approximation module was developed for accelerator support, and the problem modules were modified for testing the numerical integration algorithm with OpenMP optimizations. The framework was also supplemented with an OpenCL and CUDA module for testing GPUs.

7.2. Programming languages and extensions for multi-threaded calculations

The programming language used to create the code was the C language. This is one of the most efficient high-level languages, allowing for relatively low-level control over the hardware and how it interprets the instructions. The C language is also a model on which the CUDA and OpenCL programming languages are based.

The primary compiler used by the authors was the Intel C Compiler [22], which is a part of the Intel Parallel Studio XE package. For programming both general-purpose processors as well as Xeon Phi accelerators, the authors used the following libraries and language extensions:

- **OpenMP** – application programming interface that allows using threads and the shared memory model. It allows for dividing work between indivi-

dual threads depending on the needs. It has built-in directives to facilitate the division of work, so it is possible to balance the load depending on the available resources [43];

- **Intel Cilk Plus** – extensions of the C language, which introduce the possibility for parallelisation in a manner analogous to OpenMP. Additionally, it introduces Intel Cilk Plus Array Notation, which facilitates the handling of tables for the programmer and automatic vectorisation for the compiler (Fig. 2). Thanks to language extensions, Intel Cilk Plus allows to specify fragments for vectorisation directly and align tables to match vector registers.

```
for(i=0; i<STR; i++)
        temp[i] = shpx[i] * jac_0 + shpy[i];
```

```
temp[0:STR] = shpx[0:STR] * jac_0 + shpy[0:STR];
```

Fig. 2. Cilk Plus Array Notation (right) in comparison to the standard code (line 10 in Algorithm 1).

7.3. Accelerator programming

7.3.1. OpenCL. To compare the native algorithm implementation using OpenMP and specialised language extensions with the previously developed implementation for GPU accelerators [4, 5], the OpenCL language was used. OpenCL was developed in 2008 by the Khronos Group Corporation, established as a consortium of leading graphics processing hardware and software manufacturers, such as Apple, Intel, SGI, Nvidia, and ATI. Its internal structure is based on the same pattern as Nvidia CUDA programming extensions. However, its standard is open, allowing hardware developers to prepare their versions for a given architecture. OpenCL allows programming virtually any multi-core and vector machines – from modern CPUs to GPUs, through hybrid PowerXCell units, APUs and Intel Xeon Phi accelerators. The OpenCL specification contains a programming language based on the C99 standard, used to program accelerators, and an application programming interface (API) to support the platform (understood as a given combination of available computing hardware and system software) and run tasks on processors [20]. Each of the devices that can be used is, in the OpenCL model, referred to as *Compute Device*, and each of its cores/multiprocessors as *Compute Unit*. OpenCL defines individual threads as work-items that are grouped into work-groups. All work-groups are started on the device as an N-dimensional calculation range (NDRange). The optimal size of a work-group depends on the architecture. It is usually associated with its internal structure, i.e. the number of streaming processors in the multiprocessor or the width of vector registers. Just like in the case of the execution model, the memory model is based on the most common structure of the graphic card

memory. Work-items working within one work-group can communicate through local memory. Additionally, the global memory of the device and constant memory buffers are available. The appropriate number of registers, called private memory, is available for each work-item depending on the physical resources of the *Compute Unit* and the defined division of work between work-groups. Due to the portability of OpenCL code between devices of different types, each memory area can be physically mapped differently depending on the available hardware resources. This causes difficulties in correctly mapping the hardware when the architecture differs significantly from a standard GPU architecture. Another problem is that the actual mapping of the code to the hardware is hidden from the programmer, depriving him of total control over the program's execution. OpenCL includes procedures to maintain code portability between different hardware platforms by tailoring memory and execution models to a given architecture [45]. Direct implementation on a given machine depends on the provider of the OpenCL programming platform and the appropriate drivers.

Thanks to OpenCL technology, writing programs on many accelerator types has become possible. However, this did not solve the performance portability problem, as each architecture requires separate optimisations [3]. For our work, we have created an automatic tuning system for the numerical integration algorithm [5].

7.4. Execution analysis

The authors' built-in functions and tools provided by software and hardware suppliers were used for the code analysis.

A system for counting the number of occurrences of specific instructions has been developed to analyse the machine code. It determines what optimisations were used during code compilation and generates basic reports on memory access and arithmetic operations.

To analyse the execution flow on Intel general-purpose processors and the Intel Xeon Phi coprocessor, the Intel Vtune Amplifier profiler available in the Intel Parallel Studio XE package was used. This tool allows for deep code analysis, previewing multithreading and hardware counters. This allows identifying various problems related to program performance and more accurately using available hardware resources.

8. Obtained results

8.1. Methodology

The authors utilized their expertise in programming CPUs and GPUs to test Xeon Phi coprocessors. Although advertised as a tool for effortless code acceleration using OpenMP and semi-automatic vectorization, achieving high perfor-

mance on the tested coprocessor requires additional optimizations, as demonstrated in studies by other authors [16, 42, 49, 50]. To optimize the algorithm on the tested coprocessor, the authors ran optimized versions of the algorithm developed for Intel Xeon processors, namely *Cilk* and *Stride* algorithms, described in detail in [29]. For the *Cilk* optimization, the authors aligned data tables to 64 bytes to optimize memory access, extended tables to the size divisible by the size of a 512-bit register in the case of prismatic elements, and implemented the algorithm using the Intel Cilk Plus array notation for easier vectorization. The code areas to be vectorized were marked by *#pragma vector* and *#pragma simd*. For the *Stride* optimization, the whole algorithm was reorganized to force vectorization at the same level as parallelization, processing four (256-bit) or eight (512-bit) elements at once during one loop, reducing the final loop size for parallelization but increasing the number of local tables for temporary variables and the complexity of input and output array indexation. Performance was also affected by setting environmental variables KMP_AFFINITY = "granularity = fine, compact" and MIC_USE_2MB_BUFFERS = 2M, and using the *offload* model to send and retrieve data to and from the coprocessor. For the OpenCL version, the authors set the work-group size to a multiple of 16, as recommended in [21], to allow for automatic vectorization and reduce processing time significantly. OpenCL memory levels are mapped to coprocessor memory, and though the number of registers is small compared to GPU architectures, the available L1 and L2 cache memory has increased significantly. Separate studies were performed for tasks with standard linear approximation and discontinuous Galerkin approximation.

8.2. Standard linear approximation

8.2.1. Performance model. To develop the performance model, the performance characteristics of the tested coprocessors presented in Table 11 were used. Benchmark values were obtained from Linpack [15] and Stream [38] benchmarks.

TABLE 11. Floating points processing (left) and memory performance (right) of the tested Xeon Phi coprocessors [16, 26].

Xeon Phi version	Performance	[GFlops]	[GB/s]
5110P	Theoretical	1011	320
	Benchmark	769	165
7120P	Theoretical	1208	352
	Benchmark	999	181

For performance analysis, the Intel Vtune profiler was used. For the number of operations obtained from the profiler, the event VPU_ELEMENTS_ACTIVE

was used, following the instructions from [41]. Table 12 shows the estimated and obtained number of operations.

Table 12. Number of operations for tested Intel Xeon Phi coprocessors.

Variant		Poisson		Convection–diffusion	
		tetra	prism	tetra	prism
Estimated		608	3150	1280	4806
5110P	Cilk	595	2157	1700	1838
	Stride = 4	935	2451	1526	1417
	Stride = 8	170	2142	1156	1154
7120P	Cilk	1020	2451	2464	1838
	Stride = 4	1954	6740	4334	10 417
	Stride = 8	230	793	510	1225

Similarly, to calculate the number of memory accesses, the DATA_READ_OR_WRITE event was used, which specifies the number of accesses to the floating-point number (float or double) in the memory [41]. The obtained values are presented in Table 13.

Table 13. Number of memory accesses for tested Intel Xeon Phi coprocessors.

Variant		Poisson		Convection–diffusion	
		tetra	prism	tetra	prism
Estimated		240	690	256	704
5110P	Cilk	561	2145	756	2267
	Stride = 4	569	1961	493	1961
	Stride = 8	467	1593	544	1900
7120P	Cilk	680	2880	952	2574
	Stride = 4	892	3309	1232	3789
	Stride = 8	603	2696	680	3002

Based on the results obtained, it can be seen that the number of memory accesses is comparable to the number of calculations, which means that the arithmetic intensity of the algorithm for all its versions does not exceed 4. To facilitate data reading, *Roofline* charts were created for each of the tested accelerators (Figs. 3 and 4) [51]. The roofline graph defines the limiting factor of the algorithm's performance based on both theoretical and achieved in benchmarks performance. At the same time, according to this model, thanks to the calculated arithmetic intensity of the tested algorithms, it is easy to determine whether the factor limiting performance is memory bandwidth or the speed of

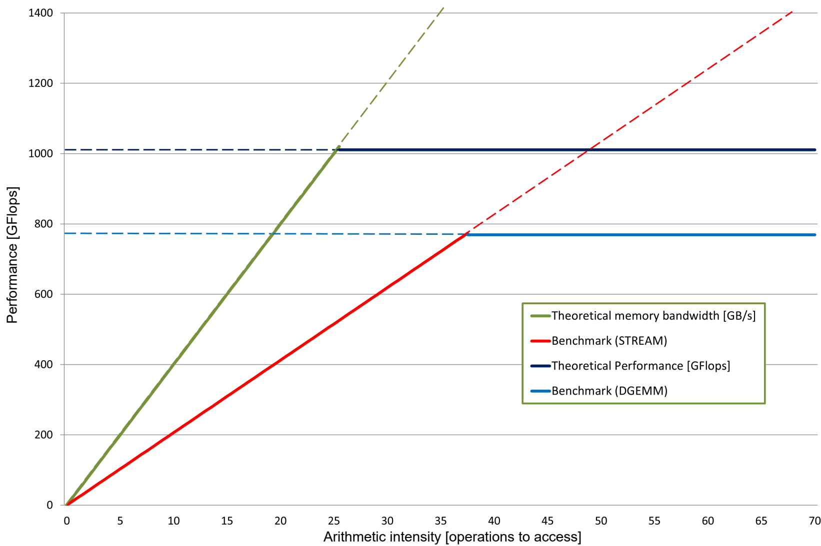

Fig. 3. Roofline graph for the Xeon Phi 5110P accelerator.

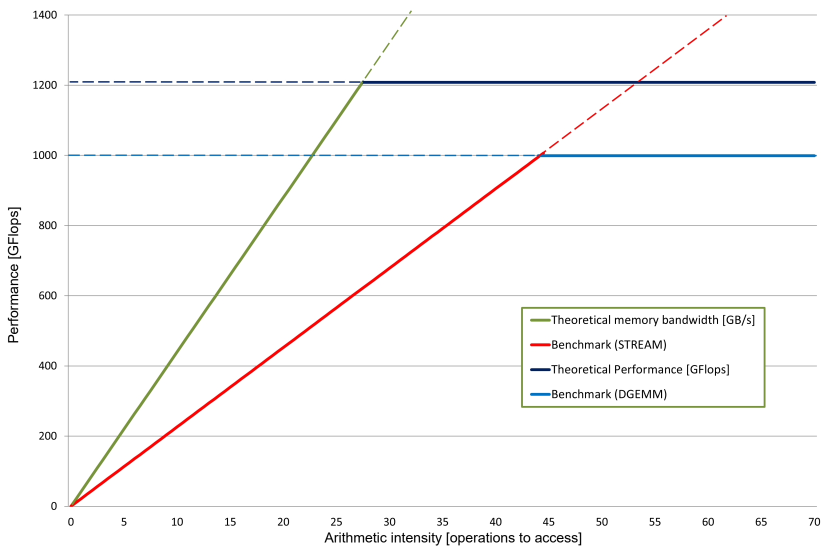

Fig. 4. Roofline graph for the Xeon Phi 7120P accelerator.

performing floating-point operations. From the created *roofline* charts, we can conclude that the algorithm is limited by memory speed for all tasks with linear approximation.

Based on the data in Table 11 (data from benchmarks) and data about the memory access obtained from the profiler, the expected algorithm execution time was calculated (Table 14).

TABLE 14. Expected execution time (in ns) of the numerical integration algorithm for each version of the algorithm.

Variant		Poisson		Convection–diffusion	
		tetra	prism	tetra	prism
5110P	Cilk	3.40	13.00	4.58	13.74
	Stride = 4	3.45	11.88	2.99	11.88
	Stride = 8	2.83	9.66	3.30	11.51
7120P	Cilk	3.76	15.91	5.26	14.22
	Stride = 4	4.93	18.28	6.81	20.94
	Stride = 8	3.33	14.90	3.76	16.59

8.2.2. Results. The results obtained for the version using OpenMP and the offload model are presented in Table 15. As can be seen in the table, the best execution times were obtained for the 5110P coprocessor and the *Stride* algorithm with eight elements processed at once. Interestingly, the theoretically faster 7120P coprocessor obtained worse results in 67% of cases, which is consistent with the results from the profiler, indicating higher memory use for this coprocessor. In this case, more chaotic results were observed, which, contrary to expectations, do not point to the *Stride* = 8 algorithm as the best for this coprocessor. As can be seen, the results obtained are low compared to the expected and amount to barely 12% of the theoretical performance.

TABLE 15. Obtained results (in ns) for the tested versions of the numerical integration algorithm using OpenMP and the offload model.

Variant		Poisson		Convection–diffusion	
		tetra	prism	tetra	prism
5110P	Cilk	58.61	212.53	68.50	221.00
	Stride = 4	56.47	166.29	61.50	170.29
	Stride = 8	**51.80**	**145.31**	**46.60**	**138.66**
7120P	Cilk	**46.00**	312.73	67.68	**170.61**
	Stride = 4	57.24	**209.10**	**59.07**	175.35
	Stride = 8	60.75	292.38	66.66	243.96

To see if the tested accelerators can be competitive against the new processor architectures, we have tested developed algorithms on the latest architecture available, the Intel i9-13900K. This processor has 32 available threads and a high-speed memory bandwidth of 89.6 GB/s. Its benchmarked performance is 1.7 TFlops, making it one of the most powerful CPUs [14]. To test the algorithms previously prepared for the Xeon Phi, we have to change the alignment of data to 32. The results obtained for the tested CPU show that it is average, 9.4 times

faster than Xeon Phi 5110P and 11.3 times faster than 7120P (Table 16). This indicates that this architecture seems outdated, but the OpenCL parameter tuning algorithm developed earlier was also launched on the tested coprocessors for comparative purposes. It tests the combination of various storage options in different types of memory and other parameters that may affect the algorithm's performance.

TABLE 16. Obtained results (in *ns*) for the tested versions of the numerical integration algorithm for the Intel Core i9 CPU.

Variant		Poisson		Convection–diffusion	
		tetra	prism	tetra	prism
i9-13900K	Cilk	5.64	11.06	7.23	23.51
	Stride = 4	5.65	21.36	7.75	16.81
	Stride = 8	5.92	26.14	7.45	17.49

The best results for the tested OpenCL versions are summarised in Table 17. As can be seen in the table, the best results were obtained for the SQS algorithm for tetrahedron elements and QSS for prismatic ones. As much as half of the Xeon Phi 7120P coprocessor options assume not using shared memory, which, with a limited number of registers for this architecture, may cause chaotic and worse results than the theoretically slower Xeon Phi 5110P.

TABLE 17. Summary of the best results in *ns* with automatic parameter tuning system for Xeon Phi coprocessors.

Variant		Poisson		Convection–diffusion	
		tetra	prism	tetra	prism
5110P	QSS	17.75	**44.69**	27.16	**46.61**
	SQS	**14.40**	62.86	**24.96**	81.55
	SSQ	17.59	129.84	25.79	175.49
7120P	QSS	**22.99**	**49.87**	32.66	**54.37**
	SQS	23.09	73.17	**31.47**	76.78
	SSQ	24.96	109.42	31.53	171.48

The results obtained in the OpenCL model are better than those obtained using OpenMP and the offload model. This may indicate better use of OpenCL's vectorisation and vector memory accesses than the classic programming method. A comparison of the obtained results is shown in Fig. 5.

Furthermore, when comparing the results achieved by the OpenCL implementation of the code with the optimal results obtained for the Intel i9 CPU, we can discern only slight disparities – with each coprocessor delivering 3.2 and

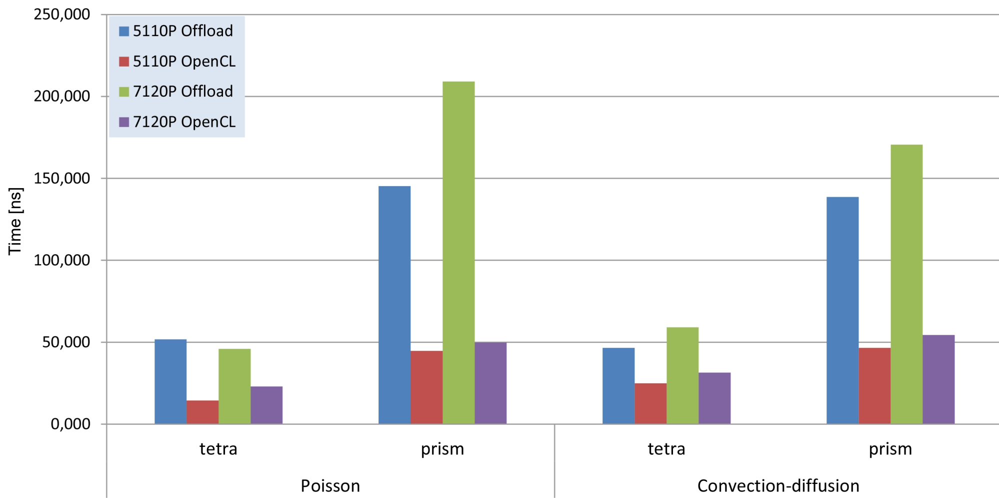

Fig. 5. Comparison of results between the Offload and the OpenCL versions of the numerical integration algorithm on Intel Xeon Phi coprocessors.

3.5 times worse performance, respectively. This indicates that despite nearly a decade of development gap, the tested coprocessors are better than expected.

The results obtained and the tests show that the tested coprocessors are very similar to GPU accelerators. In the case of numerical integration, better results are obtained for the GPU than for the CPU-dedicated code.

8.2.3. Conclusions. The investigated Intel Xeon Phi coprocessors were constructions with interesting architecture and various programming methods. However, based on the old Pentium 4 architecture extended by broad vector registers, their architecture alone was insufficient to achieve high performance for the numerical integration algorithm with linear approximation. To check whether the capabilities of the tested coprocessors will be able to use the more advanced version of the numerical integration algorithm, tests were performed for higher degrees of discontinuous Galerkin approximation and the generalised convection–diffusion–reaction task for prismatic elements.

8.3. Discontinuous Galerkin approximation

8.3.1. Performance model. In the case of testing the numerical integration with the discontinuous Galerkin approximation, an analogous analysis was performed as in the case of linear approximation. The estimated number of accesses and operations were used when developing the performance model. We were trying to obtain these values from the profiler. Because the Intel Vtune profiler caused a memory violation error for the *Stride* algorithm versions with band-

width 4 and 8, results were obtained only for the *Cilk* algorithm version. Due to the profiler version change for the Intel Xeon Phi 7120P coprocessor, the necessary data could not be obtained. Table 18 shows the obtained number of operations.

TABLE 18. Number of operations for the numerical integration algorithm with discontinuous Galerkin approximation for Intel Xeon Phi coprocessors.

Version	Degree of approximation		
	3	4	5
Estimated	770 160	4 285 200	22 156 650
Profiler	392 252	5 760 291	17 440 678

As can be seen, the number of operations obtained from the profiler is very close to estimates – only for a lower degree of approximation, it has been significantly reduced.

The number of memory accesses for the *Cilk* algorithm and the Xeon Phi 5110P coprocessor is shown in Table 19.

TABLE 19. Number of memory accesses for the *Cilk* algorithm and the Xeon Phi 5110P coprocessor.

Version	Degree of approximation		
	3	4	5
Estimated	161 506	924 354	4 839 034
Profiler	155 448	1 059 806	3 976 271

As in the previous case, we can see significant compliance of the results obtained with theoretical estimates. By dividing the received number of operations by the number of accesses, we can notice that, analogously to linear approximation, the algorithm's arithmetic intensity is low and does not exceed 6, which means that it can be treated as limited by memory performance.

Using the data obtained from the profiler and Table 11, the theoretical execution time for the Cilk algorithm was estimated (Table 20). For comparative purposes, the Xeon Phi 7120P accelerator results were presented assuming the same number of accesses.

TABLE 20. Estimated execution time for the *Cilk* algorithm with discontinuous Galerkin approximation.

Xeon Phi version	Degree of approximation		
	3	4	5
5110P	0.94	6.42	24.10
7120P	0.86	5.86	21.97

8.3.2. Results. The results obtained for the tested versions of the algorithm are presented in Table 21.

Table 21. Obtained results (in μs) for the tested coprocessors and numerical integration algorithm with discontinuous Galerkin approximation.

Variant		Degree of approximation		
		3	4	5
5110P	Cilk	3.26	22.32	223.62
	Stride = 4	20.11	140.80	1017.35
	Stride = 8	16.92	124.39	967.61
7120P	Cilk	2.61	21.15	203.63
	Stride = 4	15.70	111.08	792.89
	Stride = 8	15.31	97.03	868.14

As it can be noticed, a *Cilk* algorithm turned out to be the best for discontinuous Galerkin approximation, obtaining about 30% of the theoretical performance for lower approximation levels 3 and 4. The *Stride* algorithm can achieve high performance only for small tasks due to the limited number of wide vector registers (512-bit). Comparing the situation with standard linear approximation, we can see a significant increase in the degree of coprocessor utilisation.

As earlier, we have run our algorithms on the Intel I9-13900K CPU. The results of the execution are shown in Table 22. These results support our earlier observation that the new CPU is approximately three times faster than the tested coprocessors. The difference in performance is particularly significant for higher degrees of approximation due to the limited resources in Xeon Phi compared to i9-13900K. While computationally demanding tasks may provide a better measure of the performance of the tested coprocessors, it also highlights that the architecture of these coprocessors is outdated for modern applications.

Table 22. Obtained results (in μs) for the Intel Core i9-13900K and numerical integration algorithm with discontinuous Galerkin approximation.

Variant		Degree of approximation		
		3	4	5
i9-13900K	Cilk	1.08	7.02	34.95
	Stride = 4	2.10	13.76	111.22
	Stride = 8	2.67	29.49	197.26

As in the case of standard linear approximation, the algorithm developed in the OpenCL for the GPU was launched on the tested Xeon Phi 5110P coprocessor. Due to the abovementioned software problems, this algorithm could not be

run on the second coprocessor tested. The obtained execution times (in µs) are presented in Table 23.

Table 23. Obtained execution times [µs] for OpenCL algorithm for the Intel Xeon Phi 5110P coprocessor and discontinuous Galerkin approximation.

Xeon Phi version	Degree of approximation		
	3	4	5
5110P	4.87	46.22	201.17

Comparing the results from the OpenMP/offload and OpenCL model, it can be seen that, unlike the situation with standard linear approximation, better results were obtained for the first one, except for situations with the highest approximation degree where the OpenCL result is slightly higher.

The OpenCL model strives to maximise the number of words of the stiffness matrix processed by a single thread. This may be a factor that causes the OpenCL version results to be worse than those from OpenMP/offload.

8.3.3. Conclusions. The tested coprocessors demonstrated higher performance for the convection–diffusion task using the discontinuous approximation of higher orders than for tasks with linear approximation, indicating their potential for computation. However, it is essential to note that comparable graphics cards exhibit even better performance, and even CPUs with Haswell architecture only slightly newer than Xeon Phi were faster [5, 29]. Moreover, the new Raptor Lake architecture in i9-13900K performs much better in numerical integration tasks. The results show an even more significant disparity for the highest degree of approximation. This highlights the inadequate available resources, such as registers, to achieve high performance comparable with GPU cards or newer CPU architectures.

8.4. Final conclusions

Although the Xeon Phi coprocessors were slower than the new i9-13900K processor, they represented an interesting branch of accelerator development. The research thoroughly examined the two main advantages of the tested coprocessors: their performance and straightforward programming model. However, in both cases, the authors acknowledge that programming these coprocessors is neither easy nor does it lead to staggering performance, which is consistent with studies by other authors. The tests indicate that the design of the Xeon Phi coprocessors appears outdated and can only compete with CPU architectures from a comparable time. The main factors that limit the performance of the numerical integration algorithm are the outdated architecture and the small

number of registers available per thread. Despite this, it is worth noting that the results obtained for more complex tasks with approximation levels equal to 3 and 4 came close to those from the Nvidia Tesla K20m accelerator, whose architecture was a direct competitor for the tested coprocessors. In conclusion, while the Xeon Phi coprocessors may not perform best in modern applications, the research provided valuable insights into their potential and limitations. The findings suggest that while the coprocessors are not as powerful as more modern architectures, they may still have a role to play in specific scenarios.

References

1. N.M. Atallah, C. Canuto, G. Scovazzi, The second-generation shifted boundary method and its numerical analysis, *Computer Methods in Applied Mechanics and Engineering*, **372**(1): 113341, 2020, doi: 10.1016/j.cma.2020.113341.

2. K. Banaś, F. Krużel, Comparison of Xeon Phi and Kepler GPU performance for finite element numerical integration, [in:] *High Performance Computing and Communications, 2014 IEEE 6th Intl Symp on Cyberspace Safety and Security, 2014 IEEE 11th Intl Conf on Embedded Software and Syst (HPCC,CSS,ICESS), 2014 IEEE Intl Conf on*, pp. 145–148, Aug 2014, doi: 10.1109/HPCC.2014.27.

3. K. Banaś, F. Krużel, OpenCL performance portability for Xeon Phi coprocessor and NVIDIA GPUs: A case study of finite element numerical integration, [in:] *Euro-Par 2014: Parallel Processing Workshops*, volume 8806 of *Lecture Notes in Computer Science*, pp. 158–169, Springer International Publishing, 2014, doi: 10.1007/978-3-319-14313-2_14.

4. K. Banaś, F. Krużel, J. Bielański, K. Chłoń, A comparison of performance tuning process for different generations of NVIDIA GPUs and an example scientific computing algorithm, [in:] R. Wyrzykowski, J. Dongarra, E. Deelman, K. Karczewski [Eds.], *Parallel Processing and Applied Mathematics*, pp. 232–242, Cham, Springer International Publishing, 2018, doi: 10.1007/978-3-319-78024-5_21.

5. K. Banaś, F. Krużel, J. Bielański, Optimal kernel design for finite element numerical integration on GPUs, *Computing in Science and Engineering*, **22**(6): 61–74, 2020, doi: 10.1109/MCSE.2019.2940656.

6. E.B. Becker, G.F. Carey, J.T. Oden, *Finite Elements. An Introduction*, Prentice Hall, Englewood Cliffs, 1981, doi: 10.1002/nme.1620180613.

7. L. Buatois, G. Caumon, B. Levy, Concurrent number cruncher: A GPU implementation of a general sparse linear solver, *International Journal of Parallel, Emergent and Distributed Systems*, **24**(3): 205–223, 2009, doi: 10.1080/17445760802337010.

8. F.L. Cabral, C. Osthoff, G.P. Costa, D. Brandao, M. Kischinhevsky, S.L. Gonzaga de Oliveira, Tuning Up TVD HOPMOC Method on Intel MIC Xeon Phi Architectures with Intel Parallel Studio Tools, [in:] *2017 International Symposium on Computer Architecture and High Performance Computing Workshops (SBAC-PADW)*, pp. 19–24, 2017, doi: 10.1109/SBAC-PADW.2017.12.

9. P.G. Ciarlet, *The Finite Element Method for Elliptic Problems*, North-Holland, Amsterdam, 1978, doi: 10.1137/1.9780898719208.

10. B. Cockburn, G. Karniadakis, C. Shu [Eds.], *Discontinuous Galerkin Methods: Theory, Computation and Applications*, Vol. 11 of *Lecture Notes in Computational Science and Engineering*, Springer, Berlin, 2000, doi: 10.1007/978-3-642-59721-3.

11. B. Cockburn, G. Karniadakis, C. Shu, The development of discontinuous Galerkin methods, [in:] *Discontinuous Galerkin Methods: Theory, Computation and Applications*, Vol. 11 of *Lecture Notes in Computational Science and Engineering*, pp. 1–14, Springer, Berlin, 2000, doi: 10.1007/978-3-642-59721-3_1.

12. B. Cockburn, C.W. Shu, The local discontinuous Galerkin finite element method for convection diffusion systems, *SIAM Journal on Numerical Analysis*, **35**: 2440–2463, 1998, doi: 10.1137/S0036142997316712.

13. I. Cutress, Intel's Xe for HPC: Ponte Vecchio with Chiplets, EMIB, and Foveros on 7nm, Coming 2021, *AnandTech*, 2019.

14. R. Devine, Intel Core i9-13900K review: Retaking the performance crown for team blue, *XDA Developers*, 2022.

15. J. Dongarra, *Frequently Asked Questions on the Linpack Benchmark and Top500*, 2007.

16. J. Fang, A.L. Varbanescu, H. Sips, L. Zhang, Y. Che, Ch. Xu, Benchmarking Intel Xeon Phi to guide kernel design, 2013.

17. M. Geveler, D. Ribbrock, D. Göddeke, P. Zajac, S. Turek, Towards a complete FEM-based simulation toolkit on GPUs: Unstructured grid finite element geometric multigrid solvers with strong smoothers based on sparse approximate inverses, *Computers & Fluids*, **80**: 327–332, 2013, doi: 10.1016/j.compfluid.2012.01.025.

18. D. Göddeke, H. Wobker, R. Strzodka, J. Mohd-Yusof, P. McCormick, S. Turek, Co-processor acceleration of an unmodified parallel solid mechanics code with FEASTGPU, *International Journal of Computational Science and Engineering*, **4**(4): 254–269, 2009, doi: 10.1504/IJCSE.2009.029162.

19. R. Goodwins, Intel unveils many-core Knights platform for HPC, *ZdNet*, 2010.

20. A. Howes, L. Munshi, *The OpenCL Specification*, Khronos OpenCL Working Group, 2014, version 2.0, revision 26.

21. Intel, *OpenCL Design and Programming Guide for the Intel Xeon Phi Coprocessor*, Intel Corporation, 2014.

22. Intel, *Intel C++Compiler 16.0 User and Reference Guide*, Intel Corporation, 2015.

23. Intel, *Dane techniczne produktu* [Intel products specifications], Intel Corporation, 2017.

24. Intel, Intel Unveils New GPU Architecture with High-Performance Computing and AI Acceleration, and oneAPI Software Stack with Unified and Scalable Abstraction for Heterogeneous Architectures, *Intel Newsroom*, 2019.

25. Intel, Product change notification 116378 – 00, July 23, 2018.

26. J. Jeffers, J. Reinders, *Intel Xeon Phi Coprocessor High Performance Programming*, Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1st ed., 2013.

27. C. Johnson, *Numerical Solution of Partial Differential Equations by the Finite Element Method*, Cambridge University Press, 1987, doi: 10.1007/BF00046566.

28. Y. Kallinderis, Adaptive hybrid prismatic-tetrahedral grids, *International Journal for Numerical Methods in Fluids*, **20**: 1023–1037, 1995, doi: 10.1002/fld.1650200820.

29. F. Krużel, Vectorized implementation of the FEM numerical integration algorithm on a modern CPU, [in:] *European Conference for Modelling and Simulation*, Vol. 33, pp. 414–420, 2019, doi: 10.7148/2019-0414.

30. F. Krużel, K. Banaś, Finite element numerical integration on PowerXCell processors, [in:] *PPAM'09: Proceedings of the 8th International Conference on Parallel Processing and Applied Mathematics*, pp. 517–524, Berlin, Heidelberg, Springer-Verlag, 2010, doi: 10.1007/978-3-642-14390-8_54.

31. F. Krużel, K. Banaś, Vectorized OpenCL implementation of numerical integration for higher order finite elements, *Computers and Mathematics with Applications*, **66**(10): 2030–2044, 2013, doi: 10.1016/j.camwa.2013.08.026.

32. F. Krużel, K. Banaś, Finite element numerical integration on Xeon Phi coprocessor, [in:] M. Paprzycki M. Ganzha, L. Maciaszek [Eds.], *Proceedings of the 2014 Federated Conference on Computer Science and Information Systems*, Vol. 2 of *Annals of Computer Science and Information Systems*, pp. 603–612, IEEE, 2014, doi: 10.15439/2014F222.

33. F. Krużel, K. Banaś, AMD APU systems as a platform for scientific computing, *Computer Methods in Materials Science*, **15**(2): 362–369, 2015.

34. F. Krużel, K. Banaś, M. Nytko, Implementation of numerical integration to high-order elements on the GPUs, *Computer Assisted Methods in Engineering and Science*, **27**(1): 3–26, 2020, doi: 10.24423/cames.264.

35. F. Krużel, M. Nytko, Intel Iris Xe-LP as a platform for scientific computing, [in:] M. Ganzha [Ed.], *Communication Papers of the 17th Conference on Computer Science and Intelligence Systems*, September 4–7, 2022, Sofia, Bulgaria, Vol. 32, [in:] *Annals of Computer Science and Information Systems*, pp. 121–128, Warszawa, PTI, 2022 doi: 10.15439/2022F132.

36. J.N. Lyness, Quadrature methods based on complex function values, *Mathematics of Computation*, **23**(107): 601–619, 1969, doi: 10.2307/2004388.

37. J. Mamza, P. Makyla, A. Dziekoński, A. Lamecki, M. Mrozowski, Multi-core and multiprocessor implementation of numerical integration in Finite Element Method, [in:] *Microwave Radar and Wireless Communications (MIKON), 2012 19th International Conference*, Vol. 2, pp. 457–461, 2012, doi: 10.1109/MIKON.2012.6233633.

38. J.D. McCalpin, Memory bandwidth and machine balance in current high performance computers, *IEEE Technical Committee on Computer Architecture (TCCA) Newsletter*, pp. 19–25, December 1995.

39. K. Michalik, K. Banaś, P. Płaszewski, P. Cybułka, ModFEM – a computational framework for parallel adaptive finite element simulations, *Computer Methods in Materials Science*, **13**(1): 3–8, 2013.

40. S. Muralikrishnan, M.-B. Tran, T. Bui-Thanh, An improved iterative HDG approach for partial differential equations, *Journal of Computational Physics*, **367**: 295–321, 2018, doi: 10.1016/j.jcp.2018.04.033.

41. S. Naik, *Best Known Method: Estimating FLOP/s for workloads running on the Intel Xeon Phi coprocessor using Intel VTune Amplifier XE*, September 2013.

42. T. Olas, W.K. Mleczko, R.K. Nowicki, R. Wyrzykowski, A. Krzyzak, Adaptation of RBM Learning for Intel MIC Architecture, [in:] L. Rutkowski, M. Korytkowski, R. Scherer,

R. Tadeusiewicz, A.L. Zadeh, M.J. Zurada [Eds.], *Artificial Intelligence and Soft Computing: Proceedings of the 14th International Conference. ICAISC 2015. Part I*, Zakopane, Poland, June 14–18, pp. 90–101, Cham, Springer International Publishing, 2015, doi: 10.1007/978-3-319-19324-3_9.

43. OpenMP Architecture Review Board, *OpenMP Application Programming Interface*, version 4.5 edition, November 2015.

44. F. Roth, *System Administration for the Intel Xeon Phi Coprocessor*, Intel Corporation, 2013.

45. S. Rul, H. Vandierendonck, J. D'Haene, K. De Bosschere, An experimental study on performance portability of OpenCL kernels, [in:] *Application Accelerators in High Performance Computing, 2010 Symposium*, p. 3, Knoxville, TN, USA, 2010.

46. W.C. Schneck, E.D. Gregory, C.A.C. Leckey, Optimization of elastodynamic finite integration technique on Intel Xeon Phi Knights Landing processors, *Journal of Computational Physics*, **374**: 550–562, 2018, doi: 10.1016/j.jcp.2018.07.049.

47. L. Seiler, D. Carmean, E. Sprangle, T. Forsyth, M. Abrash, P. Dubey, Larrabee: a many-core x86 architecture for visual computing, *SIGGRAPH 08: ACM SIGGRAPH 2008 papers*, pp. 1–15, 2008, doi: 10.1109/MM.2009.9.

48. E. Strohmaier, J. Dongarra, S. Horst, M. Meuer, H. Meuer, *Top500 The List*, 2020, http://www.top500.org.

49. Ł. Szustak, K. Rojek, P. Gepner, Using Intel Xeon Phi coprocessor to accelerate computations in MPDATA algorithm, [in:] R. Wyrzykowski, J. Dongarra, K. Karczewski, J. Waśniewski [Eds.], *Parallel Processing and Applied Mathematics: 10th International Conference, PPAM 2013. Part I*, Warsaw, Poland, September 8–11, 2013, pp. 582–592, Berlin, Heidelberg, Springer, 2014, doi: 10.1007/978-3-642-55224-3_54.

50. Ł. Szustak, K. Rojek, T. Olas, Ł. Kuczyński, K. Halbiniak, P. Gepner, Adaptation of MPDATA heterogeneous stencil computation to Intel Xeon Phi coprocessor, *Scientific Programming*, **2015**: 642705, 2015, doi: 10.1155/2015/642705.

51. S. Williams, A. Waterman, D. Patterson, Roofline: An insightful visual performance model for multicore architectures, *Communications of the ACM*, **52**(4): 65–76, 2009, doi: 10.1145/1498765.1498785.

Received April 5, 2022; revised version March 14, 2023; accepted April 6, 2023.