Proceedings of the 2014 Federated Conference on Computer Science and Information Systems pp. 603–612
DOI: 10.15439/2014F222
ACSIS, Vol. 2

# Finite Element Numerical Integration on Xeon Phi coprocessor

Filip Krużel
Cracow University of Technology
ul. Warszawska 24, 31-155 Kraków, Poland
Email: fkruzel@pk.edu.pl

Krzysztof Banaś
AGH University of Science and Technology
al. Mickiewicza 30, 30-059 Kraków, Poland
Email: kbanas@pk.edu.pl

***Abstract*—In the present article we describe the implementation of the finite element numerical integration algorithm for the Xeon Phi coprocessor. The coprocessor is an extension of the idea of the many-core specialized unit for calculations and, by assumption, its performance has to be competitive with the current families of GPUs. Its main advantage is the built-in set of 512-bit vector registers and the ease of transferring existing codes from normal x86 architectures. However, the differences between standard x86 architectures and Xeon Phi do not guarantee performance portability. We choose an alternative approach and, instead of porting standard multithreaded code, we adapt to Xeon Phi previously developed OpenCL algorithms for finite element numerical integration. The algorithm is tested for standard FEM approximations of selected problems. The obtained timing results allow to compare the performance of the OpenCL kernels executed on the Xeon Phi and the contemporary GPUs.**

## I. Motivation

IN RECENT years there has been a noticeable increase of popularity of programming with the use of graphic cards. Their computing power allows for significant acceleration of calculations for properly implemented programs. However, there is a price to be paid, in the form of complex programming model with a complicated memory organization [1], [2]. Huge performance of GPUs can be seriously limited due to data transfers between different memory levels. Therefore, an important step is to design an algorithm that takes into account characteristics of memory access mechanisms for a particular architecture.

The development of multi-core architectures has resulted in many interesting ideas for further evolution of hardware for scientific and technical calculations. GPUs are an example of massively multi-core microprocessors with the large number of relatively simple cores equipped with small amount of memory. Another development trend in microprocessor architecture is to increase the amount and width of vector execution units within a single processor, clearly visible in recent general purpose cores [3]. The other idea was to combine the architecture of a general purpose processor with SIMD units encountered in graphics cards. The first example which achieved a fairly considerable popularity is the CellBE architecture used in Sony Playstation 3 consoles and successfully adapted to the scientific purposes as PowerXCell 8i processor [4]. In our previous studies, we focused on the development of an algorithm for the finite element numerical integration on the aforementioned processors which resulted in the development of highly efficient implementations for higher order elements of the Discontinous Galerkin approximation [5]. At the same time authors developed a version for the graphic card which was then successfully redesigned to use a standard approximation of the finite element method [6]. The resulting version of the algorithm has been tested on different types of graphic cards and the results of these tests will soon be presented in [7]. Both mentioned architectures can be considered as predecessors of the new Intel Xeon Phi architecture. This architecture combines large number of cores with wide vector units in each core. Opposite to standard GPUs, coprocessor cores are less numerous and their complexity lies in between standard, general purpose cores and simple GPU cores. As in GPUs, Xeon Phi shares the same way of memory organization and therefore all codes developed for graphic cards should be easily adapted to coprocessor architecture, but as in all types of such architectures, data movement between different levels of memory may become an issue of primary importance [8]. With the introduction of Intel Xeon Phi numerical coprocessors, there is a need to test the previously developed algorithms on the new architecture and verify whether the widely advertised adaptability of existing codes also applies to the transition from GPU to coprocessor.

## II. Numerical Integration Algorithm

Numerical integration algorithm is one of the most important parts of the finite element method codes. FEM assumes the division of the whole computational domain into elements for which the integrals, corresponding to pairs of element basis functions, are calculated and the results are collected in local, element stiffness matrices. Local load vectors are also obtained through integration of corresponding terms. Final structure of the formula to calculate the example entry to the element stiffness matrix depends on the form of the weak statement for the considered problem and can look as (1).

This work is developed within the project DEC-2011/01/B/ST6/00674 financed by Polish National Science Centre

978-83-60810-58-3/$25.00 © 2014, IEEE

$$A^e_{i_s j_s} = \int_{\Omega_e} \sum_{i_D, j_D} C^{i_D j_D} \frac{\partial \phi_{i_S}}{x_{i_D}} \frac{\partial \phi_{j_S}}{x_{j_D}} d\Omega \quad (1)$$

In the formula above, $C^{i_D j_D}$ are coefficients that depend on the problem with $i_D, j_D = 0,1,2,3$ and $\phi_{i_S}$ , $\phi_{j_S}$ are global basis functions.

In order to calculate the integrals we need to perform the change of variables, which means that the integration is made for a particular type of reference element. The transformation from the reference element to the real element is denoted by $x(\xi)$. For a reference element we use shape functions instead of global basis functions and apply one of the forms of quadrature. In our case, we used one of the most popular Gaussian quadrature. This quadrature allows for the transformation of the integral to the sum over integration points within the reference domain. Number of integration points is dependent on the required accuracy of the calculations and the type of the reference element. For $N_Q$ integration points with coordinates $\xi^Q$ and weights $w^Q$ we can transform the integral (1) to the sum (2).

$$A^e_{i_S j_S} = \sum_{i_Q}^{N_Q} \sum_{i_D, j_D} C^{i_D j_D} \frac{\partial \hat{\phi}_{i_S}}{\partial \xi^Q} \frac{\partial \xi^Q}{x_{i_D}} \frac{\partial \hat{\phi}_{j_S}}{\partial \xi^Q} \frac{\partial \xi^Q}{x_{j_D}} \det \boldsymbol{J}_{\boldsymbol{T}_e} w^Q \quad (2)$$

Where $\hat{\phi}_{i_S}$ and $\hat{\phi}_{j_S}$ are shape functions and $\boldsymbol{J}_{\boldsymbol{T}_e}$ is the Jacobian matrix of transformation $x(\xi)$.

Performance of numerical integration algorithm depends greatly on the problem being solved (weak formulation) and the approximation method employed. With the use of standard linear approximation the time of the creation of element stiffness matrix is relatively small. From the computational point of view, numerical integration algorithm consists of multiple independent calculations for each element. For this reason, the computational cost increases with growing number of elements. Calculated integrals correspond to the different terms in the weak formulation of the problem for which there is a need to define the matrix of coefficients for integration. Therefore, for the various problems we obtain different combinations of integration components for partial integrals of the test functions.

The problem dependent contribution mainly consists of the set of coefficient for numerical integration. Besides standard $i_D$ and $j_D$ indices that corresponds to the different spatial derivatives for test and trial functions, there can be also second pair of indices $i_E$, $j_E$. This indices are introduced, because for vector problems, the same approximation can be used for different unknowns in the solved system of partial differential equations (PDEs). Hence, for each combination of $i_D$ and $j_D$ there may be a small matrix of coefficients with the $N_E$ dimension equal to the number of equations in solved system of PDEs. Moreover, in the most general cases there may be different values of coefficients at each integration point. Hence for the generic numerical integration algorithm array of coefficient should be considered in a form $C^{i_Q i_D j_D i_E j_E}$ . The problem dependent indices indicate that element stiffness matrix entry is also dependent on the problem solved. Hence, we can define full equation for our computations, with the definition of $\frac{\partial \hat{\phi}_i}{\partial \xi}$ as $\psi^{i_Q i_D i_S}$ and $\det \boldsymbol{J}_{\boldsymbol{T}_e} w^Q$ as $vol^{i_Q}$ (3)

$$A^e_{i_E j_E i_S j_S} = \sum_{i_Q}^{N_Q} \sum_{i_S, j_S}^{N_S} \sum_{i_E, j_E}^{N_E} \sum_{i_D, j_D}^{N_D} C^{i_Q i_D j_D i_E j_E} \psi^{i_Q i_D i_S} \psi^{j_Q j_D j_S} vol^{i_Q} \quad (3)$$

The corresponding right hand side vector is calculated using the formula (4)

$$b^e_{i_E i_S} = \sum_{i_Q}^{N_Q} \sum_{i_S}^{N_S} \sum_{i_E}^{N_E} \sum_{i_D}^{N_D} D^{i_Q i_D i_E} \psi^{i_Q i_D i_S} vol^{i_Q} \quad (4)$$

As the conclusion of the numerical integration problem definition we provide the algorithm for computing stiffness matrices and load vectors for a set of elements of the same type and the order of approximation :

1: read quadrature data $\xi_Q$ and weights $w_Q$ for the reference element of particular type.
2: **for** $e=1$ **to** $N_e$ **do**
3: read problem dependent coefficients common for all integration points (e.g. material data, previous iterations (or time steps) degrees of freedom etc.)
4: read element geometry data for $x(\xi)$ transformation
5: initialize element stiffness matrix $A^e_{i_E j_E i_S j_S}$ and element load vector $b^e_{i_E i_S}$
6: **for** $i_Q=1$ **to** $N_Q$ **do**
7: read or calculate(on a basis of the coordination of the integration points) values of shape functions and their derivatives with respect to their local coordinates for a given integration point.
8: read or calculate jacobian matrix, its determinant and inverse.
9: calculate $vol^{i_Q}$
10: using the jacobian matrix calculate derivatives of shape functions $\hat{\phi}_{i_Q}$ with respect to the global coordinates for a given integration point
11: basing on the values of unknowns obtained through the use of $\hat{\phi}_{i_Q}$ calculate the $C^{i_Q}$ and $D^{i_Q}$ coefficients for a given quadrature point
12: **for** $i_S=1$ **to** $N_S$ **do**
13: **for** $j_S=1$ **to** $N_S$ **do**
14: **for** $i_E=1$ **to** $N_E$ **do**
15: **for** $j_E=1$ **to** $N_E$ **do**
16: **for** $i_D=0$ **to** $N_D$ **do**

17: **for** $j_D$=*0* **to** $N_D$ **do**
18: $A^e[i_S][j_S][i_E][j_E]$+= $C[i_Q][i_E][j_E][i_D][j_D]\times \psi[i_Q][i_S][i_D]\times \psi[j_Q][j_S][j_D]\times vol[i_Q]$
19: **end for** ($j_D$)
20: **end for** ($i_D$)
21: **if** $i_S$=$j_S$ **&&** $i_E$=$j_E$ **then**
22: **for** $i_D$=*0* **to** $N_D$ **do**
23: $b^e[i_S][i_E]$+=$D[i_Q][i_E][i_D]\times\psi[i_Q][i_S][i_D]$
24: **end for** ($j_E$)
25: **end for** ($i_E$)
26: **end for** ($j_S$)
27: **end for** ($i_S$)
28: **end for** ($i_Q$)
29: **end for** ($e$)

As we see from algorithm above, we can either read or compute most of the necessary components for numerical integration. This leads us to the conclusion that we can steer the amount of data sent from the memory and the amount of computations, depending on the available hardware resources and the problem solved.

In our case, we focused on the problem of convection-diffusion for $N_E$ = 0 in two cases - one with simple Laplace equation, where the coefficient matrix $C$ is sparse and coefficients appear only on the main diagonal in the case of $i_D = j_D$ (3 coefficients for stiffness matrices, one for the right hand side (RHS) vector) and a second with enhanced convection-diffusion problem for the full sixteen coefficients for stiffness matrix and four for RHS vector. Furthermore, for solving the Laplace task all coefficient were the same for all Gaussian integration points for stiffness matrix and different for the RHS vector. In the second, convection-diffusion task, all coefficients were constant for all Gauss points. For our reference elements we use prisms with 6 degrees of freedom. Our assumptions are illustrated by the data in Table I.

TABLE I.
NUMBER OF PARAMETERS FOR NUMERICAL INTEGRATION OF PRISMATIC ELEMENT

| | | |
|---|---|---|
| **$N_Q$** | | 6 |
| **$N_S$** | | 6 |
| **N_geo_dofs** | | 6 |
| **Nr_coeff_SM** | **Laplace** | 3 |
| **Nr_coeff_LV** | | 6 |
| **Nr_coeff_SM** | **Convection-diffusion** | 16 |
| **Nr_coeff_LV** | | 4 |

For optimization of the data transfer we need to decide which coefficients should be computed on the host system side and which on the accelerator side. This depends on the available resources and the type of the solved problem. Amount of data to send/store for one element can be observed in Table II.

TABLE II.
NUMBER OF DATA ELEMENTS FOR ARRAYS USED IN NUMERICAL INTEGRATION FOR PRISMATIC ELEMENTS

| | | |
|---|---|---|
| **Gauss data** | | 24 |
| **Shape functions at point** | | 24 |
| **Shape functions total** | | 144 |
| **Geometric data** | | 18 |
| **Jacobian terms at point** | | 10 |
| **Jacobian terms total** | | 60 |
| **Coefficients at point** | **Laplace** | 4 |
| **Coefticients total** | | 9 |
| **Coefficients at point** | **Convection-diffusion** | 20 |
| **Coefticients total** | | 20 |

For the GPU implementation the most important part is a proper way of data transfer organization and utilization of a limited resources. In order to port the code to the Xeon Phi coprocessor we need to reorganize the code, based on the experience gained when implementing the numerical integration for the PowerXCell 8i architecture.

## III. INTEL XEON PHI

With the development of multi-core architectures and a simultaneous trend of using the graphics cards for the calculation, an idea came up to combine several different architectures in a single hardware unit whose individual elements would be responsible for processing different type of code fragments. The first device of this type – mentioned earlier PowerXCell 8i processor was unveiled by IBM and was equipped with two core with IBM Power architecture (Power Processing Element) and a few specialized SIMD cores (Synergistic Processing Elements). Its hybrid design allowed for sending to SPE a pieces of code for which you can apply the SIMD paradigm in order to speed up calculations [9]. Truncated version of this processor has been successfully applied for commercial purposes in Sony Playstation 3 consoles and its scientific version was part of the Roadrunner computer which in 2008 exceeded the petaflops performance barrier [10]. PowerXCell 8i processor was a very big step in the development of architecture and despite the discontinuation of its production it has become a base used by other manufacturers for a hardware development for high-performance computing. At the same time Intel was working on its

line of graphics cards codenamed Larabee trying to eliminate the main disadvantage in programming CellBE or GPU architectures, which is complicated programming model. The main features of this architecture was the use of a very wide vector units (512bit), texture units taken from the GPU, the coherence memory hierarchy and compatibility with x86 architecture [11]. On the basis of this project Intel Many Integrated Core (MIC) architecture was developed, which was successfully applied in Intel Xeon Phi coprocessors [12]. These coprocessors are sold as a PCI-express cards (Fig 1.) and are equipped with its own operating system based on Linux, and depending on the version 57-61 cores with hard-

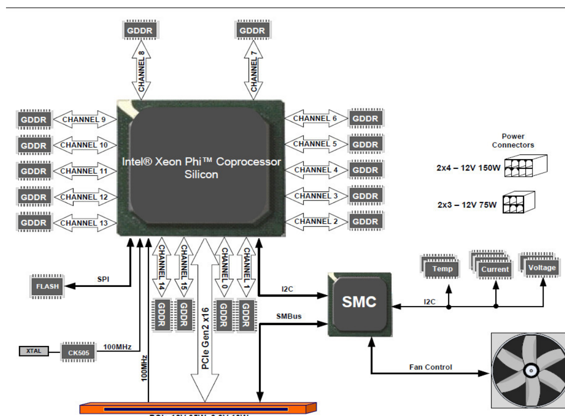


Fig 1. Xeon Phi coprocessor board schematic [13]

ware multithreading support (4 threads per core).

For testing purposes, we used 5110P coprocessor equipped with 8GB of RAM and 60 cores with a speed of 1GHz. However, in order to function properly, a single coprocessor core and 2GB of memory are reserved for the internal operating system which results in a 236 available threads and 6GB of memory for performing the calculations [14]. MIC Architecture cores design is based mainly on the Pentium architecture but it is enhanced with 512-bit vector units. The x86 compatible architecture theoretically allows for easy transfer of existing code to be used on the coprocessor with a significant increase in performance. Fig 2. shows the internal structure of the single coprocessor core.

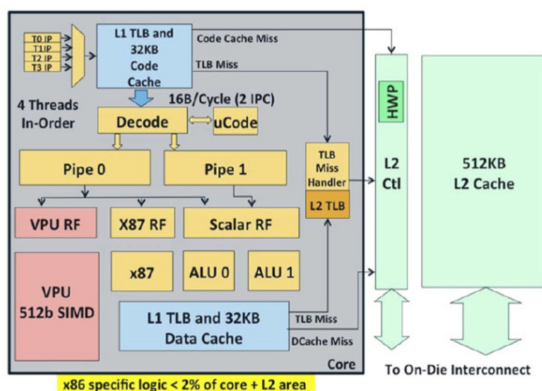


Fig 2. Single coprocessor core [15]

Every core is connected to the ultra fast interface, and thanks to a coherent cache memory, the data between cores

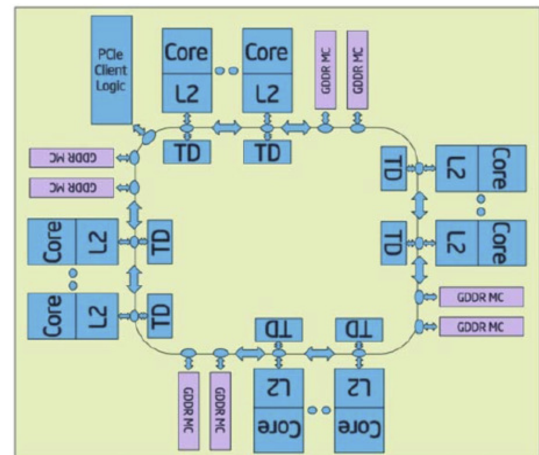


Fig 3. Xeon Phi microarchitecture [15]

are exchanged almost immediately (Fig. 3).

## IV. OpenCL Programming Model

OpenCL is a software development platform that supports many kinds of available hardware, from standard CPUs, through hybrid architectures to the GPUs [16]. In recent years this platform has gained popularity due to its portability and similarity to the previously used CUDA programming model developed by Nvidia [17]. OpenCL code is compiled and run for a given platform, representing the environment for code execution. Each platform is equipped with sets of devices of three types: CPU, GPU or Accelerator. For one host system there could be many platforms installed, varying on the vendor and supported devices. Host system runs standard code and manages the execution of an OpenCL code on device. OpenCL code is called a kernel and is written in a slightly modified C language, with the special extensions to manage different types of devices. Each device in the platform is composed of compute units, that are further divided into processing elements. Individual threads are running on processing elements with capabilities depending on the architecture of device. In OpenCL nomenclature all threads are called work items and they are grouped into work groups. This allows for direct hardware mapping for different architectures. OpenCL programming model is shown on Fig. 4. Threads within a single work group execute concurrently and can be synchronized using fast system calls. Moreover, they can share some of the data in their fast shared memory, called local memory in OpenCL nomenclature. Different work-groups are scheduled independently and have

their own resources. OpenCL execution model specifies a

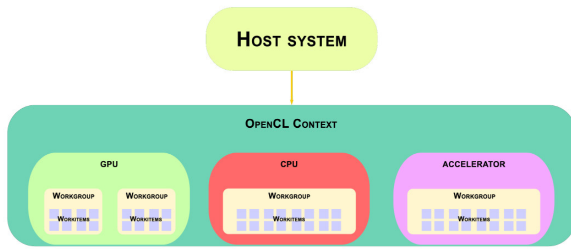


Fig 4. OpenCL programming model

set of events that has to occur in order to run a kernel.

The first phase includes initialization of the OpenCL platform, data structures and checking the available devices. Then the kernel code has to be prepared (read or compile) for running on the devices. Because of the ability of passing arguments to kernels, the space for them has to be allocated on device, before executing a kernel. Host is also responsible for preparing and allocating the space needed for variables and arrays in different types of memory that are explicitly available to programmers. All memory transactions are performed by sending a request to the OpenCL management layer. Then the requests are realized asynchronously to the host code. The same strategy is used to request kernel execution or transfer of data, back from device memory to the host memory [18].

In OpenCL the programmers have several types of memory regions explicitly available to use. Each of memory objects can be created in OpenCL memory model with different mappings to hardware resources. Individual variables defined inside kernel belong to private memory. Each thread has its own copy of each variable, and they can be stored in scalar or vector registers. Other memory regions can be assigned through specific qualifiers. Typical memory regions are divided into three types – global, constant and local. Global memory stores variables that are visible to all threads executing the kernel. Constant memory is also available for all threads but it is only accessible for reading. Variables stored in fast local memory are shared by threads in a single work-group. Because of the portability of created code OpenCL contains procedures that allows for adapting to different platforms and devices [19]. The code can query the environment to get information about many available resources. For our case we compare the available resources of all three types of devices – CPU, GPU and Accelerator. The results are presented in Table III. As we notice CPU and a Xeon Phi cards share the same amount of local and constant memory which indicates the same origins of this architectures. The Tesla K20m card used for testing our GPU implementations of numerical integration has bigger local memory size but less compute units. Hence, one can conclude that it should be slower than the other devices, but OpenCL hardware layer does not provide information on deeper division of compute units into processing elements. OpenCL in both, CPU and Intel MIC architectures treat their cores as a single compute unit but it ignores all CUDA or STREAM cores in GPUs. Our reference Tesla K20m card is equipped with 13 compute units with the Kepler architecture [20] that indicates that we have a massive amount of 192 processing elements per one compute unit, giving total 2496 cores available [21]. Despite of that all three architectures are treated as a direct opponents in the domain of high performance computing. This happens because each of these architectures has its own unique characteristics that allow for direct

TABLE III.
COMPARISON OF DIFFERENT TYPES OF DEVICES AVAILABLE IN OPENCL

| OpenCL properties | CPU | GPU | Accelerator |
|---|---|---|---|
| CL_DEVICE_NAME | Intel(R) Xeon(R) CPU E5-2620 0 @ 2.00GHz | Tesla K20m | Intel(R) Many Integrated Core Acceleration Card |
| CL_DEVICE_VENDOR | Intel(R) Corporation | NVIDIA Corporation | Intel(R) Corporation |
| CL_DEVICE_VERSION | OpenCL 1.2 (Build 67279) | OpenCL 1.1 CUDA | OpenCL 1.2 (Build 67279) |
| global memory size (MB) | 32083.020 | 4799.563 | 5773.180 |
| global max alloc size (MB) | 8020.755 | 1199.891 | 1924.391 |
| local memory size (kB) | 32 | 48 | 32 |
| constant memory size (kB) | 128 | 64 | 128 |
| cache memory size (kB) | 256 | 208 | 0 |
| cache line size (B) | 64 | 128 | 0 |
| number of compute units | 24 | 13 | 236 |

comparison between them (e.g. architecture of cores, clock speed, vector registers etc.). This determines also, that in order to fully exploit possibilities of the hardware, all existing algorithms should be adapted separately for each of the architectures. The task of numerical integration becomes non-trivial and therefore very interesting from the performance point of view.

## V. Numerical Integration on Intel Xeon Phi

For our tests we use ModFEM code - a computational framework developed for solving various scientific and engineering problems by the adaptive finite element method [22]. Due to its modular structure it allows to test different levels of the FEM. Therefore, we can easily separate the numerical integration algorithm and make a parallel versions for different architectures.

For numerical integration algorithm we have several levels of parallelism available. The chosen way of parallelisation will depend on the size of data and the number of calculations in the solved problem. Therefore, we can choose which loop from algorithm 1 should be divided. On first level we can parallelize the outermost loop over elements. Then, we can divide the loop over integration points and subsequently two inner loops over shape functions. In our previous works [5],[6] we have tested several strategies for higher order finite elements. Because of the quite big sizes of computed matrices in the problems described above, we have tested division of the inner loops over shape functions, and also a loop over Gaussian integration points. In the current case we decided to test standard approximation module. Therefore, our stiffness matrices and load vectors are quite small as we can see in Tables I and II. Hence, as the method of parallelization the most natural way of parallelizing the loop over elements is selected.

As it was mentioned above we decided to test two cases for our implementation – small Laplace and big Convection-Diffusion problem. Moreover, we have tested this problems with the use of double precision and single precision variables to check the differences between DP and SP hardware units. For our tests on graphic cards we have tried two versions of kernels – one with the stiffness matrix and load vector stored in registers and the second with the matrices stored in shared memory. In this article we will reference to them by using acronyms REG_ONE_EL for register and SHM_ONE_EL for shared memory version. Both versions has their own advantages and disadvantages. The first one allows for using very fast registers, and it saves local (shared) memory for other data, but with the limited number of registers available it can easily cause register spilling and therefore lose the efficiency of the algorithm. The second version allows for saving fast registers, but it uses a slower shared memory. Our ONE_EL versions assume that the whole element is computed by the one work-group, although one work-group can (and should) of course compute more than one element. At a first stage, host code has to compute all necessary sizes of data and thus, all needed divisions of the loops. For our reference platform we use a system equipped with NVIDIA Tesla K20m GPU, whose parameters are presented in Table III. The main difference that we must assume during the transformation of the GPU algorithm for the Xeon Phi implementation, is the size of warp/wavefront. This size (equals 32 for NVIDIA or 64 for AMD) indicates the minimal size of work-group that should be used on a given device. Due to the hardware division of every compute unit of Tesla GPU, we must also provide proper (high enough) ratio of compute unit occupancy. According to [23], Intel Xeon Phi fully utilizes its vector registers when the work-group size is set to 16. This allows for the most optimal automatic vectorization that can fully use the advantages of a very wide vector registers to store variables and use vector computations on the hardware units. Other difference lay in the use of the shared memory, because all OpenCL memory levels are mapped into Xeon Phi global memory. Hence, the use of shared and constant memory should be minimized and all possible data should be declared locally to allow proper vectorization. Of course, in the case of such a complicated algorithm there is no possibility to fit all data in registers, so we must find a proper way of preparing and storing the data. For these reasons, in opposite to GPUs SHM and REG versions that assumes only stiffness matrix allocation, we have considered more complex options for Xeon Phi.

For our tests we use a computational domain with 782336 prismatic elements. Because of the minimal work-group size that should be used for a certain architecture this indicates that we have to compute data of 785408 elements on Xeon Phi and 798720 on GPU, which in this second case is 16384 elements more than our computational domain size. While this amount seems to be very large, in fact it is only 2% more calculations and it is absolutely necessary for the proper mapping to the hardware. Due to the fact that one work-group has to compute 64 elements at once, we must divide the number of elements per compute unit by this size, so we will receive 832 work-groups that will work on 960 elements. For our Xeon Phi accelerator we have accordingly 236 work-groups with 208 elements to compute. Therefore, for GPU we have a total number of 53248 threads, while for Xeon Phi there are only 3776 threads. All precomputed values needed for calculations are shown in Table IV.

After calculations of all necessary divisions, the space needed for calculation is computed, and the data preparation phase begins. At this stage all needed buffers on the kernel side are prepared and the necessary data are computed. For our algorithm we need the following data:

- execution parameters – all values earlier computed on the host side that may be necessary for our computations – e.g. the number of elements per kernel and per work-group. This data can be stored in constant memory because we do

not need to change it. For Xeon Phi case where constatnt memory is a part of global we can assume direct read from the global memory.

TABLE IV.
PARAMETERS FOR NUMERICAL INTEGRATION OF PRISMATIC ELEMENT

| | **Xeon Phi** | **Tesla K20m** |
|---|---|---|
| **number of elements to compute** | 782336 | 782336 |
| **number of elements for kernel** | 785408 | 798720 |
| **compute units** | 236 | 13 |
| **number of elements per CU** | 3328 | 61440 |
| **number of elements per wg** | 208 | 960 |
| **wg size** | 16 | 64 |
| **number of work groups** | 236 | 832 |

- Gauss points data – all necessary Gaussian integration points data – their coordinates and associated weights can also be stored in constant memory or read from global in Xeon Phi case.
- Values of the shape functions and their derivatives on a reference element – needed for all Jacobian calculations and obtained in the same way as previous data.
- Geometric data (coordinates) for all elements – stored in global memory of the device. Here we can assume several different cases – we can copy it to local memory for each element separately (main method for REG version), copy it in coalesced way for all elements in work group (main method for SHM version) or use it directly form global memory (Xeon Phi).
- Problem dependent coefficients – send to global memory for all elements. Here we can repeat the methods from the geometric data above but for Xeon Phi we also decided to copy it directly to the registers to speed up the calculations.

After preparing the data above we can start our computations. Firstly if we are using shared and constant memory we must read all necessary execution parameters, Gauss data and values of the reference shape functions. At this stage for SHM version we have to declare local arrays for stiffness matrix and load vector. After preparation we are entering the outer loop over elements processed by a thread. According to the Table IV for Xeon Phi it is 208 elements per work-group of size 16 which indicates that each thread has to compute 13 elements, while for Tesla it will be 960 elements per work-group of size 64, that results in 15 elements per single iteration. Inside this loop we are reading all geometrical and coefficient data for one element. As it was mentioned above for SHM version we can organize this data for so-called coalescent access which theoretically enable higher performance of data transfer allowing for simultaneously read all data by all threads within one work-group. For Xeon Phi we can use global memory directly. Because of the use of the local memory on GPU after reading this data we need to establish a synchronization point with the use of a barrier, which can slow the flow of calculations a little bit in opposite to Xeon Phi and its direct global memory access. The next step includes defining (for REG and PHI versions) and zero the local stiffness matrix and load vector. Afterwards, we are entering the loop over Gauss points where we have to compute the Jacobian transformation matrix and its inverse on the basis of the previously obtained Gauss and geometrical data. After this calculations we are entering the innermost loops over the shape functions. After computing the values of shape functions and their derivatives for a real element based on their values for the reference element and earlier computed Jacobian matrix, we can compute a final entry to the stiffness matrix and load vector according to the algorithm 1. For SHM version we need to compute the right offset for storing the computed matrix in local memory. After computations for each Gauss points we can send the data to the device global memory. After all computations, the data stored are read back to the host system memory where they can be checked and used for further FEM computations. The amount of data send to and received from device global memory is shown in Table V.

TABLE V.
AMOUNT OF DATA SEND FOR NUMERICAL INTEGRATION

| **Device** | **Problem** | **Variable types** | **In data size [MB]** | **Out data size [MB]** |
|---|---|---|---|---|
| **Xeon Phi** | **Laplace** | **double** | 169,65 | 263,89 |
| | | **float** | 84,82 | 131,95 |
| | **Conv-diff** | **double** | 238,76 | 263,89 |
| | | **float** | 119,38 | 131,95 |
| **Tesla K20m** | **Laplace** | **double** | 172,52 | 268,37 |
| | | **float** | 86,26 | 134,19 |
| | **Conv-diff** | **double** | 242,81 | 268,37 |
| | | **float** | 121,41 | 134,19 |

## VI. TESTS RESULTS

For the best comparison we use the same SHM and REG algorithms for our tests on Xeon Phi. Moreover, basing on our experiments and the [23] we have prepared the more optimal version with the direct global memory use and maximization of the register usage which we refer as PHI. The performance results obtained are presented in tables VI and VII. For simplifying the comparison between our

Xeon Phi card and a reference Nvidia Tesla K20m we provide corresponding figures.

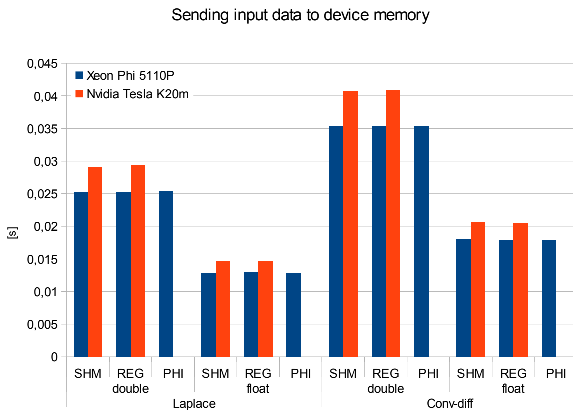


Fig 5. Sending input data to device memory

As we see from Fig. 5 the time for sending the data of comparable sizes are almost the same for Xeon Phi and Nvidia Tesla, but in all cases Xeon seems to be slightly better than Tesla.

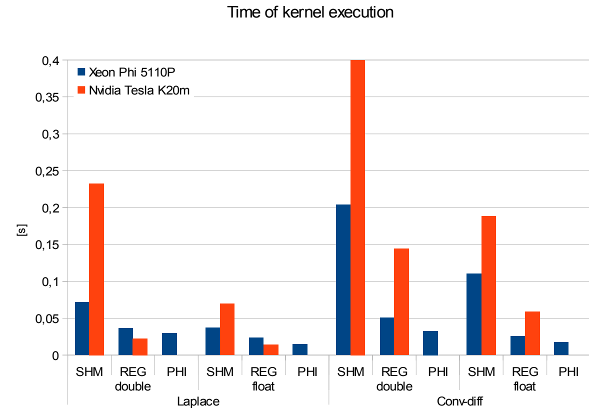


Fig 6. Time of kernel execution

Time of kernel execution (Fig. 6) shows more differences depending on the tested problem and used algorithm. A version with the use of shared memory turns out to be non optimal in our cases, but Xeon Phi is a much more faster than the Tesla card. What is more interesting we can see that the REG version with stiffness matrix stored in registers is slightly more faster than the PHI version for a small Laplace problem. All this advantage is lost when we use more complicated Convection-diffusion problem. This may indicate that Xeon Phi need quite big amount of data to fully utilize its vector registers and take advantage of it.

TABLE VI.
TEST RESULTS FOR INTEL XEON PHI

| Problem | Variable types | Kernel version | Sending Input Data to device memory | | Executing kernel [s] | Copying Output Data from device memory | |
|---|---|---|---|---|---|---|---|
| | | | [s] | [GB/s] | | [s] | [GB/s] |
| Laplace | double | SHM | 0,02531 | 6,70280 | 0,07172 | 0,85134 | 0,30998 |
| | | REG | 0,02526 | 6,71710 | 0,03618 | 0,84913 | 0,31079 |
| | | PHI | 0,02540 | 6,67933 | 0,02967 | 0,84930 | 0,31073 |
| | float | SHM | 0,01291 | 6,56829 | 0,03724 | 0,42433 | 0,31096 |
| | | REG | 0,01293 | 6,56175 | 0,02363 | 0,42609 | 0,30967 |
| | | PHI | 0,01286 | 6,59605 | 0,01492 | 0,42536 | 0,31021 |
| Conv-diff | double | SHM | 0,03538 | 6,74895 | 0,20397 | 0,85181 | 0,30981 |
| | | REG | 0,03545 | 6,73447 | 0,05066 | 0,85110 | 0,31007 |
| | | PHI | 0,03544 | 6,73691 | 0,03256 | 0,85159 | 0,30989 |
| | float | SHM | 0,01800 | 6,63413 | 0,11021 | 0,42630 | 0,30952 |
| | | REG | 0,01795 | 6,64973 | 0,02525 | 0,43486 | 0,30343 |
| | | PHI | 0,01791 | 6,66460 | 0,01755 | 0,42597 | 0,30976 |

TABLE VII.
TEST RESULTS FOR TESLA K20M

| Problem | Variable types | Kernel version | Sending Input Data to device memory | | Executing kernel [s] | Copying Output Data from device memory | |
|---|---|---|---|---|---|---|---|
| | | | [s] | [GB/s] | | [s] | [GB/s] |
| Laplace | double | SHM | 0,029054 | 5,938045 | 0,232612 | 0,094975 | 2,82569 |
| | | REG | 0,029334 | 5,881336 | 0,022464 | 0,164163 | 1,634778 |
| | float | SHM | 0,014627 | 5,897442 | 0,069616 | 0,142744 | 0,94004 |
| | | REG | 0,014691 | 5,871696 | 0,013967 | 0,047422 | 2,829597 |
| Conv-diff | double | SHM | 0,040705 | 5,965142 | 0,80109 | 0,509872 | 0,526347 |
| | | REG | 0,040874 | 5,940472 | 0,144406 | 0,094887 | 2,82831 |
| | float | SHM | 0,020599 | 5,893718 | 0,188041 | 0,046696 | 2,873589 |
| | | REG | 0,020489 | 5,925403 | 0,058719 | 0,04689 | 2,861695 |

Unfortunately, all this gained performance is lost during the copying the output data back from the accelerator to the host memory (Fig. 7). As we see on Xeon Phi the organization of the global memory has no impact on the obtained results, in opposite to the Tesla card.

Table VIII shows the obtained results in Gflops – basing on that we can see that our algorithm reaches almost 15% of theoretical peak for both double and single precision according to [24]. This can lead us to the conclusion that there is a certain margin of performance that can be used for further optimization.

TABLE VIII.
PERFORMANCE ON XEON PHI

| Problem | Variable types | Kernel version | Performance [GFLOPS] |
|---|---|---|---|
| Laplace | double | SHM | 31,92 |
| | | REG | 63,28 |
| | | PHI | 91,09 |
| | float | SHM | 61,48 |
| | | REG | 96,89 |
| | | PHI | 155,47 |
| Conv-diff | double | SHM | 18,51 |
| | | REG | 74,53 |
| | | PHI | 149,75 |
| | float | SHM | 34,26 |
| | | REG | 149,52 |
| | | PHI | 257,06 |

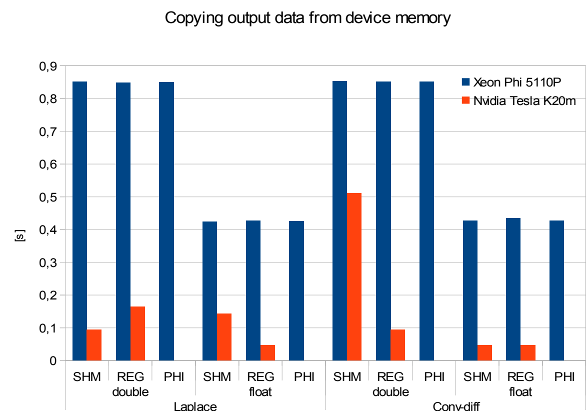


Fig 7. Copying output data from device memory

## VII. CONCLUSIONS AND FUTURE WORK

As we have shown in this article Intel Xeon Phi could be an efficient and easy to use hardware for the finite element calculations. However, it needs quite big changes in actually developed GPU codes. In our further work we will try to manually vectorize the calculations and change the data retrieving algorithm to be more efficient. The first method will allow for comparing the automatic vectorization option of the compiler and check if it fully utilizes very wide 512-bit vector registers. Second method will allow to catch up with the Tesla GPU speed of data transfer and will let to make a full comparison of the competitive architectures.

## REFERENCES

[1] NVIDIA, *CUDA C Programming Guide*, version 6.0, 2014.
[2] AMD, *AMD Accelerated Parallel Processing. OpenCL Programming Guide*, revision 2.7, 2013.
[3] Intel, *Intel 64 and IA-32 Architectures Optimization Reference Manual*, April 2012.
[4] IBM, *Cell Broadband Engine Programming Handbook Including the PowerXCell 8i Processor*, version 1.11, May 2008.
[5] F. Krużel, and K. Banaś, "Vectorized OpenCL implementation of numerical integration for higher order finite elements," Computers & Mathematics with Applications, vol. 66 (10), pp. 2030-2044, 2013, http://dx.doi.org/10.1016/j.camwa.2013.08.026
[6] K. Banaś, P. Płaszewski, and P. Macioł, "Numerical integration on GPUs for higher order finite elements", Computers & Mathematics with Applications, vol. 67 (6), pp. 1319-1344, 2014, http://dx.doi.org/10.1016/j.camwa.2014.01.021
[7] K. Banaś, and F. Krużel, "Large scale numerical integration on GPU", submitted for publication.
[8] N. K. Govindaraju, S. Larsen, J. Gray, and D. Manocha, "A memory model for scientific algorithms on graphics processors," SC 2006 Conference, Proceedings of the ACM/IEEE, Nov. 2006, http://dx.doi.org/10.1109/SC.2006.2
[9] K. Rojek, and L. Szustak, "Adaptation of double-precision matrix multiplication to the Cell Broadband Engine architecture," in: PPAM'09: Proceedings of the 8th international conference on Parallel processing and applied mathematics, Springer-Verlag, Berlin, Heidelberg, pp. 535-546, 2010.
[10] K. J. Barker, K. Davis, A. Hoisie, D. K. Kerbyson, M. Lang, S. Pakin, and J. C. Sancho, "Entering the petaflop era: The architecture and performance of Roadrunner," High Performance Computing, Networking, Storage and Analysis, pp. 1-11, Nov. 2008, http://dx.doi.org/10.1109/SC.2008.5217926
[11] L. Seiler, D. Carmean, E. Sprangle, T. Forsyth, M. Abrash, P. Dubey, et al., "Larrabee: a many-core x86 architecture for visual computing", in SIGGRAPH '08: ACM SIGGRAPH 2008 papers, pp. 1-15, 2008, http://dx.doi.org/10.1145/1399504.1360617
[12] R. Goodwins, "Intel unveils many-core Knights platform for HPC", www.zdnet.co.uk, 2010.
[13] Intel, *Intel Xeon Phi Coprocessor Datasheet*, June 2013.
[14] F. Roth, *System Administration for the Intel Xeon Phi Coprocessor*, 2013.
[15] T. P. Morgan, Intel teaches Xeon Phi x86 coprocessor snappy new tricks, www.theregister.co.uk, 2012.
[16] Khronos OpenCLWorking Group, The OpenCL Specification, Ed. A. Munshi, version 1.2, revision 19, 2012.
[17] N. Wilt, *The CUDA Handbook: A Comprehensive Guide to GPU Programming*, Addison-Wesley Professional, 2013.
[18] B. Gaster, D. Kaeli, L. Howes, P. Mistry, and D. Schaa, *Heterogeneous Computing With OpenCL*, Elsevier Science & Technology, 2011.
[19] S. Rul, H. Vandierendonck, J. D' Haene, and K. De Bosschere, "An experimental study on performance portability of OpenCL kernels", in: Application Accelerators in High Performance Computing, 2010 Symposium, Knoxville, TN, USA, p. 3, 2010.
[20] NVIDIA, "NVIDIA'a Next Generation CUDA Compute Architecture: Kepler GK110. The Fastest, Most Efficient HPC Architecture Ever Built", Whitepaper, ver. 1.0, 2012.
[21] NVIDIA, "Tesla K-Series Datasheet", Oct. 2013.
[22] K. Michalik, K. Banaś, P. Płaszewski, and P. Cybułka, "ModFem : a computational framework for parallel adaptive finite element simulations", Computer Methods in Materials Science, vol 13 (1), pp 3-8, 2013.
[23] Intel, *Intel SDK for OpenCL Applications XE 2013 R2 Optimization Guide,* 2013.
[24] Intel, *Intel Xeon Phi Product Family Performance,* revision 1.4, 12th December 2013.