PEPPHER: Efficient and Productive Usage of Hybrid Computing Systems

Siegfried Benkner, Sabri Pllana, Jesper Larsson Träff, Philippas Tsigas, Uwe Dolinsky, Cédric Augonnet, Beverly Bachmayer, Christoph Kessler, David Moloney and Vitaly Osipov

Abstract

The 3-year European FP7 project PEPPHER addresses efficient utilization and usage of hybrid (heterogeneous) computer systems consisting of multi-core CPUs with GPU-type accelerators. PEPPHER is concerned with two major aspects: programmability and efficiency on given heterogeneous systems, and code and performance portability between different heterogeneous systems. The PEPPHER approach is pluralistic and parallelization agnostic, aiming to support different parallel languages and frameworks at different levels of parallelism. The central idea of PEPPHER is to maintain multiple, tailored implementation variants of performance-critical components of the application and schedule these efficiently either dynamically or statically across the available CPU and GPU resources. Implementation variants are supplied incrementally by hand, by compilation, by component composition, by auto-tuning, or taken from expert-written, adaptive libraries.

This paper outlines the PEPPHER performance aware component model, its means for performance prediction, the PEPPHER run-time system, and other major aspects of the project concerned with algorithm and data structure support, compilation, and hardware feedback. A larger example demonstrating performance portability with the PEPPHER approach across hybrid systems with one to four GPUs is discussed: each GPU that is added to the system brings a linear performance increase, and performance aware scheduling provides for more efficient utilization of the combined CPU-GPU resources.

Benkner, Pllana and Träff (Contact author, email: `traff@par.univie.ac.at`) are with University of Vienna, Tsigas with Chalmers University, Dolinsky with Codeplay, Augonnet with INRIA, Bachmayer with Intel GmbH, Kessler with University of Linköping, Moloney with Movidius Ltd., and Osipov with Karlsruhe Insitute of Technology.

I. 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 are vital and pressing 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 [12]. A large number of current research projects are addressing these problems, and the European FP7 project PEPPHER[1] 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 with feedback monitoring for gathering empirical performance information. Specific PEPPHER contributions that will be discussed in this paper include:

- making (legacy) code written in existing programming languages and parallel APIs performance aware and portable through stepwise componentization,
- a component model with meta-data for performance awareness and adaptivity,
- algorithms and data structures for hybrid, parallel architectures, library based performance portability through algorithmic auto-tuning and compositional techniques,
- performance aware, heterogeneous run-time scheduling,
- hardware feedback mechanisms through a tailored simulator, and finally
- source-to-source transformation and OpenCL compilation.

The paper explains the vision and basic premises of PEPPHER, and highlights some concrete results to substantiate our approach. In particular, we show how a numerical kernel algorithm built from available component variants can achieve a linear performance improvement with each

[1]An acronym for "PErformance Portability and Programmability for HEterogeneous many-core aRchitectures". See also www.peppher.eu

extra GPU added to a hybrid CPU-GPU system, and at the same time more efficiently exploit the CPU. The example demonstrates performance portability across hybrid systems with different numbers of attached GPUs. Best overall performance is achieved, not by fixed, static offloading of tasks to the GPUs, but by dynamically scheduling of component tasks on either CPU or GPUs based on availability of resources and relative efficiency of the component variants.

II. The PEPPHER approach

The fundamental premise of PEPPHER for enabling performance portability is to provide performance-critical parts of the applications in multiple variants that are suitable for different types of cores, usage contexts and performance criteria. For this PEPPHER provides a flexible and powerful component model. Preselection and specialization of variants for a given heterogeneous architecture are performed statically as far as possible by *component composition* techniques, while the final selection of the most appropriate variants is delegated to a resource aware run-time system. Run-time selection is based on optimization objective, resource availability, data availability and placement, and available performance information for the variants, while respecting data dependencies between components. Component variants can be generated in part by compilation and auto-tuning mechanisms provided or enabled 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. Component variants can themselves be parallel, and in that case make explicit requirements for specific, parallel resources. Preselection, composition, and run-time selection is guided by performance information that can be incrementally provided by the application developer, and that is evaluated in the context of an explicit platform model. The PEPPHER framework and methodology in this way makes it possible to gradually make an existing application more efficient for a given, heterogeneous parallel system, as well as more performance portable across different types of heterogeneous systems, namely by progressively supplying more suitable and efficient component variants, and by componentizing more and more parts of the application.

Concretely, PEPPHER introduces a flexible and extensible component model for encapsulating and annotating performance critical parts of the application. In particular, components are made *performance aware* by association of performance models or regressions based on performance history for predicting a desired aspect of performance (execution time, power consumption, or other). Performance aspects are parameterized and evaluated relative to an abstract platform description [16]. The component model also provides for specification of resource constraints and requirements, as well as other non-functional properties of components that are nevertheless essential for the efficient execution in given contexts. The capability to maintain suitable *implementation variants* of components for different platforms under different circumstances 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 for which good ("optimal") values can be found by suitable auto-tuning tools. Such auto-tuning tools will, however, not be specifically developed in PEPPHER. Implementation variants can also be supplied manually by the expert programmer, e.g., targeted to different platforms. PEPPHER components may already have been parallelized using conventional parallel models and languages (OpenMP, OpenCL, Pthreads, etc.).

At run-time the components form a directed acyclic graph of component tasks. A component task variant can be scheduled when all data dependencies are resolved. Performance information, input information, optimization criteria, resource requirements and availability, data placement in the system, e.g., in main CPU or in GPU memory, are all used to determine which of the ready component task variants are scheduled on which part of the system. The execution model is parallel at multiple levels: ready component tasks can be executed in parallel on different parts of the system, and component tasks can themselves be parallel, e.g., OpenCL or CUDA variants for the GPU and multi-core parallel variants for the CPU.

Figure 1 gives an overview of the PEPPHER architecture and software stack, indicating how the elements of the approach fit together. The figure illustrates the PEPPHER approach to assist in development and generation of efficient, performance portable applications for heterogeneous many-core systems.

A. Performance Guidelines and Portability

Performance portability is an elusive notion. An application could be said to be performance portable if it executes with the same efficiency (fraction of theoretical peak performance) across different heterogeneous multi-core systems. This is a strong, absolute requirement. A different conception of performance portability is that no application restructuring be necessary when

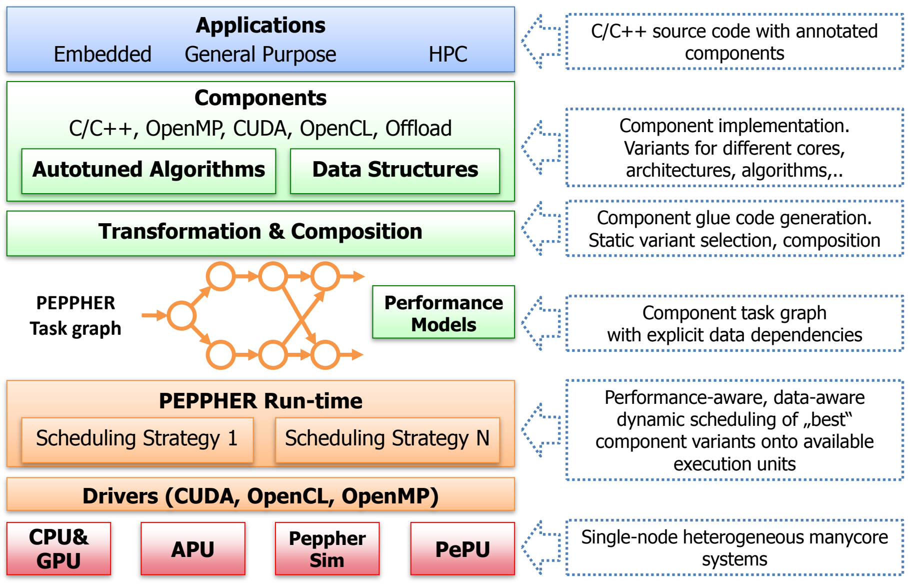

Applications
Embedded
General Purpose
HPC
C/C++ source code with annotated components
Components
C/C++, OpenMP, CUDA, OpenCL, Offload
Autotuned Algorithms
Data Structures
Component implementation. Variants for different cores, architectures, algorithms,..
Transformation & Composition
Component glue code generation. Static variant selection, composition
PEPPHER Task graph
Performance Models
Component task graph with explicit data dependencies
PEPPHER Run-time
Scheduling Strategy 1
Scheduling Strategy N
Performance-aware, data-aware dynamic scheduling of „best" component variants onto available execution units
Drivers (CUDA, OpenCL, OpenMP)
CPU& GPU
APU
Peppher Sim
PePU
Single-node heterogeneous manycore systems

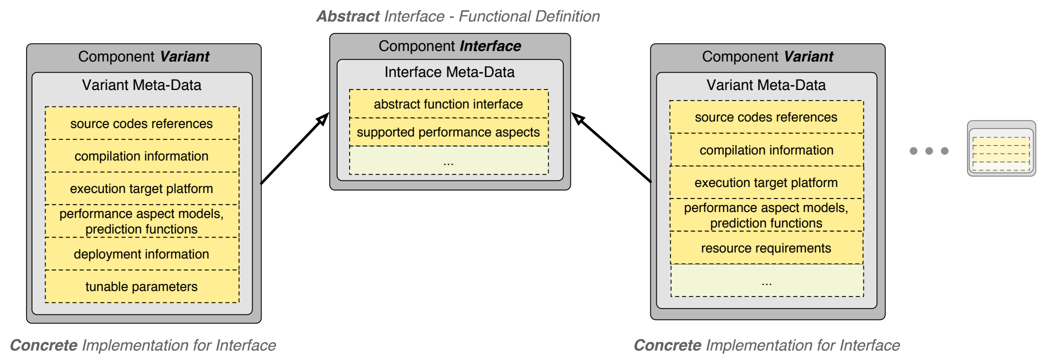
Fig. 1. The PEPPHER architecture and software stack. Applications written in C/C++ are annotated by the application programmer to enable generation of component variants. The more skilled, "expert" programmer or library writer can provide additional, highly performant component variant implementations for specific hardware, optimization criteria, or input configurations. Static composition performs component pruning and preselection and generates necessary glue code, while the final selection is done dynamically by the performance aware PEPPHER run-time system.

porting code from one architecture to another. In a library-based approach to performance portability this would be ensured by having implementations of the library routines targeted to different types of architectures, input configurations, and performance criteria. At each call the library would be responsible for invoking the best implementation for the given situation using an efficient lookup and selection procedure. In PEPPHER the components of the application, instead of only fixed library functionalities, are the units at which performance portability is enabled. Performance portability is supported and enforced by guidelines and requirements that application components and library routines must fulfill. An example guideline would prescribe that no library functionality can be improved just by expressing the same functionality by means of other, related or specialized library components. Such a guideline would ensure that no performance gain on a different system would be possible by the application programmer by simply reimplementing a library or component functionality in terms of other, related library functionality. For the PEPPHER framework this is an obligation to the library implementation that efficient implementation variants for different architectures are in place, and that general library functionalities select the most efficient special case implementations as appropriate. The guideline would further imply that the PEPPHER framework does the best possible selection among different available implementations of a library component. This is a non-trivial requirement in a heterogeneous and dynamic setting. By enforcing such requirements the user would be relieved from the temptation to try to do better and write the selection code himself. Other rules govern the use of component annotations: the more information is provided, the better the PEPPHER framework can do at selecting the most suited component variant. If no information is provided for the components, the PEPPHER framework produces default code. In this way PEPPHER provides an incremental approach to making applications performance portable.

B. 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, e.g., HyVM, SARC, AppleCore, 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 [15], and Elastic computing [22]. 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 [15] also provides variants, but focuses on MapReduce [8] as a unified, high-level programming model. Elastic computing [22] focuses on provision of large number of variants, so called elastic functions,



Abstract Interface - Functional Definition
Component Variant
Variant Meta-Data
source codes references
compilation information
execution target platform
performance aspect models, prediction functions
deployment information
tunable parameters
Concrete Implementation for Interface
Component Interface
Interface Meta-Data
abstract function interface
supported performance aspects
...
Component Variant
Variant Meta-Data
source codes references
compilation information
execution target platform
performance aspect models, prediction functions
resource requirements
...
Concrete Implementation for Interface

Fig. 2. A PEPPHER component consists of an abstract interface describing the functionality of the component, and a number of implementation variants targeted for different architectures and with different performance characteristics. Interface and variant meta-data descriptions are described in XML documents.

among which the best combination is composed mostly by static means. Selection is guided by performance profiles and models. Use of algorithmic variants has previously been studied extensively in the STAPL project [18]. Other work that has influenced PEPPHER is the Sequoia programming model for heterogeneous architectures based on a tree abstraction of the memory system [9].

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.

III. Technical aspects of PEPPHER

In this section we discuss in more detail the technical ingredients of the PEPPHER framework.

A. The PEPPHER component model

As a starting point PEPPHER has developed a comprehensive and flexible component model for specification of performance aware components, implementation variants, and tunable parameters of components. Applications for PEPPHER are currently assumed to be written in

C++. PEPPHER components are identified by language external means (e.g., pragmas) rather than language extensions and employ XML schemata for their specification. In this sense the PEPPHER framework is non-intrusive, requiring little direct application modification, for instance in legacy source code.

A PEPPHER component is an annotated software module that implements a specific functionality declared in a PEPPHER interface. A PEPPHER interface is defined by an interface descriptor, an XML document that specifies the name, parameter types and the access types (read, write or both) of a function to be implemented, and in addition which performance aspects (such as mean execution time or energy consumption) the prediction functions of component implementations must provide. The component model is illustrated in Figure 2. Interfaces can be generic in static entities such as parameter types or code; genericity is resolved statically by expansion as with C++ templates. From the interface descriptor, header files in the various component implementations' source languages can be generated.

Several component variants may implement the functionality defined in a PEPPHER interface by different algorithms or for different execution platforms; also, further component implementation variants may be generated automatically from a common source module, e.g., by special compiler transformations or by instantiation of tunable parameters. The latter would be the task of auto-tuning tools. These variants differ by their resource requirements and performance behavior, and thereby become alternative choices for composition or run-time selection whenever the interface function is called. In order to prepare and guide variant selection, component implementations need to expose their relevant properties explicitly to composition tool and run-time as will be briefly described later. Each PEPPHER component implementation variant thus provides its own component descriptor, an XML document that contains information (meta-data) about the following properties:

- The implemented PEPPHER interface.
- The required PEPPHER interfaces, i.e., other component functionality called from this component, if any.
- The source file(s) of this component implementation.
- Deployment information such as compilation commands and options and required software modules.
- A reference to the platform consisting of the programming model/language used for the

component implementation and of the target architecture.

- Type and amount of resources required for execution at the given platform.
- A reference to a performance prediction function. whose parameters are given by a context descriptor data structure.
- The structure of the context descriptor consisting of call parameter context and current resource availability information.
- Tunable parameters of the component implementation, such as buffer sizes, loop blocking factors, cut-off values and so on.
- Additional constraints for component selectability such as parameter ranges.

Performance prediction functions are usually supplied by the component developer. These can be either purely analytical, use performance data tables determined by micro-benchmarking for the target platform, or be based on historic performance data. The latter are maintained in a performance data repository. The PEPPHER run-time is able to automatically generate performance models from historic data.

The actual platform properties are defined separately in another XML document and aim at describing hardware- and software characteristics of the heterogeneous target environment. Such platform descriptions can be used at multiple levels of the PEPPHER framework [16]. Lookup of specific platform properties is done by the composition tool, the run-time, or by component developers themselves.

At run-time component invocations result in tasks that are managed by the PEPPHER run-time system and executed non-preemptively. PEPPHER components and tasks are stateless, and by default so are input/output parameters which determine the dependencies among component tasks. Parameter data placement in CPU or GPU memory is tracked by the run-time system and taken into account during variant scheduling. However, parameters can have state themselves, implemented by wrapping into STL-like container data structures, and keep track of, e.g., in which memory modules of the target system which parts of the data are currently located or mirrored. The container state then becomes part of the call context information since it is relevant for performance prediction.

The PEPPHER framework keeps track of the different component implementation variants by storing descriptors in repositories that can be explored by the composition tool and the run-time.

Composition performs preselection of a subset of appropriate implementation variants based on statically available parameter and resource information. Parameter information can be either concrete or abstract values given by constraints. Composition can in the extreme case prune the number of available variants to only one, but in general several component variants will be available for selection by the run-time system. Part of the variant selection code may also be generated by the composition tool. Executable component variants are finally generated by compilation for the platforms described in the component variant descriptions.

The final selection among the remaining component variants is done by the run-time system (see Section III-B). This is the default mechanism in PEPPHER. In the special case where sufficient meta-data for performance prediction is available for all selectable component variants, composition can be done completely statically and co-optimized with the required resource allocation [13].

To enhance programmability and provide for more variant selection and auto-tuning possibilities, portable higher-level coordination strategies for expressing computations in terms of structured execution of PEPPHER components are being added to the framework, e.g., pipeline pattern, wave-front pattern, task farming, standard skeletons, and others. First steps have been taken in SkePU [7], an auto-tunable C++ template library of data-parallel generic skeleton components such as map, reduce, or scan, each with multiple implementation variants including CUDA, OpenCL, and OpenMP, for multi-GPU based systems. By micro-benchmarking SkePU can be automatically tuned to select, depending on the call context, the expected fastest back-end and values for tunable parameters such as number of GPU threads and thread block size.

B. The PEPPHER run-time system

Execution of a PEPPHER application that consists of compiled component variants together with the parts of the application that have not been componentized is delegated to a flexible, performance and resource aware, heterogeneous run-time system. The executable forms a directed acyclic graph of component tasks with data dependencies, where each task is a set of one or more variants together with performance and other information to trigger selection. Based on system 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 can also handle scheduling of parallelized component variants over different parts of the heterogeneous system is an important aspect of the project [3].

For CPU-GPU based systems with separate memory spaces, the run-time system implements a directory based virtual shared memory system. The application registers its input and output data with the system. Scheduling of the ready component tasks is done in a centralized fashion from a CPU. At run-time the actual placement of data in either main CPU memory or GPU device memory is used together with the component performance models to decide where to launch the next component variant that uses these data. To this end, a data transfer cost model is used together with the cost estimation for the component execution. Actual data transfers between different types of memory are handled automatically by the run-time. Through so called *filters* structured non-consecutive as well as block distributed data can be handled.

The actual scheduling policy used by the run-time system can be modified by the PEPPHER user. The current default strategy is HEFT [20], but alternate, less centralized scheduling policies that would use more hints on processor resource requirements in order to execute parallel tasks on the multi-core CPU are being developed [21].

The multi-level parallel framework providing for concurrency between component variants and intra-component-variant parallelism, the static, resource and architecture aware compositional techniques, and the dynamic, flexible run-time scheduling together enable the PEPPHER framework to both efficiently utilize given, heterogeneous resources, as well as to provide performance portability (by recompilation, recomposition, re-tuning relative to a different platform description) to entirely different architectures.

C. Tunable algorithms and data structures for parallel architectures

The static compositional and dynamic run-time supported approach to performance portability is complemented by expert-written auto-tuned, architecture and context adaptive algorithmic components for further enhancing performance portability. Algorithms written by specialists with a detailed understanding of the underlying architecture and its range of tunable architecture-dependent parameters can more flexibly and efficiently adapt to architectural changes, and provide for more detailed control over performance. Through libraries such highly performance portable algorithms are made available as components to the application programmer. The highly non-trivial, auto-tunable GPU sorting algorithm developed in [14] is an example of the level of

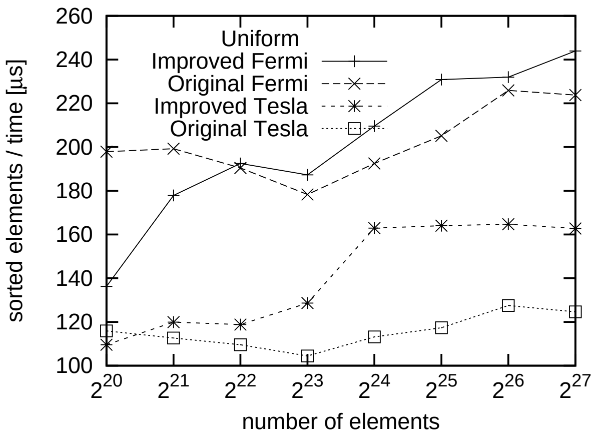


Fig. 3. Comparison of original and improved GPU sorting algorithm on Tesla and Fermi GPUs, number of sorted elements per time unit as function of number of elements to be sorted.

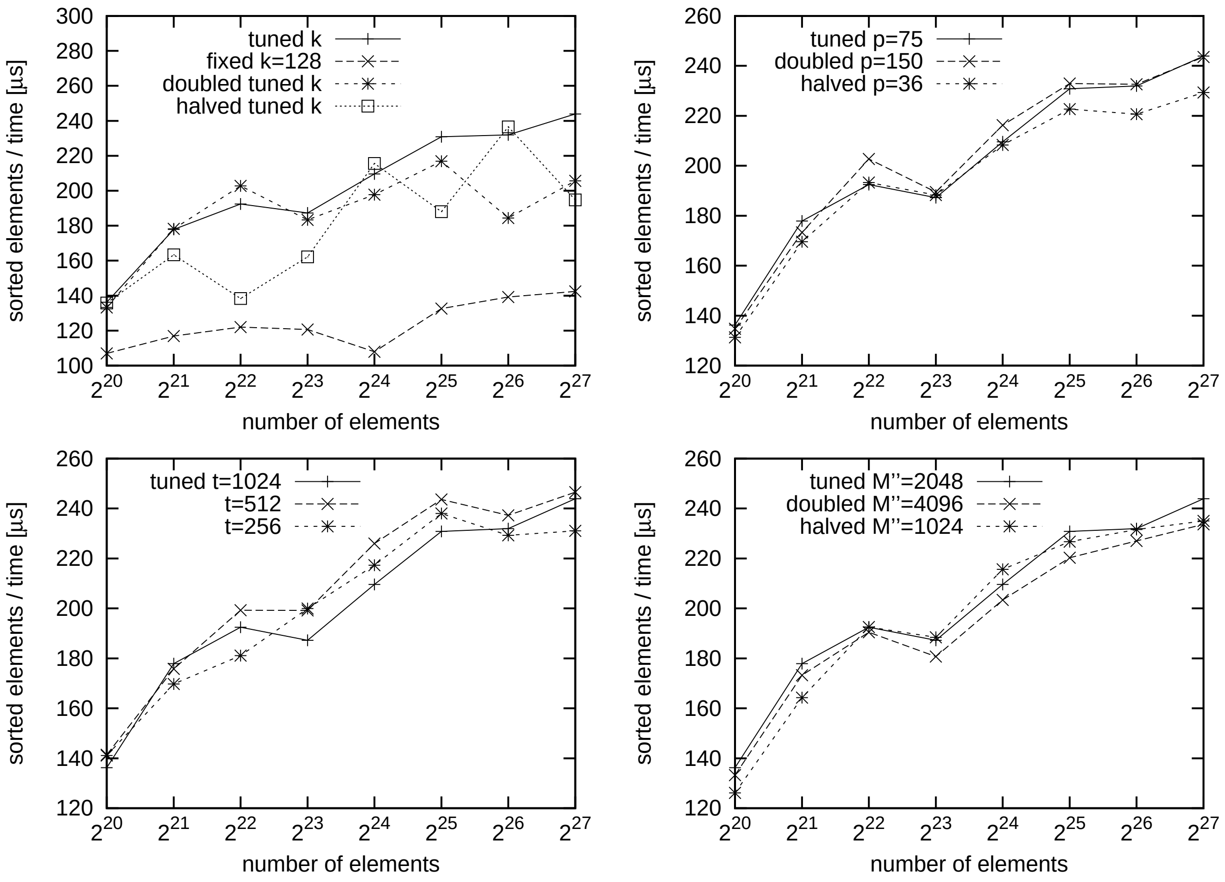


Fig. 4. Comparison of the effects of alternative parameter values against the “best”, auto-tuned values for the Fermi GPU.

	hardware parameter	Tesla/Fermi value
o	thread overload factor	5/5
T	hardware limit on threads per block	512/1024
S	number of streaming multiprocessors	15/30
E	size of shared memory	16 KB/48 KB

TABLE I

BASIC HARDWARE PARAMETERS AND RECOMMENDED SETTINGS FOR TWO CURRENT GPU ARCHITECTURES, NVIDIA TESLA AND FERMI.

parameter	constraint	Tesla/Fermi value Original	Tesla/Fermi value Improved
k	distribution degree, $k \leq E/(ro)$	$128/128$	variable/variable
t	threads per block, $t \leq T$	256 / 256	512 / 1024
ℓ	elements per thread, $\ell := n/(t \cdot p)$	8 / 8	$n/(t \cdot p)$ / $n/(t \cdot p)$
p	thread blocks, $p := oS$	$n/(t \cdot \ell)$ / $n/(t \cdot \ell)$	$150/75$
M	fallback to small case sorter, determined experimentally	2^{17} / 2^{17}	2^{16} / 2^{16}
M''	fallback to odd-even merge sort, $M'' \leq E/o, M'' \in 2^{\mathbb{N}}$	2^{10} / 2^{10}	2^{10} / 2^{11}
r	histogram replication factor, determined experimentally	8 / 8	8 / 8

TABLE II

THE TUNING PARAMETERS FOR THE TWO ALGORITHM VARIANTS OF THE GPU SAMPLE SORT ON NVIDIA TESLA AND FERMI GPUS.

adaptable, portable performance that can be achieved by the algorithm engineering expert.

Some of the basic, performance determining parameters of the NVidia GPU Tesla and Fermi architectures are summarized in Table I. Based on these, tunable, algorithmic parameters related to these architectural features are inferred by the algorithm developer as shown in Table II. These parameters are partially interrelated through an analytical performance model of the algorithm, and must partly be determined experimentally or through auto-tuning. By finding the right settings for the free parameters, superior performance of the algorithm is achieved on both architectures as shown in Figure 3. The figure compares a previous (original) [14] to a current, improved version version of the algorithm. The "best possible" values of the tunable parameters are taken from Table II, and the impact of changes to the four basic parameters k, t, p, M'' can be seen in Figure 4. Some of these parameters are determined analytically, others have been found by auto-tuning experiments. In PEPPHER concrete values for architectural parameters are looked up in the PEPPHER platform descriptors. At application deployment time Tesla and Fermi sample sort variants can thus be generated, partly through auto-tuning for determining best values for the non-determinate tuning parameters, and added to the PEPPHER component repository. The algorithm has furthermore been implemented in both CUDA and OpenCL, but only the CUDA results are shown here. We note that the CUDA and OpenCL interfaces by themselves only ensure code portability, but do not provide performance portability: this is achieved by the parameterized algorithm.

A larger selection of similarly tunable algorithms for many-core CPU architectures is incorporated in the Multi-Core STL library (MCSTL) [17] and some contributed to the Thrust GPU library. Also here, algorithms are parameterized with typical architectural features, and can thus be tuned to achieve a high degree of performance portability for the class of target architectures. Not mentioned here, algorithms and data structures for lock-free programming on CPUs and GPUs are needed and being developed in PEPPHER, both as application programmer components and for supporting the implementation of the run-time system, see, e.g., [10].

Choice between CPU and GPU variants for such library components is done either by specialized glue-code that takes the resource availability into account, that is by the library framework itself, or is delegated to the PEPPHER run-time. Thus, variant selection in PEPPHER is always external to the implementation of the variant.

D. Compilation techniques in PEPPHER

OpenCL is a possible portability layer for current heterogeneous systems, especially such that employ GPUs, but of a low abstraction level and not in itself solving the performance portability problem. Therefore, efficient compilation from C++ to OpenCL can support both the component model and the run-time system. Specifically, PEPPHER develops a C++ extension termed OffloadCL that allows for explicit compilation and offloading to GPUs but also to Cell SPEs with the compiler taking care of the necessary call graph duplication, functional duplication, and replication of host data. The Offload compiler interfaces directly with the PEPPHER run-time system by encapsulating offloadable code as PEPPHER component tasks [6].

E. Hardware support and feedback

In order to provide a larger spectrum of heterogeneous target architectures and to be able to investigate possibilities for hardware support for performance monitoring and portability, PEPPHER develops the highly configurable hardware simulator PeppherSim. This simulator makes it possible to run benchmarks/kernels on architecture configurations that are not physically available, and experiment in a controlled way with the performance portability that can be achieved through the PEPPHER framework. The simulator enables “what if” types of experiments to determine performance bottlenecks, as well as to investigate new mechanisms for more detailed performance monitoring and feedback, especially energy consumption feedback. Also, new synchronization primitives, e.g., [11], and other architecture support for algorithms and run-time can be investigated with the simulator.

F. 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, server/enterprise/general-purpose, to high-performance computing domains. In addition, important numerical kernels have been included as manageable test cases, that will also be useful as library components. Table III summarizes the benchmarks and kernels and the target architectures for which component variants already exist.

IV. A LARGER EXAMPLE

A final example illustrates how PEPPHER can provide efficient utilization of heterogeneous compute resources and performance portability across systems with varying numbers of GPU type accelerators.

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 5, and these kernels are turned into PEPPHER components by giving suitable XML interface specifications. Expert implementation variants of the BLAS kernels for CPUs and GPUs are already available through existing GPU and CPU libraries, namely PLASMA [4] and MAGMA [19], and are added as component variants

Application	x86 multi-core CPU	NVidia GPU	Cell	PeppherSim
Enterprise/General-purpose				
Suffix array construction	X	X		
Games Physics Simulation	X	X	X	
High-Performance Computing				
GROMACS	X	X		
Embedded/Multi-media				
BZIP2	X	X		
Computational photography	X	X	X	
(Numerical) kernels				
MAGMA/PLASMA	X	X		X
RODINIA	X	X		X
FFTW	X	X		X
STL Algorithms	X	X		

TABLE III
THE PEPPHER BENCHMARKS.

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

Fig. 5. QR factorization code fragment for matrices of size NT $\times$ MT using BLAS routines.

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

TABLE IV

KERNEL COMPONENTS OF THE QR APPLICATION. THE TABLE SHOWS THE PERFORMANCE OF THE KERNELS ON CPU AND GPU, RESPECTIVELY.

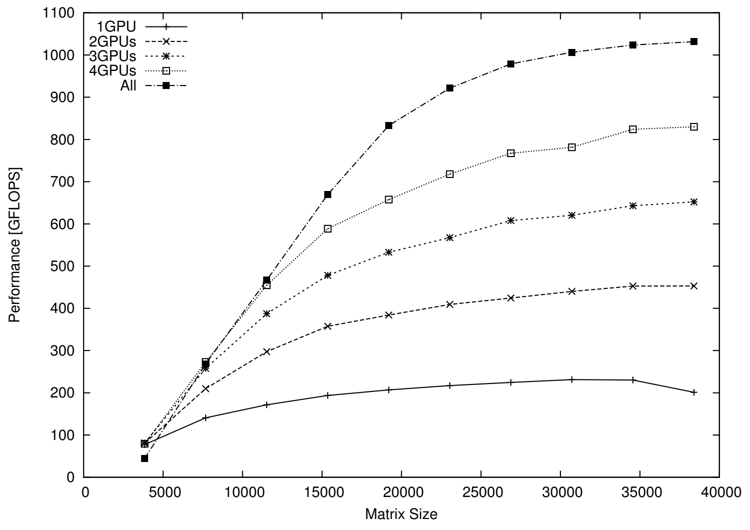


Fig. 6. QR factorization on a 16-core AMD system with up to 4 NVidia GPUs (C1060).

in the PEPPHER implementation. Figure 6 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 Opteron 8358 SE CPU cores (16 cores total) running at 2.4 GHz with 32 GB of memory divided into four NUMA nodes. It is accelerated with four NVIDIA Tesla C1060 GPUs of 240 cores each (960 GPU cores total) running at 1.3 GHz with 4 GB of GDDR3 memory (102 GB/s) per GPU. One CPU is reserved for each GPU, leaving 12 CPUs available for computational work. The performance of the kernels of the QR application is shown in Table IV, which also shows the speed-up achievable on the GPU relative to a single CPU core.

As can be seen, the SOMQR 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 is able to exploit the remaining 12 CPUs 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 from the 12 CPUs (about 200 GFlops) than would have been possible by running the whole QR application on these 12 CPUs (only 150 GFlops) alone. In the experiment 20% of the SGQRT tasks are scheduled on the GPUs, whereas for the SSSMQR tasks over 90% are executed on the GPUs, see again the speed-up ratios in Table IV.

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.

V. Conclusion

This paper outlined the PEPPHER approach to achieving performance portability and programmability for hybrid (heterogeneous) many-core architectures, particularly CPU-GPU type systems, and described the current state the project. PEPPHER combines a compositional, performance aware software framework, auto-tunable, adaptive algorithmic libraries, specific compilation techniques and an efficient run-time, and is thereby 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, and we aim to substantiate this with the PEPPHER project.

A. *Project facts*

PEPPHER is a European Union funded (FP7) project that started in January 2010. PEPPHER will last until the end of 2012. The PEPPHER partners are the Universities of Vienna (Austria), Chalmers (Sweden), Linköping (Sweden), and Karlsruhe (Germany), the French research center INRIA in Bordeaux, the European SMEs Codeplay (United Kingdom) and Movidius (Ireland), and Intel Labs Europe. More information can be found on the project web-site www.peppher.eu.

Acknowledgment

This work was supported by the EU as part of the FP7 Project PEPPHER under grant 248481. We gratefully acknowledge the contributions of all people involved in the PEPPHER project to the work presented in this paper. The authors also thank the anonymous reviewers for constructive feedback.

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] U. Dastgeer, J. Enmyren, and C. Kessler. Auto-tuning SkePU: A multi-backend skeleton programming framework for multi-GPU systems. In *4th International Workshop on Multicore Software Engineering (IWMSE11) at International Conference on Software Engineering*, 2011.

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

[9] K. Fatahalian, D. R. Horn, T. J. Knight, L. Leem, M. Houston, J. Y. Park, M. Erez, M. Ren, A. Aiken, W. J. Dally, and P. Hanrahan. Sequoia: Programming the memory hierarchy. In *ACM/IEEE Supercomputing*, page 83, 2006.

[10] A. Gidenstam, H. Sundell, and P. Tsigas. Cache-aware lock-free queues for multiple producers/consumers and weak memory consistency. In *Proceedings of the 14th International Conference on Principle of Distributed Systems (OPODIS 2010)*, volume 6490 of *Lecture Notes in Computer Science*, pages 302–317. Springer, 2010.

[11] 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.

[12] 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.

[13] 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.

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

[15] 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.

[16] 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.

[17] J. Singler, P. Sanders, and F. Putze. MCSTL: The multi-core standard template library. In *Euro-Par 2007, Parallel Processing, 13th International Euro-Par Conference*, volume 4641 of *Lecture Notes in Computer Science*, pages 682–694. Springer, 2007.

[18] N. Thomas, G. Tanase, O. Tkachyshyn, J. Perdue, N. M. Amato, and L. Rauchwerger. A framework for adaptive algorithm selection in STAPL. In *Proceedings of the ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*, pages 277–288. ACM, 2005.

[19] 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.

[20] H. Topcuoglu, S. Hariri, and M.-Y. Wu. Performance-effective and low-complexity task scheduling for heterogeneous computing. *IEEE Transactions on Parallel and Distributed Systems*, 13(3):260–274, 2002.

[21] J. L. Träff and M. Wimmer. 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.

[22] 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.

Siegfried Benkner is full professor at the Faculty of Computer Science at the University of Vienna where he heads the Research Group of Scientific Computing. He received M.Sc. and

Ph.D. degrees in Computer Science from the Vienna University of Technology in 1988 and 1994, respectively. His research interests include languages, compilers and runtime systems for parallel and distributed systems, Service-oriented software architectures, as well as Grid and Cloud computing. Siegfried Benkner contributed to several EU projects, including PPPE, PREPARE, HPF+, GEMSS, @neurIST, and was the Technical Director of the LTR project HPF+. Currently Benkner's group coordinates the EU Project PEPPHER with a focus on higher-level support for programming heterogeneous many-core architectures. Siegfried Benkner has published some 100 peer-reviewed publications and is a member of the ACM and the IEEE.

Contact Information:
University of Vienna
Faculty of Computer Science
Research Group of Scientific Computing
Nordbergstrasse 15/3C
1090 Vienna
Austria
Phone: 0043 1 4277 39417
Fax: 43 1 4277 9394
Email: `sigi@par.univie.ac.at`

Sabri Pllana is a senior research scientist at the Research Group of Scientific Computing, University of Vienna. His research is currently focused on performance-oriented software engineering for parallel and distributed systems. He holds a Ph.D. degree in computer science from the Vienna University of Technology. Sabri Pllana has contributed to several EU-funded projects and is currently serving as project coordinator for the FP7-project PEPPHER. He is member of the IEEE, member of the HiPEAC network of excellence, and member of the PlanetHPC network.

Contact Information:
University of Vienna
Faculty of Computer Science
Research Group of Scientific Computing

Nordbergstrasse 15/C3

1090 Vienna

Austria

Phone: +43 1 4277 39411

Fax: +43 1 4277 9394

Email: pllana@par.univie.ac.at

Jesper Larsson Träff received an M.Sc. in computer science in 1989, and, after two years at the industrial research center ECRC in Munich, a Ph.D. in 1995, both from the University of Copenhagen. He spent four years as a Research Associate in the Algorithms Group of the Max-Planck Institute for Computer Science in Saarbrücken, and the Efficient Algorithms Group at the Technical University of Munich. From 1998 until late 2009 he was working at the NEC Laboratories Europe in Sankt Augustin, Germany on efficient implementations of MPI for NEC vector supercomputers. This work led to a doctorate (Dr. Scient.) from the University of Copenhagen in 2009. Since 2010 he is Professor for Scientific Computing at the University of Vienna. His research interests are broadly in parallel processing and include interfaces, algorithms, and architectures. He is currently scientific coordinator for the European FP7 project PEPPHER. With Martti Forsell he organizes the annual Euro-Par Workshop on Highly Parallel Processing on a Chip (HPPC).

Contact Information:

University of Vienna

Faculty of Computer Science

Research Group of Scientific Computing

Nordbergstrasse 15/3C

1090 Vienna

Austria

Phone: 0043 1 4277 39432

Fax: 43 1 4277 9394

Email: traff@par.univie.ac.at

Philippas Tsigas's research interests include concurrent data structures for multiprocessor systems, communication and coordination in parallel systems, fault-tolerant computing, mobile computing and information visualization. He received a BSc. in Mathematics from the University of Patras, Greece and a Ph.D. in Computer Engineering and Informatics from the same University. Philippas Tsigas was at the National Research Institute for Mathematics and Computer Science, Amsterdam, the Netherlands (CWI), and at the Max-Planck Institute for Computer Science, Saarbrücken, Germany, before. At present he is a professor at the Department of Computing Science at Chalmers University of Technology, Sweden.

Contact Information:

Chalmers Tekniska Högskola AB

Department of Computer Science and Engineering

41296 Göteborg

Sweden

Phone: 0046 31 7725409

Fax: 0046 31 772 3663

Email: `tsigas@chalmers.se`

Uwe Dolinsky completed his Dipl.Inform. (FH) at the University of Wismar in 1997 and received his Ph.D. in Engineering/Computer Science from John Moores University Liverpool in 2001. Since 2001 Uwe Dolinsky has been working for Codeplay where he is currently CTO and in charge of researching and developing novel compiler and software optimisation technologies (e.g., Offload C++) for multicore processors. Besides leading compiler projects for various customers, he has worked on programming models and compiler implementations to ease the offloading of complex code onto heterogeneous accelerator cores for performance (e.g., PS3/SPU, GPU). He has authored and co-authored papers on software engineering and other subjects published by ACM, IEEE and others. He also contributed as inventor to a number of patents for Codeplay.

Contact Information:

Codeplay Software Ltd.

York Place 45

EH1 3HP Edinburgh
United Kingdom
Phone: 0044 131 466 0503
Fax: 0044 131 557 6600
Email: `uwe@codeplay.com`

Cédric Augonnet is a PhD student at the University of Bordeaux. He is part of the RUNTIME team at INRIA Bordeaux. His interests include task scheduling and hybrid accelerator-based machines. He is one of the developers of the StarPU runtime system. He holds an MSc. in computer science from the University of Bordeaux and a BSc. in computer science from the ENS Lyon.
Contact Information:
Institut National de Recherche en Informatique et en Automatique
INRIA Bordeaux Sud-Ouest
Cours de la Liberation 351
33405 Talence Cedex
France
Email: `cedric.augonnet@inria.fr`

Bev Bachmayer holds a Bachelor's degree in Computer Science from the University of Oregon (1983) and an MBA from Portland State University (1992). She is a member of the IEEE, ACM and the European Professional Women's Association (EPWN). Bev Bachmayer has worked in diverse software engineering, program management and engineering management positions in the US and Europe during her more than 28 years at Intel. Currently a technical consulting engineer, working in the Software & Solutions Group at Intel GmbH, her key area of interest is performance analysis and optimization of software on new computer architectures. Additionally, Bev supports increasing the number of professional females entering computer science/engineering programs worldwide through multiple projects.
Contact Information:
Intel GmbH
Software and Solutions Group

Dornacher Strasse 1
85622 Feldkirchen
Germany
Phone: 0049 89 99143 482
Fax: 0049 89 99143 924
Email: `bev.bachmayer@intel.com`

Christoph Kessler is a professor at the Department of Computer and Information Science of Linköping University, Sweden, where he leads the research group on compiler technology and parallel computing at the Programming Environments Laboratory (PELAB). He holds a PhD degree in Computer Science from Saarbrücken University, Germany, and a Habilitation degree in computer science from the University of Trier, Germany. His research interests include parallel programming, compilers, and software composition. He is a member of the ACM and the IEEE Computer Society.

Contact Information:
Linköpings Universitet
Department of Computer and Information Science (IDA)
Linköping University
S-58183 Linköping
Sweden
Phone: 0046 13 282406
Fax: 0046 13 285899
Email: `christoph.kessler@liu.se`

David Moloney received a B.Eng. from Dublin City University in 1985, and Ph.D. in Engineering from Trinity College Dublin in 2010. Since 1985 he worked for Siemens Halbleiter AG (Infineon) in Munich and ST Microelectronics in Milan as a DSP IC designer, before returning to Ireland 1994 to co-found a series of start-up technology companies including Parthus (CEVA) and Silansys (Frontier-Silicon). David Moloney is currently co-founder (2005) and CTO of Movidius Ltd., a fabless semiconductor company headquartered in Dublin and focused on the design of software programmable multimedia accelerator SoCs. He holds 18 US patents with many others

in process as well as authoring conference and journal papers on DSP and computer architecture. David Moloney is a member of the IEEE.
Contact Information:
Movidius Ltd.
Mountjoy Square East 19
D1 Dublin
Ireland
Phone: 00353 872837494
Fax: 00353 1 8559592
Email: `david.moloney@movidius.com`

Vitaly Osipov received his M.Sc. in mathematics from Ural State University (Yekaterinburg, Russia) in 2006, his M.Sc. in Computer Science from Saarland University (Germany) in 2007. Currently he is a Ph.D. student in Karlsruhe Institute of Technology (Germany) under the supervision of Prof. Peter Sanders. Vitaly Osipov does research in the field of graph algorithms for modern architectures such as external memory algorithms and parallel algorithms. He is a co-author of a number of papers published in IPDPS, ESA and ALENEX.
Contact Information:
Karlsruher Institut für Technologie
Am Fasanengarten 5
76128 Karlsruhe
Germany
Phone: 0049 721 608 4232
Fax: 0049 721 608 3088
Email: `osipov@kit.edu`