Using explicit platform descriptions to support programming of heterogeneous many-core systems

Martin Sandrieser *, Siegfried Benkner, Sabri Pllana

Faculty of Computer Science, University of Vienna, Austria

A R T I C L E   I N F O

Keywords:
Heterogeneous many-core architectures
Platform descriptors
Source-to-source transformation
Parallelization
GPU computing

A B S T R A C T

Heterogeneous many-core systems constitute a viable approach for coping with power constraints in modern computer architectures and can now be found across the whole computing landscape ranging from mobile devices, to desktop systems and servers, all the way to high-end supercomputers and large-scale data centers. While these systems promise to offer superior performance-power ratios, programming heterogeneous many-core architectures efficiently has been shown to be notoriously difficult. Programmers typically are forced to take into account a plethora of low-level architectural details and usually have to resort to a combination of different programming models within a single application. In this paper we propose a platform description language (PDL) that enables to capture key architectural patterns of commonly used heterogeneous computing systems. PDL architecture descriptions support both programmers and toolchains by providing platform-specific information in a well-defined and explicit manner. We have developed a prototype source-to-source compilation framework that utilizes PDL descriptors to transform sequential task-based programs with source code annotations into a form that is convenient for execution on heterogeneous many-core systems. Our framework relies on a component-based approach that accommodates for different implementation variants of tasks, customized for different parts of a heterogeneous platform, and utilizes an advanced runtime system for exploiting parallelism through dynamic task scheduling. We show various usage scenarios of our PDL and demonstrate the effectiveness of our framework for a commonly used scientific kernel and a financial application on different configurations of a state-of-the-art CPU/GPU system.

© 2011 Elsevier B.V. All rights reserved.

1. Introduction

Recent developments in computer architecture have shown that heterogeneous many-core systems are a viable way to overcome major hardware design challenges related to energy-consumption and thermal constraints while offering high computational peak performance. Prominent examples for this trend are the IBM Cell B.E., general-purpose GPU-computing or acceleration via reconfigurable hardware [1].

Heterogeneous multi-core systems comprise different processing units specialized for specific computational tasks, e.g., GPUs for data-parallel or stream computations, and thus may achieve superior performance compared to homogeneous multi-core architectures. Although heterogeneous processing units may provide exceptional performance for many workloads, programming of such systems is a challenging task. Software developers are forced to deal with a diversity of different software libraries, runtime systems and programming models. As Chamberlain et al. [2] state, a complex mixture of different...
languages, compilers and runtime systems is inherent to many heterogeneous platforms. Even though first standardization approaches exist (e.g., OpenCL [3]), the diversity of underlying hardware designs and multitude of configuration options makes it very difficult for users, let alone automatic tools, to optimize applications and effectively distribute workloads among heterogeneous processing units [4,5].

For efficient application development and compilation often detailed knowledge about the hardware configuration is required that usually exceeds the information exposed by programming models or platform layer. Commonly existing approaches employ an implicit and abstracted view on available processing resources (such as the OpenCL host-device model). Such implicit models, often based on control relationships between processing units, may not expose sufficient information that might be required to achieve efficient program execution on different classes of heterogeneous many-core systems.

In this paper, we propose explicit platform descriptors of heterogeneous many-core architectures and show how these can support high-level programming environments. To address the hierarchical aggregation of system components and the resulting need for efficient vertical data-management in modern heterogeneous systems, we developed a hierarchical machine model together with an XML-based platform description language (PDL) capable of expressing characteristics of a large class of current and future heterogeneous many-core systems. The PDL is intended to be used for making platform-specific information explicit to (1) expert programmers, and (2) to tools such as autotuners, compilers or runtime systems.

We demonstrate the usefulness of our approach with a source-to-source transformation framework in the context of a task-based programming model, where for major computational tasks different implementation variants tailored for different types of heterogeneous execution units are provided within the framework by expert programmers. The platform requirements of task implementation variants are made explicit by means of corresponding PDL descriptions. Non-expert or mainstream programmers use annotations to mark in the sequential source code functions as tasks and may specify references to PDL descriptors for supporting the selection of suitable implementation variants. Thus information provided by users can support compilers and runtime systems to optimize the mapping of computational tasks to processing units. Moreover, we provide data partitioning annotations for array parameters of tasks which are used by the transformation system to split up tasks operating on large arrays into many smaller tasks operating on corresponding sub-arrays.

We have developed a prototypical source-to-source compiler that supports our task annotations for C/C++. Our compiler takes as input an annotated serial task-based program and outputs, parametrized via PDL descriptors, code for a specific heterogeneous many-core target computing system. By varying the target PDL descriptor our compiler can generate code for different target architecture configurations without the need to modify the source program. Experiments with a commonly used scientific kernel and a financial application on different configurations of a state-of-the-art CPU/GPU system demonstrate the potential of our approach. In these experiments, our framework targets a flexible heterogeneous runtime system [6], which is capable of scheduling computational tasks to the different execution units of a heterogeneous many-core system. The runtime system takes care of data dependencies between tasks, and in case no data dependencies between a set of tasks exists, these tasks may be executed on different execution units in parallel. As a consequence, with this approach multiple levels of parallelism may be exploited; task parallelism, if two or more tasks are concurrently executed on different execution units, and data parallelism, if a task internally relies on a data parallel execution model, for example, CUDA or OpenCL.

This paper is structured as follows. Section 2 discusses the context and related work. Section 3 gives an overview of our approach and describes the platform description language. Section 4 introduces our programming framework for heterogeneous systems. Experimental results are presented in Section 5. Section 6 summarizes our approach and gives an outlook to future developments.

2. Related work

Programming heterogeneous systems is targeted by the OpenCL [3] and Nvidia Cuda [7] projects. Both programming models use a similar hierarchical platform abstraction involving a fixed control-relationship (host-device) to support portability between heterogeneous hardware elements. Our PDL can be seen as a generic approach to represent such platform patterns. It is not limited to a specific hierarchy of control-relationships that may only be well suited for specialized classes of heterogeneous hardware (e.g., GPUs).

Using high-level architectural abstractions to simplify programming of heterogeneous many-core systems has been applied by several other approaches.

Sequoia [8] introduces a programming model for heterogeneous architectures based on a tree abstraction of distinct memory modules similar to the Parallel Memory Hierarchy [9] 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) [10] 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.

Several other projects propose the provision of implementation variants of functions tailored to different architectures for programming heterogeneous many-core architectures. The Merge framework [11] offers a library-based programming model for map-reduce applications. Multiple function variants for different architectures, indicated by means of corresponding predicates, may be provided within the library, and grouped into hierarchical collections. The Merge compiler and runtime infrastructure supports dynamic selection of the best available function implementations for a given input and machine configuration. Elastic computing [12] focuses on the provision of function variants, so called elastic functions, among which the best combination is composed mostly by static means, guided by performance profiles and machine configuration.

Elastic computing [12] focuses on the provision of function variants, so called elastic functions, among which the best combination is composed mostly by static means, guided by performance profiles and models. There have been several proposals for extending C or Fortran in order to support programming of heterogeneous systems comprised of CPUs and GPUs. The OmpSS proposal [13] introduces a target directive in the context of OpenMP tasks which enables the programmer to specify that a task should be executed on a specific accelerator device (e.g., GPU). Target devices may be specified by means of keywords, e.g. CUDA or OpenCL, and corresponding implementations can be provided by the user. Moreover, extensions for the OpenMP task directive have been proposed for allowing the specification of input and output data of a task, to support the compiler in generating appropriate data transfer code. Commercial products addressing programming of heterogeneous many-core systems include CAPS HMPP and PGI Accelerate. HMPP [14] from CAPS Enterprise supports offloading of code regions to GPUs based on high-level annotations. The HMPP compiler is capable of generating corresponding CUDA or OpenCL kernels. PGI Accelerate [15] from The Portland Group features compiler directives for specifying regions of code in Fortran and C programs that can be offloaded from a host CPU to an attached GPU, delegating to the compiler the generation of corresponding GPU code.

While all these approaches employ platform abstractions of “accelerator” processing units, none of them makes the logical platform model available in a machine-processable form as intended by our PDL. Therefore our PDL could be seen as a supportive tool for such frameworks. Despite making inherent control relationships between processing units explicitly available, the PDL is also capable of capturing platform specific software and hardware parameters (e.g., memory-sizes, PU-frequencies, available runtime libraries). Hwloc [16] is a project that aims to make hardware properties and additional locality information available to tools and users. We follow a more comprehensive approach not limited to hardware information. Nevertheless, APIs like hwloc used for exploration of hardware parameters can facilitate the automatic generation of PDL descriptors. Hence, we see such efforts as important complements for our PDL.

The EU-funded project PEPPHER [17] addresses programmability and performance-portability for heterogeneous many-core systems. PEPPHER employs a component-based programming model that supports multiple implementation variants for a particular task and develops techniques required for selecting the best implementation variant for a specific target platform. We are of the opinion that PEPPHER and other related frameworks will benefit from explicit platform descriptors. For instance, the process of selecting and mapping of task implementations to processing units can be supported with information expressed in our PDL.

3. Platform description language

In this section we give an overview of our approach, describe the hierarchical machine model and corresponding XML-based platform description language (PDL).

3.1. Overview and usage scenarios

We have developed a PDL that can express hardware and software properties as well as control-relationships inherent to heterogeneous platforms. We define a logical control-relationship as the possibility for delegation of computational tasks from one processing-unit to another. Our PDL is a novel approach to capture platform patterns, which by now have often been only implicitly exposed by specific programming models. With our descriptors, platform specific information can be automatically processed and hence application development for heterogeneous platforms can be simplified. This feature is especially valuable for task-based programs that offload specific workloads to heterogeneous processing units. For such applications, the pre-selection and mapping of suitable task implementations for specific target environments can be supported, as we will show in this work.

As shown in Fig. 1, explicit platform descriptions can provide existing tools (compilers, auto-tuners, runtime systems) with additional information about target heterogeneous platforms in a generic manner. Despite being processed by tools, they allow programmers to reference abstract architectural patterns by formulating explicit control-relationships between processing-units. Implementations of the PDL enable manual as well as automatic generation of PDL descriptors. In addition to supporting automatic task-variant pre-selection and mapping to concrete systems, we envision the following additional potential usage scenarios of our PDL:
Support auto-tuners, schedulers or other tools for program optimization and performance prediction. More precisely, performance relevant observations can now be related not only to concrete hardware parameters but also to abstract architectural patterns expressed in the PDL. Moreover, expert-programmers can denote specific optimizations for abstract classes of heterogeneous systems.

Enables the expression of architectural constraints and requirements for highly optimized code. For example, highly optimized and platform specific code written by expert programmers can now be equipped with additional platform requirements expressed in our PDL. We see this as a step towards support of performance-portability guarantees for well-defined classes of target environments.

Multiple logical platform patterns can co-exist for a single target system. Our PDL supports the representation of different programming model specific control-relationships for the same physical hardware.

Provides a name-space for reference to architectural properties and platform information. Currently users have to face a diversity of different APIs to query platform information. Our PDL could serve to complement other approaches like hwloc [16] or OpenCL platform query functions [3].

### 3.2. Hierarchical machine model

To capture different characteristics of heterogeneous many-core architectures, we employ a generic machine model to describe processing units (PU), memory regions (MR) and interconnect (IC) capabilities. Our observation is that the hierarchical aggregation of system components is a main property of today’s heterogeneous systems. Hence, the generic machine model comprises different classes of PUs that allow to express a logical hierarchy between components.

We distinguish between 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 leaf nodes of the PU hierarchy and carries out a specific task. Workers must be controlled by Master or Hybrid PUs. To support the representation of 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 Hybrid or Master units. Hybrid PUs serve the construction of deeper leveled control relationships. Although such a hierarchy could also be modeled with specialized lower-level Master resources, we introduce the Hybrid entity to express the control requirement explicitly. Connections to multiple Worker and Hybrid entities are supported. Fig. 2 depicts the described PU classes in an example hierarchical organization.

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 inter-
connect entities.

Memory regions (MR)s 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 prop-
erties can include PU-affinities, latencies, memory sizes or other descriptors which are highly system dependent.

Interconnect entities describe communication facilities between processing elements. The main purpose of this entity is
the definition of PU connectivity on the abstract machine level. Concrete instances collect detailed information about com-
munication schemes, underlying bus infrastructure or other communication performance descriptors.

3.3. Specification of PDL

To utilize the generic and hierarchical machine model presented in Section 3.2, we define an XML-based platform descrip-
tion language. In the following subsection we will describe the main components of the defined PDL. Today’s heteroge-
neous platforms can be seen as a complex mixture of different hardware entities and software environments [2]. Therefore, we fol-
low a holistic approach and do not restrict description facilities to pure hardware properties. Our platform description lan-
guage is generic and extensible and hence capable of expressing a large diversity of hardware and software properties.
Depending on where a PDL description is used (i.e. expert-user, compiler, runtime), different levels of abstraction may be
provided. Based on the hierarchical machine model, generic entity descriptors enable the modeling of architectural patterns.
On this basis, instantiations via specialized platform descriptors support concrete representation of heterogeneous plat-
forms, for example a Master–Worker relationship between CPU and GPU. We emphasize that we aim at capturing platform
properties that can be queried and measured by software or users.

Starting from the hierarchical machine model, we derive an XML Schema Definition (XSD) capable of being extended with
entity descriptors for current and future heterogeneous architectures. Although concrete hardware description instances can
differ in type and accuracy, they all adhere to the abstract machine model. This approach supports the transformation and
mapping of system components onto different heterogeneous hardware environments. The initial specification of the PDL
comprises the following XML entities.

- Master, Hybrid, Worker: PUDescriptor, Interconnect, MemoryRegion, LogicGroupAttribute
- Interconnect: ICDescriptor
- MemoryRegion: MRDescriptor
- Descriptor: Property
- Property: name, value

One challenge for definition of the platform description is the ability to adapt to future, yet unknown hardware architec-
tures and their requirements. Although the high level representation provides a generic way to model large sets of architec-
tures, machine specific details need to be emphasized on lower levels of abstraction. Therefore we introduce extensible
Descriptor and Property types. A generic key/value mechanism has been employed for base properties. Specialized sub-sche-
mas for specific platforms may introduce additional normed fields (Listing 2). This approach allows the definition of abstract
architectural patterns (e.g., Master–Worker relationships) with later instantiation for different low-level platform specific
characteristics. New sub-schemas for novel platforms or extension of existing descriptors can be provided by application
programmer, tool-developer or even hardware vendors.

The LogicGroupAttribute allows to define group identifiers for sub-sets of PUs. Properties can be provided either as fixed or
unfixed entities. Values for unfixed properties are marked to be editable by other tools or users. This allows the definition of
required descriptors at program composition time with later instantiation by a runtime or other machine dependent library
for optimized program execution.

3.4. Example

To illustrate the usage of the platform description language (PDL) we provide an example for a state-of-the-art GPGPU
system. (See Listing 1)
We present an abstract logical organization of one x86 Master processing-unit comprising one attached GPU-Worker PU. Additional PUDescriptor entities refer to core-architectures in this abstract example. Due to the extensible design, such properties can be inserted by platform specific mechanisms without changing the conceptual control-view utilizing a Master–Worker pattern. Definition of available descriptors following the base property’s abstract key/value mechanism is done independently for each concrete platform. This enables the portable mapping and transformation of abstract architectural (control-view) patterns to concrete physical platform configurations with often implicit control-relationships. In Listing 2, we show concrete instantiations of additional platform dependent properties for different PU types and platforms.

Listing 1. PDL example description for x86-core (Master) and gpu (Worker).

Listing 2. Example concrete platform properties for different PUs and platforms.
4. Cascabel source-to-source translator

To show applicability of the proposed PDL for a task based programming model we have developed Cascabel, a prototypical code-generation system for offloading computational tasks to heterogeneous processing-units. The Cascabel programming model is based on sequential, high-level task based input programs with additional source-code annotations. Annotations indicate function invocations (computational tasks), suitable for execution on possibly heterogeneous processing units. We define tasks as self-contained units of work implemented as C/C++ functions with void return type and input and output parameters. Tasks may only access data passed via task parameters. A task may have multiple concrete task implementations for different heterogeneous platforms but offers same functionality and function signature for all implementations.

Our tool is parameterized with PDL descriptions to optimize high-level annotated C/C++ input programs for different hardware configurations (Fig. 3). It additionally provides a repository for storing platform description files and for task implementation variants.

The platform description language, serves to support code generation and program composition of Cascabel input programs for different target architectures.

Program portability between different target environments is reached via selection of suitable task implementation variants and target libraries supported by information expressed in the PDL. We separate between two use-cases for the PDL in this context:

**Task implementation meta-data.** We observe, that for efficient development of task implementation variants often detailed knowledge about an expected target environment is required. The PDL allows to formulate such task-specific platform constraints in an explicit manner. Programmers can equip highly-tuned task implementations with additional meta-data in the form of PDL descriptions. In what follows, we refer to such meta-data as platform expectation.

**Target platform descriptions.** To parametrize the code generator for different environments, concrete descriptions of target platform characteristics must be provided with the sequential input program. Target platform descriptions comprise concrete information about the target execution environment. By variation of the target description, different output programs can be created.

Our methodology aims at relieving users from querying platform dependent information and manual restructuring of high-level input programs which often results in a loss of application portability.

### 4.1. Task interfaces

Our approach relies on the concept that for a single well-defined functionality, multiple different task implementations may exist. In what follows, we will reference to the abstract specification of task functionality as task interface. Task interfaces are akin to a specification of function signature with additional meta-data. To avoid the redundant specification of interface meta-data for each task implementation, we define a simple interface specification schema. Interfaces comprise the following task meta-data: (1) Name, (2) Description and (3) Parameterlist. Each Parameter in Parameterlist consists of information about (3a) Name, (3b) Datatype, (3c) Access-mode, (3d) Usage-type (optional) and (3e) Relationslist (optional). Access modes (3c) are defined as read, write, readwrite. Usage-type (3d) and Relationslist (3e) enable to gain additional information about intended parameter usage. We currently support array based usage types such as Vector and Matrix. Supportive parameters (e.g., storing array dimensions) reference to other parameters (e.g., storing a reference to first array value) via...
Relations list (3e). We constrain array parameters to being contiguously stored in memory. Arrays of pointers are not supported.

4.2. Task annotations

Based on the observations made during evaluation of existing heterogeneous hardware and software environments, we come to the conclusion that offloading computational tasks is a widely-used approach to utilize heterogeneous hardware resources [3,7,13,18,15].

Our preliminary programming concept is based on serial C/C++ code with additional source-code annotations (pragmas). Presented annotations are a revision of previous proposals [19,20] for programming heterogeneous systems and try to cover the minimal requirements for our demonstration. Those requirements are: indication of task implementation variants, task identification specifiers, indication of task call-site and reference to groups of processing-units in PDL descriptors. The task pragma annotation (Fig. 4(a)) indicates a function definition suitable as task implementation variant. Multiple task implementations for the same task interface and different heterogeneous target environments may exist for possible utilization by a high-level program. Expect holds a list of references to predefined PDL descriptors capturing platform expectations. This clause enables programmers to equip task implementations with meta-data describing platform specific constraints and requirements. This aims at supporting tools such as code-generators or runtime-systems with additional information to facilitate task implementation pre-selection for a concrete execution target. The interface clause holds a single interface-specifier indicating a reference to a task interface name. All task implementations with same functionality and signature must reference this identifier. Name denotes an additional identifier for a particular task implementation.

The execute annotation (Fig. 4(b)) marks the call-site of a task and must be placed before the respective task invocation. A reference to the task interface name is given via the interface clause. Interface-specifier holds the unique interface name. The target clause holds a reference to LogicGroupAttribute as defined in Section 3.3 for processing-units. The parameter clause enables to define parameter data-partitioning information used for data-parallel execution of tasks. Partition-list holds a partitioning identifier (e.g., block, cyclic) and parameter size for each array dimension.

The reference to an LogicGroupAttribute via the target clause serves to denote sub-parts of a heterogeneous platform where specific tasks are intended to execute. This facility allows mapping of task implementations to a generic machine model. From that generic model a compiler or run-time can further automatically derive optimized mapping decisions to physical hardware elements. To give a concrete example, the Nvidia Cuda platform model defines a host running C code capable of offloading threads to a separate device [7]. In our model the host is expressed either as master or hybrid PU. The device relates to a worker element. By making that information explicit with our PDL, tools can implement transformations between different platforms and optimize task-mapping for complex heterogeneous environments.

4.3. Code generation

The current implementation of our tool utilizes the ROSE compiler framework [21]. We shortly summarize the following main steps of the code-generator.

Task registration. Code regions outlined by task annotations are registered in the task repository. In case multiple implementation variants for the same task interface exist, those are marked for potential variant selection.

Static task implementation variant pre-selection. The platform patterns specified for available task implementation variants (platform expectations) are compared to the platform description of the target environment. This serves pre-pruning of task variants not suitable for the target as well as static mapping of tasks to potentially available hardware resources.

Output generation. Based on the previously analyzed platform information, output source-files are constructed. This includes insertion of platform specific code for data-partitioning, transfer and task invocations. Our current implementation

#pragma csc task [clause [::] clause]...

where clause is one of the following:

interface(interface-specifier)
expect(target [::] target)...
name(task-name)

(a) Task annotation

#pragma csc execute [clause [::] clause]...

where clause is one of the following:

interface(interface-specifier)
target(processing-unit-group)
parameter(partition-list)

(b) Execute annotation

Fig. 4. Cascabel source code annotations.
can utilize an advanced runtime scheduling framework [6] for heterogeneous systems. Suitable task variants available in the task-repository are selected and included in output-files. At least one sequential fall-back variant must be provided by the application developer. This ensures the application can always be compiled for a Master PU in case no other implementations are available for the target platform.

**Compilation**. After all required source files have been constructed, platform specific compilers (e.g., nvcc, gcc-spu, xlc) produce one or more executables. The compilation and linking requirements are derived from information available in platform expectation and target platform description files. Our prototypical tool supports the creation of Cmake [22] configuration files for utilization of an automatic build system.

5. Experiments

To evaluate our framework, we investigate translations from task based sequential input programs to output programs tailored for parallel execution on a heterogeneous GPU-equipped system. Translation is based on the previously introduced source-code annotations in combination with different target platform descriptors. Each of the target platform descriptors expresses a different machine configuration for static task implementation variant selection and runtime-system configuration.

5.1. Experimental environment

All experiments were performed on a dual-socket 2.66 GHz Intel Xeon X5550 (quad-core) system with 24 GB DDR3-1333 main memory. The system is additionally equipped with 3 GPUs (2x Nvidia Tesla C2050, 1x Nvidia Tesla C1060) and running Linux Kernel 2.6.18–238.1.1.el5 (RHEL5). All examples have been compiled with GCC 4.1.2 compilers and -O3 optimization. Nvidia CUDA Toolkit 3.2 was used for GPU enabled experiments. We report application runtime (wall-time) for all benchmarks. We consider application runtime including runtime calls and GPU data transfers as an appropriate measure for our experiments which are based on sequential input programs with additional annotations. For all test cases mean values from five runs are reported.

5.2. Runtime system

To facilitate dynamic task scheduling and data management for generated output programs, we employ the StarPU runtime-system (ver. 0.9.1). StarPU [6] is a scheduling environment for heterogeneous systems with a portable abstraction of computational tasks. The StarPU task notion shows many similarities to the widely-used programming concept of using multiple task implementation variants implemented for different optimization strategies or target architectures. We also use a similar high-level task abstraction, therefore we consider the StarPU API as one viable target for our code generator. We observe that by using StarPU as a back-end, code generation of data management and task invocation code can be significantly simplified. In agreement with other works [23–25], we conclude that many performance relevant data-transfer and task scheduling decisions should be taken at run-time. Therefore, we aim at supporting dynamic schedulers with task variant pre-selection and task meta-data encapsulated in PDL descriptions. The StarPU runtime system offers sophisticated task scheduling policies for heterogeneous systems [26]. All of our reported experiments utilize the standard eager greedy policy with a central task queue [6], we leave detailed evaluation of different scheduling schemes in combination with different target platform descriptions to future work. For utilization of the StarPU runtime-system by our code generator, an input target platform description that specifies it’s availability must be provided.

5.3. Framework usage

1. Identification of tasks and code annotations. To utilize the framework, users need to identify computational tasks suitable for offloading to other processing units and/or data-parallel execution. In many cases this will be analogue to identification of application hot-spots. The following pseudo-code illustrates a serial program executing the task matmul.

```c
... int main(void) {
    ... matmul(A, B, C, m, n, k, lda, ldb, ldc);
    ... }
```
After code regions denoting tasks have been identified, they are annotated via our previously introduced pragmas. For a \textit{matmul} task implementation variant the according function implementation is annotated as follows:

1. \begin{verbatim}
#define csc_task expect (singlecore); interface (matmul); name (mm_impl01)
void mm_impl01 (double *A, double *B, double *C, int m, int n, int k,
    int lda, int ldb, int ldc ){
    ... matmul task implementation ...
}
\end{verbatim}

The interface specification \textit{matmul} (Subsection 4.1) comprises access modes for parameters and information about parameter usage. Associated function call sites are marked with the following exemplary annotation:

2. \begin{verbatim}
#define csc_execute interface (matmul); target (set01); parameter (A(m:block, k:*), B
    (k:*), n:block), C(m:block, n:block))
matmul (A, B, C, m, n, k, lda, ldb, ldc);
\end{verbatim}

The \textit{target} clause denotes a set processing units \textit{(set01)} where \textit{matmul} tasks are intended to execute. Concrete organization of \textit{set01} must be defined in the target platform description PDL file via \textit{LogicGroupAttribute} (step 4). The \textit{parameter} clause defines a block partition (rows) for matrix A, block partition (columns) for matrix B and a block partition (rows and columns) for matrix C. Partition block sizes are computed based on the target platform description. (*) denotes no partition.

2. \textbf{Registering platform expectations.} For the \textit{singlecore} 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 \textit{singlecore} platform expectation, a single \textit{Master PU} (CPU) was specified. New platform expectation PDL files may be registered via the external task repository helper program \textit{trmanager}.

\texttt{trmanager --add-platformexpect --name=singlecore singlecore.pdl}

3. \textbf{Registering implementation variants.} Additional task implementation variants for different platforms can also be provided via the \textit{trmanager} helper program. Annotated task implementation source-files are processed and stored in the repository.

\texttt{trmanager --add-tasks matmul_other.c}

4. \textbf{Target platform description.} To parameterize the code generator for a particular environment, a target platform description file must be provided. The user can influence task implementation selection and mapping via modification of \textit{LogicGroupAttribute} in target PDL descriptors. In this work we consider manual creation of target platform descriptions for specific target environments. In future, we will investigate automatic creation of PDL descriptors.

5. \textbf{Translator Invocation.} The annotated serial input program is supplied together with a target platform description to the prototype source-to-source translator.

\texttt{cascc --pdl-file=TargetMachine.pdl --create-cmake main_input.c}

The generated output files contain platform specific library calls for task execution and data transfer according to specified data-distributions and target execution groups. For compilation and linking, we support automatic creation of Cmake configuration files.

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 PDL allows tools to gain important additional information for utilization of such implementations.

5.4. \textit{Platform descriptions}

To evaluate our framework, we use different input target descriptions for each example program. Input target descriptions serve the pre-selection of available \textit{task implementation} variants from the task repository and configuration of the runtime-system. For the concrete hardware with up to 8 simultaneous multi-threading capable CPU cores and 3 GPUs, the exemplary target platform descriptions express four different logical system organizations. \textbf{Fig. 5} depicts the conceptual
view of different target descriptors used in our experiments. We use the LogicGroupAttribute of the PDL to denote a set of processing units for possible execution of tasks. In what follows, we will refer to this logical grouping of PUs as execution set. To benefit from dynamic information not available at program composition time, we leave the final task scheduling decision to the runtime system. The current implementation supports one execution set per application. Fig. 5(a) shows the “8 CPU” description with 8 CPU cores specified in one execution set. We translate this to a configuration where all cores, including the Master PU, may execute tasks. In the “7 CPU; 1 GPU” description (Fig. 5(b)) we add one GPU to the execution set and exclude the Master PU from execution of tasks. This reflects configurations where one CPU core is dedicated for coordination of a GPU. A similar configuration “5 CPU; 3 GPU” (Fig. 5(c)) comprises all 3 GPUs and 4 CPU Master units. One Master element is included in the execution set, leaving 3 Masters for possible coordination of GPUs. Fig. 5(d) depicts the “1 GPU” configuration with utilization of a single GPU for tasks.

5.5. DGEMM

The first example input program executes a double precision matrix–matrix multiplication of two $N \times N$ square matrices. It does so via invocation of a highly tuned level-3 BLAS[27] subroutine DGEMM from the sequential version of Intel Math Kernel Library (MKL) 10.3. Given the commodity of using efficient domain-specific libraries, we consider this setup as a realistic baseline example. DGEMM is defined as $C \leftarrow \alpha AB + \beta C$ where we use general square matrices with $\alpha = 1$ and $\beta = 0$.

Following the previously described methodology, we create a task function comprising the BLAS library call. To indicate that function definition as a computational task, we add cascabel pragma annotations specifying the interface name, reference to a platform expectation comprising a single CPU core with available MKL BLAS library and task identifier. The task invocation is annotated with the execute pragma specifying the interface name, an execution group identifier and parameter information. Parameter information holds array size and data partitioning hints. We specify a data partitioning scheme via the parameter annotation clause: $A (N:\text{block}, N:\ast)$, $B (N:\ast, N:\text{block})$ and $C (N:\text{block},N:\text{block})$. The partitioning information allows to split the DGEMM operation in data-parallel tasks, each computing a sub-block of $C$ independently. For the “8 CPU” target input description, the system selects the sequential task implementation, tailored for a single CPU and inserts runtime calls for data-parallel execution of those implementations. In addition to the sequential task implementation that calls the MKL routine, we added an GPU enabled version of DGEMM (CuBLAS 3.2) to the task repository.

Experiments where performed with three different target descriptions. We compare automatic translation for “8 CPU”, “5 CPU; 3 GPU” and “7 CPU; 1 GPU” descriptions with the sequential input program calling a sequential (“MKL seq.”) and a parallel (“MKL par.”) version of DGEMM directly. Fig. 6(a) shows the measured runtime of the DGEMM operation for different matrix sizes. We observe a maximum speedup of 48.09 (“5 CPU; 3 GPU”) over the sequential version when only the kernel runtime is measured. For the whole application, including all runtime library overheads (Fig. 6(b)), we still measure significant improvements compared to the sequential and parallel reference versions. The “8 CPU” target shows application speed-ups of up to 6.38 compared to the sequential baseline. We observe similar runtime behaviour for “8 CPU” and the parallel MKL version for input sizes larger $N = 8192$. Hybrid target descriptions comprising GPUs, show best execution times when the input dimension $N$ is larger 11264. We observe that specifying 3 GPUs and 5 CPU cores results in lower runtimes than using “7 CPU; 1 GPU” configuration for large problem sizes. A maximum application speedup of 18.08 compared to the sequential reference was measured for the largest problem size of $N = 25600$ and the “5 CPU; 3 GPU” target platform description.

5.6. Black–Scholes option pricing

Our second benchmark is a financial application derived from the Parsec benchmark suite[28] for CMP systems. The blackscholes benchmark is a financial analysis application for price calculation of European options. It computes a numerical solution for the Black–Scholes partial differential equation (PDE) [29]. This benchmark represents an important workload from the computational finance domain.

We adjusted the sequential baseline implementation to our task model via creating a wrapper function that calls the sequential implementation for a number of different options. We then indicated the function definition and call-site with annotation clause: A (N:block, N:*), B (N:*, N:block) and C (N:block,N:block). The partitioning information allows to split the DGEMM operation in data-parallel tasks, each computing a sub-block of $C$ independently. Following the previously described methodology, we create a task function comprising the BLAS library call.
our code annotations. Each of the option prices in the portfolio can be computed individually, therefore we defined a block partitioning scheme for all input and output arrays of the task implementation. In addition to the sequential CPU task implementation, we added an Nvidia Cuda version of this operation to the task implementation repository. Based on the different input target descriptions the code generator now selects task implementations tailored for the different environments. We compare the generated output programs for the “8 CPU”, “5 CPU; 3 GPU” and “1 GPU” configurations against the sequential input program and an OpenMP version provided by the Parsec suite. Results (mean) are reported for five benchmark program runs executing the blackScholes benchmark loop for 100 times. The homogeneous “8 CPU” descriptor shows speedups between 4.35 and 6.28 compared to the sequential input program. Only for the smallest input size of 1.02 million options, the OpenMP reference implementation achieves best performance. For all other test cases, output programs created by our tool show superior performance compared to the reference implementations. With increasing input size, the overhead of runtime-system calls has lower impact for all configurations (Fig. 7(a)). Results for GPU enabled configurations indicate that runtime is lower for “1 GPU” compared to using all available processing units (CPU/GPU) in a hybrid execution mode. This behavior results from the chosen naive scheduling policy and GPU data-transfer overheads. Hybrid application performance may be significantly improved by using a different runtime task scheduling policy [26]. Speedups for “5 CPU; 3 GPU” are between 2.19 (1.02 mil. options) and 9.52 (10.24 mil. options). The “1 GPU” configuration achieves runtime speedups between 2.19 (1.02 mil. options) and 9.52 (10.24 mil. options).

Fig. 6. Results for automatic translation of DGEMM based input program.

Fig. 7. Results for Black–Scholes application benchmark.
between 3.49 (1.02 mil. options) and 12.48 (10.24 mil. options). The maximum speedup of 12.48 for this application was measured for the largest input size and the “1 GPU” configuration (Fig. 7(b)).

6. Summary and outlook

The shift to heterogeneous many-core systems requires rethinking of existing programming models and software development frameworks. The diversity of different software libraries, runtime systems and architectural features inherent to such platforms raises many challenges for application developers and tool-chains.

In this paper we have presented an XML-based platform description language (PDL) that enables to capture key architectural patterns of commonly used heterogeneous computing systems. The PDL allows to express hardware and software properties as well as control-relationships inherent to heterogeneous platforms. The platform-specific information stored in a human-readable and machine-processable form in PDL descriptors may support both programmers and tools in application development and optimization for a target heterogeneous system.

We demonstrated the usefulness of our approach with a source-to-source transformation framework in the context of a task-based programming model, where for major computational tasks different implementation variants tailored for different types of heterogeneous execution units may exist. The platform requirements of task implementation variants are made explicit by means of corresponding PDL descriptions. We presented source-code annotations for sequential C/C++ programs that enable to indicate task implementation variants and reference groups of processing units in PDL descriptors. Our tool takes as input an annotated serial task-based program and outputs, parametrized via target PDL descriptors, code for a specific heterogeneous many-core target computing system.

Experiments for a commonly used scientific kernel (DGEMM) and a financial application (Black-Scholes) on different configurations of a state-of-the-art CPU/GPU system demonstrated the potential of our approach. Application speedups of 18.08 (DGEMM) and 12.48 (Black-Scholes) have been achieved via utilization of explicit platform descriptions for task variant selection and a heterogeneous runtime system [6].

In this paper we demonstrated our approach in the context of static code generation for a flexible scheduling framework [6] on a single-node CPU/GPU system. Experiments showed promising results. We also observe, that the hierarchical PDL is not limited to single-node systems. Therefore, we will further investigate the extension of our framework for clusters of heterogeneous nodes. Our future work will go beyond compile-time and will develop advanced support for optimized task implementation variant selection at runtime.

Acknowledgment

We would like to thank the INRIA Runtime Team for providing StarPU. This work received funding from the EU under grant agreement n248481 (PEPPHER Project, www.peppher.eu).

References