OpenCL Performance Portability for Xeon Phi Coprocessor and NVIDIA GPUs: A Case Study of Finite Element Numerical Integration

Krzysztof Banaś[1] and Filip Krużel[2]

[1] AGH University of Science and Technology
al. A. Mickiewicza 30, 30-059 Kraków, Poland
kbanas@agh.edu.pl
[2] Institute of Computer Modelling,
Cracow University of Technology, Warszawska 24, 31-155 Kraków, Poland

Abstract. We present the performance analysis of OpenCL kernels for three recently introduced many-core accelerator architectures: Intel Xeon Phi coprocessor and NVIDIA Kepler and Fermi GPUs. We use a case study of finite element numerical integration, a practically important and theoretically interesting algorithm used in scientific computing. We design a single parametrized kernel for all three architectures and test the performance obtained in numerical tests. We indicate possible further, architecture dependent, optimizations and draw conclusions on the performance portability for different accelerator architectures and OpenCL programming model.

Keywords: OpenCL, performance portability, performance analysis, Xeon Phi coprocessor, GPU, Kepler architecture, Fermi architecture, finite elements, numerical integration.

1 Introduction

1.1 New Processor Architectures

Accelerated computer hardware plays increasingly important role in scientific computing [18]. The most popular among recently introduced hybrid systems are those equipped with cards containing either graphics processors (mainly produced by NVIDIA) or new Intel Xeon Phi coprocessors [15]. New processor and accelerator architectures pose several problems when porting existing numerical codes. One of the most important, is the problem of programming efforts required to reach satisfactory performance levels on different platforms, the subject thoroughly investigated in [3]. It turns out that the "recompile and run" approach, used successfully for classical microprocessors during the last decades of the XXth century, either cannot be used at all, due to the differences in programming and execution models, as is the case of graphics processors, or does not bring expected results, as was reported for Intel Xeon Phi processors [16].

L. Lopes et al. (Eds.): Euro-Par 2014 Workshops, Part II, LNCS 8806, pp. 158–169, 2014.

Therefore, to reach the goal of efficiently exploiting new processor designs, at least several architecture characteristics has to be taken into account explicitly. These characteristics correspond to the development trends in microprocessor design, such as e.g.:

- the increasing number of processing cores
- the increasing role of SIMD scheduling
- the use of several levels of memory hierarchy
- the presence of vector registers and vector pipelines with increasing width

The first three development directions are well visible for massively multi-core architectures of GPUs. The last direction becomes more and more indispensable for getting the proper performance of not only special coprocessor cores [8], but also cores of standard processors [17]. It would be then advantageous to have a programming environment that would allow for exploiting all the mentioned above trends in microprocessor design. The environment should also allow for certain level of performance portability and eventually lead to performance levels in the range of several tens percent of the theoretical maximum for each considered hardware.

We choose OpenCL [6] as a programming model in order to reach the goals of our research. On one hand, it is based on CUDA model [12] designed specifically for GPUs and, thanks to this, capable of exploiting their possibilities. On the other hand, due to sufficiently broad support from hardware vendors, OpenCL software development kits exist for all popular processor and accelerator architectures, and offer opportunities for relatively easy porting of developed programs. The use of OpenCL as a tool for creating portable codes was investigated in the context of classical processors and GPUs (see e.g. [14]). We extend this research by considering the architecture of Xeon Phi and the problem of finite element numerical integration.

1.2 Finite Element Software

Finite element method is one of the most popular methods for approximating partial differential equations used in many application domains of science and engineering. For each new computing architecture, investigations are performed concerning the optimal mapping of finite element calculations.

Among the papers on finite elements on GPUs, several are of special interest when considering the general problem of code portability. The first group of papers is related to efforts to create optimized versions of codes, based on abstract specifications of weak formulations and suitable, sophisticated compilers that transform specifications into optimized procedures [10]. The research on mapping of algorithms to modern computer architectures has its own significance, as the basis for code development and further investigations concerning subsequent architectures and new development tools. One of the most important papers in

the category of analysis of finite element solution procedures is [4], where several strategies for global linear system assembly are investigated and compared. Our approach is similar. We analyse the code and formulate design guidelines that can be further used in code design for particular hardware, but also for different approximation methods and problems solved.

1.3 Current Contribution

In the current paper we investigate the possibility of solving, at least partially, the problem of performance portability among different processor architectures, by using a generic OpenCL programming environment and a proper analysis and design of the code ported to new architectures.

As an algorithm for testing the development of portable OpenCL kernels we choose finite element numerical integration. We consider low order finite elements, the most popular in practical applications. The use of high order approximations was the subject of our papers [1] and [9] where investigations were conducted separately for GPUs and PowerXCell processor, respectively, the latter being a representative of architectures having specialized cores with extended vector capabilities.

In the current paper we perform an analysis of numerical integration algorithm and try to design a parametrized OpenCL kernel, that can be used for three recent accelerator architectures: Intel Xeon Phi coprocessor and NVIDIA Kepler and Fermi GPUs. We review briefly the OpenCL programming model and the finite element numerical integration algorithm. We describe the design of a parametrized kernel for numerical integration and analyse and test its performance in practical calculations. We draw some conclusions concerning further possible optimizations and porting to other processor architectures.

2 OpenCL Programming Model

We do not describe here the OpenCL programming model as it is defined in the specification [6]. Instead, we present a model that we adopt for designing the software, in some ways simplified, but including not only abstract specification of calculations, but also the characteristics of code execution on different processors.

We assume that each piece of OpenCL code for an accelerator is specified in the form of a kernel (a function written in a slightly modified variant of C99), that after compilation is run in the form of a single thread (we use the notion of "thread", as more intuitively obvious than "work-item" notion used in OpenCL).

In CUDA and OpenCL GPU programming models threads are grouped together into sets that are executed in a SIMD fashion (we do not discuss here the problem of thread divergence, the situation that we avoid in our designs). Threads in a single set are scheduled together and each thread is executed on a single SIMD lane and the whole group is scheduled for a single SIMD (vector) unit.

This model can be useful also for looking at the execution on CPU cores equipped with wide vector execution units. In fact, this is the perspective adopted by creators of the OpenCL compiler for Xeon Phi coprocessors (that contain modified Pentium CPU cores) [7]. Although the notions adopted in the compiler's description are different, in our derivations and analyses we will reserve the notion of a thread to a subsequent execution of instructions specified in the kernel code. Hence, in our model one vector instruction executed on a CPU core corresponds to a set of threads (contrary to a common perspective used e.g. in OpenMP model and the perspective in [7], where it corresponds to a single thread).

For both types of architectures, GPUs and Xeon Phi, we will use a notion of SIMD group of threads, for a set of threads forming a unit of scheduling, with individual threads executed on either separate scalar GPU cores or SIMD lanes of a vector unit in CPU cores. The notion of SIMD groups is absent in the OpenCL specification, however it is present in all CUDA and OpenCL performance considerations (as warps for NVIDIA GPUs, wavefronts for AMD GPUs and threads executing vector instructions for Xeon Phi).

Several SIMD groups form another level of thread organization, a workgroup. The role of a workgroup in our model, is to provide access to the fast memory, that is shared by all threads forming the workgroup. Apart from being units associated with shared memory allocation, workgroups in OpenCL are used for thread synchronization (mainly to arrange memory accesses).

The notion of fast shared memory (we use the notion of shared memory, as reflecting its role in OpenCL programs, instead of an OpenCL notion of local memory) is typical for GPU architectures. It is mapped to special memory modules on GPUs. The notion of shared memory does not play an important role in the Xeon Phi OpenCL model of execution. The documentation states that it is mapped to a part of global memory. Nevertheless, the memory in CPU-like architectures is cached and one may use OpenCL shared memory to rewrite the content of data structures in global memory, so that, when properly rewritten data are used by threads, the new data arrangement allow for lower access times than in original storage (assuming that caching takes place). This may resemble e.g. repackaging used for classical processors in high performance implementations of linear algebra routines [5].

The OpenCL specification assumes that the whole workgroup is scheduled for execution on a single compute unit. Compute units in GPUs are well defined hardware blocks (e.g. streaming multiprocessors for NVIDIA GPUs). For the OpenCL model of execution on Xeon Phi the hyperthreading capabilities of its cores are utilized. Each workgroup is treated as one classical thread and, hence, four workgroups are scheduled for concurrent execution on a single CPU core (since Xeon Phi cores have 4-way hyperthreading).

Finally, a set of workgroups forms the whole set of threads executing a single kernel on an OpenCL device. Workgroups are executed in a fully MIMD fashion and no dependencies can exist between different workgroups.

Apart from shared memory discussed above, we consider two other types of memory available to threads: registers and global memory. In OpenCL (and CUDA as well) there is a special type of variables (local variables) designed to be stored in registers, whenever it is possible. However, when the number of such variables exceeds the limits imposed by the hardware or programming model, the compiler may "spill" the variables to global memory. In the first generations of GPUs, such a situation resulted in serious performance deterioration, since global memory was not cached. In recent generations (and both architectures that we consider in our paper), the global memory is cached and one can expect lower penalties for register spilling.

The last aspect of programming and execution model that we mention in this brief description is the time of accesses to shared and global memory. In classical CPU programming, when creating a single thread code, the main design guideline is to increase spatial and temporal locality. For GPUs one more aspect appears, the proper organization of memory accesses for a SIMD group of threads. We try to use in our design the safest method leading to optimal memory performance (global as well as shared). Whenever threads access memory, the slowest memory present in the instruction is accessed in such a way that subsequent threads access subsequent memory locations (32 or 64-bit).

This method of accessing memory, should also work well for Xeon Phi architecture. When subsequent threads in a SIMD group access subsequent memory locations, their accesses can be grouped into a single vectorized memory access, that in turn should speed-up code execution.

3 Finite Element Numerical Integration

Finite element codes are based on integral weak statements of the problems solved [2]. To effectively solve the problems, finite element codes transform weak statements into systems of linear equations. Each entry in the system matrix is obtained as a sum of integrals, performed for individual finite elements. The most common way of calculating integrals is to use numerical integration. Hence, numerical integration forms one of indispensable parts of generic finite element codes in any application domain.

In the current paper we leave the problem of designing a generic numerical integration procedure for different approximation methods and problems solved and concentrate on two simple test cases for which we assess the performance of an OpenCL kernel on different processor architectures.

We assume that numerical integration is performed in a loop over finite elements and for each element a small dense matrix $\boldsymbol{A}^{iE}$ is created, that is further used in calculations. The algorithm of finite element numerical integration adopted for analysis in the current paper can be represented as Algorithm 1. Its essence lies in computing the entries to subsequent matrices $\boldsymbol{A}^{iE}$ (element stiffness matrices), based on the values stored, separately for each element, in arrays $\boldsymbol{c}$ (coefficients) and $\boldsymbol{\psi}$ (element shape functions with their derivatives).

Algorithm 1. The algorithm of numerical integration used in the study

```
1: read input data common to all elements processed by a thread
2: for i_E = 1 to N_E do
3:    read input data specific to a given element (including coefficients c)
4:    initialize element stiffness matrix, A^{i_E}
5:    for i_Q = 1 to N_Q do
6:       calculate derivatives of shape functions at a given integration point, ψ[i_Q]
7:       for i_S = 1 to N_S do
8:          for j_S = 1 to N_S do
9:             for i_D = 1 to N_D do
10:               for j_D = 1 to N_D do
11:                  A^{i_E}[i_S][j_S] += c[i_D][j_D] × ψ[i_D][i_S][i_Q] × ψ[j_D][j_S][i_Q]
12:               end for
13:            end for
14:         end for
15:      end for
16:   end for
17:   store A^{i_E} in global memory
18: end for
```

One of the most important characteristics of Algorithm 1 is the range of its loops. The parameters specifying the ranges are the following:

- N_E - the number of finite elements, assumed to be in the order of millions
- N_S - the number of element shape functions, in the order of several for low order approximations analysed in the current paper
- N_Q - the number of integration points within single element, in the order of several for low order approximations analysed in the current paper
- N_D - number of space dimensions plus one (in the algorithm it is assumed that arrays ψ contain the values of functions and the values of their spatial derivatives, index value 0 corresponds to the function itself, index values different from zero correspond to its derivatives). In our investigations for 3D problems, N_D is always equal to four.

Algorithm 1 takes as the input some data stored in global memory of the device performing calculations. In the current paper we do not consider the problem of transferring the input data from finite element data structures (that may reside in a different memory). For each element the main input data consist of parameters that describe the geometry of the element and the coefficients for computing matrix entries. The geometry parameters are used for calculating the derivatives of shape functions. In Algorithm 1 it is assumed that the input coefficient matrices $\boldsymbol{c}$ are used in calculations without changes.

The output of the algorithm is represented as a set of element stiffness matrices, that can be further assembled to the global matrix or used directly in matrix-free linear system solvers [13].

4 Computational Aspects of Numerical Integration Algorithm

From the computational point of view, the algorithm of finite element numerical integration is interesting as the one that combines relative simplicity with many ways for introducing different optimizations. The difficulty of optimizing it lies in the fact that PDE coefficients used in final calculations usually have different non-zero structure for different types of approximated problems and may be (e.g. for quasi-linear or non-linear problems) computed at each integration point based on input matrices $\boldsymbol{c}$ (the option not considered in the current paper). Moreover, the entries of arrays $\boldsymbol{\psi}$ are computed in different ways for different types of finite element approximations. All these facts influence significantly the optimizations that can be applied to the algorithm and the performance that can be achieved as a result [10,4].

4.1 Parallelization

In the form presented in Algorithm 1, the most suitable for parallelization is the loop over elements. The number of elements for large scale problems exceeds many times the number of threads necessary for optimal usage of computing resources (even for clusters with GPUs). When considering numerical integration alone, the algorithm is embarrassingly parallel with no dependencies between calculations for any two different elements (when considered as a part of finite element calculations, special techniques, such as colouring, has to be often applied to avoid dependencies).

In the current paper we consider only the parallelization of the loop over elements. The parallel code obtained from Algorithm 1 does not change at all, the only thing that changes is the range of element indices assigned to a thread. We pose the question how to design a portable OpenCL kernel for Algorithm 1, that would properly map to computing resources of different processor architectures. We test the performance obtained when the same, simple design guidelines are applied for different architectures.

These guidelines are the following: we try to limit the number of global memory accesses and maximize the use of registers in main calculations. We utilize the ability, offered by the OpenCL programming model, of explicitly managing the fast shared memory. However, we use shared memory with caution. For GPUs, despite the fact that it is usually one order of magnitude faster than global memory, it is several times slower than registers and, when its size for a single workgroup grows, it can limit the number of concurrently working SIMD groups and, in consequence, slow down execution by not allowing the concurrent execution of multiple SIMD groups to hide instruction and memory access latencies.

4.2 Arithmetic Operations and Register Accesses

In analysing the parallel version of Algorithm 1 we accept, in the usual way, the numbers of operations performed and the numbers of memory accesses, as

the most important characteristics of code execution. The number of operations depends on the non-zero pattern of array $\boldsymbol{c}$ (with all optimizations that it induces taken into account) and the number of additional operations performed in line 6 of Algorithm 1. The number of global memory accesses, in the version adopted in our study, is related only to the operations in lines 1,3 and 17 of Algorithm 1 (assuming that there is no register spilling to global memory). The number of shared memory accesses depends on the details of operations in line 6 of Algorithm 1, as well as the ability of the hardware to store all the data used in main calculations in line 11 of Algorithm 1 in registers.

Typical for the situation when the number of required registers exceeds the limits of the GPU hardware, is to consider the use of shared memory for some of data used in calculations or even change the algorithm [1]. In the current study, for the purpose of analysing the portability of the code, we leave to further papers more elaborate investigations considering the optimal mapping of calculations for different architectures and design the code assuming that all the data in main calculations in line 11 of Algorithm 1 reside in registers and, in a manner typical for CPU programming, relying on the compiler for the optimization of register variable usage.

4.3 Memory Accesses

Reading input data in lines 1 and 3 of Algorithm 1 is assumed as reading from global memory to shared memory. The accesses to global memory from different threads in a SIMD group are organized in an optimal way with subsequent threads accessing subsequent memory locations. In a similar way, accesses to shared memory storing read data are organized during further calculations. The accesses to global memory when writing output data are also performed in the optimal manner. In that way, not optimal memory accesses are reduced to shared memory accesses during reading of input data from global memory.

4.4 Arithmetic Intensity

Table 1 presents arithmetic intensity parameters for executing Algorithm 1 for a single prismatic 3D finite element with linear approximation and two test cases selected for the paper, associated with two example forms of arrays $\boldsymbol{c}$. The first case, corresponding to e.g. Laplace equations, has only 3 non-zero entries, all equal to one, for all 16 combinations of indices i_D and j_D and lead to 7 operations performed for calculations in lines 9–13 of Algorithm 1 (for off-diagonal stiffness matrix entries symmetry can be taken into account). The second case, corresponding e.g. to full convection-diffusion-reaction PDEs, has all 16 entries non-zero and results in more than two times more operations performed in loops over indices i_D and j_D in Algorithm 1. The relatively high ratios of the number of floating point operations to the number of global and fast memory accesses allow one to expect performance figures possible to obtain in the range of several tens of maximum performances for floating point operations.

Table 1. The ratio of the number of floating point operations to the number of global and fast (shared and constant) memory accesses for an implementation of Algorithm 1

	Type of problem:	
For single finite element:	Laplace	conv-diff
The number of floating point operations	2916	4806
The number of global memory accesses	60	74
The arithmetic intensity for global memory	≈48	≈65
The number of fast memory accesses	276	276
The arithmetic intensity for fast memory	≈10	≈17

However, the numbers in Table 1 are obtained assuming that there are no global memory accesses due to register spilling . Another factor that can limit the performance, especially in the case of GPUs, is the fact that large register and shared memory requirements, related to the optimal execution of individual statements, can induce low "processor occupancy", i.e. low number of concurrently executed SIMD groups, that in turn will not allow for fully hiding the latency of arithmetic and memory operations.

5 Numerical Experiments

5.1 Parametrized Implementation of Numerical Integration Algorithm

We design a single OpenCL kernel implementing a specific version of Algorithm 1, based on the OpenCL model of programming and the design guidelines and execution performance analysis described earlier. We parametrize the kernel with several parameters that are specified either at compile time or runtime. There are two parameters that adapt the kernel to processor architectures. The first is the size of workgroups. Based on recommendations in programming guides ([12,7]) we choose 64 threads for a single workgroup for NVIDIA GPUs and 16 threads for Xeon Phi. The second is the number of workgroups. We assume that at least 8 workgroups are assigned to each compute unit of GPUs, while there is only one workgroup for one compute unit for Xeon Phi (i.e. there are four workgroups for each of its cores).

5.2 Hardware Used for Testing

We performed numerical tests for Intel Xeon Phi coprocessor working in 5110P accelerator card and NVIDIA GPUs working in Tesla accelerator cards: Tesla M2075 for Fermi GPU and Tesla K20 for Kepler GPU. All cards are connected to systems running Linux with kernel 2.6.32. For OpenCL code development on NVIDIA GPUs, compilers and libraries from CUDA 5.5 SDK were used, while for Xeon Phi we employed compilers and libraries from Intel SDK for OpenCL

Table 2. Characteristics of accelerators used in computational experiments

OpenCL device	Fermi Tesla M2075	Kepler Tesla K20m	Xeon Phi 5110P
Number of compute units	14	13	236
Number of cores per comp. unit	32	192	1/4
Total number of cores	448	2496	59
Shared (local) memory size [KB]	48	48	32
Number of registers per comp. unit	32768x32bit	65536x32bit	32x512bit
Device memory size [MB]	5375	4800	5773
Global max alloc size [MB]	1343	1200	1924
Peak DP performance [TFlops]	0.515	1.17	1.01
Benchmark (DGEMM) performance	0.36	1.10	0.84
Peak SP performance [TFlops]	1.03	3.52	2.02
Benchmark (SGEMM) performance	0.51	2.61	1.74
Peak memory bandwidth [GB/s]	150	208	320
Benchmark (STREAM) bandwidth	105	144	165

Applications XE 3.0. Table 2 presents several characteristics of the accelerators used for testing[1].

5.3 Results

Table 3 presents the results of test runs for all three accelerators, single precision and double precision calculations and two problem types introduced above: Laplace and convection-diffusion-reaction. Several parameters are given for each run: execution time for a single finite element, performance in GFLOPS and as a percentage of the theoretical peak (the results are reported for the best of several executions). Additionally for GPUs the table contains the information provided by the *nvcc* compiler and concerning the number of registers used by each thread and the size of stack frame in global memory related to spilled loads and stores.

Several observations follow:

- the results vary significantly for both problems, different architectures and different precision of data
- for Fermi architecture, the resources are sufficient for single precision calculations (especially for Laplace test case, for which the calculated occupancy equals 33% and the performance reaches very high values around 60% of the theoretical peak), but the number of registers and the size of shared memory are too small to allow for high performance of the kernel for double precision calculations (where small occupancy and register spilling occurs)

[1] For Xeon Phi architecture the number of compute units reported by the OpenCL compiler is four times larger than the number of cores. This is related to the "hyperthreading" form of SMT for Intel x86 cores [11], where each core is seen as four "logical processors" (and each logical processor is considered as a compute unit).

Table 3. Finite element numerical integration execution characteristics and performance results for two test cases: Laplace equation and convection-diffusion PDE and three accelerator architectures: Fermi, Kepler and Xeon Phi. The same OpenCL kernel is used for all calculations, execution times are reported for one element .

	M2075 – Fermi		K20 – Kepler		5110P–Xeon Phi	
	SP	DP	SP	DP	SP	DP
Laplace						
Execution time [ns]	4.5	43.1	3.79	10.73	18.75	32.0
Performance [GFLOPS]	648	67	769	272	155	91
Performance [% of peak]	62.9	13.0	21.8	23.2	7.6	9.0
The number of registers used	63	63	92	158	–	–
The size of stack frame [B]	40	320	0	0	–	–
convection-diffusion						
Execution time [ns]	13.3	119.5	4.25	11.9	18.7	32.1
Performance [GFLOPS]	361	40	1131	404	257	150
Performance [% of peak]	35.0	7.7	32.1	34.5	12.7	14.8
The number of registers used	63	63	126	196	–	–
The size of stack frame [B]	120	616	0	0	–	–

- for Kepler architecture the results are consistent for single and double precision, while the performance is approximately 50% higher for the test case with higher arithmetic intensity (reaching more than 30% of the theoretical peak)
- the same observation holds for Xeon Phi, while the obtained performance, as the percentage of the peak, is more than two times lower than for the Kepler architecture
- the portable kernel used in the study, turned out to be the fastest for GPUs, but not the best for Xeon Phi, for which the kernel with no explicit usage of shared memory performed calculations approx. 20% faster

6 Conclusions

The analyses presented in the paper show how OpenCL notions can be used for designing a single code for such different architectures as NVIDIA GPUs and Xeon Phi. The same code, with only two parameters adapted to different architectures, was created for an example algorithm of finite element numerical integration. The performance results show that, using design process based on several simple, general optimization guidelines, it is possible to obtain for each architecture a reasonable performance, sometimes above 50% of its theoretical maximum. However, at the current stage, with the performance for several cases below 10% of the theoretical maximum, it cannot be concluded that full performance portability, if defined as obtaining high performance with a single code for all considered architectures, has been reached.

Acknowledgements. This work was supported by the Polish National Science Centre under grant no DEC-2011/01/B/ST6/00674.

References

1. Banaś, K., Płaszewski, P., Macioł, P.: Numerical integration on GPUs for higher order finite elements. Computers and Mathematics with Applications 67(6), 1319–1344 (2014)
2. Becker, E., Carey, G., Oden, J.: Finite Elements. An Introduction. Prentice Hall, Englewood Cliffs (1981)
3. Benkner, S., Pllana, S., Traff, J., Tsigas, P., Dolinsky, U., Augonnet, C., Bachmayer, B., Kessler, C., Moloney, D., Osipov, V.: Peppher: Efficient and productive usage of hybrid computing systems. IEEE Micro 31(5), 28–41 (2011)
4. Cecka, C., Lew, A.J., Darve, E.: Assembly of finite element methods on graphics processors. International Journal for Numerical Methods in Engineering 85(5), 640–669 (2011), http://dx.doi.org/10.1002/nme.2989
5. Goto, K., van de Geijn, R.A.: Anatomy of high-performance matrix multiplication. ACM Trans. Math. Softw. 34(3), 12:1–12:25 (2008), http://doi.acm.org/10.1145/1356052.1356053
6. Group, K.O.W.: The OpenCL Specification, version 1.1 (2010), http://www.khronos.org/registry/cl/specs/opencl-1.1.pdf
7. Intel: Intel SDK for OpenCL Applications XE 2013 R3. User's Guide (2013)
8. Jeffers, J., Reinders, J.: Intel Xeon Phi Coprocessor High Performance Programming, 1st edn. Morgan Kaufmann (2013)
9. Krużel, F., Banaś, K.: Vectorized OpenCL implementation of numerical integration for higher order finite elements. Computers and Mathematics with Applications 66(10), 2030–2044 (2013)
10. Markall, G.R., Ham, D.A., Kelly, P.H.: Towards generating optimised finite element solvers for gpus from high-level specifications. Procedia Computer Science 1(1), 1815–1823 (2010); iCCS 2010
11. Marr, D.T., Binns, F., Hill, D.L., Hinton, G., Koufaty, D.A., Miller, A.J., Upton, M.: Hyper-Threading Technology Architecture and Microarchitecture. Intel Technology Journal 6(1), 4–15 (2002)
12. NVIDIA: NVIDIA CUDA C Programming Guide Version 5.0 (2012)
13. Reguly, I., Giles, M.: Finite element algorithms and data structures on graphical processing units. International Journal of Parallel Programming, 1–37 (2013), http://dx.doi.org/10.1007/s10766-013-0301-6
14. Rul, S., Vandierendonck, H., D'Haene, J., De Bosschere, K.: An experimental study on performance portability of opencl kernels. In: Application Accelerators in High Performance Computing, 2010 Symposium, Papers, Knoxville, TN, USA, p. 3 (2010)
15. Top500, http://www.top500.org
16. Wienke, S., an Mey, D., Müller, M.S.: Accelerators for technical computing: Is it worth the pain? A TCO perspective. In: Kunkel, J.M., Ludwig, T., Meuer, H.W. (eds.) ISC 2013. LNCS, vol. 7905, pp. 330–342. Springer, Heidelberg (2013)
17. Williams, S., Waterman, A., Patterson, D.: Roofline: An insightful visual performance model for multicore architectures. Commun. ACM 52(4), 65–76 (2009), http://doi.acm.org/10.1145/1498765.1498785
18. Yuen, D., Wang, L., Chi, X., Johnsson, L., Ge, W., Shi, Y. (eds.): GPU Solutions to Multi-scale Problems in Science and Engineering. Springer (2013)