Improving Programmability of Heterogeneous Many-Core Systems via Explicit Platform Descriptions

Martin Sandrieser
Faculty of Computer Science
University of Vienna, Austria
ms@par.univie.ac.at

Siegfried Benkner
Faculty of Computer Science
University of Vienna, Austria
sigi@par.univie.ac.at

Sabri Pllana
Faculty of Computer Science
University of Vienna, Austria
pllana@par.univie.ac.at

ABSTRACT

In this paper we present ongoing work towards a programming framework for heterogeneous hardware- and software environments. Our framework aims at improving programmability and portability for heterogeneous many-core systems via a Platform Description Language (PDL) for expressing architectural patterns and platform information. We developed a prototypical code generator that takes as input an annotated serial task-based program and outputs, parametrized via PDL descriptors, code for a specific target heterogeneous computing system. By varying the target PDL descriptor, code for different target configurations can be generated without the need to modify the input program. We utilize a simple task-based programming model for demonstration of our approach and present preliminary results indicating its applicability on a state-of-the-art heterogeneous system.

Categories and Subject Descriptors

D.2.6 [**Software Engineering**]: Programming Environments

General Terms

Design

Keywords

Many-core, programming framework, platform description

1. INTRODUCTION

Currently a trend towards heterogeneous platforms can be observed. These developments are motivated by an exceptional computational performance potential in combination with significant gains in energy efficiency of modern heterogeneous computing systems. The use of graphics processing units (GPUs), field programmable gate-arrays (FPGAs) or heterogeneous chip-multiprocessors for general-purpose computations has become prevalent in many application domains [8,9,12]. Heterogeneity of hardware resources also has led to a diverse landscape of different programming models, runtime systems and supportive tools for application development. This raises programmability challenges for software developers and makes efficient application development for heterogeneous many-core architectures often error-prone and tedious. In addition to programmability challenges, portability of applications tailored for a specific heterogeneous environment remains challenging.

Current approaches to unify software development for heterogeneous computing often employ logical platform models (e.g., OpenCL [11] host-device model) that may not be suitable for all classes of heterogeneous systems to the same extent. Due to the diversity of hardware designs and configuration options, achieving efficient program execution on different heterogeneous systems is still challenging for many applications [10, 15].

In this paper, we present our ongoing work on a programming framework for heterogeneous many-core systems that utilizes a platform description language (PDL) to transform programs for different configurations of heterogeneous hardware and software environments. We identify *heterogeneous platforms* as complex hierarchical systems with diverse *hardware and software* properties. Our preliminary programming concept is based on serial input programs that offload computational tasks to heterogeneous processing units. We introduce simple source code annotations for a task-based programming model that can relate computational tasks to hierarchical platform descriptions.

Our approach assumes a separation between *mainstream* and *expert* programmers, where *mainstream* programmers develop applications at a higher level of abstraction relying on software artifacts developed by *expert* programmers that have comprehensive platform-specific knowledge. Our preliminary programming concept supports the definition of high-level programs with limited platform-specific knowledge and optimization of tasks considering platform details. Therefore, our programming framework supports multiple task implementation variants of same functionality stored in a task repository together with meta-data capturing information about the expected heterogeneous execution environment. For different target platforms and configurations different meta-data formulated in our PDL can be provided. Supported by these external platform descriptions our prototypical source-to-source translator transforms high-level serial input programs to utilize optimized task implementation variants for a specific environment without the need for manual modification of the input program. We see this approach as a step towards improving programmability of

IWMSE'11, May 21, 2011, Waikiki, Honolulu, HI, USA

heterogeneous many-core systems for mainstream application developers. The machine-processable implementation of our PDL facilitates the selection and mapping of task implementations for automatic tools such as our prototype code generator.

Experimental evaluation of our approach was performed for different scientific kernels on a heterogeneous CPU/GPU system. We show that speedup over baseline implementations can be achieved by varying target PDL descriptors for different configurations of the execution environment without manual modification of the input program.

This paper is structured as follows. Section 2 discusses context and related work. Section 3 outlines the used programming model and Section 4 gives an overview of our PDL. Our tool infrastructure and a usage scenario are described in Section 5 and preliminary experimental results are presented in Section 6. Finally, we present conclusions and an outlook to future work in Section 7.

2. BACKGROUND AND RELATED WORK

Heterogeneous many-core systems have become ubiquitous and hence many related projects for application development exist. The OpenCL [11] specification offers a unified programming environment for heterogeneous many-core systems. It aims at providing portability of OpenCL programs across a wide variety of different heterogeneous hardware resources via definition of unified platform-, execution- and memory models. The Nvidia Cuda [13] programming framework, evolving from general-purpose computation on GPUs, uses similar platform abstractions to support application development. Both projects utilize programming models based on offloading of computational kernels from a general-purpose (*host*) processing unit to other *devices* often used as accelerators. It remains unclear if one single unified platform abstraction based on *host-device* relationships can provide an efficient abstraction for the large diversity of existing heterogeneous hardware and software configurations. Our approach considers a diversity of existing platform models and hardware configurations based on adaptable descriptions of possible execution environments. To some extent, our work also shows similarities to existing programming concepts that utilize explicit platform abstractions based on memory hierarchies. Sequoia [6] defines a programming model for heterogeneous architectures based on a tree abstraction of distinct memory modules similar to the Parallel Memory Hierarchy [1] model. Portability between different target systems is achieved via explicit descriptions of memory configurations and mapping of computational tasks to such trees of memories. Akin to Sequoia, Hierarchical Place Trees (HPT) [18] follow an approach based on abstraction of memory hierarchies. HPT offers additional extensions for different data transfer mechanisms as well as dynamic task scheduling. Both projects define programming models for heterogeneous architectures that are parameterized with machine specific descriptions of memory regions. We also aim at parametrization of applications via platform descriptions but follow a different modeling approach. Our hierarchical platform abstraction is based on *control relationships* between processing units. Control-relationships are defined as the *possibility for offloading computational tasks from one processing unit to another*. We believe, that this abstraction is suitable for a wide variety of different programming models and can also reflect the software diversity inherent to modern heterogeneous platforms. Our approach is driven by the observation that for efficient application development, often different platform and vendor specific programming models must be considered. One goal of our approach is to support such diversity.

High-level offloading of computational tasks to specialized processing units is a widely used concept for programming heterogeneous many-core architectures. HMPP [5] is a hybrid programming environment to enable application portability via different task implementations for specific hardware accelerators. It employs a similar programming model to StarSS [7] utilizing source code annotations to indicate task implementations and usage. Offload [4] is a programming model for C++ offering specialized program scopes for function execution on heterogeneous accelerator cores.

Given the commodity of outlining tasks for offloading via additional source code directives, we also adopt a similar programming model. In comparison to existing approaches that relate outlined tasks to often implicitly defined platform models (e.g., Nvidia Cuda, Cell SDK, OpenCL) our annotations support explicit reference to generic platform descriptors formulated in a hierarchical and expandable PDL. This can facilitate the automatic selection and mapping of task implementation variants from task variant and platform description repositories.

Several other frameworks that optimize applications based on task implementation repositories exist. Focus often lies on optimization and selection of tasks from large quantities of available implementation variants. Elastic Computing [17] shows a similar separation of functionality and implementation details of compute tasks for heterogeneous many-core systems. Even though we do not explore fine-grained tuning of task implementations in this work, we are of the opinion that relating task implementations to generic platform descriptors could support such efforts.

We contribute to the EU-funded project PEPPHER [16] which focuses on programmability and performance portability for single-node heterogeneous many-core systems. PEPPHER adopts a component-based approach to programming heterogeneous many-core systems where components encapsulate multiple implementation variants for computational tasks. Components and implementation variants are augmented with meta-data to enable performance prediction on different processing units of a heterogeneous system, which is a pre-requisite for selecting the best implementation variants for specific target platforms by means of advanced compilation and runtime techniques. We believe that task meta-data in form of explicit platform descriptions can support PEPPHER and other related programming frameworks that employ similar methodologies.

3. PROGRAMMING MODEL

It has been shown previously that task-based programming models are a viable approach for programming heterogeneous systems [7]. Therefore we also adopted a task-based programming model and extended it to support generic platform descriptions. With our approach we do not claim to introduce a novel programming paradigm but try to facilitate the mapping of computational tasks to heterogeneous processing units via meta-data in the form of explicit platform descriptions.

Based on the observation that development of efficient task implementations in many cases requires detailed knowl-

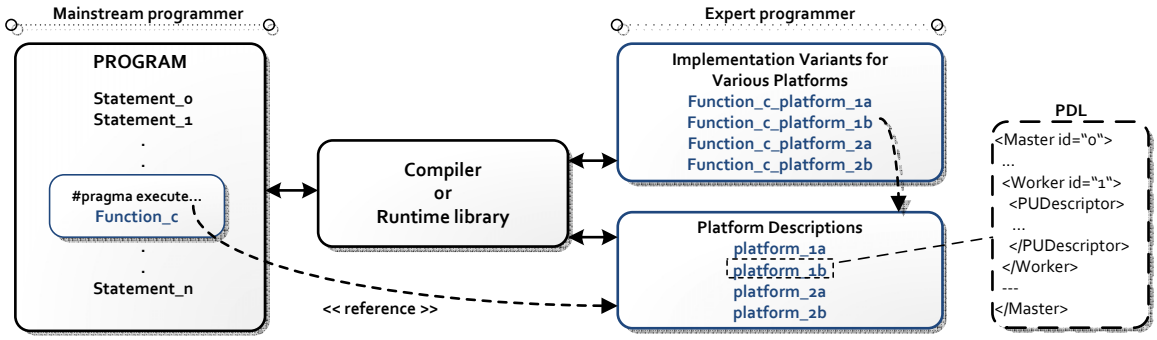

Figure 1: Methodology overview. Compute intensive tasks of serial input programs can be outlined via source code directives. Directives reference generic groups of processing units in platform descriptors. Task implementations for heterogeneous platforms written by expert programmers can be equipped with platform description meta-data. In conjunction with platform descriptions of concrete target environments, this facilitates automatic task variant selection for tools like compilers or runtime libraries.

edge about the expected execution environment, we assume a separation of programmers into *mainstream* and *expert* programmers. High-level serial input programs utilizing specialized implementation variants of computational tasks are often developed or maintained by *mainstream programmers* without detailed knowledge about low-level platform characteristics and implementation details. In contrast, efficient implementations of computational tasks are often written by *expert programmers* considering platform specific details of possible execution environments.

Our intention is to supply task implementations with meta-data describing possible execution environments. This allows *expert programmers* to expose important aspects of platform specific information to tools such as compilers or runtime systems. In addition to supporting automatic decisions by tools, explicit platform descriptions can be referenced from high-level programs via source code directives to support selection of task implementations. This methodology aims at closing the information gap between users and tools existing in many heterogeneous environments. With our approach information (e.g., control relationships, processing unit or memory properties) used for optimizing task implementations on specific platforms can be made explicit.

3.1 Tasks

Following a common programming technique, we introduce source code annotations in the form of C/C++ pragmas to indicate code-regions relevant for our framework. We assume that applications are at the highest level formulated as serial C/C++ programs utilizing self-contained computational tasks written for specific environments. Tasks are implemented as functions that must not access global variables. Tasks may only communicate via input and output parameters and must return *void*. A *task* can have multiple *task implementation variants* of the same functionality. Such *task implementations* can be optimized for different heterogeneous platforms but must implement the same *task interface*.

3.2 Annotation Requirements

To support generation of memory management and task invocation code, we identify the following minimal requirements for task annotations:

- Outlining of code-regions to define task implementation variants.
- Reference from task implementation variants to expected execution environments formulated in PDL.
- Access mode specifiers (read, write, read-write) for task parameters.
- Distribution specifiers for data-parallel task execution.
- Specifiers to relate task implementations to task interfaces.
- Annotation of task call sites with reference to groups of processing units in PDL descriptors.

We identify this as the minimal required functionality for demonstration of our framework using a simplified task based programming model. Annotations are subject to future enhancements. Possible increments could additionally include specifications of parameter meta-data (data-structures) or hints for runtime scheduling of computational tasks.

3.3 Task Implementation Annotation

```
#pragma cascabel task
                    : targetplatformlist
                    : taskidentifier
                    : taskname
                    : parameterlist
```

The *task* annotation indicates a function (C/C++ function definition) suitable for execution on a heterogeneous platform. *targetplatformlist* references one or more target platform environments the code is intended for. Platforms are expressed in PDL descriptors. This allows to specify platform characteristics and requirements expected by a particular task implementation. *taskidentifier* denotes a reference to the task interface name. All task implementations with same functionality and function signature must reference this name. *taskname* specifies an additional unique name for task implementations, in order to support grouping of associated code portions to be compiled for different types of processing units. *parameterlist* holds a list of function parameter-names with explicit *access* specifiers. Access modes include *read*, *write* and *readwrite*. Optional information about expected storage characteristics can be provided following the access specifier.

3.4 Execute Annotation

```
#pragma cascabel execute taskidentifier
                         : executiongroup
                         (distributionslist)
```

The *execute* annotation marks the call site of a task and must be placed before the respective function invocation. It holds a reference to the task interface name via *taskidentifier*. *executiongroup* is an important property which allows grouping of tasks for sub-sets of processing units in abstract platform descriptions. Corresponding group organization can be specified via PDL descriptors. Finally, *distribution* specifiers hold references to input and output parameter data-distributions (e.g., block, cyclic, block-cyclic) and optional sizes. They serve as additional information to a compiler and runtime system for possible data-decomposition of data-parallel tasks.

4. PLATFORM DESCRIPTION LANGUAGE

In this section we describe a hierarchical machine model and corresponding XML-based platform description language (PDL) for heterogeneous many-core architectures. We identify a hierarchical aggregation of system components as inherent to many of today's heterogeneous systems. Hence, we employ a generic machine model that enables to express a logical hierarchy between system components. The model supports the definition of *processing units* (PU), *memory regions* (MR) and *interconnect* (IC) capabilities.

The logical system hierarchy is captured by control relationships between processing units. We specify such a relationship as the *possibility for offloading computational tasks* from one PU to another.

4.1 Processing Units

We introduce three different classes of processing units. The *Master* PU refers to a feature rich, general-purpose processing unit that marks a possible starting point for execution of a program. *Master* entities can only be defined on the highest hierarchical level but may co-exist with other *Masters* within the same system. A *Worker* entity describes a specialized compute resource which is present at lower hierarchy levels (leaf nodes) and carries out a specific task. *Workers* must be controlled by *Master* or *Hybrid* PUs. To support the representation of highly hierarchical systems, *Hybrid* PU can act as *Master* and *Worker* PU at the same time. *Hybrid* PUs are present at inner nodes of the PU hierarchy and must always be controlled by other *Hybrid* or *Master* units.

4.2 Memory Region and Interconnect

The control relationship hierarchy between PUs captures possible task execution patterns appearing in heterogeneous systems. However, for efficient program execution on such hierarchical environments, hierarchical data-transfer and access mechanisms must be considered. To cover this important requirement, we introduce associated memory region and interconnect entities.

Memory regions (MRs) can be present for all processing units within the abstract machine. While the abstract model only supports the definition of directly addressable MRs, concrete instantiations could express qualitative properties. Such properties can include affinities, relative speeds to PUs,

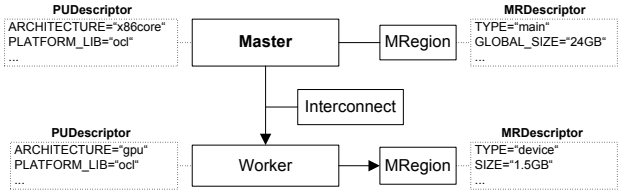

Figure 3: Visualization of a PDL excerpt for a GPGPU system.

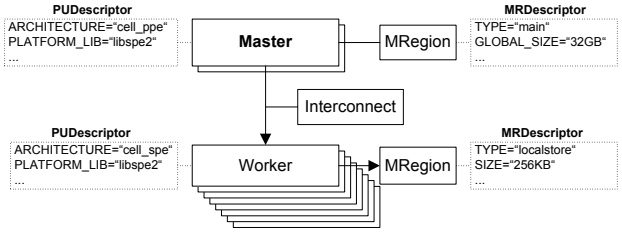

Figure 4: Visualization of a PDL excerpt for a 2-socket Cell B.E. system.

sizes and others. *Interconnect* entities describe communication facilities between processing elements. The main purpose of these entities is the definition of PU connectivity at the abstract machine level. Concrete instances can collect detailed information about communication schemes, underlying bus infrastructure or other communication performance descriptors.

4.3 XML Schema for PDL

We derive an XML Schema Definition (XSD) based on the hierarchical machine model. The PDL definitions are designed to capture multiple aspects of heterogeneous environments and are therefore not restricted solely to hardware properties. Our XSD specification defines three classes of PUs (*Master*, *Hybrid*, *Worker*), *Interconnect* and *MemoryRegion* entities stored in a multi-level XML document tree. Each of these nodes can be equipped with own *Descriptor* entities that hold sequences of *Property* nodes. Descriptor types are specified to cover a huge diversity of hardware *and* software properties. To support representation of future platform requirements, we consider standard techniques such as XML schema inheritance and entity polymorphism for base *Descriptor* and *Property* types. A generic key/value mechanism has been employed for base properties with specialized sub-schemas for specific platforms. This approach allows the definition of abstract architectural patterns (e.g., Master-Worker relationships) with later instantiation for different low-level platform specific characteristics.

A *LogicGroupAttribute* supports the definition of group identifiers for sub-sets of PUs in hierarchical platform descriptions. We use this attribute to relate execution groups from the previously introduced *task execute* pragma annotation to PUs in target environment descriptors.

4.4 Examples

Figure 3 sketches an example PDL description of a x86 processing unit with one attached GPU-*Worker*. The second example (Figure 4) shows a similar control relationship between *Master* and *Worker* PUs for a system comprising

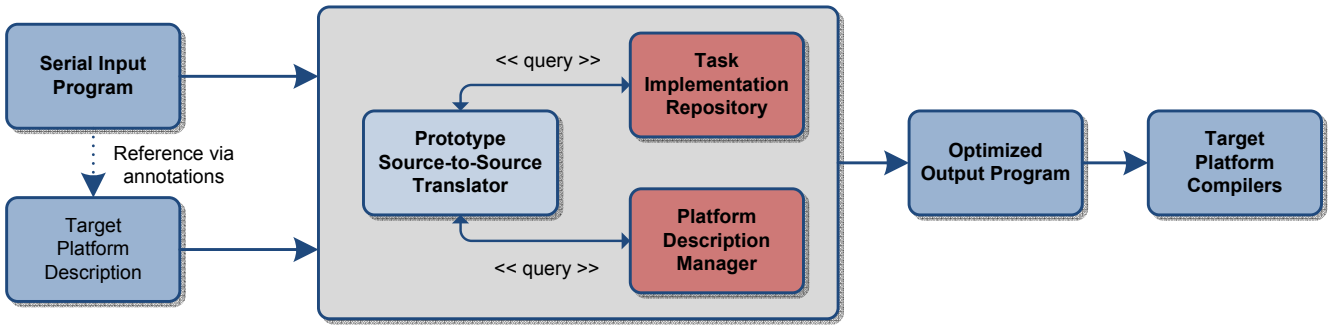

Figure 2: Prototype system infrastructure.

two Cell B.E. [8] multiprocessors. For processing unit and memory region elements, *Descriptors* store concrete platform specific properties. Such properties can be inserted for differnet platforms individually. The following listings show different instantiations of additional platform dependent properties. Listing 1 depicts information for the GPU-Worker acquired via Nvidia OpenCL [11] runtime. Listing 2 shows an example for one PPU *Master* element observable in Figure 4. Platform specific descriptors can be utilized at multiple levels. The *fixed* attribute indicates whether modification of property values by other users or tools is allowed.

```
<PUDescriptor>
 <Property fixed="false"
  xsi:type="ocl:oclDevicePropertyType">
  <ocl:name>DEVICE_NAME</ocl:name>
  <ocl:value>GeForce GTX 480</ocl:value>
 </Property>
 <Property fixed="false"
  xsi:type="ocl:oclDevicePropertyType">
  <ocl:name>MAX_COMPUTE_UNITS</ocl:name>
  <ocl:value>15</ocl:value>
 </Property>
 <Property fixed="false"
  xsi:type="ocl:oclDevicePropertyType">
  <ocl:name>MAX_WORK_ITEM_DIMENSIONS</ocl:name>
  <ocl:value>3</ocl:value>
 </Property>
</PUDescriptor>
```

Listing 1: An excerpt of concrete platform properties defined for OpenCL on GPUs.

Example properties for one *Master* PU shown in Figure 4 have been manually pre-defined.

```
<PUDescriptor>
 <Property fixed="false"
  xsi:type="ppu:ppuPropertyType">
  <ppu:name>CORE_ARCH</ppu:name>
  <ppu:value>ppc64</ppu:value>
 </Property>
 <Property fixed="false"
  xsi:type="ppu:ppuPropertyType">
  <ppu:name>MAX_CLOCK</ppu:name>
  <ppu:value unit="GHz">3.2</ppu:value>
 </Property>
 <Property fixed="false"
  xsi:type="ppu:ppuPropertyType">
  <ppu:name>L1_CACHE</ppu:name>
  <ppu:value unit="KB">32</ppu:value>
 </Property>
</PUDescriptor>
```

Listing 2: An excerpt of concrete platform properties defined for Cell B.E. PPU.

5. CASCABEL INFRASTRUCTURE

To evaluate our approach we implemented a prototypical source-to-source compiler, a task implementation repository and a simple platform query API. In what follows, we describe these major components of our prototypical framework named *Cascabel*.

5.1 Source-to-source Translator

This part of our infrastructure is based on the ROSE [14] compiler framework. The serial input program is parsed and the SAGE III [14] intermediate representation constructed. After processing of code annotations that indicate computational tasks, implementations are registered in the task implementation repository (Figure 2). If additional implementation variants for employed task interfaces exist in the repository, they are marked for possible program inclusion. In addition to the serial input program, a target platform description PDL file (Section 4) needs to be provided as input. By providing different input PDL descriptors, the translator can be parameterized for different heterogeneous target environments without changing the input program.

Each *targetplatform* listed in a task implementation annotation references a PDL description capturing specific platform characteristics expected by the implementation. Such *platform expectations* are queried via the *platform description manager* and must be present for each task implementation. By comparing entity properties as well as structural patterns in *target platform description* and *platform expectation* PDL files, our tool selects implementation variants appropriate for the target environment. Utilizing information gathered from *target platform descriptions* and *source annotations* about data distributions, target runtime libraries, interconnect and PU group organization the code-generator inserts platform specific code to manage data transfers and task executions. Our preliminary implementation utilizes the StarPU runtime system [3] to facilitate data management and task scheduling.

Depending on the target platform descriptor, one or more output source files are constructed. Final executables are produced by platform specific compilers (e.g., nvcc, xlc-spu). While automation seems feasible using the PDL, our preliminary framework requires the final compilation and linking plan to be provided manually.

5.2 Task Repository

For retrieval and storage of task implementation variants we utilize a simple task repository. It is based on the lightweight SQLite database engine and holds task variant source

code as well as associated *platform expectations* in our PDL. For each task interface, multiple task implementations can co-exist in the repository. Currently, selection of task implementations is based on comparison between *target platform descriptor* of input program and *platform expectation descriptors* of task implementations. In future, we plan to extend the database schema with support for historic execution data to further support task variant selection.

5.3 Platform Description Manager

This component consists of a simple query API for the XML based platform descriptions. It allows to retrieve entity properties via key/value lookup. Available keys are retrieved from xml-schema definitions as outlined in Section 4.3. Supplementary important structural queries have been implemented. Most importantly, this serves the lookup of *interconnect* entities between processing units and associated *memory-regions*. In combination with PU groups (*LogicGroupAttributes*) this allows inserting platform dependent library calls for task execution and data management.

5.4 Framework Usage

In this section we outline a possible usage scenario of our framework on basis of a concrete example.

1. **Identification of tasks.** To utilize the framework, the user (*mainstream programmer*) needs to identify computational tasks suitable for offloading to other processing units. In many cases this will be analogue to identification of application hot-spots. The following pseudo-code illustrates a serial program executing the task *spmv_base*.

```
int main(){
    ...
    spmv_base(A, nnz, nrows, col_idx, row_ptr,
        first, vec, vec_result);
    ...
```

2. **Code annotations.** After code regions denoting tasks have been identified, they are annotated via our previously introduced pragmas. For *spmv_base* the according function implementation is annotated as follows:

```
#pragma cascabel
    task:x86single:Ispmv_csr:spmv01:(
    A:read_csr_nzval,nnz:read_csr_nnz_A, ...)
```

   Matrix A is stored in CSR format in the original implementation. To support this representation we provide additional parameter information required by this format using specialized access-mode specifiers.

   Associated function call sites are marked with the following exemplary annotation:

```
#pragma cascabel execute Ispmv_csr:group01(
    A:csr_block:nnzA,vec::nrows,
    vec_result:block:nrows)
```

3. **Registering platform expectations.** For the *x86single* platform identifier referenced in the task annotation, a platform description capturing the expected execution environment must be available in the repository. This PDL file describes a sub-part of a generic platform with expected PU control-relationships and additional properties. For the *x86single* target platform a single *Master* PU with x86 ISA was specified. New platform expectation PDL files may be registered via the external task repository helper program *trmanager*.

```
trmanager --add-platformexpect -pname x86single
    x86single.pdl
```

4. **Registering implementation variants.** Additional task implementation variants for different platforms can also be provided via the *trmanager* helper program. Annotated task implementation source-files are parsed and stored in the repository.

```
trmanager --add-tasks spmv_other.c
```

5. **Target platform description.** To parameterize the code generator for a particular environment, a target platform description file must be provided. Via modification of *LogicExecutionGroups* in target PDL descriptors, the user can influence task implementation selection and mapping. In this work we consider manual creation of target platform descriptions for specific target environments. In future, automatic PDL descriptor creation will be investigated.

6. **Translator Invocation.** The annotated serial input program is supplied together with a target platform description to the prototype source-to-source translator. The generated output files contain platform specific library calls for task execution and data transfer according to specified data-distributions and target execution groups.

An example for the code generated by our source-to-source translator is shown in Listing 3 targeting the StarPU runtime system [3], which has been specified in the example PDL descriptors. StarPU requires specific buffer configuration and data management calls for task scheduling and execution. This platform specific code is generated automatically with information from PDL descriptors and source code annotations.

```
// Computation start
#pragma cascabel execute Ispmv_csr:group01(A:
    csr_block:nnzA,vec::nrows,vec_result:block:
    nrows)
// Create tasks and submit to run-time
int i;
for (i = 0; i < blocks; i++) {
    struct starpu_task *task = starpu_task_create
        ();
    task->cl = &cl_spmv_base;
    task->buffers[0].handle =
        starpu_data_get_sub_data(A_handle, 1, i);
    task->buffers[0].mode = STARPU_R;
    task->buffers[1].handle = vec_handle;
    task->buffers[1].mode = STARPU_R;
    task->buffers[2].handle =
        starpu_data_get_sub_data(vec_result_handle
        , 1, i);
    task->buffers[2].mode = STARPU_W;
    task->use_tag = 1;
    task->tag_id = (starpu_tag_t)(++
        starpu_globaltag);
    task->synchronous = 0;
    int ret = starpu_task_submit(task);
}
//Original call
/*spmv_base(A, nnz, nrows, col_idx, row_ptr, first
    , vec, vec_result);*/
// Computation end
```

Listing 3: Excerpt from example translation. The original function call is replaced by platform specific runtime (StarPU) calls. Target runtime parameters are gathered from PDL descriptors.

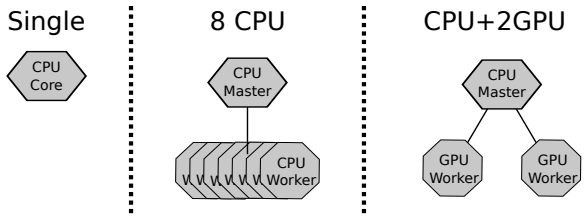

Figure 5: Conceptual view of different PDL input files used in the experiments.

Although our methodology requires additional steps for initial registration of platform descriptions and task implementations, we consider this separation of high-level input programs and platform specific task variants with generic platform meta-data a step towards improved programmability of heterogeneous systems. Expert programmers can equip highly specific task implementations with meta-data defining an expected execution environment. The machine processable explicit PDL allows tools to gain important additional information for utilization of such implementations.

6. PRELIMINARY RESULTS

In this section we report on initial experiments with our framework for two scientific application kernels (matrix multiplication and sparse matrix-vector multiplication) targeting different configurations of a CPU/GPU system with the StarPU runtime system.

Experiments were performed on a dual-socket 2.66GHz Intel Xeon X5550 (Quad-Core) system comprising two Nvidia GPUs (GTX480 and GTX285). The first serial input program performs a double precision matrix multiplication of two 8192x8192 matrices (DGEMM). It does so via calling a highly optimized Blas library (GotoBlas2 1.13). This call was annotated with our source code annotations. The source-to-source translator is parameterized with PDL platform descriptions to select suitable DGEMM implementations from a task implementation repository and construct the output program with platform specific runtime code (Listing 3).

The single threaded input program is translated to two different test programs (CPU and CPU+2GPUs) using different PDL descriptions. Input PDL descriptions used for this example are depicted in Figure 5.

The program generated from the "8 CPU" PDL description comprises the input task-implementation of DGEMM and is optimized for data-parallel execution via StarPU on 8 CPU-Cores. Data-parallel work distribution is derived from our previously defined source code annotations supporting distribution specifiers. Speedup of 3.16 is achieved over the base input program. The second platform description "CPU + 2GPU" produces a GPU-enabled program version which executes a CuBlas DGEMM implementation (Cuda Toolkit 3.2). A speedup of 6.20 was achieved with that parametrization. Both output programs were generated automatically using different PDL descriptions without modification of the serial input program.

The second example program executes a sparse matrix-vector multiplication kernel (SpMV). A sequential input test program is translated for execution with the StarPU runtime based on a hybrid PDL input file comprising all available cpu-cores and gpus in a single execution group. Based on

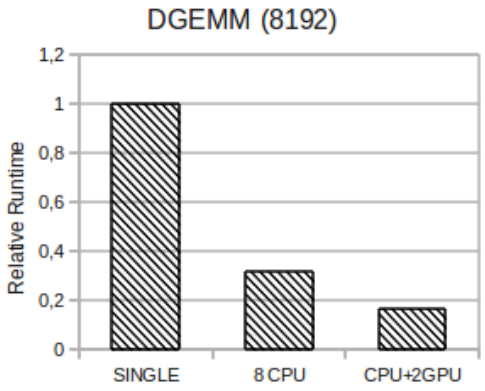

Figure 6: Program runtime after translation from single threaded input program (SINGLE) to multithreaded (8 CPU) and GPGPU (CPU + 2GPU) versions.

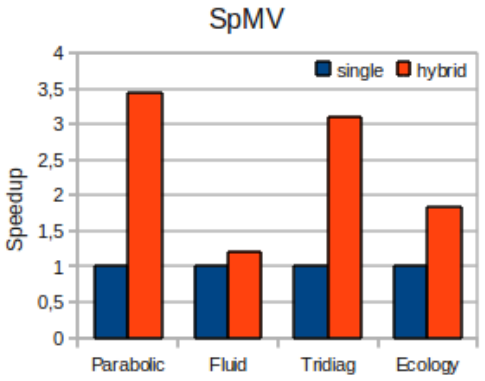

Figure 7: SpMV kernel speedup after translation from single threaded kernel (single) to multithreaded GPGPU-enabled version (hybrid).

this configuration, the runtime system is capable of automatically distributing workloads to the processing units [2]. We measure spmv kernel speedup for the following different workloads:

Name	Non-zeros
Parabolic	2.1M
Fluid	553K
Tridiag	53.3M
Ecology	2.9M

All matrices use compressed sparse row (csr) storage. The sequential SpMV implementation is translated for hybrid execution on a maximum of 8 cpu-cores and 2 gpus. We observe speedups (Figure 7) between 3.43 (Parabolic) and 1.21 (Fluid).

Our experiments indicate possible performance gains via selection of specialized task implementation variants with help of PDL descriptors at compile time. It should be noted that improving programmability due to automatic program composition may result in performance drawbacks compared to fully hand-crafted applications tailored for a particular environment by expert programmers. In future, we will investigate performance critical optimizations of task implementations and data management in our framework.

7. CONCLUSIONS AND FUTURE WORK

Heterogeneous many-core systems pose a great challenge to application developers and tool-chains. The complex mixture of different platform libraries and execution models inherent to these systems raises many challenges with respect to programmability and application portability. In this paper we have presented a prototypical framework that utilizes platform descriptions to transform annotated task-based programs for execution on heterogeneous target environments. We outlined our programming methodology, which is based on a separation between mainstream application programmers unaware of platform-specific details and expert programmers considering platform-specific knowledge for optimizing kernels. Our approach assumes that for a specific algorithmic task different implementation variants, each targeted for a specific processing unit of a heterogeneous architecture, are made available by expert programmers. By explicity formulating platform-specific information via our PDL, compilation frameworks and/or runtime systems can select the best implemenation variant of a task for a specific platform, without requiring mainstream users to adapt their applications.

Our work represents a fist step towards a programming framework for heterogeneous many-core systems that aims at improving programmability for mainstream programmers by hiding platform-specific implementation details, while allowing expert programmers to supply highly optimized implementations for specific execution units with explicit platform dependencies. Our prototype framework consist of a source-to-source translator, a task implementation repository, and a platform description query API. So far, our framework has been utilized together with the StarPU runtime system which has been specifically designed for single-node heterogeneous platforms supporting memory management and dynamic, data-aware task scheduling. As our framework currently supports only a single implementation variant for a specific platform expectation, providing support for multi implementation variants will be the focus of our future work. Also, we plan to investigate how our PDL can be used to further support the interoperability between different programming models available in heterogeneous systems for existing applications.

Acknowledgment

This work received funding from the European Union Seventh Framework Programme (FP7/2007-2013) under grant agreement n248481 (PEPPHER Project, www.peppher.eu).

8. REFERENCES

[1] B. Alpern, L. Carter, and J. Ferrante. Modeling parallel computers as memory hierarchies. In *Programming Models for Massively Parallel Computers, 1993. Proceedings*, pages 116–123, 1993.

[2] C. Augonnet, J. Clet-Ortega, S. Thibault, and R. Namyst. Data-Aware Task Scheduling on Multi-Accelerator based Platforms. In *The 16th International Conference on Parallel and Distributed Systems (ICPADS)*, Shanghai, China, Dec. 2010.

[3] C. Augonnet, S. Thibault, R., and P. A. Wacrenier. StarPU: a unified platform for task scheduling on heterogeneous multicore architectures. *Euro-Par 2009 Parallel Processing*, pages 863–874, 2009.

[4] P. Cooper, U. Dolinsky, A. Donaldson, A. Richards, C. Riley, and G. Russell. Offload – Automating code migration to heterogeneous multicore systems. *High Performance Embedded Architectures and Compilers*, pages 337–352, 2010.

[5] R. Dolbeau, S. Bihan, and F. Bodin. HMPP: a hybrid multi-core parallel programming environment. In *First Workshop on General Purpose Processing on Graphics Processing Units*, 2007.

[6] K. Fatahalian, D. R. Horn, T. J. Knight, L. Leem, M. Houston, J. Y. Park, M. Erez, M. Ren, A. Aiken, W. J. Dally, et al. Sequoia: Programming the memory hierarchy. In *Proceedings of the 2006 ACM/IEEE Conference on Supercomputing*, page 83, 2006.

[7] R. Ferrer, P. Bellens, V. Beltran, M. Gonzalez, X. Martorell, R. M. Badia, E. Ayguade, J. S. Yeom, S. Schneider, K. Koukos, et al. Parallel programming models for heterogeneous multicore architectures. *Micro, IEEE*, 30(5):42–53, 2010.

[8] M. Gschwind. The cell broadband engine: Exploiting multiple levels of parallelism in a chip multiprocessor. *International Journal of Parallel Programming*, 35(3):233–262, 2007.

[9] S. Hauck and A. DeHon. *Reconfigurable computing: the theory and practice of FPGA-based computation.* Morgan Kaufmann Pub, 2008.

[10] K. Komatsu, K. Sato, Y. Arai, K. Koyama, H. Takizawa, and H. Kobayashi. Evaluating performance and portability of OpenCL programs. *The Fifth International Workshop on Automatic Performance Tuning (iWAPT)*, 2010.

[11] A. Munshi. The OpenCL specification, version 1.1, Sept. 2010.

[12] J. Nickolls and W. J. Dally. The gpu computing era. *Micro, IEEE*, 30(2):56–69, 2010.

[13] Nvidia. NVIDIA CUDA C Programming Guide v3.2. http://developer.download.nvidia.com/compute/cuda/3_2_prod/toolkit/docs/CUDA_C_Programming_Guide.pdf, 2010.

[14] D. Quinlan, C. Liao, T. Panas, R. Matzke, M. Schordan, R. Vuduc, and Q. Yi. ROSE user manual: A tool for building Source-to-Source translators. draft user manual (version 0.9.5a, 2011). http://www.rosecompiler.org/ROSE_UserManual/ROSE-UserManual.pdf.

[15] S. Rul, H. Vandierendonck, J. D'Haene, and K. D. Bosschere. An experimental study on performance portability of OpenCL kernels. In *Symposium on Application Accelerators in High Performance Computing (SAAHPC '10)*, 2010.

[16] The PEPPHER Consortium. Performance portability and programmability for heterogeneous many-core architectures - PEPPHER. EU FP7 project 248481. http://www.peppher.eu/.

[17] J. R. Wernsing and G. Stitt. Elastic computing: a framework for transparent, portable, and adaptive multi-core heterogeneous computing. *ACM SIGPLAN Notices*, 45(4):115–124, 2010.

[18] Y. Yan, J. Zhao, Y. Guo, and V. Sarkar. Hierarchical place trees: A portable abstraction for task parallelism and data movement. *Languages and Compilers for Parallel Computing*, pages 172–187, 2010.