CAMES, **27**(1): 3–26, 2020, doi: 10.24423/cames.264

Implementation of Numerical Integration to High-Order Elements on the GPUs

Filip KRUŻEL[1)*], Krzysztof BANAŚ[2)], Mateusz NYTKO[1)]

[1)] *Cracow University of Technology, Department of Computer Science*
Warszawska 24, 31-155 Kraków, Poland
*Corresponding Author e-mail: fkruzel@pk.edu.pl

[2)] *AGH Science and Technology University*
Department of Applied Computer Science and Modelling
Adama Mickiewicza 30, 30-059 Kraków, Poland

This article presents ways to implement a resource-consuming algorithm on hardware with a limited amount of memory, which is the GPU. Numerical integration for higher-order finite element approximation was chosen as an example algorithm. To perform computational tests, we use a non-linear geometric element and solve the convection-diffusion-reaction problem. For calculations, a Tesla K20m graphics card based on Kepler architecture and Radeon r9 280X based on Tahiti XT architecture were used. The results of computational experiments were compared with the theoretical performance of both GPUs, which allowed an assessment of actual performance. Our research gives suggestions for choosing the optimal design of algorithms as well as the right hardware for such a resource-demanding task.

Keywords: GPU, numerical integration, finite element method, OpenCL, CUDA.

1. Introduction

In the development of computing accelerators, one can notice that most of them are derived directly from graphics cards. The key fact which enabled the observation of the computing capabilities of the graphics cards was the combination of many functional units responsible for different stages of graphics processing in a single GPU (Graphics Processing Unit). The evolution of graphics cards began with the concept of a stream (information processing on the data stream), on which early graphics processing was based. The three-dimensional data processing required a whole sequence of operations – from mesh generation to its transformation into a two-dimensional image composed of pixels. Over the years, the next steps of this processing have been gradually transferred from the CPU

to the supporting graphics card. The first GPU that allowed for the processing of the entire "stream" of operations needed to display graphics was the Nvidia GeForce 256. The downside of this design was the fact that it prevented any changes to the previously prepared instruction stream. For this reason, in 2001, Nvidia created the GeForce 3 architecture, where it was possible to create special programs (shaders) that could manipulate data in the pipe.

Units performing individual operations such as triangulation, lighting, mesh transformation, or final rendering were implemented as physically separate pieces of the graphics card – called shading units. The next step in the evolution of GPU architectures was the unification of all shading units and the creation of fully programmable streaming multiprocessor. Thanks to this, graphics card programming got closer to CPU programming, and in many applications, it became more efficient. The G80 architecture became the basis of the first accelerator of the engineering calculations named Nvidia Tesla, on which the architectures all modern GPU accelerators are based [8].

2. Numerical integration

One of the most demanding engineering tasks to br carried out on a GPU turns out to be the procedures of the Finite Element Method (FEM). One of the most important parts of FEM is numerical integration used for the preparation of an elementary stiffness matrix for a system solver. Usually, the past studies about using the computing power of the GPU focused on the most time-consuming part of FEM, which is solving the final system of linear equations [6, 9, 10]. However, it should be noted that after the optimization of the procedure mentioned above, the earlier steps of calculations such as, e.g., numerical integration and assembling, also begin to affect the execution time significantly [17]. Numerical integration in FEM is strictly correlated with the given elemental shape functions that are connected with the used geometry of an element and an approximation type. Therefore, there is a need to apply suitable geometric transformation in mesh geometry used for computing. Each element of the mesh is treated as a transformed reference element. By denoting physical coordinates in the mesh as $\boldsymbol{x}$, the transformation from the reference element with coordinates $\boldsymbol{\xi}$ is denoted as $\boldsymbol{x}(\boldsymbol{\xi})$. It is usually obtained through the general form of linear, multi-linear, square, cubic, or other transformation of basic geometry functions and a set of degrees of freedom. Those degrees are usually related to specific points in the reference element (depending on the approximation used) – for example, a vertex for linear or multi-linear transformation.

The integration transformation from the reference element requires using the Jacobian matrix $J = \frac{\partial \boldsymbol{x}}{\partial \boldsymbol{\xi}}$. Computing the Jacobian matrix, its determinant and

inverse, are the typical components of the numerical integration in FEM. The presence of such an operation significantly differentiates this algorithm from the other integration and matrix-multiplication algorithms, where there is no need for such an additional operation within the main loop of calculations.

The method of calculating the elements of the Jacobian matrices differs depending on the type of the element and approximation used. For multi-linear elements and higher-order approximation, the calculations of these elements can become an essential factor affecting the overall performance of the algorithm.

The integrals calculated for the shape function of individual elements form local stiffness matrices for each element. Words from these matrices may appear in several places in the global stiffness matrix. As a result of integration, we get a small elementary stiffness matrix, whose single expression can be calculated in the form (1):

$$(A^e)^{i_s j_s} = \int\limits_{\Omega_e} \sum_{i_D} \sum_{j_D} \boldsymbol{C}^{i_D j_D} \frac{\partial \boldsymbol{\phi}^{i_s}}{\partial \boldsymbol{x}_{i_D}} \frac{\partial \boldsymbol{\phi}^{j_s}}{\partial \boldsymbol{x}_{j_D}} d\Omega, \tag{1}$$

where i_s and j_s are the local indexes that change in the range from 1 to N_S (the number of shape functions for a given element) and i_D and j_D are the indexes that change in the range from 1 to N_D (dimension of the Ω).

Similarly, the right-hand side vector is calculated (2):

$$(b^e)^{i_s} = \int\limits_{\Omega_e} \sum_{i_D} \boldsymbol{D}^{i_D} \frac{\partial \boldsymbol{\phi}^{i_s}}{\partial \boldsymbol{x}_{i_D}} d\Omega. \tag{2}$$

Applying the variable change to the reference element $\widehat{\Omega}$ for the selected integral example leads to the formula (3):

$$(A^e)^{i_s j_s} = \int\limits_{\widehat{\Omega}} \sum_{i_D} \sum_{j_D} \boldsymbol{C}^{i_D j_D} \sum_{k} \frac{\partial \widehat{\phi}^{i_s}}{\partial \boldsymbol{\xi}_{k_D}} \frac{\partial \boldsymbol{\xi}_{k_D}}{\partial \boldsymbol{x}_{i_D}} \sum_{z} \frac{\partial \widehat{\phi}^{j_s}}{\partial \boldsymbol{\xi}_{l_D}} \frac{\partial \boldsymbol{\xi}_{l_D}}{\partial \boldsymbol{x}_{j_D}} \det \boldsymbol{J} d\widehat{\Omega}, \tag{3}$$

where $\widehat{\phi}^{i_s}$ denotes the shape functions for the reference element and k_D and $l_D = 1, ..., N_D$. In this formula, the determinant of the Jacobi matrix $\det \boldsymbol{J} = \det\left(\frac{\partial \boldsymbol{x}}{\partial \boldsymbol{\xi}}\right)$ and the components of the Jacobian inverse transformation matrix $\boldsymbol{\xi}(\boldsymbol{x})$, from real to reference elements, are used.

The next step is to replace the analytical integral with the numerical quadrature. In this work, a Gaussian quadrature was used with coordinates in the reference element marked as $\boldsymbol{\xi}^{i_Q}$ and weights $\boldsymbol{w}^{i_Q}$ where $i_Q = 1, .., N_Q$ (N_Q – the

number of Gauss points, which depends on the type of the element and the degree of approximation used). For integral (3) this leads to the formula (4):

$$\int_{\Omega_e} \sum_{i_D} \sum_{j_D} \boldsymbol{C}^{i_D j_D} \frac{\partial \boldsymbol{\phi}^{i_s}}{\partial \boldsymbol{x}_{i_D}} \frac{\partial \boldsymbol{\phi}^{j_s}}{\partial \boldsymbol{x}_{j_D}} d\Omega$$

$$\approx \sum_{i_Q=1}^{N_Q} \left(\sum_{i_D} \sum_{j_D} \boldsymbol{C}^{i_D j_D} \frac{\partial \boldsymbol{\phi}^{i_s}}{\partial \boldsymbol{x}_{i_D}} \frac{\partial \boldsymbol{\phi}^{j_s}}{\partial \boldsymbol{x}_{j_D}} \det \boldsymbol{J} \right) \Bigg|_{\boldsymbol{\xi}^{i_Q}} \boldsymbol{w}^{i_Q}. \quad (4)$$

In order to create the general numerical integration algorithm, some modifications of the above formulas are made. They are made to make the algorithm more easy to be implemented into the programming language code. They are as follows:

- $\boldsymbol{\xi}[i_Q]$, $\boldsymbol{w}[i_Q]$ – tables with local coordinates of integration points (Gauss points) and weights assigned to them, $i_Q = 1, 2, ..., N_Q$,
- $\boldsymbol{G}^e$ – table with element geometry data (related to the transformation from the reference element to real element),
- $\mathbf{vol}[i_Q]$ – table with volumetric elements $\mathbf{vol}[i_Q] = \det\left(\frac{\partial \boldsymbol{x}}{\partial \boldsymbol{\xi}}\right) \times \boldsymbol{w}[i_Q]$,
- $\boldsymbol{\phi}[i_Q][i_S][i_D]$, $\boldsymbol{\phi}[i_Q][j_S][j_D]$ – 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,
  - $i_D, j_D = 0, 1, ..., N_D$, for i_D, j_D different from zero, the tables refer to the derivatives relative to the coordinate at index i_D, and for $i_D, j_D = 0$ to the shape function, so $i_D, j_D = 0, 1, 2, 3$,
- $\boldsymbol{C}[i_Q][i_D][j_D]$ – table with values of problem coefficients (material data, values of degrees of freedom in previous nonlinear iterations and time steps, etc.) at subsequent Gauss points,
- $\boldsymbol{D}[i_Q][i_D]$ – table with D^{i_D} coefficient values (Eq. (2)) in subsequent Gauss points,
- $\boldsymbol{A}^e[i_S][j_S]$ – an array storing the local, elementary stiffness matrix,
- $\boldsymbol{b}^e[i_S]$ – an array storing the local, elementary right-hand side vector.

By introducing the presented notation we can present a general formula for the elemental stiffness matrix:

$$\boldsymbol{A}^e[i_S][j_S] = \sum_{i_Q}^{N_Q} \sum_{i_D, j_D}^{N_D} \boldsymbol{C}[i_Q][i_D][j_D] \times \boldsymbol{\phi}[i_Q][i_S][i_D] \times \boldsymbol{\phi}[i_Q][j_S][j_D] \times \mathbf{vol}[i_Q], \quad (5)$$

and right-hand side vector:

$$\boldsymbol{b}^e[i_S] = \sum_{i_Q}^{N_Q} \sum_{i_D}^{N_D} \boldsymbol{D}[i_Q][i_D] \times \boldsymbol{\phi}[i_Q][i_S][i_D] \times \mathbf{vol}[i_Q]. \tag{6}$$

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

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

```
- determine the algorithm parameters – N_EL, N_Q, N_S;
- load tables ξ^Q and w^Q with numerical integration data;
- load the values of all shape functions and their derivatives relative to local
  coordinates at all integration points in the reference element;
for e = 1 to N_EL do
    - load problem coefficients common for all integration points (Array C^e);
    - load the necessary data about the element geometry (Array G^e);
    - initialize element stiffness matrix A^e and element right-hand side vector b^e;
    for i_Q = 1 to N_Q do
        - calculate the data needed for the Jacobian transformations (∂x/∂ξ, ∂ξ/∂x, vol);
        - calculate the derivatives of the shape function relative to global coordinates
          using the Jacobian matrix;
        - calculate the coefficients C[i_Q] and D[i_Q] at the integration point;
        for i_S = 1 to N_S do
            for j_S = 1 to N_S do
                for i_D = 0 to N_D do
                    for j_D = 0 to N_D do
                        A^e[i_S][j_S]+ = vol
                          ×C[i_Q[[i_D][j_D] × φ[i_Q][i_S][i_D] × φ[i_Q][j_S][j_D];
                    end
                end
                if i_S = j_S and i_D = j_D then
                    b^e[i_S]+ = vol ×D[i_Q][i_D] × φ[i_Q][i_S][i_D];
                end
            end
        end
    end
    - write the entire matrix A^e and vector b^e;
end
```

The optimal structure of the algorithm must include the capabilities of the hardware for which this algorithm should be developed. In the case of external accelerators like GPU, the cost of sending data to the accelerator and back can be very high and should be hidden by sufficiently saturated calculations. The

arrangement of Algorithm 1, in which the outer loop is a loop after all elements, favors this. The general form of Algorithm 1, in which we do not consider data locations in different memory levels, allows us to treat each of the internal loops as independent and to change their order to achieve optimal performance. Such a performance can also be achieved by calculating all the necessary data in advance and use them further. This allows for the creation of different variants of the algorithm depending on the hardware resources.

2.1. The model problem

Our previous work focused on the implementation of the numerical integration algorithm on various types of accelerators, including PowerXCell [12, 13], AMD Accelerated Processing Unit [15], Intel Xeon Phi numeric coprocessors [14], and GPU-based architectures [2, 4]. Most of them focused on the implementation of the standard linear approximation for several problems and geometry types. This work leads us to develop a very efficient auto-tuning mechanism that can adapt the code to the architecture. This is possible because standard linear approximation is not a very resource-consuming way of solving numerical integration. There is a different situation when we are using higher-order elements associated with the discontinuous Galerkin method used to approximate the solution of FEM. The main problem connected with such elements is that the amount of data needed for numerical integration grows rapidly with an increasing degree of approximation. This leads to the problem of managing, often limited resources of various types of GPU accelerators to obtain an efficient implementation of the algorithm.

The main aim of this study is to check the possibility of using the GPU – based accelerators to maintain the larger and more complicated tasks than those in our previous studies with the use of standard linear approximation. To achieve it, we are using non-linear prismatic element type geometry, and we are solving an artificial convection-diffusion-reaction problem with the coefficient matrix $\boldsymbol{C}[i_Q][i_D][j_D]$ fully filled with values. Matrices of this type appear, for example, in convective heat transfer problem, after applying the SUPG stabilization. The general form of weak formulation for these types of problems can be found in [11] and [5]. In this paper, the authors focused on calculating the integral over the Ω area since the boundary integrals are usually less computationally demanding, and, from an algorithmic point of view, they repeat a similar integration scheme. The part of calculations responsible for the boundary conditions in the code used by the authors is always performed by the CPU cores, using standard FEMs. In the discontinuous Galerkin method, the degrees of freedom of an element are related to its interior, and their number depends on the degree of approximation. For the reference element of the prismatic type tested, the

number of shape functions is given by the formula $\frac{1}{2}(p+1)^2(p+2)$ [21]. In this type of element, shape functions are obtained as the products of polynomials from a set of complete polynomials of a given order for triangular bases and 1D polynomials associated with the vertical direction. The number of integration points is determined by the requirement to accurately compute the products of shape functions, neglecting non-linearity of coefficients, and element geometry. It increases as the degree of approximation increases to maintain solution convergence [7], and for three-dimensional prisms, it is of the order $O(p^3)$ just like the number of shape functions. Basic Algorithm 1 parameters, depending on the degree of approximation, are presented in Table 1.

Table 1. The number of Gauss points and shape function depending on the degree of approximation.

Parameter	Degree of approximation						
	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

2.2. Complexity

When estimating the number of operations in the numerical integration algorithm with the higher-order prismatic elements, we can specify the following calculation stages:

1) Jacobian calculations (line 9 of the Algorithm 1) – in the case of the convection-diffusion problem this stage is repeated N_Q times
   - Calculation of the derivatives of basic shape functions – due to its repetitive nature, we can assume that each compiler will reduce the number of operations to 14.
   - Matrix J and its inverse, determinant, and volume element (**vol**) – 18 operations for each of the geometric degrees of freedom to create a Jacobian matrix + 43 operations related to its inverse and determinant.
2) Calculation of the derivatives of shape functions relative to global coordinates (line 10 of the Algorithm 1) – usually performed before double loop over shape functions to avoid redundancy of calculations (although the algorithm version with derivatives calculated inside the loop over the i_S index can be considered) – 15 operations for each shape function (N_S times).
3) Calculation of the product of matrix coefficients $C[i_Q]$ with shape functions dependent on the first loop over the shape functions (i_S) – optimization

reducing repetition of calculations by pulling out before j_S loop expressions that are independent of its index – 22 operations. This stage will be repeated $N_Q \times N_S$ times.

4) Calculation of the right-hand side vector (line 24 of the Algorithm 1) – 9 operations. We are omitting this stage because these calculations are similar to the calculation of the stiffness matrix and take less than 2% of the total calculation time.
5) Final calculation of the elements of the stiffness matrix (line 18 of the Algorithm 1) – 9 operations for convection – diffusion repeated for $N_Q \times N_S \times N_S$ times.

The final number of operations performed for the algorithm will depend on the applied optimization, both manual and automatic. Since some coefficients can be constant (e.g., convection-diffusion-reaction problem coefficients) and the obtained matrices are symmetrical, some compilers can further reduce the estimated number of operations. For prismatic elements, some calculations are performed separately for each integration point, due to which similar optimizations may be complicated, which should result in a much larger number of operations. By using parameters N_Q and N_S presented in Table 1 and the calculated number of repetitions for each part of the algorithm, calculations of the number of operations depending on the degree of approximation were made. The results for each of the steps presented above are shown in Table 2.

Table 2. The number of operations for the convection-diffusion problem with discontinuous Galerkin approximation.

Calculation stage	Degree of approximation						
	1	2	3	4	5	6	7
1	990	2970	7920	13200	24750	38115	55440
2	540	4860	28800	90000	283500	679140	1451520
3	792	7128	42240	132000	415800	996072	2128896
5	1944	52488	691200	4050000	21432600	79866864	250822656
$\sum$	**4266**	**67446**	**770160**	**4285200**	**22156650**	**81580191**	**254458512**

Another aspect requiring analysis is the size of data needed to store current calculations. For a numerical integration algorithm with higher-order elements, the size of the data needed increases as the degree of approximation increases as it can be seen in Table 3.

An important element of performance analysis is the study of the number of accesses to various levels of the memory hierarchy during the execution of the algorithm. This study is only possible by taking into account the details

TABLE 3. Memory requirements for numerical integration in the convection-diffusion problem and different degree of approximation.

	Degree of approximation						
	1	2	3	4	5	6	7
Integration data	24	72	192	320	600	924	1344
Coefficients $\boldsymbol{C}$	16	16	16	16	16	16	16
Geometric data	18	18	18	18	18	18	18
Shape functions $\boldsymbol{\phi}$	144	1296	7680	24000	75600	181104	387072
Stiffness matrix $\boldsymbol{A}^e$	36	324	1600	5625	15876	38416	82944
$\sum$	**238**	**1726**	**9506**	**29979**	**92110**	**220478**	**471394**

of the calculation on specific architectures. As an example architecture, which theoretically allows for explicit memory management, a GPU was chosen.

3. Graphics Processing Unit (GPU)

During these studies, the authors examined two competing solutions – one from Nvidia and the other from AMD/ATI. The first card is the Nvidia Tesla K20m accelerator. It is based on Kepler GK110 core architecture (Fig. 1). As

FIG. 1. Tesla K20m [18].

can be seen in the figure, this accelerator has 13 SMX multiprocessors sharing a common second-level cache (L2). The scheduler named GigaThread Engine is responsible for the distribution of work between individual multiprocessors. This accelerator is equipped with six separate memory controllers for RAM support. It connects to the host using a PCI Express 3.0 interface.

Each of the SMX units is equipped with 192 single-precision (SP) processing cores and 64 double-precision (DP) cores, which in total gives 2496 SP cores and 832 DP cores (Fig. 2). Due to the huge number of cores, we refer to such constructions as the multi-core architectures. Each multiprocessor is equipped with 32 load and store units (LD/ST), which allows storing the data in 65536 32-bit registers (256 kB). In addition, SMX has 32 special units (Special Function Units) used to perform the so-called transcendental instructions (sin, cos, inverse, and similar).

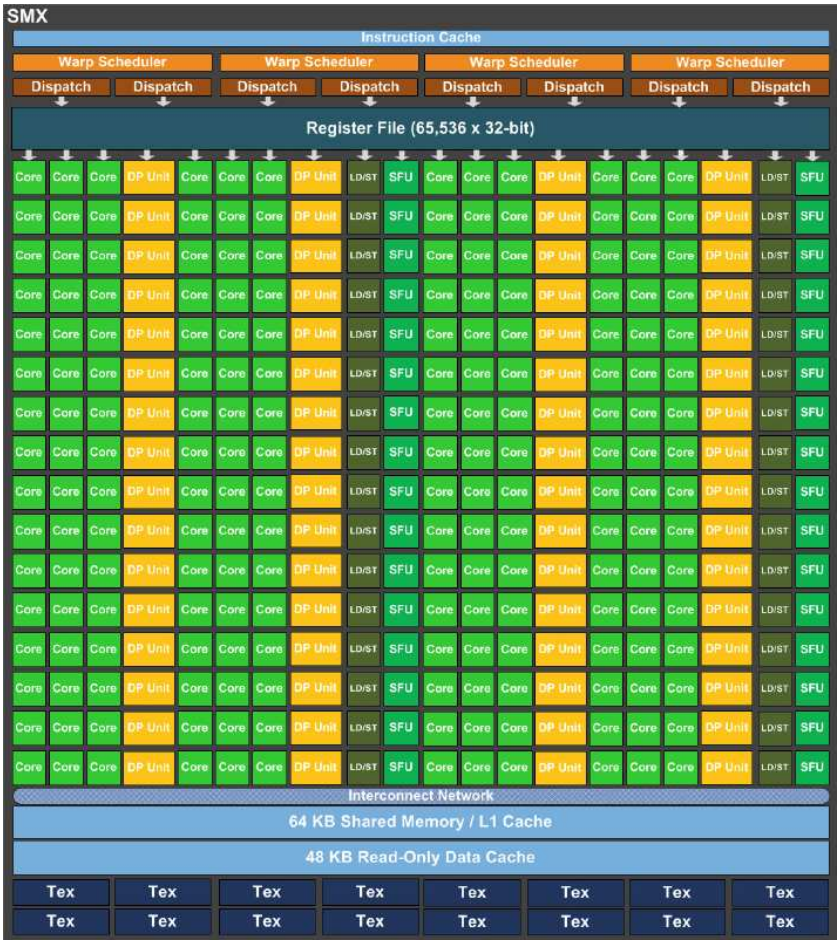

Fig. 2. Next Generation Streaming Multiprocessor (SMX) [18].

Threads are grouped in the so-called *warps* (32 threads), and four warp schedulers manage their work. Each SMX is equipped with 64 KB of memory, which is divided between the L1 cache and shared memory for threads. Their size can be managed by the programmer, depending on the needs. Also, each multiprocessor has 48 KB of read-only and texture memory, which can be treated as a high-speed read-only cache. The performance parameters of the tested hardware are presented in Table 4.

Table 4. Parameters of the tested Tesla K20m card [18].

Theoretical performance of double-precision calculations [TFlops]	1.17
Theoretical performance of single-precision calculations [TFlops]	3.52
Theoretical memory bandwidth [GB/s]	208

Theoretical performance for the tested Tesla K20m card is calculated from the formula *2* (operations in FMA instructions in each core) ** number of cores * clock frequency* (706 GHz). Memory bandwidth was calculated from the formula *2* (data transfers in tact – 2 for Double Data Rate memory) ** bus width* (320 bit) **2* (number of memory channels) ** clock frequency* (1300 MHz).

Another tested GPU was the Gigabyte Radeon R9 280X, which is based on the Tahiti XT architecture (Fig. 3). It is equipped with 32 cores in the Graphics Core Next (GCN) microarchitecture and 3GB of RAM. It has two asynchronous calculation engines (ACE), two units for geometry and rasterization calculations, and a single command processing unit (Command Processor).

The accelerator has eight rendering units (ROPs) equipped with memory buffers. Also, data can be stored in L2 memory and special 64 KB GDS (Global Data Share) memory. Groups of four GCN units are sharing 16 KB read-only cache and 32 KB instruction cache. Six separate two-channel memory controllers are used to communicate with the card's RAM. Also, at the host's memory communication level, the PCI Express 3.0 interface and special video decoding units (Video Codec Engine and Unified Video Decoder) were used.

GCN units consist of four 16 core vector units equivalent to cores in the Nvidia Kepler architecture (Fig. 4). In total, this gives 2048 cores whose performance depends on whether they support single- or double-precision and architecture-specific settings. Each GCN unit contains a scheduler responsible for a task distribution between threads grouped into waves (64 threads). Each vector unit contains a 64 KB register file, and a single scalar unit has an 8 KB register file. Individual GCN units are equipped with 16 KB L1 cache. Also, each GCN has 64 KB of LDS (Local Data Share) memory for thread synchronization and storage of shared data. The performance parameters of the tested card are presented in Table 5.

FIG. 3. Radeon R9 280X [1].

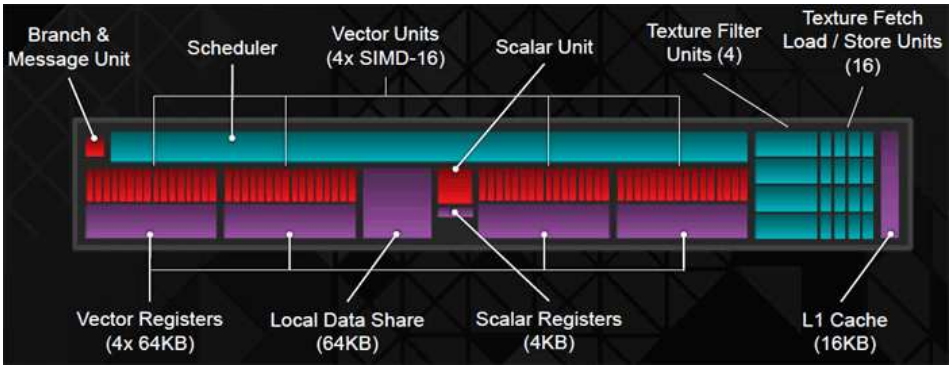

FIG. 4. Graphics Core Next Multiprocessor [1].

TABLE 5. Parameters of the tested Radeon R9 280X card [20].

Theoretical performance of double-precision calculations [TFlops]	0.87
Theoretical performance of single-precision calculations [TFlops]	3.48
Theoretical memory bandwidth [GB/s]	288

The theoretical performance for the tested Radeon R9 280X card is calculated from the same formula as in the case of the Tesla card, i.e., *2* (operations in FMA instructions in each core) ** number of cores * clock frequency* (850 GHz). For the tested equipment, the DP calculation efficiency is a $\frac{1}{4}$ of the SP calculation efficiency. Memory bandwidth was calculated from the formula *2* (data transfers in time) ** bus width* (384 bit) ** 2* (number of memory channels) ** clock frequency* (1500 MHz).

As it can be seen in the Tables 4 and 5, both cards are characterized by similar performance, even though that the Nvidia solution is dedicated to high-performance calculations, and the AMD card is a GPU designed for graphically advanced games.

3.1. Performance analysis

To analyze machine code (or pseudo-machine code in the case of an Nvidia's closed solution), a system that counts the number of occurrences of specific instructions was developed. It allows for analysis of what optimizations were used during a compilation and generates basic reports on memory access and arithmetic operations.

Nvidia Visual Profiler was used for profiling the code (Fig. 5). It works in the CUDA environment and performs dynamic analysis of the program during its

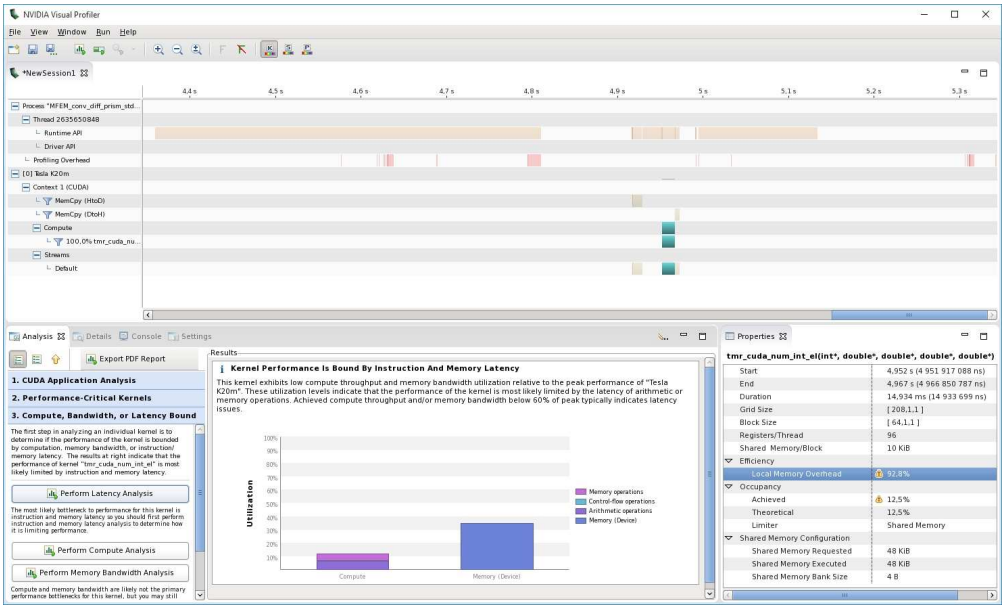

Fig. 5. Nvidia Visual Profiler.

execution. This application visualizes the instructions flow and allows for finding areas that need optimization. It also generates reports to suggest changes that may affect program performance. Thanks to the direct hardware-based, Nvidia Visual Profiler allows for measuring the effectiveness of applications directly from hardware counters, which for Nvidia's closed solutions is the only method for a more in-depth analysis of performance [19].

A complementary tool for AMD's accelerators is CodeXL, which allows debugging and code analysis in the OpenCL environment. It is an open solution that allows the analysis of performance on GPU-based accelerators, ordinary CPUs as well as hybrid APUs.

4. Results

Our previous study focused on the implementation of numerical integration for standard linear approximation. On GPUs, this approach allows for storing the whole computational domain within one kernel. The computations were divided within the whole workgroup (block) of threads, usually allowing for the calculation of the whole stiffness matrix of one element by one thread. We called this approach One Element – One Thread model, and its details can be found in [3]. For a convection-diffusion task using high-order prismatic elements, a different approach was needed due to the limited number of registers and memory on the graphics card.

During our studies, we have decided to develop a new, more resources saving, method which can be referred to as One Element – Several Kernels model. In the case of higher-order elements, the most suitable scenario is the one in which one element is processed by the entire (relatively large) working group. In this way, we are avoiding redundancy in the calculation of the Jacobian transformations, which are constant throughout the entire element. Conducted tests showed that the calculations of the Jacobian transformations take less than 1% of the time of the entire algorithm. This fact prompted us to separate them and calculate with the kernel with the One Element – One Thread division type. The calculated values are stored in the card's memory and then used by the next kernel calculating the stiffness matrix in the One Element – One Working Group model. Other aspects that attempted to affect the final performance were skipping the calculation of the right-hand side vector (less than 2% of the total computation time) and the use of a *float* type as more native to the GPU and not significantly affecting the accuracy of the results obtained. A modified numerical integration algorithm for high-order prismatic elements is presented by Algorithm 2.

Algorithm 2: Modified numerical integration algorithm for high order prismatic elements.

- determination of algorithm parameters – N_{EL}, N_Q, N_S, $N_{\text{entries\_per\_thread}}$;
- calculation of the number of stiffness matrix parts: $N_P = \lceil \frac{N_S * N_S}{N_{\text{entries\_per\_thread}} * WG\_SIZE} \rceil$;

KERNEL 1:

for $e = 1$ *to* N_{EL} **do**

- load the element geometry data (Table $\boldsymbol{G}^e$) ;

for $i_Q = 1$ *to* N_Q **do**

- read the coordinates of the integration point and its weight from table with numerical integration data;
- calculate the needed data for the Jacobian transformations $\left(\frac{\partial \boldsymbol{x}}{\partial \boldsymbol{\xi}}, \frac{\partial \boldsymbol{\xi}}{\partial \boldsymbol{x}}, \mathbf{vol}\right)$;
- save the calculated matrix of the Jacobian transformations and the volumetric element for a given Gauss point in the card's RAM ;

end

end

KERNEL2:

for $e = 1$ *to* N_{EL} **do**

- load problem coefficients common for all integration points (Table $\boldsymbol{C}^e$);

if *JACOBIAN_DATA_IN_SHARED_MEMORY* **then**

- load previously calculated data of the Jacobian transformations for all Gauss points in the element;

end

for $i_P = 1$ *to* N_P **do**

- calculate index j_S based on the thread number, part number i_P, and the number of the calculated stiffness matrix elements ($N_{\text{entries\_per\_thread}}$) ;
- initialize a fragment of the element stiffness matrix of the size $N_{\text{entries\_per\_thread}}$ - $\boldsymbol{A}^e_w$;

for $i_Q = 1$ *to* N_Q **do**

if *JACOBIAN_DATA_IN_SHARED_MEMORY* **then**

- load the Jacobian transformation data for a single Gauss point;

else

- load the Jacobian transformation data for a single Gauss point (form RAM with the use of cache);

end

- calculate derivatives of shape functions relative to global coordinates using the Jacobian matrix (each thread counts some derivatives and saves them in shared memory);
- calculate problem coefficients $\boldsymbol{C}[i_Q]$ at the integration point;

for $i_w = 0$ *to* $N_{entries\_per\_thread}$ **do**

- calculate the appropriate index i_S;
- load previously calculated derivatives of the shape function;

for $i_D = 0$ *to* N_D **do**

for $j_D = 0$ *to* N_D **do**

$\boldsymbol{A}^e_w[i_S][j_S] += \mathbf{vol} \times \boldsymbol{C}[i_Q[[i_D][j_D] \times \times \boldsymbol{\phi}[i_Q][i_S][i_D] \times \boldsymbol{\phi}[i_Q][j_S][j_D];$

end

end

end

end

end

- write a fragment of the matrix $\boldsymbol{A}^e_w$ to the output buffer in the accelerator global memory;

end

end

As one can see, this approach is characterized by an additional level of parallelism at the level of the double loop over shape functions. Additionally, three modifiable tuning options for the selected algorithm has been added:

1) NENTPT (Number of ENTries Per Thread) – means how the thread computes many entries to the stiffness matrix – it divides the stiffness matrix into smaller parts and significantly reduces the time needed for calculations.
2) NR_PARTS – directly dependent on the previous one according to the formula:
$N_P = \lceil \frac{N_S * N_S}{N_{\text{entries\_per\_thread}} * WG\_SIZE} \rceil$, where WG_SIZE means the size of the workgroup – an additional division of the stiffness matrix into smaller parts to reduce the number of registers needed per thread.
3) USE_WORKSPACE_FOR_JACOBIAN_DATA – allows to pre-rewrite the Jacobian transformation data into the shared memory for a single element calculated in kernel 1.

4.1. Performance model

In the newer GPU architectures, in addition to the standard division into shared/local, constant and global memory, the presence of cache (usually two-level) memory should also be taken into account. Cache memory makes a large part of the execution non-deterministic, and its presence in such architectures makes the overall design similar to the standard model known from the CPU.

Table 6 shows the basic hardware parameters of the tested accelerators. As can be seen from the table, a relatively large number of registers for architectures based on GPUs is paid for by the small size of a fast memory level (especially L2 cache). Inadequate resource allocation can result in fewer threads running simultaneously or, in the worst case, even prevent a startup. To develop a per-

Table 6. Characteristics of the tested accelerators [18, 20].

	Tesla K20m	Radeon R9 280X
Cores (multiprocessors)	13	32
L2 cache	1536 kB	768 kB
Global memory (DRAM)	5GB	3GB
Core characteristics		
SIMD (SP/DP) units	192/64	64/16
L1 cache	16 kB	16 kB
Registers	256 kB	256 kB
Constant memory	48 kB	64 kB
Shared memory	48 kB	32 kB

formance model, comparing the capabilities of the tested accelerators with the memory demand for the algorithm is necessary. The performance parameters of the tested accelerators, theoretical and obtained in the Stream and Linpack tests, are presented in Table 7.

Table 7. Performance of the tested accelerators [18, 20].

Tested accelerators	Radeon	Tesla
Theoretical performance of double-precision calculations [TFlops]	1.13	1.17
Performance achieved in the DGEMM benchmark	0.65	1.10
Theoretical performance of single-precision calculations [TFlops]	4.51	3.52
Performance achieved in the SGEMM benchmark	1.70	2.61
Theoretical memory bandwidth [GB/s]	288	208
Memory bandwidth from the STREAM benchmark	219	144

The use of shared memory may be necessary due to the insufficient number of registers for the kernel. If the number of registers used is exceeded, thread-local variables are stored in cache and DRAM (register spilling). Since shared memory is several times faster than DRAM, its proper use has a tremendous impact on performance. Since the minimum number of threads in GPU accelerators is usually large and in our case is set to 64, it should be noticed that storing n local variables for each thread will require $n \times 64$ places in shared memory.

In order to develop a performance model and verify it with data from the profiler, the number of operations was calculated for each of the higher approximation degrees and the modified numerical integration algorithm. The calculated and obtained from the profiler results are presented in Table 8.

Table 8. The number of operations for the high-order prismatic elements in the convection-diffusion problem.

	Degree of approximation		
	3	4	5
Estimated	762240	4272000	22131900
Profiler	816000	4665600	22828800

As it can be noticed, the estimated values are very close to the real values obtained by the profiler. This can indicate that the chosen model is accurate.

The number of accesses to global memory can be estimated from Table 3. Total amount of access to the RAM can be computed according to the formula (7):

$$\begin{aligned} &N_G \text{ (geometrical data} - 3 \cdot N_{geo} \text{ (6 for prisms))} + \\ &N_P \text{ (problem coefficients} - 16 \text{ for convection-diffusion problem)} + \\ &N_Q \cdot N_S \cdot 4 \text{ (shape functions and derivatives for a reference element)} + \\ &N_S \cdot N_S \text{ (size of the stiffness matrix).} \end{aligned} \tag{7}$$

Access to shared memory was estimated by taking into account proprieties that can affect optimization (e.g. constant problem coefficients for the convection-diffusion task, relative to integration points) according to the following formulas:

- geometrical data – $3 \cdot N_S \cdot N_Q$,
- problem coefficients – $16 \cdot N_S$,
- shape function derivatives – $N_Q \cdot N_S \cdot 3$ (write) + $N_Q \cdot N_S \cdot 3$ (read),
- local Stiffness matrix A^e – $(N_S \cdot N_S)$ (zeroing) + $N_Q \cdot N_S \cdot N_S \cdot 2$ (A^e).

The estimated number of accesses to the accelerator RAM was calculated and compared with the values obtained from the profiler. The theoretical values from Table 3 were compared in the same way with the access to shared memory. Since the numbers used in the calculations were of single precision, the number of transactions in shared memory obtained from the profiler had to be multiplied by the size of a single transaction (128 bytes) and divided by the size of a single variable (4 bytes). Similarly, in the case of DRAM, the obtained transactions had to be multiplied by $\frac{32}{4}$. The obtained results are presented in Table 9.

Table 9. The number of accesses to individual memory levels in the convection-diffusion task with the high-order elements.

	Degree of approximation		
	3	4	5
Estimated RAM	9314	29659	91510
Estimated Shared	173120	960825	4950792
Shared	166400	1659648	7489024
RAM	3999	11095	6293449

As can be seen, in the case of approximation degrees 3–4, the number of RAM accesses is reduced through the cache memory. In the case of approximation degree equal to 5, the numerical integration task turned out to be too large, and register spilling took place. In the profiler, this was marked by using "local" memory, which is a separate fragment of RAM. This resulted in a significant increase in memory usage compared to theoretical estimates. To illustrate the ratio of the number of accesses to the number of operations for the examined algorithm, its arithmetic intensity was calculated (Table 10). It is defined as the ratio of the number of operations to the number of accesses to memory.

TABLE 10. Arithmetic intensity for individual approximation levels.

	Degree of approximation		
	3	4	5
IA	204	421	4

To facilitate data reading, *Roofline* charts were created for each of the accelerators tested (Figs 6 and 7) [22]. The roofline graph defines the limiting factor

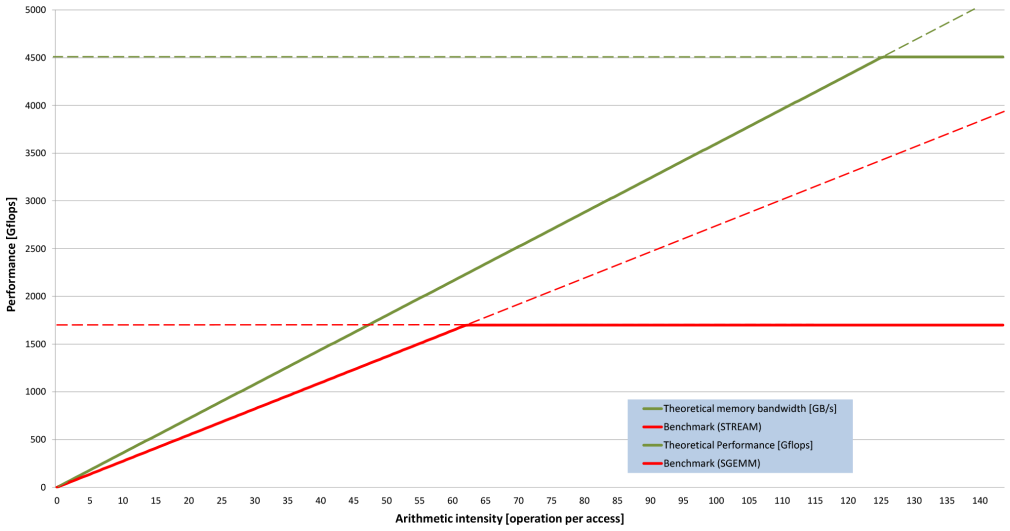

FIG. 6. Roofline chart for double-precision calculations for the Radeon R9 280X accelerator.

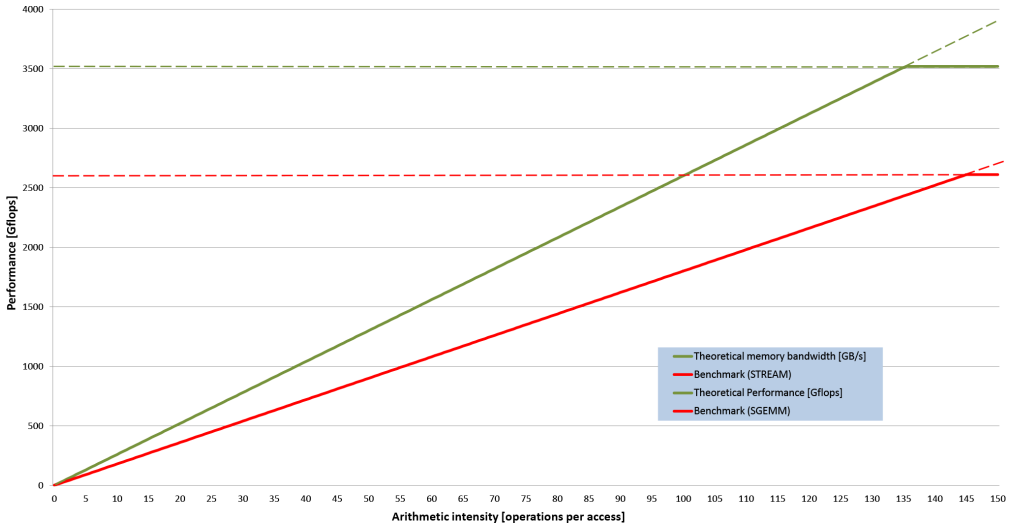

FIG. 7. Roofline chart for double-precision calculations for the Tesla accelerator.

of the algorithm's performance based on both theoretical as well as achieved in benchmarks, performance. The enclosed charts with red lines mark the performance limit obtained in the STREAM and LINPACK benchmarks, and the theoretical performance of the tested cards is marked in green. 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 performing floating-point operations. On this basis, for both examined architectures, it was found that in the case of approximation degrees 3 and 4, the algorithm performance limiting factor is the floating-point processing speed. In the case of approximation degree 5, the speed of downloading data from the accelerator memory is the same factor. Using the data from Table 7 and previously indicated performance limiting factors, estimated algorithm execution times were calculated (Table 11).

Table 11. Estimated algorithm execution times in μs.

	Degree of approximation		
	3	4	5
Tesla	0.313	1.788	43.705
Radeon	0.480	2.744	28.737

4.2. Results

The best results obtained for the examined algorithm are presented in Table 12.

Table 12. Obtained results for the numerical integration algorithm with discontinuous Galerkin approximation in μs.

	Degree of approximation		
	3	4	5
Tesla	3.130	26.790	99.805
Radeon	2.196	9.594	31.793

As can be seen from the results, the AMD card turned out to be better than the Nvidia card despite its theoretically lower performance in benchmarks. The ratio of the obtained results to the theoretical estimates is presented in Table 13.

Table 13. Percentage of achieved performance of tested cards.

	Degree of approximation		
	3	4	5
Tesla	9.99%	6.67%	43.79%
Radeon	21.86%	28.61%	90.39%

As one can see in the case of the Radeon card and the highest degree of approximation limited by memory performance, up to 90% of theoretical performance was obtained. In the case of the Tesla card, the performance obtained for both compute-bound and memory-bound problems is much worse. To analyze the obtained results, Table 14, with the algorithm parameters and the results of usage registers and card occupancy obtained from the Nvprof and CodeXL profilers, was created.

Table 14. Parameters of the numerical integration algorithm for the obtained execution times.

		Degree of approximation		
		3	4	5
Workgroup Size		64	96	128
Tesla	NENTPT	20	25	128
	NR_PARTS	2	3	1
	registers	64	70	48
	Occupancy	19%	33.2%	49.9%
Radeon	NENTPT	8	5	8
	NR_PARTS	4	12	16
	registers	253	181	101
	Occupancy	10%	10%	20%

As can be seen from the table, in the case of the Tesla card, the best results were obtained by increasing the number of entries of the stiffness matrix calculated by the threads while minimizing the size of the processed parts of the stiffness matrix. It should be noted that for the degrees of approximation *3* and *4*, the workgroup size used was much higher than the number of shape functions relative to the division of work between threads. This caused uneven load-balance and a large amount of redundant work. Stiffness matrix division strategy related to the NENTPT and NR_PARTS resulted in a decrease in the number of registers used by the thread on the Tesla card, as was expected. In the case of the tested Radeon, the compiler increased the use of registers to the highest value causing a significant decrease in the occupancy factor. Despite this, the results for the Radeon card can be considered much more satisfactory than the results for the Nvidia Tesla K20m accelerator. In the case of the highest degree of approximation and the Tesla card, the best time was obtained for a non-optimal way of work division, where each thread calculates 128 elements of the stiffness matrix. Detailed analysis of the results showed that for arrays larger than 128 bytes (32 float numbers), the compiler put them in slow local memory instead of registers, which caused a decrease in performance. However, in this case, all lost profit was recovered because the table, with a fragment of the

stiffness matrix equal in size to the size of the workgroup, was saved to the card's RAM most optimally. In the case of the Radeon card, the most significant gains were brought by a different tactic indicating the minimization of the number of entries to the stiffness matrix calculated by individual threads, while increasing the number of parts into which this matrix was divided. Low card occupancy result – reaching 20% does not significantly affect the high performance obtained, both with floating-point processing as well as memory bandwidth.

5. Conclusions

In contrast to the tasks based on the linear elements approximation, the numerical integration with higher-order elements, turns out to be too large for the tested accelerators based on GPU architecture. Our previous study showed the opposite situation for CPUs, where a higher degree of approximation allowed us to achieve better results [16]. A very positive outcome here are the results of efficient AMD Radeon R9 280X graphics card, which turned out to be matching, and in some cases, even better than the costing ten times more, Nvidia Tesla K20m accelerator. The results of this research can, therefore, be an important guide for people designing similar algorithms and facing the choice of what accelerator would be best for their applications. In the case of accelerators based on GPUs, a key factor turns out to be the minimized amount of accesses to the shared memory along with the maximization of the use of registers. This should be taken into account when choosing this type of architecture for transferring specific algorithms because the use of shared memory is also the only way to synchronize threads, which in some types of algorithms can be very problematic. Our future work will focus on improving the results for GPU-based architectures by testing their newer versions with an increased amount of resources available.

References

1. *AMD. White paper: AMD Graphics Cores Next (GCN) Architecture*, Advanced Micro Devices Inc., Sunnyvale, CA, 2012.

2. 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 Work-shops*, vol. 8806 of *Lecture Notes in Computer Science*, Springer International Publishing, pp. 158–169, 2014.

3. K. Banaś, F. Krużel, J. Bielański, Optimal kernel design for finite element numerical integration on GPUs, *Computing in Science and Engineering*, 2019 [in print].

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:] *Parallel Processing and Applied Mathematics*, R. Wyrzykowski, J. Dongarra, E. Deelman, K. Karczewski [Eds], Springer International Publishing, pp. 232–242, 2018.

5. E. Becker, G. Carey, J. Oden, *Finite Elements. An Introduction*, Prentice Hall, 1981.

6. 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.

7. P. Ciarlet, *The finite element method for elliptic problems*, North-Holland, Amsterdam, 1978.

8. P.K. Das, G.C. Deka, History and evolution of GPU architecture, *Emerging Research Surrounding Power Consumption and Performance Issues in Utility Computing*, pp. 109–135, 2016.

9. 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 (Part of Special Issue: Selected contributions of the 23rd International Conference on Parallel Fluid Dynamics ParCFD2011).

10. 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.

11. C. Johnson, *Numerical solution of partial differential equations by the finite element method*, Cambridge University Press, 1987.

12. 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*, Springer-Verlag, pp. 517–524, 2010.

13. 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.

14. F. Krużel, K. Banaś, Finite element numerical integration on Xeon Phi coprocessor, [in:] *Proceedings of the 2014 Federated Conference on Computer Science and Information Systems*, Warsaw, Poland, M.P.M. Ganzha, L. Maciaszek [Eds], vol. 2 of *Annals of Computer Science and Information Systems*, IEEE, pp. 603–612, 2014.

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

16. F. Krużel, Vectorized implementation of the FEM numerical integration algorithm on a modern CPU, [in:] *Proceedings of the 33rd International ECMS Conference on Modelling and Simulation: ECMS 2019*, 11–14 June 2019, Caserta, Italy, **33**(1): 414–420, 2019.

17. J. Mamza, P. Makyla, A. Dziekoński, A. Lamecki, M. Mrozowski, Multi-core and multi-processor implementation of numerical integration in Finite Element Method, [in:] *2012 19th International Conference on Microwave Radar and Wireless Communications*, vol. 2, pp. 457–461, 2012.

18. NVIDIA Corporation, *NVIDIAs Next Generation CUDA Compute Architecture: Kepler GK110*, Whitepaper, 2012.

19. NVIDIA Corporation, *Profiler User's Guide*, 2015.

20. R. Smith, AMD Radeon HD 7970 Review: 28nm and Graphics Core Next, Together As One, *AnandTech,* 2011, retrieved from https://www.anandtech.com/show/5261/amd-radeon-hd-7970-review on 12.09.2019

21. P. Šolín, K. Segeth, I. Doležel, *Higher-order finite element methods*, Chapman & Hall/CRC, 2004.

22. S. Williams, A. Waterman, D. Patterson, Roofline: An insightful visual performance model for multicore architectures, *Communications in the ACM*, **52**(4): 65–76, 2009.

Received September 27, 2019; revised version January 25, 2020.