Explicit Platform Descriptions for Heterogeneous Many-Core Architectures

Martin Sandrieser, Siegfried Benkner and Sabri Pllana
Department of Scientific Computing
Faculty of Computer Science - University of Vienna, Austria
Email: {ms, sigi, pllana}@par.univie.ac.at

***Abstract*—Heterogeneous many-core architectures offer a way to cope with energy consumption limitations of various computing systems from small mobile devices to large data-centers. However, programmers typically must consider a large diversity of architectural information to develop efficient software. In this paper we present our ongoing work towards a Platform Description Language (PDL) that enables to capture key architectural patterns of commonly used heterogeneous computing systems. PDL architecture patterns support programmers and toolchains by providing platform information in a well-defined and explicit manner. We have developed a source-to-source compiler that utilizes PDL descriptors to transform sequential task-based programs to a form that is convenient for execution on heterogeneous many-core computing systems. We show various usage scenarios of our PDL and demonstrate our approach for a commonly used scientific kernel.**

***Keywords*-platform description language; task-based programming; heterogeneous many-core systems; code-generation;**

I. Introduction

Recent developments in computer architecture show that heterogeneous many-core systems are a viable way to overcome major hardware design challenges related to energy-consumption and thermal requirements 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].

Although heterogeneous processing-units may provide exceptional performance for many workloads, programming of such systems is a challenging task [2]. Software developers are exposed to a huge diversity of different software-libraries, runtime-systems and programming-models. As Chamberlain et al. [3] state, a complex mixture of different languages, compilers and run-time systems is inherent to many heterogeneous platforms. Even though first standardization approaches exist (e.g., OpenCL [4]), the diversity of underlying hardware designs and configuration options makes it very difficult for automatic tools to optimize applications and effectively distribute workloads among heterogeneous processing-units [5], [6].

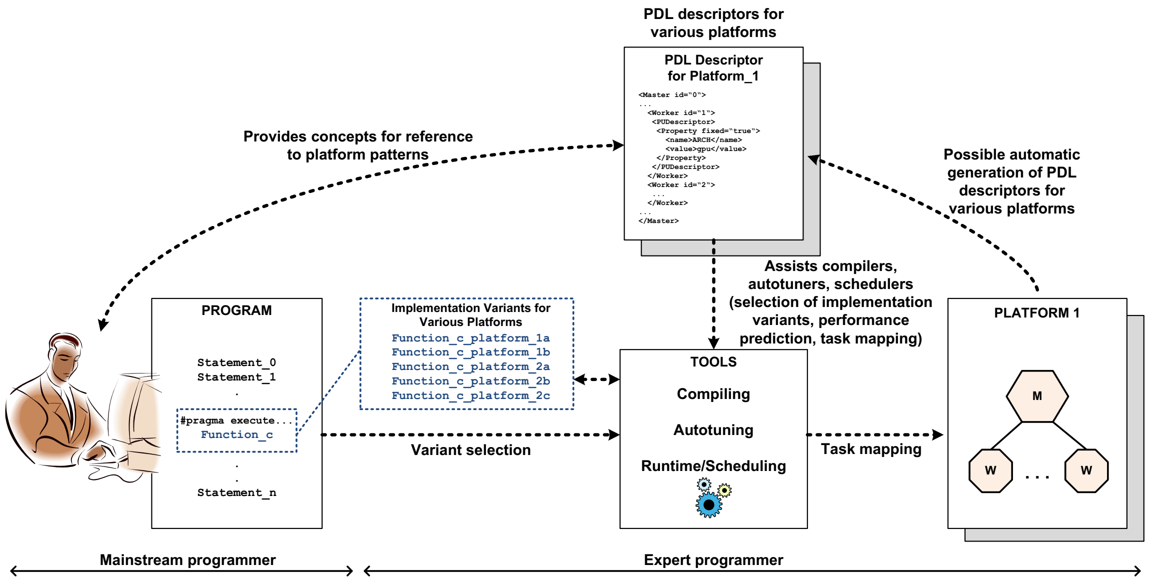
For efficient application development and compilation often detailed knowledge about the hardware configuration is required that usually exceeds the information exposed by programming models or platform layer. Commonly existing approaches employ an implicit and abstracted view on available processing 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.

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

In this paper, we show how *explicit platform descriptors* of heterogeneous many-core architectures 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) tools such as auto-tuners, compilers or runtime-systems. In this paper the usefulness of our approach is illustrated in the context of source-to-source compilation together with a task-based programming model. Tasks are indicated using *source-code annotations* which may reference information in PDL descriptors to support *delegation of tasks* to a particular part of a target platform. Thus information provided by users can support compilers and runtime-systems to optimize *mapping* of computation to processing units. 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 target heterogeneous many-core computing system. By varying the target PDL descriptor our compiler can generate code for different target architectures without the need to modify the source program. Preliminary experiments are performed with a commonly used scientific kernel.

This paper is structured as follows. Section II gives an overview of our approach and describes possible usage scenarios. Section III describes the platform description language. Our approach is evaluated in Section IV. Sections V and VI discuss related work and conclusions.

1530-2075/11 $26.00 © 2011 IEEE
DOI 10.1109/IPDPS.2011.280


PDL descriptors for various platforms
PDL Descriptor for Platform_1
Provides concepts for reference to platform patterns
Possible automatic generation of PDL descriptors for various platforms
Assists compilers, autotuners, schedulers (selection of implementation variants, performance prediction, task mapping)
PROGRAM
Statement_0
Statement_1
#pragma execute...
Function_c
Statement_n
Implementation Variants for Various Platforms
Function_c_platform_1a
Function_c_platform_1b
Function_c_platform_2a
Function_c_platform_2b
Function_c_platform_2c
TOOLS
Compiling
Autotuning
Runtime/Scheduling
Variant selection
Task mapping
PLATFORM 1
M
W
W
Mainstream programmer
Expert programmer

```
<Master id="0">
...
  <Worker id="1">
   <PUDescriptor>
    <Property fixed="true">
      <name>ARCH</name>
      <value>gpu</value>
    </Property>
   </PUDescriptor>
  </Worker>
  <Worker id="2">
   ...
  </Worker>
...
</Master>
```

Figure 1: PDL usage scenarios. In a high-level program mainstream programmers may use annotations to reference platform patterns. Expert programmers provide implementation variants for specific platforms. Tools can use PDL descriptors for selection of implementation variants, performance prediction or task mapping.

II. Overview and Usage Scenarios

In this section we provide an overview of our approach and highlight various possible usage scenarios.

Programmability of heterogeneous computing systems is addressed in several frameworks [7]–[9], by providing annotations for outlining computational tasks, a source-to-source compiler with corresponding run-time systems. We are involved in the EU-funded project PEPPHER [10] which addresses programmability and performance-portability for heterogeneous many-core systems. PEPPHER targets a task-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.

We have developed a platform description language (PDL) that can express hardware and software properties as well as *control-relationships* inherent to specific 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 Figure 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 preselection and mapping to concrete systems, we identify 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 logic 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 [11] or OpenCL platform query functions [4].

III. Platform Description Language

In this section we describe the hierarchical machine model and the corresponding XML-based platform description language (PDL).

A. 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.

Processing units are divided into three different classes, as shown in Figure 2.

- 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 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 either by other *Hybrid* or *Master* units. 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.

Figure 2 depicts the described PU classes in an example hierarchical organization.

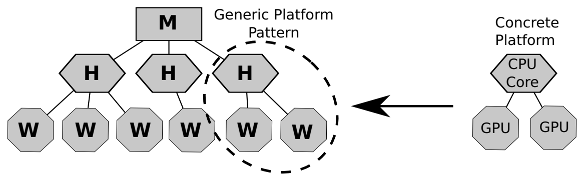


Figure 2: The PDL allows the definition of generic control patterns between Master (M), Hybrid (H) and Worker (W) processing-units. Concrete platforms are mapped to generic processing-unit hierarchies to support portability.

The control-relationship hierarchy, in combination with distinct memory spaces, reflects the need for efficient vertical data-movement observable in the majority of today's heterogeneous applications [12]. To support the definition of data-paths and possible transfer requirements, the abstract machine model incorporates the following communication 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 properties can include affinities, relative speeds to PUs, 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 communication schemes, underlying bus infrastructure or other communication performance descriptors.

B. Specification of PDL

To utilize the generic and hierarchical machine model presented in Section III-A, we define an XML-based platform description language. As stated previously, today's *heterogeneous platforms* can be seen as a complex mixture of different hardware entities and software environments [3]. Therefore, we follow a holistic approach and do not restrict description facilities to pure hardware properties. Our platform description language 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 platforms, for example a *Master*-*Worker* relationship between CPU and GPU. Besides support for genericity and extensibility, platform description facilities have been carefully specified to reflect 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.

Initial specification comprises the following XML entities.

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

Figure 3 shows a conceptual overview of the PDL with entity relationships. One challenge for definition of the platform description is the ability to adapt to future, yet unknown hardware architectures and their requirements. Although the high level representation provides a generic way to model large sets of architectures, machine specific details need to be emphasized on lower levels of abstraction. Therefore we introduce extensible *Descriptor* and *Property* types. The specification of these types can be altered to cover software and hardware specific information only available at lower layers of the employed toolchain. This ensures the availability of low-level performance relevant information while still adhering to the base abstract platform description schema.

We apply standard techniques such as schema inheritance and XML entity polymorphism together with predefined platform descriptors. Predefined *Descriptor* and *Property* subschemas have unique identification and versioning support provided by the XSD. New subschemas for novel platforms or extension of existing descriptors can be provided by application programmer, tool-developer or even hardware vendors.

Concrete platform information can be made available at multiple levels of heterogeneous toolchains. This means, that the utilization of platform information may not only be relevant for static code-generation frameworks, as presented in this work, but also for runtime-libraries or user-code. The *LogicGroupAttribute* allows to define group identifiers for sub-sets of PUs. Hand written 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.

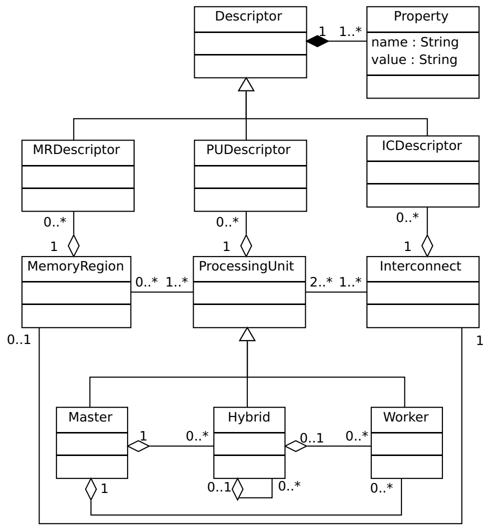


Figure 3: UML class diagram of the platform description language with generic descriptors.

C. Example

To demonstrate the usage of the previously defined platform description language (PDL) we provide an example for a state-of-the-art GPGPU system.

```
<!-- XML HEADER -->
<Master id="0" quantity="1">
  <PUDescriptor>
    <Property fixed="true">
      <name>ARCHITECTURE</name>
      <value>x86</value>
    </Property>
    <!-- Additional properties -->
  </PUDescriptor>
  <Worker quantity="1" id="1">
    <PUDescriptor>
      <Property fixed="true">
        <name>ARCHITECTURE</name>
        <value>gpu</value>
      </Property>
      <!-- Additional properties -->
    </PUDescriptor>
  </Worker>
  <Interconnect type="rDMA" from="0" to="1" scheme=""/>
</Master>
```

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

As observable from Listing 1, the XML description facilities are very well suited for hierarchical representation of elements. 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 what follows, we show a concrete instantiation of additional platform dependent properties for the GPU-Worker gathered by Nvidia OpenCL [4] run-time libraries.

```
<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>
  <Property fixed="false" xsi:type="
        ocl:oclDevicePropertyType">
    <ocl:name>GLOBAL_MEM_SIZE</ocl:name>
    <ocl:value unit="kB">1572864</ocl:value>
  </Property>
  <Property fixed="false" xsi:type="
        ocl:oclDevicePropertyType">
    <ocl:name>LOCAL_MEM_SIZE</ocl:name>
    <ocl:value unit="kB">48</ocl:value>
  </Property>
</PUDescriptor>
```

Listing 2: Excerpt of concrete platform properties defined for OpenCL on GPUs. Generated from OpenCL run-time libraries.

While base descriptors for common platforms may be provided a priori, the extension for specialized information required by specific applications or tools is possible at any time via XML schema inheritance.

IV. Case Study

To show applicability of the featured concepts for a task based programming model we introduce *Cascabel*, a prototypical code-generation system for offloading suitable tasks to heterogeneous processing-units. Our platform description language, in combination with a simple query API, can support code generation and program composition for task-based input programs on a variety of target architectures. We observe that the PDL facilitates mapping of code written for specific heterogeneous environments to actual hardware configurations. This shifts the burden of querying complex and platform dependent information away from user-space to external mechanisms supported by our PDL. Subsequently, high-level task-based applications can be optimized for heterogeneous platforms without restructuring or loss of application portability.

A. 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 ([4], [7]–[9], [13], [14]). We identify the following key benefits of high-level task programming models:

- Explicit task outlining with parameter access-specifiers helps compilers and runtime-systems to derive inter-task data-dependencies.
- High-level task parallel work distribution eases handling of distinct, non-coherent memory spaces often present in heterogeneous systems.
- Nesting of explicit tasks facilitates mapping of computation and data-decomposition onto hierarchically organized system resources.

Considering the commodity of high-level task offloading, we introduce a simplified tasking model. It is not our intention to define a novel programming style but to demonstrate the capability of our PDL to assist tools in the process of optimizing programs for heterogeneous hardware and software environments. To the best of our knowledge, this parametrization of supportive tools based on external platform descriptions goes clearly beyond existing task based programming approaches for heterogeneous systems. Integration into existing programming models (e.g., OpenMP-Tasks) seems also feasible.

Our preliminary programming concept is based on serial C/C++ code with additional source-code annotations (pragmas). Presented annotations are based on our experiences with programming heterogeneous systems and try to cover the minimal requirements for our demonstration. Those requirements are: outlining of task implementation variants with additional parameter information (access-mode, data-distribution), task identification specifiers, indication of call-site and reference to groups of processing-units in PDL descriptors. *Tasks* are defined as self-contained units of work (algorithmic solutions) with input and output parameters. A *task* can have multiple *task implementations* for different heterogeneous platforms but offers same functionality and function signature for all implementations. We specify the following pragma syntax:

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

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

The following listings show the example usage of task definition and execution annotations:

```
// Task definition
#pragma cascabel task :x86
                      :Ivecadd
                      :vecadd01
                      :(A:readwrite,
                        B:read)
void vector_add(double *A, double *B){...};
```

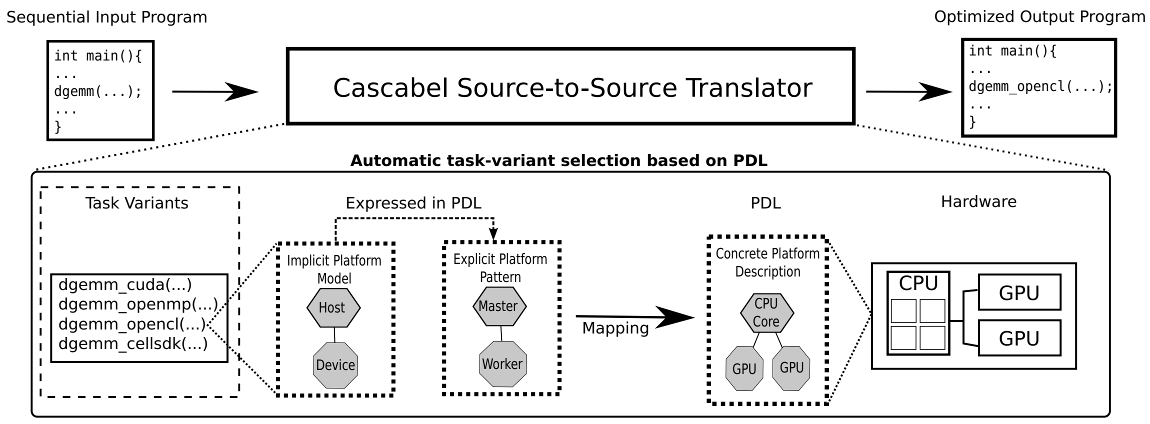


Figure 4: Overview of the prototype source-to-source translator code-named *Cascabel*. Suitable task variants for concrete hardware are selected with help of the PDL. Platform patterns in the form of control-relationships can be expressed for task variants. These patterns are mapped to concrete platform descriptions also expressed in the PDL (e.g., a *Master-Worker* relationship between CPU/GPU).

```
// Task execution
#pragma cascabel execute Ivecadd
                          :execution_set01
                          (A:BLOCK:N,
                           B:BLOCK:N)
vector_add( A, B );
```

The *task* annotation indicates a function (C/C++ function definition) suitable for execution on a heterogeneous platform. Following the identifier, *targetplatformlist* specifies one or more concrete platforms (e.g., OpenCL, Cuda, Cell-SDK) the code is intended for. *Taskidentifier* denotes a reference to the task interface name. All task implementations with same functionality and function signature must reference to this name. *Taskname* specifies an additional unique name for task implementations. This can serve the grouping of associated portions of *master, hybrid* and *worker* code required for many heterogeneous platforms (e.g., Nvidia Cuda). *Parameterlist* holds a list of function parameter-names with explicit *access* specifiers. Access modes include *read*, *write* and *readwrite*.

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 subsets of PUs in abstract platform descriptions. It refers to a *LogicGroupAttribute* as defined in Section III-B for processing-units. 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 run-time system for possible data-decomposition of data-parallel tasks.

At a high level of abstraction, the application is defined as a sequential program with additional task definitions and execution annotations. If fall-back sequential task implementations for *Master* processing-units are provided, the high-level input program can be executed on all systems where an appropriate C/C++ compiler is available.

B. Task Mapping

In complex heterogeneous system configurations with diverse hardware elements, libraries and execution models, mapping of different task implementations to available processing units is a complex job. Most platform libraries force the user to handle this mapping manually and expose only a narrow and static view on available hardware resources. This may be sufficient for simple hardware configurations, but with increasing system complexity task-mapping problems become predominant for application developers. Although some static and dynamic approaches exist ([12], [15], [16]), we identify additional need to alleviate the mapping process. Our previously defined PDL can support static and dynamic task-mapping via the expression of generic architectural patterns. In what follows we illustrate this with an static example.

The *execute* annotation enables via the *LogicGroupAttribute* the specification of execution groups for denoting 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* [13]. 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.

C. Code Generation

We developed a prototypical source-to-source compiler based on the Rose framework [17]. Our tool is parameterized with PDL descriptions to optimize high-level annotated C/C++ input programs for different hardware configurations (Figure 4). It additionally provides a repository for managing task implementation variants tailored for different heterogeneous platforms.

We shortly summarize the following main steps of the code-generator:

1) **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.
2) **Static task pre-selection**. The platform patterns specified for available task implementation variants 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.
3) **Output generation**. Based on the previously analyzed platform information, output source-files are constructed. This includes insertion of highly platform specific code for data-partitioning, transfer and task invocations. The PDL allows us to derive data-transfer paths between memory-regions and communication between processing-units via the explicitly specified *interconnect* entity. If a runtime-system for automatic data-management is employed, the PDL information can be exposed to the runtime for additional optimizations.
   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.
4) **Compilation**. After all required source-files have been constructed, platform specific compilers (e.g., nvcc, gcc-spu, xlc) produce one or more executables. The required compilation and linking plan is derived from information available in the platform description file. While for the preliminary prototype this knowledge has to be provided by the user, further automation of selecting optimized compiler and linker settings is feasible.

D. Preliminary Experiment

In what follows, we show results of an example translation from a serial high-level input program to output programs that utilize the *StarPU* [16] runtime-system. StarPU is a scheduling and data-management framework for heterogeneous-manycore systems. Experiments were performed on a dual-socket 2.66GHz Intel Xeon X5550 (Quad-Core) system comprising two Nvidia GPUs (GTX480 and GTX285). The 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 selects other available DGEMM implementations from the task implementation repository and constructs output programs based on PDL platform descriptions. Figure 5 depicts speedup against the single threaded input program (single). It is translated to two different test programs (starpu and starpu+2gpu). The ”starpu” version comprises the input task-implementation of DGEMM and is optimized for data-parallel execution via StarPU on 8 CPU-Cores. The second program ”starpu+2gpus” utilizes the GPUs and executes a CuBLAS DGEMM implementation (Cuda Toolkit 3.2). Both output programs were created using different PDL descriptions without modification of the serial input program.

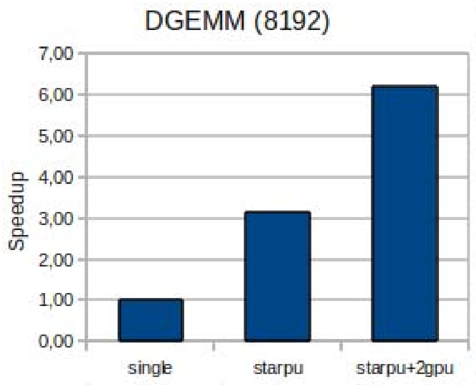


Figure 5: Speedup after translation from single threaded input program (single) to multithreaded (starpu) and GPGPU (starpu+2gpu) versions.

V. Related Work

Using high-level architectural abstractions to simplify programming of heterogeneous many-core systems has been applied by several previous approaches. *Sequoia* [12] introduces a programming language based on a hierarchical abstraction of distinct memory spaces present in many of today's heterogeneous systems. It addresses the issue of efficient vertical data-movement in a tree of memories via a specific programming model. While our hierarchical machine model shows a similar tree abstraction, it emphasizes control relationships between processing-units. Important memory-hierarchy characteristics are additionally captured via PDL *MemoryRegion* and *Interconnect* entities. This gives our approach the flexibility to adopt to a wide variety of programming- and execution models which need not necessarily be bounded to recursive data-decomposition.

Programming heterogeneous systems is also targeted by the OpenCL [4] and Nvidia Cuda [13] 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., GPGPUs). Other approaches addressing heterogeneous many-core systems are the IBM Accelerated Library Framework [14], CAPS HMPP [7] and PGI Accelerate [18]. They all employ platform abstractions of "accelerator" processing units. To the best of our knowledge, 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* [11] 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.

VI. Conclusions and Future Work

The shift to heterogeneous many-core systems requires rethinking of existing programming models and software development frameworks. In this paper we have presented preliminary results that show how a Platform Description Language enables to capture key architectural patterns of commonly used heterogeneous computing systems. Programmers as well as toolchains can benefit from explicitly exposed platform information. We have presented our prototype source-to-source compiler that uses PDL architecture descriptors to transform annotated sequential task-based programs for execution on heterogeneous many-core computing systems.

We have observed that tracking dynamically changing system resources via platform descriptors can be difficult. In future we will investigate how platform descriptors could be utilized for supporting highly dynamic run-time schedulers.

References

[1] A. R. Brodtkorb, C. Dyken, T. R. Hagen, J. M. Hjelmervik, and O. O. Storaasli, "State-of-the-art in heterogeneous computing," *Scientific Programming*, vol. 18, no. 1, pp. 1–33, 2010.

[2] S. Pllana, S. Benkner, E. Mehofer, L. Natvig, and F. Xhafa, "Towards an intelligent environment for programming multicore computing systems," in *Euro-Par 2008 Workshops - Parallel Processing*, ser. LNCS. Springer, 2009, vol. 5415, pp. 141–151.

[3] R. D. Chamberlain, M. A. Franklin, E. J. Tyson, J. Buhler, S. Gayen, P. Crowley, and J. H. Buckley, "Application development on hybrid systems," in *2007 ACM/IEEE conference on Supercomputing-Volume 00*, 2007, pp. 1–10.

[4] A. Munshi, "The OpenCL specification, version 1.1," Sep. 2010.

[5] 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 (iWAPT2010)*, 2010.

[6] S. Rul, H. Vandierendonck, J. D'Haene, and K. D. Bosschere, "An experimental study on performance portability of OpenCL kernels," in *Application Accelerators in High Performance Computing, 2010 Symposium, Papers*, 2010.

[7] F. Bodin and S. Bihan, "Heterogeneous multicore parallel programming for graphics processing units," *Scientific Programming*, vol. 17, pp. 325–336, Dec. 2009.

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

[9] J. Planas, R. M. Badia, E. Ayguade, and J. Labarta, "Hierarchical Task-Based programming with StarSs," *International Journal of High Performance Computing Applications*, vol. 23, no. 3, pp. 284–299, 2009.

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

[11] F. Broquedis, J. Clet-Ortega, S. Moreaud, N. Furmento, B. Goglin, G. Mercier, S. Thibault, and R. Namyst, "hwloc: a generic framework for managing hardware affinities in HPC applications," in *PDP, 2010 18th Euromicro International Conference on*, 2010, pp. 180–186.

[12] 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 *2006 ACM/IEEE Conference on Supercomputing*, 2006, p. 83.

[13] J. Nickolls, I. Buck, M. Garland, and K. Skadron, "Scalable parallel programming with CUDA," *Queue*, vol. 6, no. 2, pp. 40–53, 2008.

[14] IBM, "Accelerated library framework programmers guide and API reference v3.1," 2009. Available: http://public.dhe.ibm.com/software/dw/cell/ALF_Prog_Guide_API_v3.1.pdf

[15] C. K. Luk, S. Hong, and H. Kim, "Qilin: exploiting parallelism on heterogeneous multiprocessors with adaptive mapping," in *Microarchitecture, 2009. MICRO-42. 42nd Annual IEEE/ACM International Symposium on*, 2010, pp. 45–55.

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

[17] "ROSE." [Online]. Available: http://www.rosecompiler.org/

[18] Portland Group, "PGI fortran & c accelerator programming model v1.2," 2010.