Available online at www.prace-ri.eu

Partnership for Advanced Computing in Europe

Porting FEASTFLOW to the Intel Xeon Phi: Lessons Learned

Ioannis E. Venetis[a], Georgios Goumas[b]*, Markus Geveler[c], Dirk Ribbrock[c]

[a]University of Patras, Greece
[b]National Technical University of Athens – NTUA, Greece
[c]Technical University of Dortmund – TUD, Germany

Abstract

In this paper we report our experiences in porting the FEASTFLOW software infrastructure to the Intel Xeon Phi coprocessor. Our efforts involved both the evaluation of programming models including OpenCL, POSIX threads and OpenMP and typical optimization strategies like parallelization and vectorization. Since the straightforward porting process of the already existing OpenCL version of the code encountered performance problems that require further analysis, we focused our efforts on the implementation and optimization of two core building block kernels for FEASTFLOW: an *axpy* vector operation and a sparse matrix-vector multiplication (*spmv*). Our experimental results on these building blocks indicate the Xeon Phi can serve as a promising accelerator for our software infrastructure.

1. Introduction

The difficulties in further optimizing execution speed of single processors have recently driven hardware vendors to produce multi-core systems. For example, increasing further the clock speed leads to large increases in power consumption and heat dissipation issues. Furthermore, the ever increasing numbers of available transistors are difficult to be exploited in existing architectural features of processors to improve their execution speed. In this context, Intel has recently released the Xeon Phi coprocessor. It consists of 60 cores, with some features not being typically found on previous generations of Intel processors.

This work is part of an approach to develop efficient, reliable and future-proof numerical schemes and software for the parallel solution of partial differential equations (PDEs) arising in industrial and scientific applications. Here, we are especially interested in technical flows including Fluid-Structure interaction, chemical reaction and multiphase flow behavior which can be found in a wide variety of (multi-) physics problems. These are incorporated in the software package FEASTFLOW [1] which is both a user oriented as well as a general purpose subroutine system for the numerical solution of the incompressible Navier-Stokes equations in two and three space dimensions. We use a paradigm we call 'hardware-oriented numerics' for the implementation of our simulators: Numerical- as well as hardware-efficiency are addressed simultaneously in the course of the augmentation process. On the one hand, adjusting data-structures and solvers to a specific domain-patch dominates the asymptotic behavior of a solver. However, utilising modern multi- and many-core architectures as well as hardware accelerators such as GPUs has become state-of-the-art recently. Here, we aim at evaluating the Intel Xeon Phi coprocessor as a backend-expansion for these codes.

Our attempts have focused on two directions: programmability and efficiency. Regarding programmability, we have evaluated the potential of OpenCL to serve as “quick-and-dirty” back-end of the FEASTFLOW framework

---

* Corresponding author. *E-mail address*: goumas@cslab.ece.ntua.gr

for the Intel Xeon Phi. Our initial attempts were rather discouraging as the low performance of the straightforward OpenCL version porting indicates that elaborate optimization and fine-tuning is required (a task left for future work). In the sequel, we focused on native programming models and tools including OpenMP with the use of Xeon Phi intrinsics and worked on two key building blocks for FEASTFLOW, i.e. scalar-vector product (*axpy*) and sparse-matrix vector multiplication (*spmv*) [2, 3, 4, 7] . Our experimental results indicated that the Intel MIC architecture can serve as a promising accelerator for our software infrastructure.

The rest of this paper is organized as follows: In section 2 we provide some background information on the FEASTFLOW framework and the Intel Xeon Phi architecture. In section 3, we report our experiences with OpenCL, Pthreads and OpenMP as programming models to port FEASTFLOW and then we briefly describe our implementation of the two building blocks in section 4. Experimental results are presented in section 5. We conclude this paper in section 6.

2. Background

Feastflow

The program package FEASTFLOW [1] is both a user oriented as well as a general purpose subroutine system for the numerical solution of the incompressible Navier-Stokes equations in two and three space dimensions. It has been developed for education and research as well as industrial applications. Functionality comprises the description of the partial differential equations and the Finite Element Method (FEM) as well as domain decomposition, mesh generation, Finite Element shape functions, assembly and (parallel) solution. In addition to this CFD part of the package, the subsystem FEAST [8] is designed for both, efficiency in terms of good scalability across a wide variety of MPI implementations and cluster computers, and efficiency targeting single-node performance with platform-specific SparseBanded BLAS implementations for good memory performance and cache efficiency. This module also exploits the tremendous floating point performance and memory bandwidth of modern graphics processors (GPUs) as numerical co-processors. The solvers are based on a combination of domain decomposition and parallel multigrid and features include automatic partitioning and loadbalancing. Finally the subsystem HONEI [9] is a collection of libraries offering a hardware oriented approach to numerical calculations. Here, hardware abstraction models are developed and applications written on top of these can be executed on a wide range of computer architectures such as common CPUs, GPUs and the Cell processor and clusters thereof. The runtime components implemented so far aim at heterogeneous compute nodes in clusters using MPI in combination with PThreads and/or accelerators.

Intel Xeon Phi architecture

The architecture of Intel Xeon Phi is shown in Figure 1 and key features of the architecture are summarized in Table 1. Intel Xeon Phi is a many-core coprocessor based on the Intel MIC architecture. It consists of x86-based cores, caches, Tag Directories (TD), and GDDR5 Memory Controllers (MC), all connected by a high speed bidirectional ring. Each core is based on an in-order architecture, where instructions are fetched and executed in the order they appear in an application. This contrasts to previous designs of Intel, where processors were based on out-of-order designs, with the order of instructions executed rearranged internally by the processor. In addition, every 240 core supports execution of up to 4 hardware threads, leading to support for the simultaneous execution of up to 244 threads in an application. Furthermore, special attention is paid to vectorization, i.e., the execution of a single instruction on multiple data at the same time.

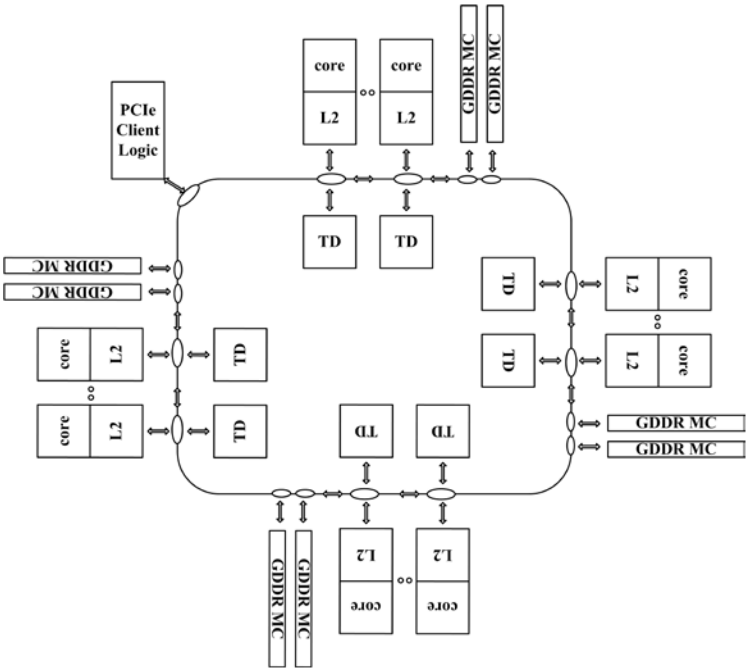

Figure 1: Intel Xeon Phi architecture.

Core	– *In-order architecture, based on the Intel processor family.* – *Support for 4 hardware threads.* – *Support for 32-bit and 64-bit execution environments.* – *No support for previous vectorization extensions (MMX, SSE, SSE2, SSE3, SSE4, SSE4.1, SSE4.2 and AVX).* – *New vectorization extension to support the dedicated 512-bit wide vector floating-point unit (VPU) that is provided for each core.* – *Special support for square root, power, exponent, reciprocal, scatter, gather and streaming store operations.* – *Execution of 2 instructions per cycle on the U-pipe and the V-pipe (restrictions apply to several instructions on which pipeline they can execute).* – *L1 I-cache and D-cache (32KB each).*
Vector Processing Unit (VPU)	– *Execution of 16 single precision, 8 double precision or 16 32-bit integer operations per cycle.* – *Support for Fused Multiply-Add instruction.* – *32 vector registers, each 512-bit wide.*
Core Ring Interface (CRI)	– *Connects each core to a Ring Stop (RS), which in turn connects to the interprocessor core network.* – *Hosts the L2 cache and the Tag Directory (TD).*
Ring	– *The Xeon Phi has 2 rings, each sending data to one direction.* – *Includes component interfaces, ring stops, ring turns, addressing and flow control.*
SBOX	– *Gen2 PCI Express client logic.* – *System interface to the host CPU or PCI Express switch.* – *DMA engine.*
GBOX	– *Coprocessor memory controller.* – *Contains the FBOX (interface to the ring interconnect), MBOX (request scheduler) and PBOX (physical layer that interfaces to the GDDR devices).* – *8 memory controllers, each supporting 2 GDDR5 channels.*
Performance Monitoring Unit (PMU)	– *Allows data to be collected from all units in the architecture.*

Table 1: Basic architectural features of the Intel Xeon Phi accelerator.

3. Programming models

Programmability and performance portability are key concepts when implementing software infrastructures for multicore, manycore and massively parallel systems. To this direction, FEASTFLOW supports an OpenCL backend. In this section we report preliminary experiences with the execution of the OpenCL backend on the Xeon Phi.

OpenCL application runs

When incorporating the Xeon Phi board on a single node with FEASTFLOW OpenCL backends, we were unable to obtain a reasonable application performance. Even compared to single threaded CPU code, our OpenCL backends performed worse. Several efforts to investigate the issue on the application level have not shed adequate light to the problem. Hence, we incorporated a building-block kernel (see below) for further investigation.

Single node single- and multicore (PThreads, OpenCL) runs on the host CPU / OpenCL microbenchmarks on the Intel Xeon Phi

In order to examine the above-mentioned problem, we concentrated on micro-benchmarking a single BLAS1 like operation which is a core building block in the FEASTFLOW framework computing $r = a*x + y$ with vectors r, x, y. For convenience, we compare fully optimized versions of the code on the CPU with handcrafted SSE intrinsic code, additionally parallelized on the shared memory level using OpenMP versus the OpenCL version on the Xeon Phi. All vectors are aligned and in order to rule out granularity effects, we measure with vector length of 64^4 (double precision) and take the average of ten runs each comprising 20 calls to the operation kernel. The results of the measurements are shown in the figure below.

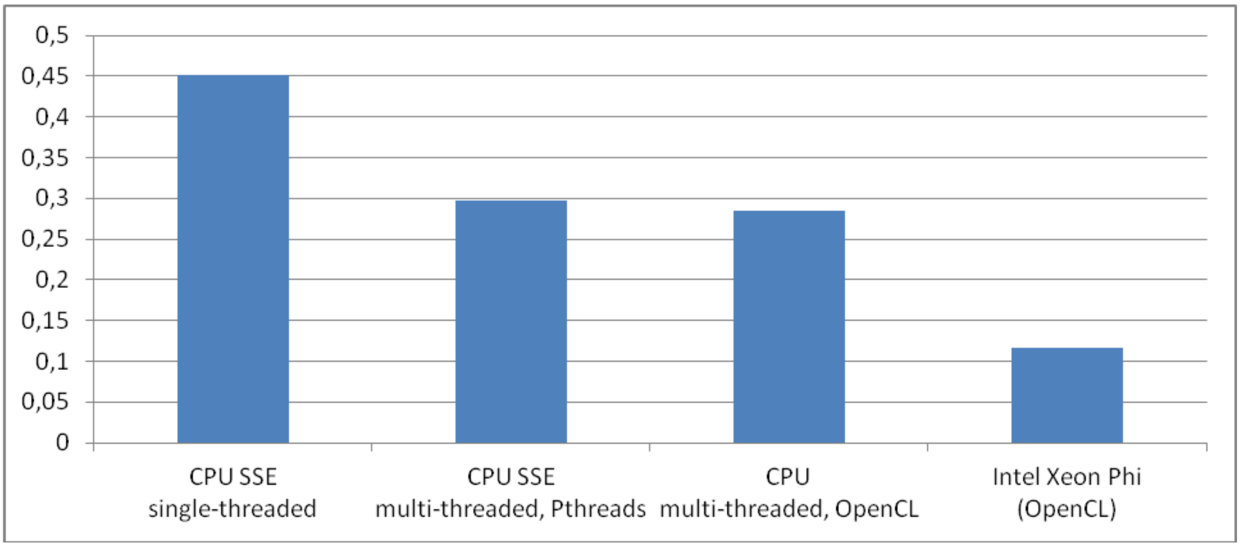

Figure 2: Execution times (in sec) for building block *axpy* ($r = a*x + y$, vector size: 64^4).

Results indicate that the irregularity obtained on the application level is not present in this micro-kernel benchmark. Roughly 5 to 6 times the performance of the multi-threaded, SSE optimised CPU version can be reached with the OpenCL backend on the accelerator, which is a reasonable speedup given the bandwidth difference between the architectures and a bandwidth-bound code. However, this brought no light into the issue with the CFD application and the subject has to be further examined in future work.

4. Implementation of building blocks

To proceed with the implementation of FEASTFLOW on the Xeon Phi coprocessor, we followed a bottom-up approach, focusing on two key kernels of the infrastructure that can be used to build full solvers like CG or GRRES and their variations. *axpy*, a typical BLAS1 operation computing $r = a*x + y$ and *spmv* computing a matrix-vector multiplication $y = A*x$ with matrix A being sparse. The kernels are implemented in C and parallelized with OpenMP, while vectorization was applied both for the multicore node architecture and the Xeon Phi accelerator. Table 2 summarizes the code versions implemented.

		Parallel	**Compiler (pragma) vectorization**	**Manual vectorization**
axpy	**Sandy Bridge**	√ *(OpenMP)*		√ *(AVX)*
	Xeon Phi	√ *(OpenMP)*		√ *(VPU)*
spmv	**Sandy Bridge**	√ *(OpenMP)*	√	√ *(AVX)*
	Xeon Phi	√ *(OpenMP)*	√	√ *(VPU)*

Table 2: Implemented code versions for the building blocks *axpy* and *spmv*.

5. Experimental results

In this section we report our experimental results for the two building blocks using the versions described in the previous section. We used one node equipped with Xeon Phi coprocessor from the Eurora Supercomputer in CINECA. The host node includes 2 eight-core Intel(R) Xeon(R) CPU E5-2658 @ 2.10GHz (Sandy Bridge) with 16 GB RAM and 2 Intel Xeon Phi 5120D accelerators with 8 GByte RAM.

Results for *axpy*

Table 3 summarizes our experimental results for *axpy* both on Sandy Bridge and Xeon Phi. The vector size is $64*1024^2$. Lowest times are reported over the number of threads tested. This was achieved for 8 threads in Sandy Bridge and 256 threads in Phi. As expected, given the memory-bound nature of the kernel, the main node system fails to provide considerable speedup, reaching up to *2.31* on 8 cores (and not improving up to 16 cores). For the same reason, vectorization does not provide any significant boost and is capable only of increasing the total speedup to 2.43. On the other hand, the Phi execution behavior is much more promising: Both parallelization and vectorization have a significant impact on performance leading to an aggregate speedup of: a) *41.88* over the serial, unvectorized Phi version, b) *11.4* over the serial Sandy Bridge version and c) *4.68* over the best Sandy Bridge version. Thus, regarding the *axpy* kernel, Phi provides an efficient alternative to the main multicore node both in terms of scalability and overall execution time, especially if energy efficiency is also taken into consideration.

	Sandy Bridge	**Xeon Phi**
serial	*0.285*	*1.047*
parallel	*0.123*	*0.369*
vector	*0.263*	*0.455*
parallel + vector	*0.117*	*0.025*

Table 3: Execution times (sec) for *axpy* on Sandy Bridge and Phi for various versions.

Results for *spmv*

In this set of experiments we evaluated the *spmv* versions reported in Table 2 for two sparse matrices collected from the University of Florida Sparse Matrix Collection [6]. Their characteristics are reported in Table 4. Matrices are stored in the CSR format [7]. Each *spmv* benchmark averages 128 iterations of the matrix-vector multiplication.

name	number of rows	number of non-zero elements	size in memory (MB)	domain
xenon2	*157,464*	*3,886,688*	*44,85*	*material*
Hamrle3	*1,447,360*	*5,514,242*	*68,6*	*circuit*

Table 4: Matrices used.

In our first set of experiments we evaluated the scheduling impact as implemented natively in OpenMP with the static, dynamic and guided options accompanied with the selection of the chunk size. Figures 3-6 demonstrate the execution times for various numbers of threads. In each case the version of the code with the overall lowest execution time is shown. Several chunk sizes per strategy were tested and the best result is reported. Two interesting observations can be made: a) As *spmv* is input-specific and can potentially lead to imbalanced executions, sweeping over the search space of scheduling algorithms and chunk sizes can provide different results. b) There is no "best" option of the scheduling configuration and this should be tuned separately for architecture and input matrix.

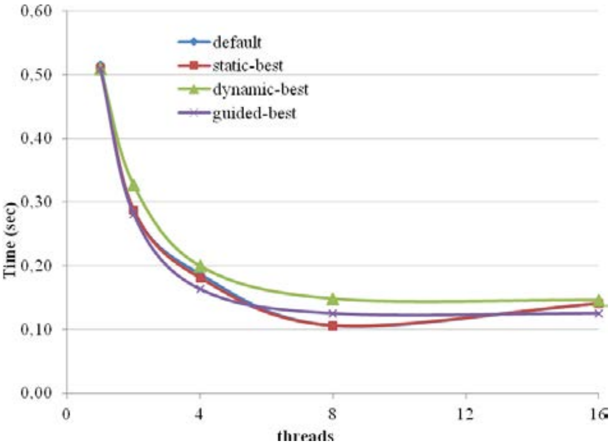

Figure 3: The impact of scheduling on Sandy Bridge for the xenon2 matrix (pure OpenMP version).

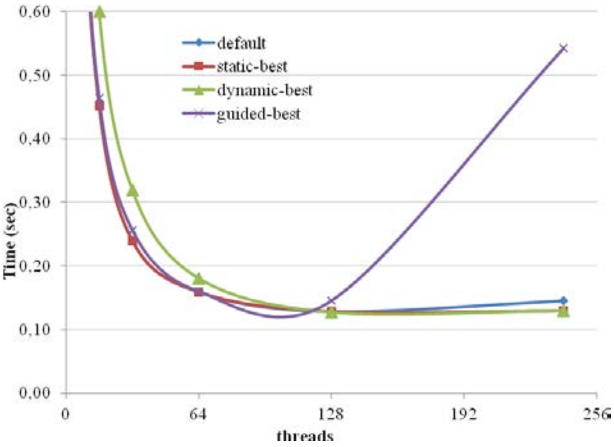

Figure 4: The impact of scheduling on Phi for the xenon2 matrix (OpenMP + manual vectorization version).

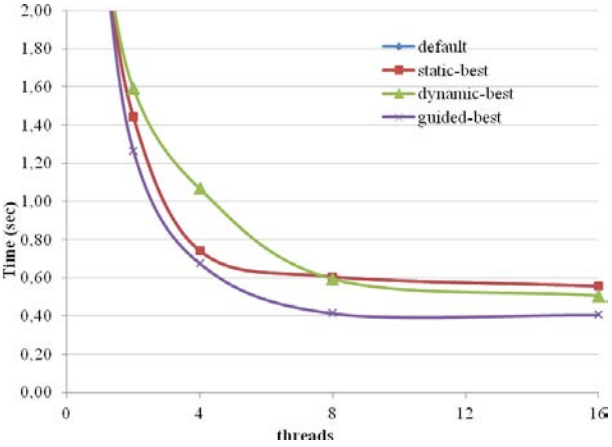

Figure 5: The impact of scheduling on Sandy Bridge for the Hamrle3 matrix (OpenMP+pragma vectorization version).

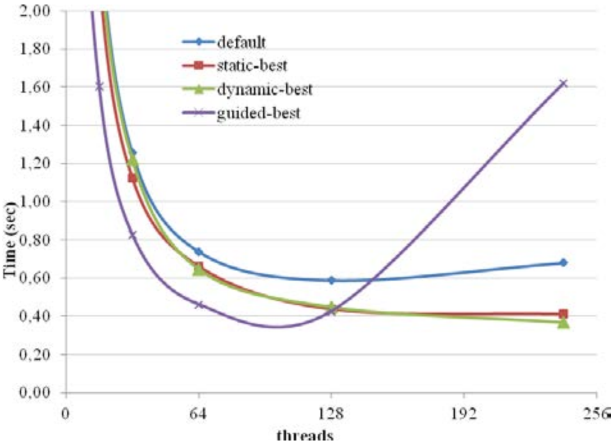

Figure 6: The impact of scheduling on Phi for the Hamrle matrix (OpenMP + manual vectorization version).

In our second set of experiments we evaluated the impact of parallelization and vectorization on the Intel Xeon Phi coprocessor and compare the results with the best performing version on Sandy Bridge. Note that, as reported in the literature [3], *spmv* is a heavily memory-bound kernel in traditional multicore architectures and thus both multithreading and vectorization (targeting the computational part of the kernel), are not expected to have a dramatic positive impact in performance in Sandy Bridge. Indeed, our results confirm that vectorization has a minimal impact on performance, as pure OpenMP was the best option for matrix xenon2, while the compiler (pragma) vectorization offered a marginal improvement for matrix Hamrle3.

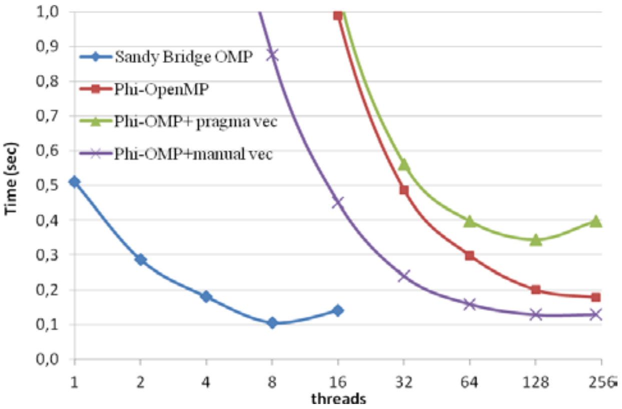

Figure 7: Comparison of Sandy Bridge (best) and Phi versions for the xenon2 matrix.

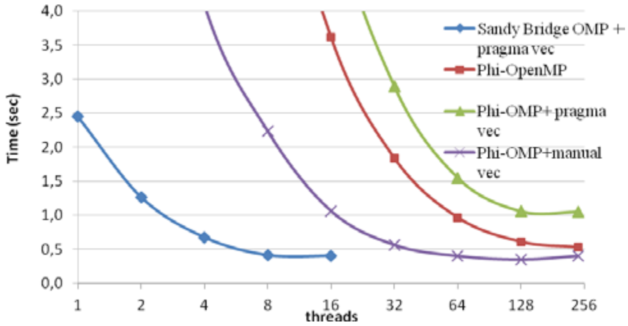

Figure 8: Comparison of Sandy Bridge (best) and Phi versions for the Hamrle3 matrix.

On the other hand, vectorization has a significant impact on the Phi system, provided this is applied manually. Compiler vectorization has been ineffective for *spmv* and led to higher execution times than pure multithreading. Regarding the comparison between the two execution platforms, we report a draw for the two matrices: Sandy Bridge outperforms Phi for matrix xenon2 by *~20%*, while Phi outperforms Sandy Bridge for matrix Hamrle3 by *~15%*. Again here, it would be interesting to include energy efficiency in the assessment of the two platforms, a task that is left for future work. Overall, *spmv* is an extremely challenging kernel as its performance is memory-bandwidth bound both for Sandy Bridge and Phi. Additionally, the performance is also input dependent [3] with the sparsity pattern of the matrix affecting issues like load balancing and the impact of vectorization [4].

6. Conclusions

In this paper we reported our experiences on porting the FEASTFLOW application on the Intel Xeon Phi coprocessor platform. Initially, we attempted to utilize the already existing OpenCL version of the code directly on the Phi, but the performance results were discouraging. In future work we aim to shed more light on this aspect. In the sequel we implemented two key building blocks for FEASTFLOW, namely *axpy* and *spmv*. Although memory bandwidth is a major limitation for both kernels, our preliminary results are interesting and render Phi a promising platform for our software infrastructure. Here we report a few lessons learned from our experimentation: a) The Xeon Phi coprocessor was able to provide significant performance improvement for *axpy* reaching a 4.68 speedup over the best Sandy Bridge implementation. b) In *spmv* Phi was not able to provide spectacular improvements over Sandy Bridge, leading to a *~15%* improvement in one of the two matrices used (Sandy Bridge outperformed Phi in the other matrix by *~20%*). Yet, *spmv* is an extremely challenging kernel, memory bound and input dependent and further experimentation is required with a wider set of matrices to draw safer conclusions. c) searching the search space for scheduling strategies and parameters for *spmv* is meaningful. d) energy efficiency is a parameter that should be considered for a better assessment of the accelerator platform (left for future work).

References

[1] S. H. M. Buijssen, H. Wobker, D. Göddeke, S. Turek: FEASTSolid and FEASTFlow: FEM applications exploiting FEAST`s HPC technologies, Nagel, W.; Resch, M., Transactions of the High Performance Computing Center Stuttgart (HLRS) 2008, 425--440, High Performance Computing in Science and Engineering 2008, Springer, doi: 10.1007/978-3-540-88303-6/_30, 2008

[2] S. Williams, L. Oliker, R. W. Vuduc, J. Shalf, K. A. Yelick, J. Demmel: Optimization of sparse matrix-vector multiplication on emerging multicore platforms. SC 2007: 38

[3] G. Goumas, K. Kourtis, N. Anastopoulos, V. Karakasis and N. Koziris, "Performance Evaluation of the Sparse Matrix-vector Multiplication on Modern Architectures," The Journal of Supercomputing, Vol 50, No 1, 2009
[4] X. Liu, M. Smelyanskiy, E. Chow, and P. Dubey, "Efficient Sparse Matrix-Vector Multiplication on x86-based Many-core Processors," In 27th International Conference on Supercomputing (ICS), Eugene, USA, June 10-14, 2013.
[5] Eurora User Guide, http://www.hpc.cineca.it/content/eurora-user-guide.
[6] T. Davis, 1997. University of Florida sparse matrix collection. NA Digest 97, 23, 7.
[7] R. Barrett, M. Berry, T. F. Chan, J. Demmel, J. Donato, J. Dongarra, V. Eijkhout, R. Pozo, C. Romine, H. V. der Vorst, 1994. Templates for the Solution of Linear Systems: Building Blocks for Iterative Methods. SIAM, Philadelphia.
[8] Finite Element Analysis and Solution Tools, http://www.feast.tu-dortmund.de
[9] HONEI: Hardware oriented numerics, efficiently implemented http://www.honei.org

Acknowledgements

This work was financially supported by the PRACE project funded in part by the EUs 7th Framework Programme (FP7/2007-2013) under grant agreement no. RI-261557.