# The PEPPHER Approach to Programmability and Performance Portability for Heterogeneous many-core Architectures [1]

Siegfried BENKNER [a], Sabri PLLANA [a,2], Jesper LARSSON TRÄFF [a], Philippas TSIGAS [b], Andrew RICHARDS [c], Raymond NAMYST [d], Beverly BACHMAYER [e], Christoph KESSLER [f], David MOLONEY [g] and Peter SANDERS [h]

[a] *University of Vienna, Austria*
[b] *Chalmers University, Sweden*
[c] *Codeplay Software Ltd., UK*
[d] *INRIA and University of Bordeaux 1, France*
[e] *Intel, Germany*
[f] *Linköping University, Sweden*
[g] *Movidius Ltd., Ireland*
[h] *Karlsruhe Institute of Technology, Germany*

**Abstract.** The European FP7 project PEPPHER is addressing programmability and performance portability for current and emerging heterogeneous many-core architectures. As its main idea, the project proposes a multi-level parallel execution model comprised of potentially parallelized components existing in variants suitable for different types of cores, memory configurations, input characteristics, optimization criteria, and couples this with dynamic and static resource and architecture aware scheduling mechanisms. Crucial to PEPPHER is that components can be made performance aware, allowing for more efficient dynamic and static scheduling on the concrete, available resources. The flexibility provided in the software model, combined with a customizable, heterogeneous, memory and topology aware run-time system is key to efficiently exploiting the resources of each concrete hardware configuration. The project takes a holistic approach, relying on existing paradigms, interfaces, and languages for the parallelization of components, and develops a prototype framework, a methodology for extending the framework, and guidelines for constructing performance portable software and systems – including paths to migration of existing software – for heterogeneous many-core processors. This paper gives a high-level project overview, and presents a specific example showing how the PEPPHER component variant model and resource-aware run-time system enable performance portability of a numerical kernel.

**Topic Area**: Section 3, Software and Architectures

**Keywords.** Programmability, performance portability, heterogeneous architectures.

[1] This work was supported by the EU under grant No. 248481, FP7 Project PEPPHER, `www.peppher.eu`

[2] Project Coordinator: Research Group of Scientific Computing, Nordbergstrasse 15/C308, 1090 Vienna, Austria; E-mail: pllana@par.univie.ac.at.

## Introduction

With the proliferation of radically different computer architectures (many-core CPUs, embedded CPUs, SIMD instruction sets, Cell Broadband Engine Architecture, Intel MIC and SCC architectures, etc.), the fusion of different architectures into hybrid systems, as well as the rapid succession of architecture generations (e.g., NVidia GPUs) ensuring both a reasonable level of performance and a sufficient degree of functional *and* performance portability between different hybrid systems pose fundamental challenges to current computer science research and engineering. Due to the large architectural parameter space it is also clear that ensuring programmability and portability for such heterogeneous systems cannot be tackled manually, but must be handled or assisted by automatic means [14,9]. A large number of current research projects [10] are addressing these problems, and the European FP7 project PEPPHER[3] is one of them. The PEPPHER project attacks the problems of performance portability and efficient use of heterogeneous systems at several levels at the same time. PEPPHER proposes solutions that involve a combination of static and dynamic scheduling, automatic adaptation of (library) components, compilation and transformation techniques, and a resource aware run-time that is aided by performance information and performance feedback mechanisms.

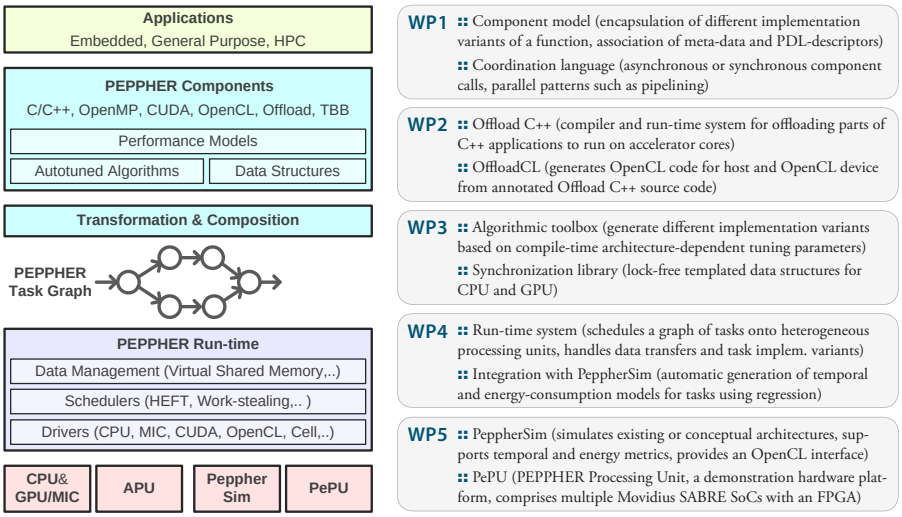
This paper explains the approach and basic premises of PEPPHER. To substantiate, we show how a numerical kernel algorithm built from available component variants can achieve a linear performance improvement with each extra GPU added to a hybrid CPU-GPU system, and at the same time more efficiently exploit the CPU. The example demonstrates performance portability of systems with different numbers of attached GPUs.

## 1. The PEPPHER Approach

A fundamental working hypothesis of PEPPHER for enabling performance portability is to provide performance-critical parts of the applications in multiple variants suitable for different types of cores and performance criteria. Preselection and specialization of variants for a given heterogeneous architecture are performed statically as far as possible, while the selection of the most appropriate remaining variants is delegated to a resource aware run-time system. Variants can be generated in part by compilation and autotuning strategies provided by the PEPPHER framework, or supplied directly by the more skilled ("expert") programmer, for instance as part of a performance portable library of algorithms and data structures. Variants can themselves be parallel, and make requirements for specific, parallel resources. Preselection, composition, and run-time selection is guided by performance information that can be successively provided by the application developer. The PEPPHER framework and methodology will in this way make it possible to gradually make an existing application both more efficient for a given, heterogeneous parallel system as well as more performance portable across different types of heterogeneous systems.

More concretely, PEPPHER introduces a flexible and extensible component model for encapsulating and annotating performance critical parts of the application. Compo-

[3] An acronym for "PErformance Portability and Programmability for HEterogeneous many-core aRchitectures" (see `www.peppher.eu`)




(a) PEPPHER stack (b) Work-packages

**Figure 1.** The holistic PEPPHER approach and associated work-packages. PEPPHER applications currently written in C++ are made performance aware and portable by turning critical parts into components with implementation variants for different hardware, input characteristics, and optimization criteria (WP1). Variants can themselves be parallelized in the most suitable framework. Variants can in part be generated automatically, and in part supplied by "expert" programmers as part of libraries (WP3). Transformation and compilation techniques support the variant generation (WP2). Scheduling and dynamic variant selection for the available resources is done by the PEPPHER run-time (WP4). Hardware mechanisms for performance monitoring and scheduling are also investigated in the project (WP5).

nents are made *performance aware* by association of analytical performance models or interpolations based on performance history for predicting an aspect of performance (execution time, power consumption, or other). Performance aspects are parameterized and evaluated relative to an abstract platform description. The component model also provides for specification of resource constraints and requirements and other non-functional component properties that may be essential for the execution in a given context.

The capability to manage *implementation variants* of components suitable for different platforms under different circumstances (availability of resources, structure of input) and for different optimization objectives is essential. Variants can be generated automatically by compilation to different platforms and by auto-tuning techniques. For the latter, the component model makes it possible to expose tunable component parameters. Implementation variants can also be supplied manually, e.g. targeted to different platforms, by the expert programmer. PEPPHER components may already have been parallelized using a conventional parallelization language or framework (OpenMP, OpenCL, pThreads, and others).

Figure 1 shows the PEPPHER architecture and software stack, and indicates how the elements of the approach fit together.

### *1.1. The PEPPHER Component Model*

A detailed component model allowing the specification of performance aware components and component variants, as well as specification of tunable parameters of com-

ponents has been developed. Currently, PEPPHER applications have to be written in C/C++. PEPPHER components are identified by *pragmas* and further described by language external means (XML-based descriptors). This limits the intrusiveness of the PEPPHER framework. An extensible platform description language for capturing properties of a broad range of target platforms is described in [15]. The PEPPHER framework maintains a rich variety of component variants. By static *composition*, preselection for given input (size and other known characteristics, performance aspects) and platforms can be sometimes be performed as described in, e.g., [11].

### *1.2. The PEPPHER Run-time System*

A PEPPHER executable consists of compiled component variants together with the non-componentized parts of the application. Execution is delegated to a flexible, performance aware, heterogeneous resource aware run-time system. Based on core availability, resource requirements, estimated performance, execution history, and input availability the run-time system schedules the most promising component variant on the best available resource. The further development of such a performance, resource and architecture/memory aware run-time system, that is flexible enough to handle scheduling of parallelized component variants over different parts of the heterogeneous system is an important aspect of the project [3,18].

### *1.3. Tunable Algorithms and Data Structures for Parallel Architectures*

The compositional and run-time supported approach to performance portability is complemented by auto-tuned, architecture and context adaptive algorithmic library components for further enhancing performance portability.

Such highly performance portable algorithms are made available as components to the application programmer through libraries. The auto-tunable GPU sorting algorithm in [12] is an example of the level of adaptable, portable performance that can be achieved by the algorithm engineering expert. Algorithms and data structures for common tasks on a wide range of multi-core architectures also support the efficient implementation of the PEPPHER run-time.

### *1.4. Compilation Techniques in PEPPHER*

OpenCL is a possible portability layer for current heterogeneous systems, especially such that employ GPUs. OpenCL is a low-level model and does not alone solve the performance portability problem. However, efficient compilation from C++ to OpenCL supports the component model and the run-time system. Specifically, a C++ extension termed Offload C++ that is used in PEPPHER allows for explicit compilation and offloading to accelerator devices like GPUs and IBM Cell SPEs with the compiler taking care of the necessary call graph duplication, functional duplication, and replication of global data [6].

### *1.5. Hardware Support and Feedback*

Finally, a highly configurable hardware simulator (PeppherSim) provides a larger spectrum of possible heterogeneous target architectures and makes it possible to investigate

```
for (step = 0; step < min(MT, NT); step++){
  for (p = proot; p < P; p++) {
    SGEQRT(step, step, ...);
    for (j = step+1; j < NT; j++)
      SORMQR(j, step, ...);
    for (i = i_beg+1; i < MT; i++){
      STSQRT(i, step, ...);
      for (j = step+1; j < NT; j++)
        SSSMQR(i, j, step, ...);
    }
  }
}
```

**Figure 2.** QR factorization code fragment for matrices of size `NT` $\times$ `MT` using BLAS routines.

possibilities for hardware support for performance monitoring and portability. Also, new synchronization primitives, e.g. [8], and other architecture support for algorithms and run-time can be investigated with the simulator.

### *1.6. Application Benchmarks*

A small set of larger application benchmarks has been selected to experiment with and validate the performance portability that can be achieved with the PEPPHER framework. The benchmarks are intended to cover relevant application areas ranging from embedded (e.g., BZIP2, Computational photography), server/enterprise/general-purpose (e.g., Games physics), to high-performance (e.g., GROMACS) computing domains. In addition, important numeric and non-numeric kernels (e.g., MAGMA/PLASMA, MCSTL) have been included as manageable test cases, that will also be useful as library components.

## 2. Example: Tile QR Factorization

We illustrate with an example the promise of PEPPHER for more efficient utilization of heterogeneous compute resources, and per implication for improved performance portability.

The application is a standard Tile-QR factorization algorithm based on BLAS components that has been implemented on top of the PEPPHER run-time [1]. The application is constructed from BLAS kernel functions as shown in Figure 2, and these kernels are turned into PEPPHER components by giving suitable XML interface specifications. These kernels are sequential, and the PEPPHER run-time handles the parallelism by scheduling calls to them. Expert implementation variants of the BLAS kernels for CPUs and GPUs are already available through existing GPU and CPU libraries, namely PLASMA [4] for CPU and MAGMA [16] for GPU, and are added as component variants in the PEPPHER implementation. Figure 3 shows the results of running the QR factorization application on CPU-GPU system with one to four GPUs. The test platform is composed of four quad-core AMD CPU cores (16 cores total) that are accelerated with four NVIDIA Tesla GPUs (960 GPU cores total). One CPU core is reserved for each GPU, leaving 12 CPU cores available for computational work.

**Table 1.** Kernel components of the QR application. The table shows the performance of the kernels on CPU and GPU, respectively.

| BLAS kernel | CPU performance (GFlops) | GPU performance (GFlops) | Speed-up ratio |
|---|---|---|---|
| SGEQRT | 9 | 30 | 3 |
| STSQRT | 12 | 37 | 3 |
| SORMQR | 8.5 | 227 | 27 |
| SSSMQR | 10 | 285 | 28 |

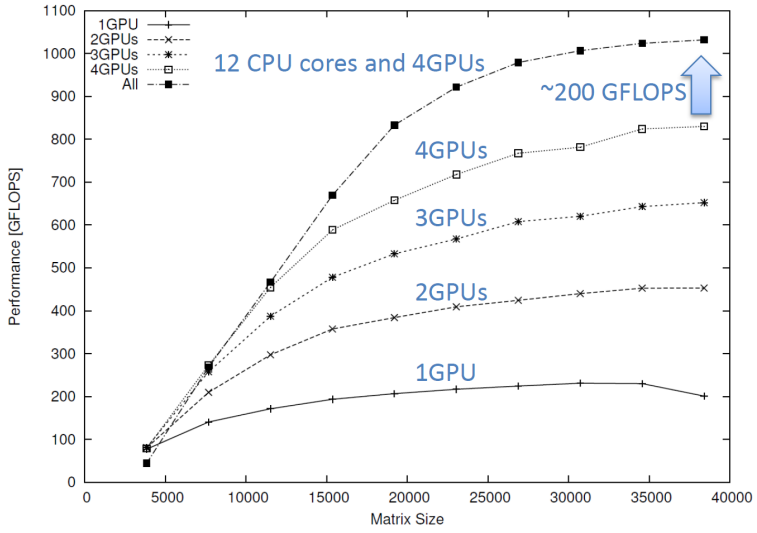



**Figure 3.** Single-precision performance of QR factorization on a 16-core AMD system (four quad-core AMD Opteron 8358 SE 2.4GHz) with up to 4 NVidia GPUs (C1060).

The performance of the kernels of the QR application is shown in Table 1, which also shows the speed-up achievable on the GPU relative to a single CPU core. As can be seen, the SORMQR and SSSMQR kernels are particularly efficient on the GPU, and the GPU variants are therefore likely to be selected by the PEPPHER run-time.

The plot shows that the PEPPHER run-time is able to exploit the GPUs efficiently in that each additional GPU brings a linear increase in performance of about 200GFlops. In addition, the run-time exploits the remaining 12 CPU cores for execution of the most suitable BLAS components. The latter is interesting: by automatically avoiding to waste GPU processing power with GPU-unfriendly BLAS kernels (that are scheduled on CPUs instead), more performance can be achieved than expected. Single-precision performance of the QR application on 12 CPU cores is about 150 GFlops, but the achieved performance increase when we add 12 CPU cores to four GPUs is about 200 GFlops. In the experiment about 20% of the SGEQRT tasks are scheduled on the GPUs, whereas for the SSSMQR tasks over 90% are executed on the GPUs (see speed-up ratios in Table 1).

The experiment demonstrates efficiency and performance portability of this particular application across systems with one to four GPUs. It is important to note that it relied on existing expert written components, so no extra implementation effort was needed for the kernels.

## 3. Related Work

A large number of projects are currently concerned with aspects of multi-core programmability as mentioned above. In contrast to many other (European) projects [10], PEPPHER is not focusing on providing a common programming model or virtual machine type portability layer. In PEPPHER the application programmer provides performance information by annotating components and describing characteristics of the actual environment/architecture, using the most convenient API for implementation variants that are tailored to different types of CPU, GPU and other cores. Likewise PEPPHER is not concerned with automatic parallelization per se. PEPPHER is not an auto-tuning project, but enables auto-tuning techniques to be used by exposing tunable parameters of both components and parameterized, adaptive library algorithms. PEPPHER is taking a general-purpose approach in contrast to implicit parallelization and performance portability via domain specific languages, as in e.g., [5].

Many other projects also take the provision of implementation variants of functions, methods, or components tailored to different architectures as basic premise for addressing performance and performance portability issues. Three recent such projects are PetaBricks [2], Merge [13], and Elastic computing [17]. PetaBricks [2] is an auto-tuning project that addresses performance portability mostly across homogeneous multi-core architectures by focusing on auto-tuning methods for different types of optimization criteria. Parallelism is implicit. Merge [13] also provides variants, but focuses on MapReduce [7] as a unified, high-level programming model. Elastic computing [17] focuses on provision of large number of variants, so called elastic functions, among which the best combination is composed mostly by static means. Selection is guided by performance profiles and models.

In contrast to some of these works PEPPHER is distinguished by a holistic approach, which attacks performance portability at multiple layers from high-level component based programming, compilation, library and run-time support, to hardware mechanisms for performance monitoring and feedback. Ultimately, PEPPHER aims to become language and parallelization agnostic and support different implementation languages and parallelization interfaces. Thus, the programmer can choose the most convenient language and API for implementing the components.

## 4. Conclusion

This paper outlined the PEPPHER approach to achieving performance portability and programmability for hybrid (heterogeneous) many-core architectures. PEPPHER combines a compositional, performance aware software framework, auto-tunable, adaptive algorithmic libraries, specific compilation techniques and an efficient run-time, and will thereby become independent of specific programming models, virtual machines and architectures. The combination of component based adaptation with a performance and resource aware run-time system provides for the necessary degrees of freedom, possibly not found in many auto-tuning projects, where scheduling decisions may be hard-coded at an early stage. We contend that neither best performance nor performance portability can be achieved by fixed, static offloading of promising tasks of the component based application onto preselected cores. Instead, component tasks must be scheduled dynamically on available cores for which a variant giving the best performance exists.

## References


[1] E. Agullo, C. Augonnet, J. Dongarra, M. Faverge, H. Ltaief, S. Thibault, and S. Tomov. QR Factorization on a Multicore Node Enhanced with Multiple GPU Accelerators. In *25th IEEE International Parallel & Distributed Processing Symposium (IEEE IPDPS 2011)*, Anchorage, Alaska, USA, May 2011.

[2] J. Ansel, C. P. Chan, Y. L. Wong, M. Olszewski, Q. Zhao, A. Edelman, and S. P. Amarasinghe. PetaBricks: A language and compiler for algorithmic choice. In *Proceedings of the 2009 ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2009*, pages 38–49. ACM, 2009.

[3] C. Augonnet, S. Thibault, R. Namyst, and P.-A. Wacrenier. StarPU: A unified platform for task scheduling on heterogeneous multicore architectures. *Concurrency and Computation: Practice and Experience*, 23(2):187–198, 2011.

[4] A. Buttari, L. Langou, J. Kurzak, and J. Dongarra. A class of parallel tiled linear algebra algorithms for multicore architectures. *Parallel Computing*, 35(1):38 – 53, 2009.

[5] H. Chafi, A. K. Sujeeth, K. J. Brown, H. Lee, A. R. Atreya, and K. Olukotun. A domain-specific approach to heterogeneous parallelism. In *16th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*, pages 35–46, 2011.

[6] P. Cooper, U. Dolinsky, A. F. Donaldson, A. Richards, C. Riley, and G. Russell. Offload - automating code migration to heterogeneous multicore systems. In *High Performance Embedded Architectures and Compilers, 5th International Conference (HiPEAC 2010)*, volume 5952 of *Lecture Notes in Computer Science*, pages 337–352. Springer, 2010.

[7] J. Dean and S. Ghemawat. MapReduce: Simplified data processing on large clusters. *Communications of the ACM*, 51(1):107–113, 2008.

[8] P. H. Ha, P. Tsigas, and O. J. Anshus. NB-FEB: A universal scalable easy-to-use synchronization primitive for manycore architectures. In *Principles of Distributed Systems, 13th International Conference (OPODIS 2009)*, volume 5923 of *Lecture Notes in Computer Science*, pages 189–203. Springer, 2009.

[9] M. W. Hall, Y. Gil, and R. F. Lucas. Self-configuring applications for heterogeneous systems: Program composition and optimization using cognitive techniques. *Proceedings of the IEEE*, 96(5):849–862, 2008.

[10] HiPEAC. Related Projects, `http://www.hipeac.net/related/`, accessed on July 2011.

[11] C. W. Kessler and W. Löwe. A framework for performance-aware composition of explicitly parallel components. In *Parallel Computing: Architectures, Algorithms and Applications, (ParCo 2007)*, volume 15 of *Advances in Parallel Computing*, pages 227–234. IOS Press, 2007.

[12] N. Leischner, V. Osipov, and P. Sanders. GPU sample sort. In *24th IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, 2010.

[13] M. D. Linderman, J. D. Collins, H. Wang, and T. H. Y. Meng. Merge: A programming model for heterogeneous multi-core systems. In *Proceedings of the 13th International Conference on Architectural Support for Programming Languages and Operating Systems, (ASPLOS 2008)*, pages 287–296. ACM, 2008.

[14] S. Pllana, S. Benkner, E. Mehofer, L. Natvig, and F. Xhafa. Towards an intelligent environment for programming multi-core computing systems. In *Euro-Par 2008 Workshops - Parallel Processing*, volume 5415 of *Lecture Notes in Computer Science*, pages 141–151. Springer Berlin / Heidelberg, 2009.

[15] M. Sandrieser, S. Benkner, and S. Pllana. Explicit platform descriptions for heterogeneous many-core architectures. In *16th International Workshop on High-Level Parallel Programming Models and Supportive Environments (HIPS), International Parallel and Distributed Processing Symposium (IPDPS 2011)*, 2011.

[16] S. Tomov, R. Nath, H. Ltaief, and J. Dongarra. Dense linear algebra solvers for multicore with gpu accelerators. In *Parallel Distributed Processing, Workshops and Phd Forum (IPDPSW), 2010 IEEE International Symposium on*, pages 1 –8, april 2010.

[17] J. R. Wernsing and G. Stitt. Elastic computing: A framework for transparent, portable, and adaptive multi-core heterogeneous computing. In *Proceedings of the ACM SIGPLAN/SIGBED 2010 conference on Languages, compilers, and tools for embedded systems (LCTES)*, pages 115–124. ACM, 2010.

[18] M. Wimmer and J. L. Träff. Work-stealing for mixed-mode parallelism by deterministic team-building. In *23rd ACM Symposium on Parallelism in Algorithms and Architectures (SPAA 2011)*, pages 105–115, 2011.