In: *Proc. Second EXCESS Workshop*, Chalmers, Gothenburg, Sweden, 26 Aug. 2016.
©Copyright held by the authors.

Programming Frameworks for Optimized Software Composition for Parallel Systems

Christoph KESSLER [a]
[1] and Lu LI [a] and Erik HANSSON [a] and Nicolas MELOT [a] and August ERNSTSSON [a]
[a] *Linköping University, 58183 Linköping, Sweden*

Abstract. EXCESS (Execution Models for Energy-Efficient Computing Systems) is a European research project funded by EU FP7 2013–2016. The EXCESS project developed a holistic approach to energy optimization across the entire hardware/system software/application software stack. Targeting architectures and applications in both HPC and the embedded domain, its goal has been to bridge the gap between such systems and their tool chains by providing a generic, retargetable framework for programming and optimization. Work package WP1 of EXCESS, led by Linköping University, Sweden, investigates execution, platform and programming models for energy optimization. This presentation gives a high-level overview of the EXCESS programming and optimization framework.

Keywords. Energy optimization, energy modeling, performance modeling, platform modeling, XPDL, heterogeneous parallel system, context-aware software composition, skeleton programming, SkePU, smart containers, VectorPU, microbenchmarking, MeterPU, parallel streaming computations, global optimization, Drake

Beyond the traditional goal of pure computational performance, energy efficiency is currently the most important quality concern both in high-end embedded computing and in high-performance computing systems. For a best-effort optimization of programs and their execution environments for energy efficient execution, the EU FP7 project EXCESS (2013–2016) investigated a holistic approach considering the whole stack consisting of hardware, system software, libraries and application software.

Traditionally, programming frameworks and tool chains have been developed separately in the embedded computing domain and the HPC domain. One goal of the EXCESS project has been to develop techniques for modeling and optimization of computer systems, energy and performance behavior, and software building blocks that can span both domains, thereby bridging the gap in tool chain design.

In the following, we give a high-level overview of the EXCESS modeling, programming and optimization framework, towards

- systematic modeling of software components, performance, energy, and platforms, for high-level portable programming at multiple layers of abstraction;
- cross-layer, static and dynamic optimization for time and performance;
- composition of applications from their building blocks; and
- automatic deployment of programs for performance- and energy-aware execution especially on heterogeneous parallel target systems.

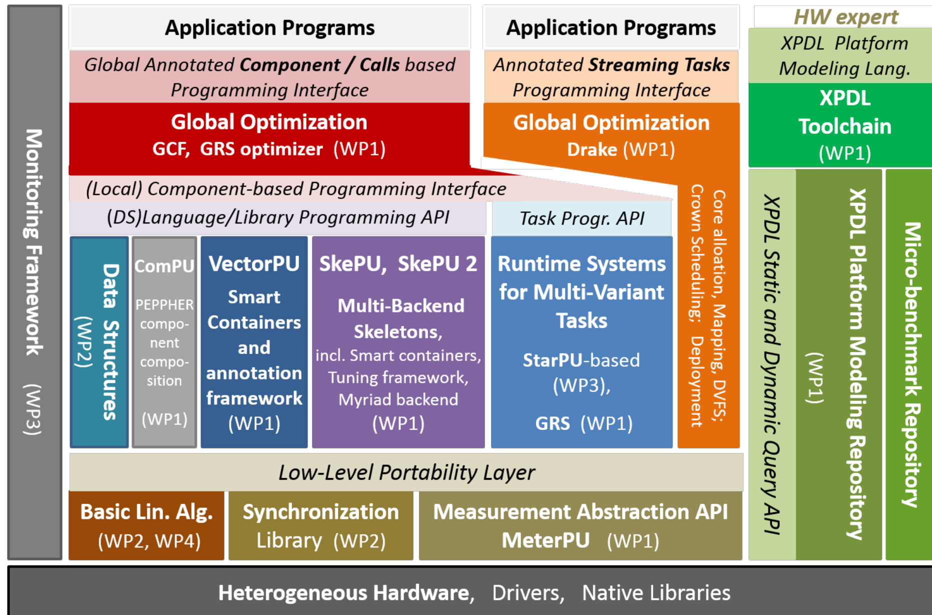

Figure 1. Layering structure of programming interfaces and prototypes in the EXCESS programming and tool infrastructure, with special emphasis on WP1 frameworks.

Figure 1 shows an overview of the different programming models / interfaces and prototypes developed in EXCESS WP1 and some other work packages towards a multi-layer, multi-programming-model infrastructure.

Overall, we distinguish between six different layers of EXCESS programming, i.e., different APIs, as shown in Figure 1. Let us consider them from bottom to top:

1. Native platform programming layer

The native platform programming layer is provided by the hardware and native system software like nvcc, icc, gcc or MDK. It has been our goal in WP1 to raise the level of abstraction and portability beyond that of native APIs. We achieved that goal by designing and implementing new, higher-level programming frameworks (such as those given below) on top of the native layer(s), so that non-expert programmers should, in most cases, not need to write code in such platform-specific low-level APIs any more. At the same time, such frameworks should not generally prevent expert programmers from occasionally and locally escaping to using such low-level APIs where they consider it beneficial for performance or energy efficiency and where a default implementation alternative always remains for guaranteeing application portability.

2. Low-level hardware abstraction layer

The low-level hardware abstraction layer provides portable basic functionality atop the native EXCESS platforms' programming environments. These include portable libraries,

techniques, and tools. Portable libraries wrap native implementations such as basic linear algebra functionality and basic synchronization mechanisms, and which could then be called from constructs in higher-level programming frameworks. Another example of such functionality is the *MeterPU* portable measurement abstraction library [11].

MeterPU

MeterPU [11] is a generic portable measurement abstraction library for C++. The API is simple and platform and metric independent; platform-specific measurement methods (using software or hardware infrastructure) are realized by plug-ins. Switching between metrics or measurement methods then only requires changing a single line of code. The current version includes platform-specific plug-ins supporting multicore CPU and multiple CUDA GPUs and several measurement techniques. In contrast to the *ATOM* Monitoring Framework developed in WP3, which is mainly intended for remote observation of long-running applications on servers and clusters, MeterPU is light-weight, has very low runtime overhead and is intended to be used for (node-)local measurements only, in particular for self-tuning applications and system software components. For instance, SkePU(1.2) has been made energy-tuneable with the help of MeterPU [11]. MeterPU has been released as open-source software (GPL v3) in July 2015, and is available at `http://www.ida.liu.se/labs/pelab/meterpu`.

```
using namespace MeterPU;

// Initialize a meter for GPU
// energy of default device id 0:
Meter< NVML_Energy<GPU_0> > meter;

meter.start();
... // part to be measured
meter.stop();
meter.calc();
... = meter.get_value();
```

3. Task programming API layer

The task programming API layer, implemented by task-based runtime systems, provides functions and data types to describe multi-variant tasks and their operands, and to submit them to the runtime system's scheduler. A typical representative of such a runtime system is StarPU [1], which has been adapted in WP3 for energy tuning. Moreover, also other runtime systems can be used for direct task-level programming, such as the *global composition runtime system* (*GRS*) prototype designed for synchronous task execution, which we developed in order to experiment with global optimization techniques for variant selection (see Sect. 6a).

In contrast, some other high-level programming frameworks developed in EXCESS WP1, such as *Drake* (see Sect. 6b), provide their own, light-weight runtime support. For example, the Drake runtime library is to be used specifically with the Drake-generated code for energy-efficient execution of parallel streaming tasks on multicore systems; it is not intended to be used as an external API for low-level task programming.

4. Library-level programming API layer

The library-level programming API layer provides portable multi-variant implementations of more high-level, energy-tunable constructs for computations and for data struc-

tures, such as *SkePU skeletons* and its *smart containers* [4] with the very recent version *SkePU 2* [6] that provides a modernized, type-safe and more flexible programming interface; the new VectorPU [12] smart container framework for flexible, fine-grained control of operand coherence with (GPU) accelerator execution; or energy-tunable *data structures* such as lock-free queues developed in EXCESS WP2.

SkePU 1.x

The SkePU (1.x) skeleton programming library [5] provides generic program building blocks, so-called *skeletons* such as map, reduce, scan, stencil, for high-level portable parallel programming. Aggregate operand data is wrapped in so-called *smart containers* [4] for transparent memory management and optimization of the data transfer volume between the memory units in a heterogeneous system at runtime. For each skeleton, SkePU provides multiple *backends* (implementation variants) for various execution units and programming models in a heterogeneous system, and supports automated selection tuning for performance or energy efficiency. SkePU is available as open-source software under GPL v3. For download and documentation see `http://www.ida.liu.se/labs/pelab/skepu`.

```
#include "skepu/vector.h"
#include "skepu/reduce.h"

BINARY_FUNC(plus_f, double, a, b,
    return a+b;
)

int main()
{
  // instantiate Reduce skeleton:
  skepu::Reduce<plus_f>
           globalSum(new plus_f);
  skepu::Vector<double> v1(...);
  ...
  double r = globalSum(v1);
  ...
}
```

While originally implemented for multicore and single and multi GPGPU-based systems (with OpenMP, CUDA and OpenCL backends), a new SkePU backend supporting Movidius Myriad2 has been developed recently [16], providing seamless portability of the same source code on multicore, GPGPU, and Myriad.

SkePU 2

The SkePU programming interface was defined in 2010 [5], i.e. before C++11 was well supported on CUDA systems. Being a C++ include library without any further tool support, its objective of full code portability across a wide variety of C/C++ based programming models including CUDA could, at that time, only be achieved with the help of C preprocessor macros to define user functions in a portable way and generating the necessary glue code towards the parallel and accelerator implementations (i.e., backends) of the skeletons. This worked, but came at the price of reduced flexibility, unnecessary complexity, and a risk of type errors that only could be caught at runtime.

SkePU 2 [6] is a partial rewrite of SkePU with a new, fully C++11 compliant programming interface (i.e., frontend). It uses a precompiler (LLVM clang) to allow for programmer-friendly, type-safe and flexible specification of user functions. SkePU 2 preserves the architecture and mature runtime system of SkePU 1.2, such as highly optimized skeleton algorithms for each supported backend target, multi-GPU support, smart containers, tuning etc. The increased programmer friendliness and type safety is demonstrated, and the increased flexibility even leads to performance improvements in certain cases. For download and documentation see `http://www.ida.liu.se/labs/pelab/skepu`.

5. Local component-level programming interface

The **local component-level programming interface** generalizes the library and task-level programming interfaces by allowing the programmer to define (possibly, multi-variant) components for arbitrary functionality with arbitrary implementation variants for supported programming models, as long as they adhere to a simple *component model* that uses smart containers for aggregate data operands like vectors and matrices and that states the direction of operand data flow (in, out or inout).

Representatives are the new *Call* skeleton in *SkePU 2* and the *VectorPU annotation framework*. Also the *ComPU* framework can be mentioned here, an XML-based annotation framework for multi-variant C++ components based on the PEPPHER Composition Tool [2] that has been adapted for constrained composition with XPDL. Moreover, our EXCESS research contribution of an improved *adaptive sampling* technique for effective modeling of relative performance and energy for selection tuning [10] has been implemented and evaluated atop ComPU.

ComPU
ComPU is an XML-based annotation framework for multi-variant C++ components for GPU-based systems. It is technically based on the *PEPPHER Composition Tool* [2] that we developed in the previous FP7 project PEPPHER 2010–2012. ComPU allows to expose and annotate multi-variant components, i.e., functions with multiple implementation variants for the various programming models (sequential C++, OpenMP, CUDA, OpenCL) coexisting on a GPU-based system. Annotations of components and of their variants are done externally in XML descriptor files. From this metadata, glue code is generated for calls to multivariant components, which contains context-dependent variant selection logic that attempts to optimize for expected execution time. The code can be provided on request. Examples and documentation can be found at `http://www.ida.liu.se/labs/pelab/ctool`.

VectorPU
VectorPU is a library based framework for high-level programming of heterogeneous systems in C++ providing a shared data abstraction for array operands by leveraging generic smart containers and a domain-specific embedded language for data access annotations. By declaring specific operand access modes to aggregate data containers (e.g., vectors), the requirements for memory coherence can be programmed very flexibly such that the amount of really required invalidations and update data transfers in the coherence protocol can be kept to a minimum. We could show that VectorPU can achieve speedups over Nvidia's *Unified Memory* of 1.40x to 13.29x, and achieves almost the efficiency of manually programmed data movements while maintaining a unified memory abstraction.

6. Global component-level programming interface

The global component-level programming interface allows to extend local (multi-variant) component models by mechanisms and techniques to coordinate optimization (such as variant selection, voltage and frequency scaling, core allocation for parallelizable tasks, core shutdown, settings for tunable parameters, etc.) at a more global scope. At this level,

the high-level programmer (or, in future, a tool such as a compiler frontend) needs to specify *groups* of component calls (i.e., tasks) to be considered together in the optimization to take inter-component dependences properly into account. Here, we provide prototype global optimization frameworks for two very different styles of programming:

(a) **Global selection for component calls**, represented by the global composition runtime system (GRS) with its semi-static optimizer. Groups of calls to multi-variant components to be considered together for selection optimization are connected by control flow and, most often, also by operand data flow, and form a contiguous program region of interest, which is then analyzed at runtime to calculate a dispatch table with globally best selections for all calls in the region. Optimization attempts to minimize overall execution time or energy usage for the whole region.

GRS Global Composition Runtime System
GRS is a C-based runtime system for prototyping techniques for the global optimization of variant selection in multi-variant functions (components). The global selection optimizer in GRS, based on the formalization and algorithmic solution in [7], performs at runtime a semi-static global optimization of variant selections for groups of calls to multi-variant functions (components), in particular, for program regions consisting of loops and sequences containing calls to multi-variant components. GRS has been used with GPU-based systems and with an ARM big.LITTLE based platform. The GRS source code is available upon request.

(b) **Global optimization of streaming task collections**, represented by the *Drake* framework. Drake is applicable to programs consisting of (possibly, complex) task pipelines also known as *Kahn Process Networks*, where a certain throughput (e.g., frames per second) is to be maintained while latency (pipeline depth) is only of secondary interest. Optimization attempts to minimize overall energy usage across all tasks while maintaining the throughput constraint. We have also developed several global optimization algorithms, most of them based on the *Crown Scheduling* technique [15,14], which can be used as a plug-in to Drake.

Drake
Drake [13] is a framework for the specification, global static optimization and deployment of collections of streaming tasks for energy-efficient execution on multi-/many-core systems. The programming model can be considered as a special skeleton programming system for modeling *complex pipelines*, where each pipeline task can be internally parallel, i.e., use parallel algorithms and data structures on multiple cores to gain speedup. Drake takes a target architecture description and the input program in the form of a streaming task graph with the tasks' specific code and operand access information as input, runs an optimization algorithm (plug-in) and deploys the tasks with the calculated mapping, core allocations and DVFS settings to produce an executable code. Drake, together with the required support software packages such as *Pelib*, is publically available as open-source (GPL v3) at `http://www.ida.liu.se/labs/pelab/drake`. Drake comes with a backend for Intel IA multicore architectures.

7. Layering and Dependences

Generally, the upper levels of the software stack in Figure 1 can use functionality provided by the lower levels. Task programming API and library / local component programming API are at the same level because they may mutually use each other's functionality: tasks may contain library or skeleton calls, while skeleton implementations in turn may create multiple tasks from a single invocation and use the heterogeneous runtime system's dynamic scheduler and data flow driven synchronization in order to provide load balancing and hybrid execution.

8. Interfaces for access to support functionality

The described infrastructure for portable, high-level programming and optimization is complemented by auxiliary infrastructure. For instance, the EXCESS platform modeling language *XPDL* [9] allows, in principle, to make the entire infrastructure retargetable and to query platform and performance/energy models from applications and from optimization control both at deployment and run time. Moreover, the external *ATOM monitoring framework* [8] allows to observe and visualize the time and energy usage of a (possibly, remotely) running EXCESS application at a relatively coarse level of granularity. The ATOM visualizer is also used by MeterPU.

XPDL eXtensible Platform Description Language
XPDL [9] is an XML-based platform description language that allows to write scalable formal specifications of the main structure and the main performance and energy relevant properties of target systems' hardware and system software. Such specifications are to be used as machine-readable data sheets to provide platform information to retargetable toolchains for platform-specific performance/energy modeling, code generation and optimization, and to allow runtime systems to query platform information for automatic adaptation and application self-tuning. The XPDL design encourages reuse by features such as model crossreferencing, multiple inheritance, parameters and constraints in models. XPDL is used within several prototype frameworks within WP1. Further information on XPDL is available at `http://www.ida.liu.se/labs/pelab/xpdl`. Our prototype toolchain for XPDL is available under LGPL v3 license.

9. Conclusions and Outlook

We have contributed a number of techniques and software frameworks in the EXCESS project. These are of course also applicable outside EXCESS, and we expect that most of them will be used and/or further developed in future projects. We encourage other researchers to try them out and send us feedback, and we welcome cooperations on future development.

Acknowledgments

This work has been funded 2013–2016 by EU FP7 project EXCESS under grant #611183, and co-funded by *SeRC* (www.e-science.se) and the CUGS graduate school.

The authors thank all who contributed to the research, development, software and documentation of the systems described, including Usman Dastgeer, Johan Enmyren, Oskar Sjöström, Mudassar Majeed, Claudio Parisi, Rosandra Cuello, Sebastian Thorarensen, Lukas Gillsjö, Johan Janzén, and Ming-Jie Yang.

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*, 23:187–198, February 2011.

[2] Usman Dastgeer and Christoph Kessler. The PEPPHER composition tool: Performance-aware composition for GPU-based systems. *Computing*, 96(12):1195–1211, 2014. (online: Nov. 2013).

[3] Usman Dastgeer and Christoph Kessler. Performance-aware composition framework for GPU-based systems. *The Journal of Supercomputing*, 71(12):4646–4662, December 2015. (online: Jan. 2014).

[4] Usman Dastgeer and Christoph Kessler. Smart containers and skeleton programming for GPU-based systems. *International Journal of Parallel Programming*, June 2016. doi: 10.1007/s10766-015-0357-6.

[5] Johan Enmyren and Christoph W. Kessler. SkePU: A multi-backend skeleton programming library for multi-GPU systems. In *Proc. 4th Int. Workshop on High-Level Parallel Programming and Applications (HLPP-2010), Baltimore, Maryland, USA*, pages 5–14. ACM, September 2010. doi: 10.1145/1863482.1863487.

[6] August Ernstsson, Lu Li, and Christoph Kessler. SkePU 2: Flexible and type-safe skeleton programming for heterogeneous parallel systems. *Accepted for International Journal of Parallel Programming*, Special issue for selected papers from HLPP-2016, to appear (2017).

[7] Erik Hansson and Christoph Kessler. Optimized variant-selection code generation for loops on heterogeneous multicore systems. In Gerhard R. Joubert, Hugh Leather, Mark Parsons, Frans Peters, and Mark Sawyer, editors, *Advances in Parallel Computing, Volume 27: Parallel Computing: On the Road to Exascale. Proc. of ParCo-2015 conference, Edinburgh, UK, Sep. 2015.*, pages 103–112. IOS Press, April 2016.

[8] Dennis Hoppe *et al.* ATOM Monitoring Framework. http://mf.excess-project.eu/about, 2016.

[9] Christoph Kessler, Lu Li, Aras Atalar, and Alin Dobre. XPDL: Extensible Platform Description Language to Support Energy Modeling and Optimization. In *2015 44th International Conference on Parallel Processing Workshops (ICPPW)*, pages 51–60. IEEE, Sept 2015.

[10] Lu Li, Usman Dastgeer, and Christoph Kessler. Pruning Strategies in Adaptive Off-Line Tuning for Optimized Composition of Components on Heterogeneous Systems. *Parallel Computing*, 51:37–45, January 2016. doi: 10.1016/j.parco.2015.09.003.

[11] Lu Li and Christoph Kessler. MeterPU: A Generic Measurement Abstraction API Enabling Energy-tuned Skeleton Backend Selection. *Journal of Supercomputing*, pages 1–16, 2016. doi: 10.1007/s11227-016-1792-x.

[12] Lu Li and Christoph Kessler. VectorPU: A generic and efficient data-container and component model for transparent data transfer on GPU-based heterogeneous systems. In *Proc. PARMA-DITAM'17 workshop, Stockholm*. ACM, January 2017.

[13] Nicolas Melot, Johan Janzen, and Christoph Kessler. Mimer and Schedeval: Comparison tools for static schedulers and streaming applications on concrete manycore architectures. In *Proc. 8th Int. Workshop on Parallel Programming Models and Systems Software for High-End Computing (P2S2) at ICPP'15*. IEEE, September 2015.

[14] Nicolas Melot, Christoph Kessler, and Jörg Keller. Improving energy-efficiency of static schedules by core consolidation and switching off unused cores. In Gerhard R. Joubert, Hugh Leather, Mark Parsons, Frans Peters, and Mark Sawyer, editors, *Advances in Parallel Computing, Volume 27: Parallel Computing: On the Road to Exascale. Proc. of ParCo-2015 conference, Edinburgh, UK, Sep. 2015.*, pages 285–294. IOS Press, April 2016.

[15] Nicolas Melot, Christoph Kessler, Jörg Keller, and Patrick Eitschberger. Fast crown scheduling heuristics for energy-efficient mapping and scaling of moldable streaming tasks on many-core systems. *ACM Transactions on Architecture and Code Optimization (TACO)*, 11(4), January 2015. Article No. 62.

[16] Sebastian Thorarensen, Rosandra Cuello, Christoph Kessler, Lu Li, and Brendan Barry. Efficient execution of SkePU skeleton programs on the low-power multicore processor Myriad2. In *Proc. Euromicro PDP-2016 Int. Conf. on Parallel, Distributed, and Network-Based Processing, Heraklion, Greece*, pages 398–402. IEEE, February 2016.