Vectorized Implementation Of The FEM Numerical Integration Algorithm On A Modern CPU

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

KEYWORDS

CPU; Optimization; Parallelization; Vectorization

ABSTRACT

The main aim of this study is to answer the question: how to effectively implement the creation of the finite element stiffness matrix in parallel simulations of Finite Element Method using the full advantages of modern multiprocessors such as parallelization combined with vectorization. In this work, an efficient method for implementation of a Finite Element Method numerical integration algorithm on a modern Intel Haswell CPU architecture was developed. This algorithm was chosen, due to its non-trivial structure and the fact that its optimization is often omitted in research in favour of accelerating the other phases of FEM. Tests included two types of tasks to solve, with the use of two types of approximation and two types of finite elements. During this study, several methods for the implementation of the chosen algorithm was investigated, including Intel Cilk Plus, Intel Intrinsics and other computing techniques. Results were compared with an older Sandy Bridge architecture, showing a significant impact of vectorization and large cache on the performance of the modern CPUs. Our research gives suggestions for choosing the optimal design of algorithms and effectively using all of the features of the modern CPUs.

INTRODUCTION

The development of modern general purpose processors is associated with continuous attempts to increase their power through various treatments. The initial frequency increase and the growing number of transistors were supplemented by the expansion of the number of computing cores. This is related to Moore's law saying that the number of transistors in integrated circuits doubles every 24 months (Moore, 1965). The current trend in the use of many cores in the production of general purpose processors began with IBM introducing in 2001 Power4 processor which was the first unit with two cores in one chip (Tendler et al., 2002). At the same time, the concept of multithreading has been developed (Ungerer et al., 2002). Both concepts were combined in the "Niagara" architecture of the UltraSparc T1 processor, where four-thread cores were combined into one CPU. For home applications, the first multithreaded processor was the Pentium 4 presented in 2002, which was equipped with the Hyper-Threading technology, thanks to which the work could be divided between two virtual threads. The next step on the way to multi-core was the Intel Pentium Extreme Edition processor, which physically glued two Pentium 4 cores into one chip. This design faced with major performance problems, mainly associated with the separation of the cache memory of each core. This problem was solved in its successor, the Core 2 architecture. Simultaneously with Intel's designs, its main competitor, Advanced Micro Devices (AMD) presented its dual-core Athlon 64 X2 processors.

Since the introduction of the first multi-core processors, it has been possible to observe continuous attempts to increase the number of cores, along with reducing the size of the transistors forming them. Already in 2007, Intel unveiled a prototype Polaris processor with 80 cores, and in 2009, a 48-core Single-Chip Cloud Computer. Hardware solutions presented in these processors have been successfully implemented in newer architectures such as Sandy Bridge and Many Integrated Core (MIC).

The next step to increase the computational capabilities of modern general purpose processors is to equip them with specialised units for operations on vectors. Thanks to this, it has become possible to make fragments of programs corresponding to the Single Instruction Multiple Data (SIMD) paradigm, that is, where a single instruction is used simultaneously for many data. The first unit of this type was the MMX (MultiMedia eXtensions) introduced in 1996 in Pentium processors, allowing operations on 64-bit integers. The next units that expand the vector capabilities of modern processors were 128-bit Streaming SIMD extensions made for the single precision floating point operations. Subsequent versions of SSE introduced support for double precision numbers and added a large number of vector processing support commands. The presence of such registers also enables the packaging of independent data and its processing by the vector units.

Communications of the ECMS, Volume 33, Issue 1,
Proceedings, ©ECMS Mauro Iacono, Francesco Palmieri,
Marco Gribaudo, Massimo Ficco (Editors)
ISBN: 978-3-937436-65-4/978-3-937436-66-1(CD) ISSN 2522-2414

NUMERICAL INTEGRATION

Numerical integration is an important part of many engineering problems. One of the computing methods that is widely using numerical integration is the Finite Element Method. FEM is used to calculate the approximate solution of partial differential equations defined for a given, computational area Ω with given conditions on the margin $\partial\Omega$ (Zienkiewicz and Taylor, 2000; Becker et al., 1981). For the purpose of calculations, it is assumed that the calculated area should be divided into elements with simple geometry (tetrahedrons, cubes, prismatic elements). The whole system matrix, called the stiffness matrix, is obtained by assembling element stiffness matrices, with the entries corresponding to pairs of element basis functions. The form of the basis functions depends on the chosen approximation method. In this research, discontinuous Galerkin approximation and standard linear approximation methods were used. The first of these methods define basis functions as an extension for the entire computational area of the element shape functions ϕ^r $(r = 1, ..., N_S)$, which are the local polynomials of the order p, independently for each element. In the standard linear approximation method, the basis functions are created by the appropriate "glueing" of the shape functions defined for individual elements. In a single element, the shape functions are in the form of linear or multi-linear functions depending on the used element (tetrahedral or prismatic).

The integrals calculated for the shape function of individual elements form local stiffness matrices for each element. The elements from these matrices can appear in some places in the global stiffness matrix.

To calculate each entry of the element stiffness matrix we are using the equation (1).

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

In the next step we are changing the variables from the real to the reference element and in order to perform numerical integration we are using numerical quadrature to transform the equation (1) into the sum (2).

$$A^e_{i_S j_S} = \sum_{i_Q}^{N_Q} \sum_{i_S, j_S}^{N_S} \sum_{i_D, j_D}^{N_D} C^{i_Q i_D j_D} \times \psi^{i_Q i_D i_S} \psi^{j_Q j_D j_S} vol^{i_Q}, \qquad (2)$$

In the above formula C is an array of a problem dependent coefficients, each ψ are global derivatives of element shape functions and vol^{i_Q} is the determinant of Jacobian transformation matrix multiplied by a Gaussian quadrature weights. Indices denoted by i_Q (j_Q), i_S (j_S) and i_D (j_D) are in turn responsible for: Gaussian points, shape functions and different spatial derivatives for test and trial function. The corresponding formula for the right hand side vector, called the load vector, is calculated using (3)

$$b^e_{i_S} = \sum_{i_Q}^{N_Q} \sum_{i_S}^{N_S} \sum_{i_D}^{N_D} D^{i_Q i_D} \psi^{i_Q i_D i_S} vol^{i_Q}, \qquad (3)$$

A single element of the global stiffness matrix and load vector is the integral over the entire computational area, which can be presented as the sum of integrals over individual elements:

$$\int_\Omega ... d\Omega = \sum_{e_i} \int_{\Omega_{e_i}} ... d\Omega \qquad (4)$$

Due to the zeroing of each of the basis functions in the entire computational area except one or several elements, in most cases, elements in global stiffness matrix are equal (or near) to zero. This means that the created stiffness matrix is a rare matrix. The number of non-zero elements depends on the mesh type, the type of elements and the selected approximation. For a linear approximation, the number of non-zero elements in the row of the matrix is of the order of a dozen or so. In the case of discontinuous Galerkin approximations of higher degrees, the number of non-zero elements increases to hundreds in a row. Nevertheless, in any case, most of the matrix elements are equal to zero.

As a consequence of the formulas and conclusions above, we can present a generic numerical integration algorithm (Fig. 1).

1: read quadrature data ξ_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
5: initialize element stiffness matrix $A^e_{i_S j_S}$ and element load vector $b^e_{i_S}$
6: **for** i_Q=*1* **to** N_Q **do**
7: read or calculate(on a basis of the coordinates of integration points) the 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 the derivatives of shape functions with respect to the global coordinates for a given integration point
11: based on the values of unknowns obtained through the use of shape functions derivatives 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_D=*0* **to** N_D **do**
15: **for** j_D=*0* **to** N_D **do**
16: $A^e[i_S][j_S]$ +=$C[i_Q][i_D][j_D]\times\psi[i_Q][i_S][i_D]\times\psi[j_Q][j_S][j_D]\times vol[i_Q]$
17: **end for** (j_D)
18: **end for** (i_D)
19: **if** i_S=j_S **then**
20: **for** i_D=*0* **to** N_D **do**
21: $b^e[i_S]$+=$D[i_Q][i_D]\times\psi[i_Q][i_S][i_D]\times vol[i_Q]$
22: **end for** (j_S)
23: **end for** (i_S)
24: **end for** (i_Q)
25: **end for** (e)

Fig. 1. *FEM numerical integration algorithm*

In our previous work, we focused on transferring the selected algorithm to various types of hardware accelerators. This included some hybrid processors like PowerXCell (Krużel and Banaś, 2010; Krużel and Banaś, 2013) and AMD Accelerated Processing Unit (Krużel

and Banaś, 2015), Intel Xeon Phi numeric coprocessors (Krużel and Banaś, 2014), and GPU-based architectures (Banaś and Krużel, 2014; Banaś et al., 2018). The main problem considered in previous works was the method of managing, often limited, resources of various types of accelerators in order to obtain an efficient implementation of the numerical integration algorithm. This resulted in the development of several efficient implementations of the algorithm for individual accelerators and the creation of an automatic code tuning system (Banaś et al., 2016) in order to maximise the universalisation of the generated code. This work aims to take advantage of a series of low-level programming mechanisms in order to achieve the highest possible efficiency for the surveyed architectures of general purpose processors. This allows for obtaining the appropriate reference results for further accelerator studies. It will also check whether the mechanisms appropriate for modern processors, such as vectorisation and parallelisation will allow using the full power of the hardware for non-trivial algorithms such as numerical integration in FEM.

MODERN CPU

During the research, the author analysed the numerical integration algorithm on a CPU based on the Intel Sandy Bridge architecture and its successor with the code name Haswell. Intel Sandy Bridge was the first CPU architecture equipped with 256-bit AVX (Advanced Vector eXtension) units allowing for the processing of 4 double-precision or 8 single-precision floating point numbers at once. The main differences between the two architectures lay in the increased numbers of processing units in the newer architecture. These additional processing units are supporting AVX2 instructions - of which the most important is the so-called FMA (fused multiply-add) (5), which was previously available only in digital signal processors (DSPs) and accelerators (PowerXCell 8i).

$$a = b + c * d \tag{5}$$

The use of FMA instructions considerably accelerates algorithms that use the multiplication and addition in one step and improves their accuracy by operating on non-rounded components. In addition, in Haswell architecture, arithmetic and logic units that were operating on 128 bits were replaced with 256 bits units. A significant improvement applied in the Haswell architecture is the upgrade of the cache memory handling with an additional address generation management unit allowing two reads and one write to be performed during each clock cycle, which was only possible for the Sandy Bridge architecture with the appropriate data alignment of memory (256-bit) (Kanter, 2012). As noted, from the computing power point of view, the main improvements in the new Haswell architecture are based on adding a set of AVX2 instructions expanding the previous one, which is associated with the increase of the number of available registers from 144 to 160 per core and adding FMA instructions.

TABLE I: Tested Intel Xeon processors (Intel Corporation, 2018)

	Xeon E5 2620 (Sandy Bridge)	**Xeon E5-2699 v.3 (Haswell)**
Frequency	2 GHz	2,3 GHz
Cores	6	18
Threads	12	36
L2 Cache	2 MB	4,5 MB
L3 Cache	15 MB	45 MB

The configuration used in this work included both multi-core, multi-threading and multiprocessor configuration (2 processors per system). The characteristics of the tested processors are presented in table I.

PROGRAMMING TOOLS

ModFEM

As the primary tool, a modular software framework for engineering calculations using the Finite Element Method (ModFEM) was used (Michalik et al., 2013). Thanks to its modular structure, it allows for modification of individual fragments of FEM calculations such as approximation, mesh operations or solver solutions. The main module managed by the user is the problem module, which is used to define the FEM weak formulation and determines what other modules will be used.

For the current work, the approximation module of the ModFEM framework was modified to support optimisations made to the numerical integration algorithm. The problem modules of the analysed problems have also been modified, in order to properly prepare the data structures and compare the obtained results with and without optimisation.

Programming languages and extensions

The programming language used to create the code was C as one of the most efficient high-level languages, allowing at the same time a relatively low-level control over the hardware and how it interprets the issued orders.

The primary compiler used by the author was the Intel C Compiler (Intel Corporation, 2015) that is a part of the Intel Parallel Studio XE package and the GNU C Compiler (gcc) with extensions supporting parallelism and vectorisation. During programming of general purpose processors, the following libraries and language extensions were used:

OpenMP - application programming interface allowing the use of threads in the shared memory model. It allows for dividing work between individual threads. It has built-in structures that facilitate the division of work so that the programmer can balance the load depending on the available resources (OpenMP Architecture Review Board, 2015)

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

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

Fig. 2. Intel Cilk Plus Array notation (right) compared with standard notation (left)

Cilk Plus - C language extensions introducing the possibility of parallelisation in a way analogous to OpenMP. Also, Cilk Plus introduces an array notation (Intel Cilk Plus Array Notation), which facilitates the handling of tables for the programmer and automatic vectorisation for the compiler (Fig. 2). Thanks to language extensions, Intel Cilk Plus allows for a direct identifying fragments for vectorisation and allows for a proper aligning of tables to match vector registers.

Intel Intrinsic:

```
shp1=_mm256_mul_pd(coeff03,shp1);
shp2=_mm_mul_pd(_mm256_castpd256_pd128(coeff03),shp2);
coeff03 = _mm256_set1_pd(vol);
shp1=_mm256_mul_pd(coeff03,shp1);
load_vec1=_mm256_add_pd(load_vec1,shp1);
shp2=_mm_mul_pd(_mm256_castpd256_pd128(coeff03),shp2);
load_vec2=_mm_add_pd(load_vec2,shp2);
```

Intel Cilk Plus:

```
load_vec[0:NSHAP] += (coeff03 * shp[0:NSHAP])*vol;
```

Fig. 3. Intel Intrinsics instruction

Intel Intrinsics - a set of instructions that are directly translated to the machine language (assembler) of the processor. They allow for total control over the way of execution while giving full access to the arithmetic logic units of the processor. This allows for direct optimisation of the code without using the compiler option (Fig. 3).

SOLVED PROBLEMS

Within this work, the chosen processors were tested with two FEM problems - the Poisson equation and the generalised convection-diffusion equation. The first problem is quite simple and less computationally demanding, while the second one is more comprehensive and needs more resources (memory and compute power). For a simpler standard linear approximation, we used two types of elements - tetrahedral which are geometrically linear and prismatic with more complex geometry. For the high-order discontinuous Galerkin approximation, we were solving the convection-diffusion problem with the use of prismatic elements. This allowed for a comprehensive examination of the optimisation possibilities and the full use of the available hardware.

Arithmetic Intensity

An important element of the performance analysis is the examination of the number of accesses to different levels of the memory hierarchy during the execution of the algorithm. Such a study is possible only taking into account the details of calculations on specific architectures. In specific cases, a limited analysis is possible, allowing to characterise the relationship between the requirements of the algorithm regarding the number of operations and the number of memory accesses.

For an in-depth performance analysis of our algorithm we have computed the theoretical number of floating points operations and the minimal numbers of accesses to the main memory. It allows for calculating the arithmetic intensity of the numerical integration algorithm, expressed in the number of operations per single access to memory. The arithmetic intensity is a significant performance parameter - its value depends on whether the processor will always have the necessary data available (for high-intensity values), or its processing pipelines will wait for the data to be delivered from memory (for low-intensity values). For each of the processors, it is possible to calculate the arithmetic intensity limit value, called the processor (or machine) balance. It separates the cases when the intensity is high from the cases where data from memory are not supplied quickly enough to ensure that processing pipelines work without downtime (McCalpin, 1995). For the estimated numbers of floating point operations and accesses to memory, in the case of integration for linear approximation, the arithmetic intensity values are presented in the table II.

TABLE II: Theoretical arithmetic intensity of the numerical integration algorithm for a standard linear approximation

Poisson		convection-diffusion	
tetrahedron	*prism*	*tetrahedron*	*prism*
16,89	47,73	24,62	60,08

In the same way, the arithmetic intensity value was calculated for the convection-diffusion problem with a discontinuous Galerkin approximation and a high degree of approximation (Tab. III).

TABLE III: Theoretical arithmetic intensity of the numerical integration algorithm for a Discontinuous Galerkin approximation

Degree of approximation		
3	*4*	*5*
471,33	757,24	1392,62

Due to the fact that the estimated values of the number of arithmetic operations and RAM accesses do not take into account any optimisation, the actual arithmetic intensity value for the optimised algorithm may be slightly lower.

IMPLEMENTATION

Linear approximation

In order to obtain the highest efficiency and the fullest possible use of the available capabilities of the chosen processors, several versions of the algorithm were tested.

The first version was implemented with the use of OpenMP framework with the parallelisation of the loop over elements (line 2 in the algorithm form figure 1) and tested with automatic optimisation options which include vectorisation (-O3).

The next step was to optimise the algorithm for architecture. In order to optimise access to memory, the data tables have been aligned to the 32 bytes and, in the case of prismatic elements, they have been extended to the size which is divisible by the size of an AVX register (256-bit). In order to facilitate vectorisation, the tested version of the algorithm was implemented with the use of the Intel Cilk Plus array notation discussed earlier. At the same time, areas of code that should be

vectorised were marked by *#pragma vector* and *#pragma simd*. Using all possible CPU functions was forced on the compiler using the *-xHost* switch (Intel Corporation, 2012).

Another applied method of optimisation was the use of Intel Intrinsics instructions to direct control over the execution. This method was mainly used to compare the possibilities of classic optimisations, along with the special functions of the compiler, with manual control of the hardware.

Since the version based on Intel Intrinsics seems to be impossible for further optimisation in the current algorithm form, in the next step, the whole algorithm was reorganised. This optimisation forced vectorisation at the same level as parallelisation. Thanks to this, during one loop, four elements are processed at once (for double precision floating points). This reduced the final loop size for parallelisation, but it was connected with the increase in the number of local tables for temporary variables and the complex indexation of input and output arrays, which may affect the final performance. The modified algorithm was named *Stride* for further reference.

Execution model and performance analysis

In order to develop the performance model, both the official characteristics of the tested hardware and the results of performance tests were used. To test the used architectures, the Linpack (Dongarra, 2007) and Stream (McCalpin, 1995) benchmarks were used.

TABLE IV: Tested systems performance parameters (T-theoretical, B-benchmark)(Intel Corporation, 2018)

	Perf.	**Compute [GFlops]**	**Memory [GB/s]**
2×Xeon E5 2620 (Sandy Bridge) 24 Threads	**T**	240,0	85,2
	B	200,9	35,2
2×Xeon E5 2699 v.3 (Haswell) 72 Threads	**T**	1324,8	136,0
	B	964,8	96,7

Obtained results together with theoretical values are presented in the table IV. Based on performance obtained in benchmarks, we can calculate the aforementioned "machine balance" using the formula (6) (McCalpin, 1995).

$$\frac{\text{Floating points processing performance } [Flops]}{\text{Number of data accesses given per second}} \quad (6)$$

According to the equation, we can calculate that the machine balance for a processor with Sandy Bridge architecture is **45.66** and for Haswell architecture is **79.82**. Comparing these values with the values presented in the table II we can notice that in the case of the Sandy Bridge architecture for tetrahedral elements the limiting factor is the memory. In the case of prismatic elements and the convection-diffusion problem, the limiting factor is the efficiency of the floating point operations processing. In the case of the Poisson task, the calculated intensity is on the border of the floating point processing efficiency and the speed of RAM. In the case of Haswell architecture, the factor limiting the efficiency of the algorithm is obtaining the data from memory.

Results

The results of the tests performed for the developed algorithms and each of the processors are presented in the tables V and VI. The best results have been bolded.

TABLE V: Achieved execution times in ns for processor with Sandy Bridge architecture

SB	**Poisson**		**conv-diff**	
	tetra	*prism*	*tetra*	*prism*
O3 vec	18,72	70,90	31,77	140,40
Cilk	24,64	79,19	31,49	112,54
Intrinsic	**18,08**	54,39	31,49	95,19
Stride	18,66	**52,40**	**31,34**	**84,33**

TABLE VI: Achieved execution times in ns for processor with Haswell architecture

Haswell	**Poisson**		**conv-diff**	
	tetra	*prism*	*tetra*	*prism*
O3 vec	8,60	18,35	11,79	26,69
Cilk	**8,45**	19,84	11,65	27,92
Intrinsic	8,56	**18,33**	**11,62**	26,97
Stride	8,47	21,57	11,63	**23,58**

As we can notice the higher speedup (1,66) was obtained for the prismatic elements and the convection-diffusion problem for the Sandy Bridge processor. For the Haswell processor, small speedups were obtained - in the best case of the same most demanding task as in older architecture it was a factor of 1,13. We can notice that the speedup grows with the complication of the mesh geometry and the solved problem.

Comparing the performance between individual processors, we can notice that the Haswell processor is approximately three times faster than the processor with the Sandy Bridge architecture. From this dependence, it can be concluded that the factor limiting the efficiency is the speed of downloading data from RAM, because according to the results obtained in the benchmarks, Haswell's memory is about three times faster than Sandy Bridge. For more complicated tasks (convection-diffusion) and elements with non-linear geometry (prisms), it achieves a speedup of 5×, which indicates the use of both its computational capabilities and the faster acquisition of data from memory.

Thanks to previously calculated factors limiting the performance for individual tasks and data from table IV, we can calculate the achieved percentage of theoretical performance (from benchmarks). The result of this comparison is presented in the tables VII and VIII.

TABLE VII: The obtained percentage of the theoretical performance for the Sandy Bridge processor

	Poisson		***conv-diff***	
	tetra	*prism*	*tetra*	*prism*
O3 vec	43,70%	22,11%	37,19%	17,03%
Cilk	33,20%	19,79%	37,52%	21,25%
Intrinsic	45,23%	28,82%	37,53%	25,12%
Stride	43,84%	29,91%	37,70%	28,36%

TABLE VIII: The obtained percentage of the theoretical performance for the Haswell processor

	Poisson		***conv-diff***	
	tetra	*prism*	*tetra*	*prism*
O3 vec	34,66%	29,76%	36,51%	24,80%
Cilk	35,25%	27,53%	36,94%	23,71%
Intrinsic	34,81%	29,79%	37,02%	24,55%
Stride	35,18%	25,32%	37,02%	28,08%

As can be seen, in most cases satisfactory results have been achieved with the maximum 45% efficiency on the Sandy Bridge architecture and 37% on the Haswell architecture. Summing up the obtained results, we can conclude that the benefits of using vectorisation are dependent on the task and architecture. However, it can be expected that for the more computationally demanding tasks, they should be more substantial. It should be noted that in the majority of cases, the factor limiting performance turned out to be downloading and writing data to RAM. In connection with this, in order to fully utilise the capabilities of the tested processors (especially the very fast Haswell), it should be tested with the higher arithmetic intensity task such as numerical integration for the higher degrees of discontinuous Galerkin approximation.

Discontinous Galerkin approximation

For the case of a higher order approximation, a generalised convection-diffusion problem for prismatic elements was tested. In assumptions, it should allow for adequate saturation the tested processors with calculations and, as a result, more fully use their power. The algorithm compiled with automatic optimisations (-O3) was considered as the reference. Also, *Cilk* algorithm was modified for the higher order approximation.

Execution model and performance analysis

In order to verify the estimated number of operations for the convection-diffusion problem with a discontinuous Galerkin approximation, the Intel VTune profiler was used. It should be noted, that used hardware counters are not too accurate and often make erroneous calculations (McCalpin, 2014). In our case, however, the obtained results are very close to the estimated one, which means that the model is well developed and the high *hit ratio* is achieved. In the case of the lowest approximation level for the algorithm *Normal* and *Cilk*, the compiler was able to slightly reduce the number of operations, which was not possible in the case of the higher degree of approximation.

In the case of algorithm design, one should also take into account the vectorised data acquisition - in the case of Sandy Bridge architecture a single download is performed on 128 bits data, and in Haswell's case on 256 bits (Kanter, 2012). For the best-optimised *Cilk* algorithm, this should result in an almost two-fold decrease in the number of downloads for older architecture and four times for the newer one.

Results

The results from the tests are presented in table IX.

TABLE IX: Obtained times for executing different versions of the algorithm on tested processors (in μs)

		Degree of approximation		
		3	*4*	*5*
SB	*O3 vec*	9,88	50,32	307,83
	Cilk	**7,02**	**43,87**	**229,28**
Haswell	*O3 vec*	2,08	12,10	68,79
	Cilk	**1,24**	**7,39**	**44,19**

As can be seen from the table, the *Cilk* algorithm achieves very good accelerations in relation to automatic optimisation. In the case of the Sandy Bridge processor, the speedup reaches 1,41 and in the case of Haswell processor as much as 1,68. Comparing the performance between processors, we can see that, analogously to the case of linear approximation, for a more complex task, the Haswell processor is 4-6 times faster than the Sandy Bridge processor.

Comparing the obtained times with the theoretical performance (Tab. X), we can notice that in contrast to the standard linear approximation, the Galerkin approximation is a more computationally demanding task, which results in a more complete use of the hardware (up to 64% theoretical).

TABLE X: The obtained percentage of the theoretical performance

		Degree of approximation		
		3	*4*	*5*
SB	*O3 vec*	38,78%	42,38%	35,82%
	Cilk	54,57%	48,61%	48,09%
Haswell	*O3 vec*	38,37%	36,72%	33,38%
	Cilk	64,53%	60,10%	51,97%

As can be seen from the obtained results, the same algorithm developed for the standard linear approximation (*Cilk*) gives excellent results for the task of a much higher intensity of calculation. For the best case, it reaches approximately 600 GFlops for the Haswell processor. This indicates a good number of downloads from memory to the number of calculations ratio, and also the optimal use of memory access, performed in the form of 256-bit downloads. It also makes very good use of available AVX registers and arithmetic-logic units operating on them. In conclusion, we can indicate

that the numerical integration algorithm for more complicated problems and a higher degree of approximation is a task sufficiently demanding to use the capabilities of modern CPUs.

CONCLUSIONS

For CPUs, the key factor limiting performance in the case of the numeric integration algorithm is the speed of obtaining and writing data from and to RAM. Thanks to the implemented optimisations, it was possible to obtain very good performance results of the tested algorithm. For the task with higher arithmetic intensity, it was possible to make good use of the computational capabilities of the tested processors, reaching over 60% of the theoretical performance. It should be noted that in many cases the key to optimisations of numerical algorithms is the way of memory access. The use of programming language functions that help vectorisation and parallelisation can be a necessity in case of full use of the hardware. Other possibilities lay in the reorganisation of the algorithm using the characteristics of the given hardware.

REFERENCES

Banaś, K. and Krużel, F. (2014). Opencl performance portability for Xeon Phi coprocessor and NVIDIA GPUs: A case study of finite element numerical integration. In *Euro-Par 2014: Parallel Processing Workshops*, volume 8806 of *Lecture Notes in Computer Science*, pages 158–169. Springer International Publishing.

Banaś, K., Krużel, F., and Bielański, J. (2016). Finite element numerical integration for first order approximations on multi- and many-core architectures. *Computer Methods in Applied Mechanics and Engineering*, 305:827 – 848.

Banaś, K., Krużel, F., Bielański, J., and Chłoń, K. (2018). A comparison of performance tuning process for different generations of nvidia gpus and an example scientific computing algorithm. In Wyrzykowski, R., Dongarra, J., Deelman, E., and Karczewski, K., editors, *Parallel Processing and Applied Mathematics*, pages 232–242, Cham. Springer International Publishing.

Becker, E., Carey, G., and Oden, J. (1981). *Finite Elements. An Introduction.* Prentice Hall, Englewood Cliffs.

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

Intel Corporation (2012). *A Guide to Vectorization with Intel C++ Compilers.*

Intel Corporation (2015). *Intel C++ Compiler 16.0 User and Reference Guide.*

Intel Corporation (2018). *https://ark.intel.com/pl.* Intel products specification.

Kanter, D. (2012). Intels Haswell CPU microarchitecture. *Real World Technologies.*

Krużel, F. and Banaś, K. (2014). Finite element numerical integration on Xeon Phi coprocessor. In M. Ganzha, L. Maciaszek, M. P., editor, *Proceedings of the 2014 Federated Conference on Computer Science and Information Systems*, volume 2 of *Annals of Computer Science and Information Systems*, pages 603–612. IEEE.

Krużel, F. and Banaś, K. (2010). Finite element numerical integration on PowerXCell processors. In *PPAM'09: Proceedings of the 8th international conference on Parallel processing and applied mathematics*, pages 517–524, Berlin, Heidelberg. Springer-Verlag.

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

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

McCalpin, J. (2014). How to measure flops using vtunes? Intel Devoloper Zone Forum.

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

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

Moore, G. E. (1965). Cramming more components onto integrated circuits. *Electronics*, 38(8).

OpenMP Architecture Review Board (2015). *OpenMP Application Programming Interface*, version 4.5 edition.

Tendler, J. M., Dodson, J. S., Fields, J. S., Le, H., and Sinharoy, B. (2002). Power4 system microarchitecture. *IBM Journal of Research and Development*, 46(1):5–25.

Ungerer, T., Robic, B., and Silc, J. (2002). Multithreaded processors. *The Computer Journal*, 45(3):320–348.

Zienkiewicz, O. and Taylor, R. (2000). *Finite element method. Vol 1-3.* Butterworth Heinemann, London.

FILIP KRUŻEL is an assistant professor at the Institute of Computer Science of the Cracow University of Technology. His scientific interests focus on the high-performance computing and the use of various types of accelerators and non-standard hardware architectures. His e-mail address is: *fkruzel@pk.edu.pl* and his Webpage can be found at *http://www.pk.edu.pl/~fkruzel).*