Programmability and Performance Portability Aspects of Heterogeneous Multi-/Manycore Systems

Christoph Kessler*, Usman Dastgeer*, Samuel Thibault†, Raymond Namyst†,
Andrew Richards‡, Uwe Dolinsky‡, Siegfried Benkner§, Jesper Larsson Träff¶ and Sabri Pllana§

*Linköping University, S-58183 Linköping, Sweden. Email: firstname.lastname@liu.se
†University of Bordeaux, INRIA, Bordeaux, France. Email: firstname.lastname@labri.fr
‡Codeplay Software Ltd., Edinburgh, UK. Email: {andrew,uwe}@codeplay.com
§University of Vienna, Austria. Email: {sigi,pllana}@par.univie.ac.at
¶Technical University of Vienna, Austria. Email: traff@par.tuwien.ac.at

***Abstract*—We discuss three complementary approaches that can provide both portability and an increased level of abstraction for the programming of heterogeneous multicore systems. Together, these approaches also support performance portability, as currently investigated in the EU FP7 project PEPPHER. In particular, we consider (1) a library-based approach, here represented by the integration of the SkePU C++ skeleton programming library with the StarPU runtime system for dynamic scheduling and dynamic selection of suitable execution units for parallel tasks; (2) a language-based approach, here represented by the Offload-C++ high-level language extensions and Offload compiler to generate platform-specific code; and (3) a component-based approach, specifically the PEPPHER component system for annotating user-level application components with performance metadata, thereby preparing them for performance-aware composition. We discuss the strengths and weaknesses of these approaches and show how they could complement each other in an integrational programming framework for heterogeneous multicore systems.**

I. Introduction

The need to improve the performance/energy ratio has caused a general trend towards increased heterogeneity in multi- and manycore systems, where general-purpose computing cores are complemented with energy-efficient special-purpose accelerators located either on or off-chip, as in, e.g., GPU-supported systems. However, this trend has also brought new, fundamental problems for the design, optimization and maintenance of software for such systems. Current programming approaches are either platform-specific, such as CUDA for Nvidia GPUs, or are portable but at a low level of abstraction, as with, e.g., OpenCL [11]. OpenCL requires explicit coding of data transfer, memory management, kernel launch etc., *and* re-optimization and adaptation to obtain decent performance when migrating to a new device configuration. We see a main challenge in how to combine two seemingly conflicting goals, namely (1) *programmability* of heterogeneous multi-/manycore systems, i.e., guaranteeing portability while raising the level of abstraction and leveraging modern software engineering technology, and (2) *performance portability*, i.e., supporting a best-effort automated adaptation of code to effectively utilize the underlying architecture.

This paper elaborates on this challenge and summarizes our ideas and contributions towards a solution, as presented in a hot-topic special session at DATE-2012. In particular, we present several key approaches and tools:

- the StarPU runtime system, based on performance-aware dynamic scheduling and selection of task variants to automatically run on the most suitable type of execution unit in the context of heterogeneous systems, combined with the SkePU skeleton programming library as front-end to provide abstraction and more parallelism (Section II);
- the Offload-C++ language, an extension of C++ for high-level portable programming of accelerator-based systems, and its compiler to OpenCL (Section III);
- the PEPPHER component model and methodology to achieve performance portability for applications based on annotated software components that expose their implementation variants, performance-relevant metadata and tunable parameters in an extended composition interface (Section IV).

We conclude in Section V by comparing these approaches to each other and describing how their strenghts could be combined in an integrated programming framework for heterogeneous multi-/many-core systems.

II. Skeleton programming with heterogeneous run-time system

A. The StarPU runtime system

StarPU [1] is a C-based runtime system capable of scheduling graphs of tasks onto heterogeneous multicore platforms. It uses the concept of *codelet*, a C structure containing different implementations of the same functionality for different computation units (e.g., CPU and GPU). A StarPU *task* is then an instance of a codelet applied to some data. The programmer has to explicitly submit all tasks and register all the input and output data for all tasks. Thanks to a dynamic database of autotuned per-task performance models, the runtime system can perform intelligent task scheduling on variety of platforms without requiring any manual modifications in the program. StarPU provides a software virtual shared memory abstraction

978-3-9810801-8-6/DATE12/©2012 EDAA

by keeping track of data copies in accelerator-embedded memory and features a data-prefetching engine.

B. The SkePU skeleton programming library

Skeletons are generic components derived from higher-order functions that describe common computation structures that could be mapped to parallel or platform-specific implementations. Skeletons are instantiated with problem-specific sequential user code.

SkePU [7] is a generic, tunable skeleton programming framework developed in C++ for single- and multi-GPU systems. It currently implements several data-parallel skeletons including map, reduce, mapreduce, map-with-overlap, map-array, and scan, providing multiple implementations for each skeleton type (CUDA, OpenCL, OpenMP and sequential C). The SkePU skeletons accept SkePU generic containers (Vector, Matrix) as arguments, which implicitly manage the data transfers between host and GPU memory and keep track of multiple copies of the data residing on different memory units. As shown in the earlier work [5], SkePU provides higher programming abstraction while retaining performance close to hand-written code for real-world applications.

C. SkePU-StarPU integration

To support performance portability as offered by the StarPU runtime system at a higher programming abstraction, we have integrated the SkePU library with StarPU. Behind the generic SkePU interface, the SkePU skeleton calls are mapped to one or more StarPU tasks, generating task-parallelism for the runtime system. Several programming techniques (e.g., usage of static member functions etc.) are employed to integrate SkePU written in C++ with the pure C-based StarPU system. Furthermore, support for different StarPU features, such as data partitioning and different scheduling policies (e.g., history based performance models) is implemented to exploit performance gains for a variety of situations. Experiments on several real-world applications have shown the usefulness of this integration, in terms of programmability and performance portability.

Figure 1 shows the performance of executing a Coulombic potential application written using SkePU skeletons on a hybrid system containing one GPU and multi-core CPUs. Thanks to the integration with the runtime system, we are able to seamlessly use both CPU and GPU devices present in the system in parallel for the computation work. As shown in Figure 2, the same application when ported to a system containing two high-end and one low-end GPU is able to use all devices in the system and scales well over the GPU devices present in the system. During this porting process, no changes in the application code are required as the runtime system is managing the selection and scheduling decisions. Figure 2 also shows the overhead of the SkePU-StarPU integrated approach by comparing it to a hand-written CUDA code.

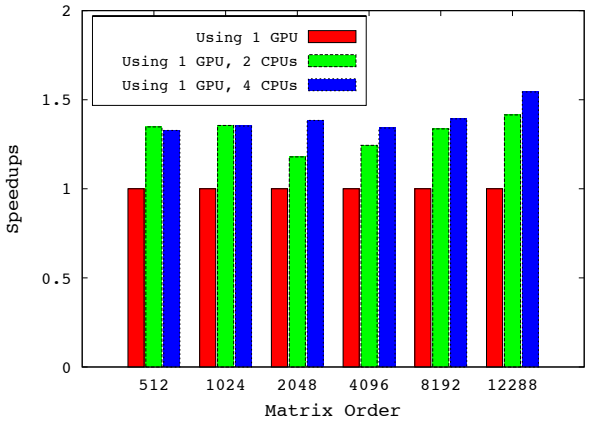

Fig. 1: Coulombic potential grid execution on a heterogeneous platform (CPUs and GPU) for different matrix sizes. The base-line is SkePU-generated StarPU code using CUDA.

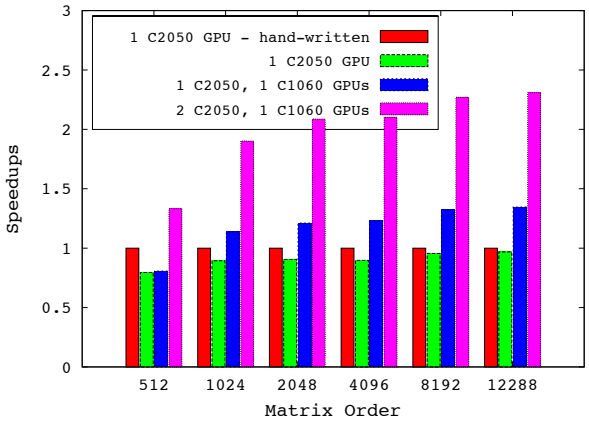

Fig. 2: Coulombic potential grid execution on a heterogeneous platform containing three NVIDIA (2 C2050 and 1 C1060) GPUs; the base-line is hand-written CUDA code.

III. High-level programming and compiling for heterogeneous systems

Using Offload [4] Codeplay has developed flexible techniques for using C++ to achieve performance portability, and ease of programming across a range of different accelerator devices, including Cell BE, GPUs and FPGAs. These techniques are already used by game developers and are becoming available for use by wider range of software developers. In this section we describe the key features of Offload and how different requirements of different hardware are addressed.

A. Fundamentals of Offload C++

The key language feature of Offload C++ are `__offload` blocks which are sections of code that run on accelerator processors. Without significant code changes this enables complex C++ accelerator code to be embedded inside host code and interact with data and code on the host. This interaction is usually transparent to the programmer as the

Offload C++ compiler transforms any data accesses to host data into calls into the Offload runtime (software cache or DMA), whereas called functions are transparently cloned for the accelerator processor employing a process named call-graph duplication [4]. These transparent mechanisms not only enable the development of portable software across different heterogeneous devices as Offload extensions can be buried inside libraries with standard C++ interfaces. This automated offloading process also enables the programmer to focus more on performance tuning for example by placing different types of offload blocks (synchronous or asynchronous) in different parts of the code and by using local data caching techniques [3].

Codeplay's Offload technology was originally developed for PlayStation©3 developers. The intention was to offload single threads from the main CPU core onto the accelerator cores of the PlayStation©3 (the cores called SPUs). In this use-case no parallelization was necessary, as the task-parallelism used in games engines just requires individual tasks to be moved onto SPUs to gain maximum performance.

B. *Offload C++ for OpenCL Devices*

As part of the PEPPHER project Codeplay has adapted its Offload technology to work with OpenCL-based GPUs. The main features added to Offload compiler and runtime were support for data-parallelism, OpenCL- specific memory spaces and OpenCL buffers. Essentially offload blocks are compiled by the OffloadCL compiler into OpenCL kernels[1]. The Offload runtime has been adapted to work on top of the OpenCL runtime (indirectly through the PEPPHER runtime system, i.e., StarPU) as OpenCL performs runtime compilation of kernels.

```
void offloadCLExample ( int width, int height,
                        float *myFloatArray )
{ GpuBuffer<float, 2> myGpuBuff
                (width, height, myFloatArray);
  myGpuBuff.unmap(); //move data onto device
  parallel_for( width, height,//process buff
     processBuffer (myGpuBuff) //in parallel
  );
  myGpuBuff.map(); // map result to host
}
```

In the above example, we process an array of data (this is the data parallelism of GPUs). The array has a width, a height, and we get a pointer to the data, myFloatArray. We need to un-map the array from host (CPU) onto the device (GPU), then process it in parallel on the device, then map it back to host. What we have done here is hide most of the complexity of an OpenCL buffer object inside a C++ class.

This enables developers to put any array they want to process on GPU inside such a class, and most of the complexity of mapping to from GPU can be handled. We could use a RAII[2] class that automatically handles map and unmap, but for best performance optimization, we sometimes need to leave it to the programmer. Figure 3 shows the data mapping between host and device memory.

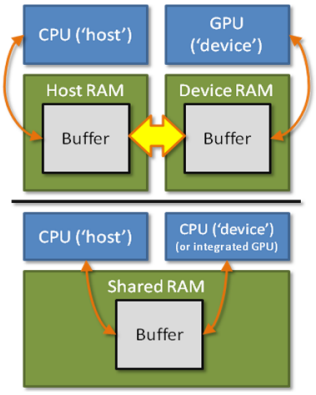

Fig. 3: Mapping data between host and GPU memory

To handle the data parallelism we use the `parallel_for` function. This is a standard C++ way of handling parallelism. In C++11, we can use a lambda-function, which makes programming easier. Here, we have used a C++ functor, which is an object that can be used like a function. In OpenCL, a *buffer* is a way of handling the fact that data shared between host and device might be in a different memory chip, at a different address, copied or shared, or use different pointer sizes (32-bit vs 64-bit, etc.). This requires that on the device side we need a different data-type for the buffer. We therefore provide a translation system where any GpuBuffer parameter is translated to a GpuKernelBuffer value on the device.

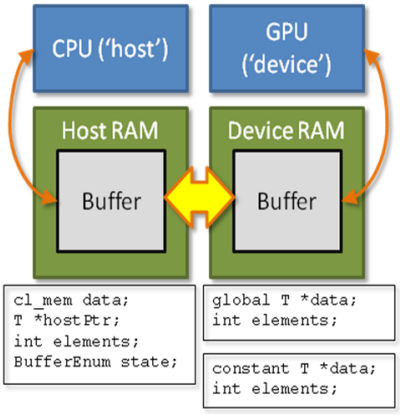

Fig. 4: Data elements on CPU and GPU

On the host side, a buffer has a `cl_mem` object, has a pointer to the host data, has a size, has some state (mapped/unmapped/in-use/in-transit etc). On the kernel side, a buffer is a pointer to global or constant memory, and has a size. In OpenCL, the transition from buffer to global or constant pointer is only handled by clSetKernelArg. The transition cannot be done anywhere else. It cannot be called in a kernel, and cannot be called beforehand. So, we need to create a parallel function call that passes in some buffers, but the

[1]Restrictions imposed by the current OpenCL specification limit the set of compilable C++ features inside an offload block

[2]Resource Acquisition Is Initialization

function receives global and local pointers. Any data accessed by a kernel must go through this process.

C++ lets us write a lot of complex forms of parallelism in a way that is portable between different compilers, by hiding implementation details in generic classes. For example, we have been able to compile example programs produced for Microsoft's C++AMP[3] GPU compiler by implementing compatible version of the C++AMP classes. The critical issue is the translation of types between CPU and GPU, which we implement by defining special linked classes with a CPU and GPU implementation.

IV. The PEPPHER Component Framework

PEPPHER targets programmability and performance portability for single-node heterogeneous manycore architectures by means of a component-based approach to parallel program development in combination with advanced compilation and run-time technology. In the following we overview the PEPPHER component model, and describe high-level language features for constructing applications using components together with the associated transformation framework.

A. The PEPPHER Component Model

A basic premise underlying PEPPHER is that for the efficient utilization of heterogeneous parallel architectures, different programming models and APIs, tailored and optimized for the different types of architectural components, have to be combined within an application. In PEPPHER performance-critical parts of a C/C++ application (typically functions) are realized by means of multi-architectural components that encapsulate behind an interface different implementation variants of a function tailored to different architectural components (e.g., CPU and GPU) of a heterogeneous manycore system.

Component interfaces and implementation variants are accompanied with rich meta-data, kept in external XML descriptors, describing the parameter intent (in, out, inout) and non-functional properties of implementation variants, including information about resource requirements, possible target platforms, and performance relevant parameters [13].

PEPPHER components have been designed to support the prediction of performance aspects (e.g., execution time) by associating corresponding performance prediction models with implementation variants. Based on these models, the PEPPHER framework attempts to optimize performance by selecting (at runtime) best suited component variants and target execution units based on performance predictions.

Figure 5 depicts our performance modeling approach in the context of PEPPHER. The performance model for a PEPPHER component is built by an *expert programmer* (that is the component developer) or is generated automatically by the PEPPHER framework using historical performance data. The performance data needed to build the model is provided by the PEPPHER simulator or by measurements of component execution on a specific platform.

[3]Accelerated Massive Parallelism; specification for heterogeneous computing by Microsoft

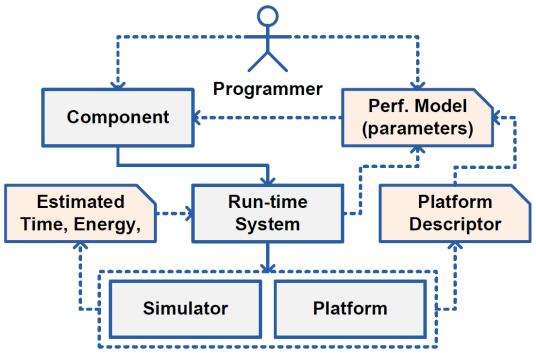

Fig. 5: Overview of performance modeling in PEPPHER. Performance models are provided by the component developer or automatically generated by the run-time system.

Beside analytical performance modeling techniques, in PEPPHER we use an instruction level simulator (PeppherSim) for performance evaluation. An interesting feature of PeppherSim is estimation of energy consumption in addition to the execution time of a program. PEPPHER run-time system uses the output of PeppherSim to generate analytical models of energy consumption based on linear or quadratic regression.

Component implementation variants are usually written by expert programmers using different programming APIs (e.g., CUDA, OpenCL) or are taken from optimized vendor-supplied libraries. Non-expert programmers (e.g., domain scientists) construct applications at a higher level of abstraction by invoking component functionality using conventional interfaces and simple source code annotations to delineate asynchronous (or synchronous) component calls.

B. The PEPPHER Composition Tool

The PEPPHER composition tool [6] performs basic composition, i.e., call-to-callee binding with optional static preselection of execution unit and implementation variant) of PEPPHER components. It parses the XML-based metadata specifications of components and their implementation variants and generates stubs (glue code) that intercept calls and prepare for selection and execution on the appropriate target platform, by default by creating a task for the (StarPU) runtime system.

C. The PEPPHER Coordination Language

The PEPPHER coordination language aims to enable incremental transformation of existing (sequential) C/C++ applications for efficient execution on heterogeneous manycore architectures by offering a set of annotations (pragmas) for coordinating invocations of PEPPHER components. The basic coordination abstractions enable synchronous (blocking) and asynchronous (non-blocking) invocation of component-provided functionality in PEPPHER applications and components. This allows to express inter-component parallelism while delegating to the runtime the actual exploitation of parallelism through dynamic task scheduling.

The following code snippet shows two annotated call sites, each of which are realized by means of multi-architectural

components. Selection of the best implementation variant is delegated to the runtime system.

```
#pragma pph call
cf1(A, N);  // non-blocking component call
...
#pragma pph call
cf2(B, M);
```

Here, the `call` annotation indicates a non-blocking call. Since no data dependencies exist between the two calls, both components may be scheduled for parallel execution, provided enough execution resources are available at runtime. Further, optional features are provided to support the specification of performance requirements and constraints, data partitioning information and access patterns, and preferred execution targets for components.

A number of higher-level coordination primitives are provided for supporting parallel patterns, e.g., pipelining [10]. The `pipeline` construct indicates that the subsequent `while` loop is a pipeline. Within the loop body, each stage of the pipeline corresponds to a call to a multi-architectural component with different implementation variants. By means of the `buffer` clause the order strategy (*priority*, *random*, and *fifo*) and size of buffers may be specified. The required data structures to implement buffers between pipeline stages are generated automatically by the transformation system based on an analysis of the data packets passed between pipeline stages. The `stage` construct may be used to merge several component calls into a single stage. The `replicate` clause is provided to control stage replication, by automatically generating multiple instances of a stage, which may operate in parallel on different data packets. Stage replication aims at increasing pipeline throughput by replicating stages with (relatively) high execution times. — The following code excerpt shows an example of a face detection pipeline which has been implemented within the PEPPHER framework using the OpenCV image processing library [8].

```
unsigned int N = get_max_execution_units();
#pragma pipeline with buffer(PRIORITY,N*2)
while(image.number < 32) {
  readImage(file,image);
  #pragma stage replicate(N) {
     resizeAndColorConvert(image);
     detectFace(image,outImage);
  }
  writeFaceDetectedImage(file,outImage);
}
```

D. The PEPPHER Transformation System

We have developed a prototype source-to-source compiler that transforms C/C++ applications with the described annotations into C++ with calls to a runtime coordination layer that utilizes the StarPU heterogeneous runtime system [1]. The runtime system schedules stages for parallel execution onto the execution units of each heterogeneous manycore system. The source-to-source compiler has been implemented using the ROSE compiler framework [12]. An overview of the framework is shown in Figure 6.

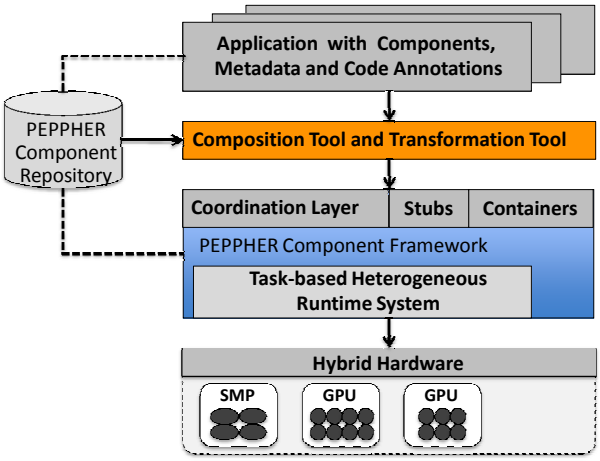

Fig. 6: Overview of the PEPPHER Composition and Transformation System

Below we show initial performance results for the face detection pipeline shown above on a CPU/GPU architecture and compare it to an implementation using Intel Threading Building Blocks (TBB). With our framework, for the detection stage two different implementation variants for CPUs and GPUs, are provided, while Intel TBB cannot take advantage of GPUs. Performance evaluation was performed for different image resolutions including VGA(640x480), SVGA(800x600), XGA(1024x768), and QXGA(2048x1536) on a a heterogeneous system consisting of two Intel Xeon X5550 (4 cores, 2.67 GHz), one Nvidia GeForce GTX 480 (480 Cores, 1.5GB, 1.40GHz), and one Nvidia GeForce GTX 285 (240 Cores, 1GB, 1.48GHz). The system uses the latest CUDA 4.0, and runs Red Hat Enterprise Linux 5.6.

	VGA	SVGA	XGA	QXGA
Intel TBB (1 Core)	12.75	20.07	35.15	145.68
PEPPHER (1 Core)	9.62	14.33	24.94	111.45
PEPPHER (1 Core + 1 GPU)	3.94	5.91	10.35	46.30
PEPPHER (1 Core + 2 GPUs)	2.95	2.72	6.53	30.81
Intel TBB (8 Cores)	1.47	2.29	4.13	17.40
PEPPHER (8 Cores)	1.18	1.78	3.58	13.69
PEPPHER (7 Cores + 1 GPU)	1.13	1.63	2.91	11.89
PEPPHER (6 Cores + 2 GPUs)	0.94	1.40	2.44	10.71

TABLE I: OpenCV execution times (seconds)

Table I shows performance results for the two different OpenCV image processing code versions. Using only the CPU cores, we get slightly better results than with TBB. As opposed to TBB, however, our approach can automatically take advantage of the GPUs by utilizing the GPU implementation variant for the (merged) middle pipeline stage. Since for each GPU an additional CPU core is required, the number of usable general purpose cores is reduced accordingly. With one CPU core and one GPU (GTX 460) the execution time is reduced by a factor of up to 3.14 compared to the TBB version using one CPU core. Using now a second GPU results, however, in

Approach	Code Portab.	Flexi-bility	Progr. Effort	Abstrac-tion	Perform. Portability
Platform-specific e.g. CUDA	No	High	Medium	Low	No
OpenCL	Yes	High	High	Low	No
StarPU	Yes	Medium	Medium	Medium	Yes
SkePU	Yes	Low	Low	High	Yes
SkePU+StarPU	Yes	Medium	Low	High	Yes
Offload-C++	Yes	Medium	Low	High	No
PEPPHER components	Yes	High	Medium	High	Yes
PEPPHER coordination	Yes	High	Low	High	Yes

TABLE II: Summary of the programming systems mentioned

only modest further performance improvements. These initial results indicate that our high-level approach to pipelining has the potential to outperform TBB, while significantly improving programmability. Based on our concept of multi-architectural components together with a versatile heterogeneous runtime system, we can take advantage of a heterogeneous CPU/GPU-based architecture without modifying the high-level application code.

V. Conclusion

We have considered three different and complementary approaches to high-level portable programming of heterogeneous multicore systems (see also Table II):

- The library approach (here, SkePU skeleton programming library and StarPU run-time system),
- The language approach (here, Codeplay's Offload-C++ language and compiler), and
- The component approach (here, the PEPPHER component model and transformation system).

Each of these approaches can be used stand-alone and provides portability, compared to OpenCL with better programmability thanks to a higher level of abstraction. When starting from a (well-written) sequential legacy (C/C++) code, each of these approaches allows for an incremental parallelization process and requires only moderate restructuring of the code. While Offload requires, in principle, adding a few additional keywords only, the other two approaches are designed to additionally support performance portability. SkePU skeletons are ready-to-use generic components for most common data- and task parallel computation patterns, while PEPPHER components can encapsulate user-provided code of arbitrary complexity but require comprehensive metadata to support optimized composition [9].

We are therefore working towards an integration of these approaches in the PEPPHER framework. For instance, Offload-C++ can be used to code certain accelerator-based implementation variants of a PEPPHER component at high level and compile it to OpenCL or platform-specific target code, while either leaving selection and scheduling to the underlying (StarPU) runtime system or being wrapped as PEPPHER components with additional metadata for static composition. As long as Offload-C++ code and coordination-annotated code are kept in separate source files and compiled separately, with the Offload compiler and the transformation tool respectively, these can be combined in the same application.

Where they fit the structure of the computation, SkePU skeleton calls can be used in the same way as PEPPHER components but are more light-weight and do not require user-supplied XML descriptors for their metadata specification. Both SkePU, the PEPPHER composition tool and the PEPPHER transformation framework target (also) StarPU as runtime system, and hence tasks coded with these different frameworks can be mixed at runtime. Finally, the transformation framework for annotations of SkePU or component calls can provide complex coordination patterns at a high level of abstraction while still allowing for optimized configuration, selection and scheduling of variants, thereby also achieving performance portability.

Acknowledgment

This research has been funded by EU FP7 project PEPPHER, www.peppher.eu, grant #248481.

References

[1] Cédric Augonnet, Samuel Thibault, Raymond Namyst, and Pierre-André Wacrenier. StarPU: A Unified Platform for Task Scheduling on Heterogeneous Multicore Architectures. *Concurrency and Computation: Practice and Experience, Special Issue: Euro-Par 2009* **23**(2), Feb 2011.

[2] Siegfried Benkner, Sabri Pllana, Jesper L. Träff, Philippas Tsigas, Uwe Dolinsky, Cedric Augonnet, Beverly Bachmayer, Christoph W. Kessler, David Moloney, and Vitaly Osipov. PEPPHER: Efficient and Productive Usage of Hybrid Computing Systems. *IEEE Micro* **31**(5), September/October 2011.

[3] Codeplay Software Ltd. *Cache classes reduce code changes* Offload C++ Knowledge Base Entry, http://offload.codeplay.com/kb/136.html, 2011

[4] Pete Cooper, Uwe Dolinsky, Alastair F. Donaldson, Andrew Richards, Colin Riley, and George Russell. *Offload - Automating Code Migration to Heterogeneous Multicore Systems*. 5th Int. Conf. on High Performance and Embedded Architectures and Compilers (HiPEAC'10), 2010, pp. 337–352

[5] Usman Dastgeer, Christoph Kessler, and Samuel Thibault. Flexible runtime support for efficient skeleton programming. Proc. ParCo-2011 conference, Ghent, Belgium, Sep. 2011, to appear.

[6] Usman Dastgeer, Lu Li, and Christoph Kessler. Performance-aware dynamic composition of applications for heterogeneous multicore systems with the PEPPHER Composition Tool. Proc. 16th Workshop on Compilers for Parallel Computers (CPC'12), Padova, Italy, January 2012.

[7] Johan Enmyren and Christoph Kessler: SkePU: A Multi-Backend Skeleton Programming Library for Multi-GPU Systems. Proc. 4th Int. Workshop on High-Level Parallel Programming and Applications (HLPP-2010), Baltimore, USA, Sep. 2010. ACM.

[8] B. Gary, *Learning OpenCV: computer vision with the OpenCV library*. O'Reilly USA, 2008.

[9] Christoph W. Kessler and Welf Löwe. Optimized composition of performance-aware parallel components. *Concurrency and Computation: Practice and Experience*, to appear. Published online in Wiley Online Library, DOI: 10.1002/cpe.1844, Sep. 2011.

[10] Erich Marth and Siegfried Benkner. Language support for pipelined applications on heterogeneous many-core architectures. In *Workshop on Hybrid Multi-core Computing, in conjunction with HiPC*, 2011.

[11] A. Munshi et al., The OpenCL Specification version 1.1, www.khronos.org/opencl, 2010.

[12] D. Quinlan, ROSE: Compiler Support for Object-Oriented Frameworks, *Parallel Processing Letters*, vol. 49, 2005.

[13] M. Sandrieser, S. Benkner, and S. Pllana. Improving programmability of heterogeneous many-core systems via explicit platform descriptions. In *Proc. 4th Int. Workshop on Multicore Software Engineering* (IWMSE'11), Hawaii, USA, May 2011, pp. 17–24, ACM.