Exploiting Multi-Level Parallelism in Streaming Applications for Heterogeneous Platforms with GPUs

Balevic, A.

Chapter 1

Introduction

What started off as a simple graphics card is nowadays a pervasive programmable accelerator called a *Graphics Processing Unit* (GPU). The success of GPU as an accelerator for data parallel computationally intensive tasks inspired numerous application designers to parallelize their application hot-spots and port them as *kernels* to the GPU architecture. Modern GPUs exhibit massive parallelism and can be used for general purpose computing, i.e. they are compute-capable. As such, GPUs are well-suited to address the ever increasing computational demands in automotive, medical, entertainment and bio-informatics fields. Nowadays, CPUs are often combined with GPUs and intellectual property (IP) cores implementing special functions, to form a *heterogeneous platform*. The emergence of heterogeneous platforms increasingly blurs the distinction between the embedded systems domain and the high performance computing (HPC) domain.

While heterogeneous platforms such as NVIDIA Tegra, Apple A5X ARM system-on-chip (SoC), and TI's OMAP 5430 offer the dual benefits of higher performance and better energy efficiency due to the presence of a CPU and a GPU in a single design, they also pose unprecedented design and implementation challenges for application programmers. Tapping into the parallelization potential of heterogeneous platforms is a challenging task. Even parallel programming for a single multicore processor is difficult [139], since the programmer is faced with numerous questions - from finding the parallelism and partitioning the application, to the correct and efficient realization of communication and synchronization between tasks. Modern platforms embrace both increasing degrees of parallelism and heterogeneous execution units. Parallelism in architectural components ranges from fine-grain *instruction-level parallelism* (ILP) typically found in superscalar pipelined processors and VLIW processors, *data parallelism* found in vector processor or processor array architectures, to coarse-grain *task-level parallelism* (also referred to as process-level parallelism [31]

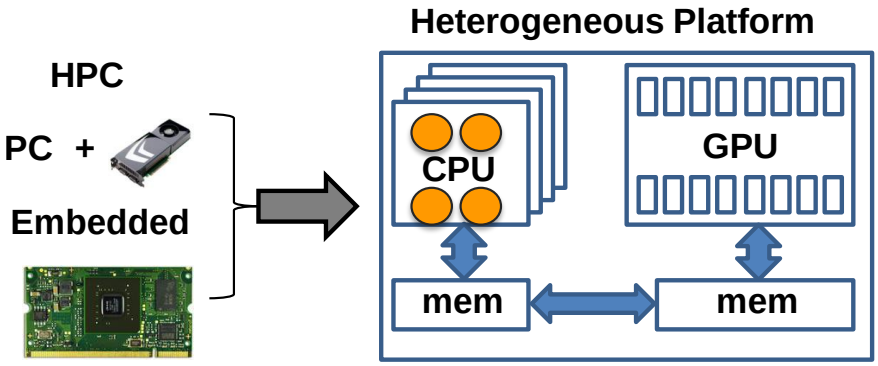

Figure 1.1: A heterogeneous platform featuring a quadcore CPU and a GPU.

and thread-level parallelism [72]) typically found in multiprocessor systems. A combination of parallelism at different levels of the platform results in *multi-level parallelism* [31].

Let us examine some forms of parallelism that can be found in a heterogeneous platform featuring a quadcore central processing unit and a single GPU accelerator shown in Figure 1.1. To exploit this platform, it is necessary to find parallelism in the application and map it onto different architectural components of the platform, i.e. in this case the four CPU cores and the GPU. The classical approach to application parallelization involves *functional decomposition* of the application into different tasks, thus revealing *task parallelism*. Task parallelism is a form of parallelism where different tasks execute concurrently on multiple platform components, e.g., processor cores. After task parallelism in the application is revealed, some classes of applications, such as *streaming applications*, can benefit from yet another, closely related form of parallelism - *pipeline parallelism*. Streaming applications are the applications that transform an input stream of data (tokens) into an output stream. As a result of streaming, the processing of consecutive tokens by different tasks can occur at the same time, thus resulting in pipeline parallelism. Programming massively parallel computing accelerators, such as GPUs, requires a fundamentally different parallelization model. For example, GPUs feature a data parallel architecture based on an array of parallel processors. To exploit the computational power of GPUs, it is necessary to find data parallelism in the application and to transform it into the form of a GPU code (*kernel*) that GPU threads execute in parallel. By *data parallelism*, we refer to the type of parallelism where the same operation is applied to multiple elements in a data set [31]. Further, the *hierarchical decomposition* of the application leads to *multi-level parallelism*. Multi-level parallelism lets us to combine different forms and granularities of parallelism on the platform, such as task, data, and pipeline parallelism.

Leveraging task, pipeline and data parallelism on the same platform makes it necessary to use different programming models and APIs. The architectural diversity between platform components puts additional pressure on the designer, since different platform components typically require different programming models and skill sets. Numerous languages, libraries and tools have been developed which share the same goal - to make the parallelization process easier for the designer. While it is hard to set a clear boundary, we roughly classify the parallelization approaches into three main categories according to the degree of automation:

- Explicit parallel programming
- Semi-automatic approaches (parallel languages, directive-based compilation, run-time environments)
- Automatic parallelization (transformation frameworks)

Explicit parallel programming using the programming model and the application programming interface (API) provided by the vendor is still the most dominant category used for parallelizing applications today. Parallelization for multicore processors includes writing a multi-threaded application using C/C++ POSIX Pthreads, Win32MT, or Boost libraries, which provide concepts for expressing task parallelism (i.e. using threads) and implementing synchronization in the parallel program. GPUs are typically programmed using a vendor specific programming language such as NVIDIA's CUDA [37], or using the OpenCL standard [122] for heterogeneous parallel programming.

The semi-automatic approaches have been rapidly gaining momentum in the last few years. Examples of semi-automatic parallelization approaches include languages for expressing parallelism explicitly, e.g. StreamIt [127], directive-based compilation, e.g. OpenMP [40, 102], OpenACC [103], run-time libraries and task-scheduling environments, such as StartPU [9, 10], or a combination of a run-time environment and directive-driven code parallelization, as in HMPP [47] and OmpSs [11, 48]. While the designer still needs to partition the application and identify parallel tasks, semi-automatic approaches relive the programmer of having to explicitly write parallel code using vendor specific APIs and libraries. Instead, the parallel code is generated following the designer's parallelization directives.

The automatic transformation of sequential code into a parallel executable is one of the ultimate parallel computing goals [49]. While we are still a long way from automatic parallelization of arbitrary code, significant progress has been made in parallelization of more specific applications under some constraints (see Chapter 2).

The challenge addressed in this dissertation is compile-time generation of structured, multi-level parallel programs which exploit different forms of parallelism, such

as task, data, and pipeline parallelism, for efficient mapping onto heterogeneous platforms with massively parallel computing accelerators (such as GPUs).

1.1 Research Framework

The research work that resulted in this dissertation began in late 2009 within an industry-academia research cooperation in the framework of the Tera-Scale Multi-Core Architectures (TSAR) project. The TSAR project (2008-2012) is an European project in the MEDEA+ framework conducted by an international consortium of industrial and academic institutions comprised of BULL, UPMC/LIP6, Thales Communications, FZI, Philips Medical Systems, ACE Associated Compiler Experts bv, Compaan Design bv, and Leiden Embedded Research Center (LERC).

During the TSAR project, there was a close collaboration between LERC and Leiden University's spin-off company Compaan Design on novel solutions for the challenges of automatic parallelization for heterogeneous platforms with GPU accelerators. The role of LERC was to capture the total system view of high-performance streaming applications on platforms with a GPU accelerator, while using and further advancing tools and techniques developed at LERC and Compaan.

LERC and Compaan have a long tradition of research [2,81,97,98] on the Kahn Process Networks (KPNs) [76] model of computation and its polyhedral variant called Polyhedral Process Networks (PPNs). The Kahn Process Network (KPN) model of computation was introduced in 1974 by the prominent French scientist Gilles Kahn [76]. The initial purpose of the KPN model was modeling parallel programs in distributed systems. In the period 2000 - 2009, the Kahn Process Networks (KPN) and its variants gained large acceptance in embedded systems design due to the clear separation of communication and computation [83]. A Kahn Process Network describes an application as a network of concurrent autonomous processes that communicate via tokens over channels. For example, on heterogeneous platform in Figure 1.1, each KPN process can be concurrently executed on a different architectural component that has its own private memory. In the KPN, the tokens are transmitted over unidirectional communication channels. Each KPN communication channel has exactly one writer process and exactly one reader process. Furthermore, each KPN channel is an infinite, first-in first-out (FIFO) queue of tokens.

An important milestone in compiler research has been the introduction of the *polyhedral model* which is mathematical representation of program code that enables program analysis and transformation [42, 55, 86]. In the polyhedral model, the iteration domains of program statements are represented as polytopes. The mathematical representation of the program code provides a powerful basis for further transformations. To be represented in the polyhedral model, the program code needs to satisfy certain

properties (see e.g., Chapter 2). While significant progress has been made on extending the boundaries of programs that could be represented in the polyhedral model, see e.g. [119], in this thesis we consider only the programs that can be expressed as *Static Affine Nested Loop Programs* (*SANLPs*) (Section 2.1). A Polyhedral Process Network (PPN) [134] emerged as an important variation of the KPN model, where program statements, dependence edges, and the input and output arguments of the statements are described as polytopes automatically obtained by polyhedral analysis of SANLPs [3, 134]. A PPN of a SANLP can be automatically derived from its C code using, e.g., the COMPAAN compiler. A simple two-node PPN is illustrated in

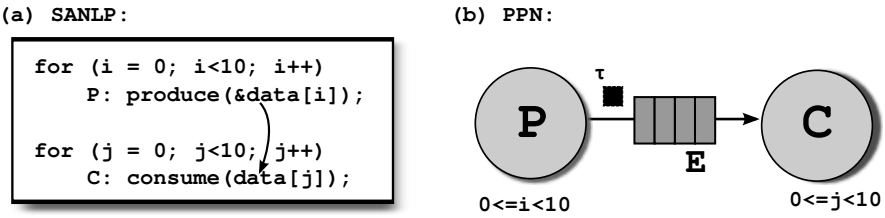

Figure 1.2: SANLP containing two loop nests surrounding statements P and C, and the corresponding PPN model.

Figure 1.2. The produce function writes an element of `data` array, and the consume function reads an element of `data` array. The PPN containing two nodes is depicted in Figure 1.2(b). Nodes P and C represent a *producer-consumer* (P/C) pair of tasks. The tasks are automatically derived from the SANLP shown in Figure 1.2(a). The nodes P and C sequentially execute iterations of their loops in the program source code, i.e. node P executes iterations of the i-loope, and node C executes iterations of the j-loop. The dependence relationship between a producer iteration and a consumer iteration is known exactly, i.e. we know on which iterations it exists. The nodes communicate over the channel E by sending and receiving data packets called tokens. After the node P fires, i.e. executes statement `produce`, the token τ_k is written into channel E. After the consumer node C fires and reads token τ_k from channel E, it executes the next iteration of the statement `consume`. If no data exists on E, process C blocks. The PPN model briefly described above presents the basis for our research, as it provides a parallel, polyhedral specification of the application as a starting point.

LERC contributed to the TSAR project with several technical reports, software prototypes and international peer-reviewed publications [15, 17–20]. The close collaboration between LERC, Compaan Design and ACE Associated Compiler Experts on the TSAR project shaped the research questions and set the framework for the research and development done during the course of this thesis. In the next section, we give further details on the research challenge and the questions posed in this thesis.

1.2 Problem Statement

One of the major challenges in the domain of parallel programming is how to automatically parallelize sequential streaming applications and map them onto heterogeneous platforms, such as the platform illustrated in Figure 1.3 that features architecturally diverse components (such as CPUs, GPUs, FPGAs). Heterogeneous platforms with accelerators provide numerous parallelization opportunities but require different programming models and APIs.

To address the parallelization challenge for mapping streaming applications onto heterogeneous platforms we build upon the work done at LERC and Compaan on the parallelization of C code [81, 131]. The tools developed in the scope of the prior work give us a parallel specification of the application as the starting point. More specifically, applying the COMPAAN compiler to a sequential streaming application leads to a parallel model of the input application in the form of a PPN specification.

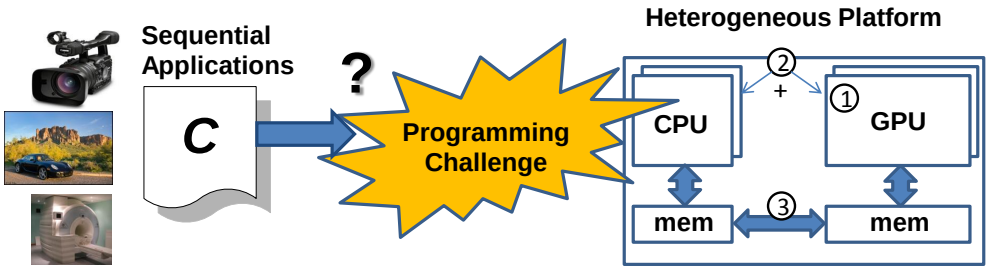

Figure 1.3: The Programming Challenge.

This parallel model can be instantiated, for example, as task-parallel C/C++ code. Compaan generates tasks that are mapped on threads, e.g. implemented using POSIX Pthreads or Intel's Thread Building Blocks (TBB) libraries, for parallel execution on one or more microprocessors. As indicated in Section 1, GPUs became the off-the-shelf hardware of choice for accelerating computationally intensive data parallel workloads on heterogeneous platforms. The natural next question that arises in the context of PPN-based parallelization is how to make use of the tremendous computational power of data parallel accelerators such as GPUs, while still reaping the benefits of task and pipeline parallelism at the platform level. In this context, we formulate our main research question as follows:

How to parallelize sequential streaming applications and efficiently map them onto heterogeneous platforms with massively parallel computing accelerators (such as a GPUs) using a model−based approach?

To address this question we broke it down into the following three sub-questions:

- **How to generate data parallel kernels for execution on massively parallel computing accelerators, such as GPUs, from the PPN model.** To exploit the computational power of massively parallel accelerators, such as GPUs, the data parallelism in the application must be made explicit and brought into a form that is compliant with the accelerator's programming model. Although the PPN model provides a parallel specification, it primarily captures task parallelism. The question is how to identify data parallel operations in the PPN, and transform the components of the PPN model into the form of data parallel kernels that can be executed on a massively parallel accelerator, such as GPU.

- **How to derive multi-level parallel programs featuring task, pipeline, and data parallelism from the standard polyhedral specification.** A heterogeneous platform contains parallelism at many levels - from platform-level task parallelism between different cores to data parallelism within a GPU accelerator and vector processing units. Parallelizing compilers today enable us to target parallelism within a single platform component, such as a single CPU, a GPU, or an FPGA. To efficiently exploit heterogeneous platforms it is necessary to exploit parallelism within different platform components. The question is how to derive at compile-time structured, multi-level programs in which each module can be transformed into a well-suited parallelism form for the given target architecture.

- **How to efficiently solve the problem of host-accelerator communication overhead.** As numerous experiments in the literature have shown (see e.g., [66]), computational acceleration of kernels is only one side of the performance coin. For streaming applications, the actual performance is often dominated by the time to transfer the input data to the accelerator and transfer the results back. The data transfer time can easily outweigh the benefits of the GPU acceleration. The question that we want to address is how to reduce host-accelerator communication overhead by overlapping data transfers and computation on host and its accelerator in a model-based manner.

The solutions to these challenges would enable us to extend the range of parallelization options in Compaan's heterogeneous compilation toolflow to include the increasingly popular data parallel accelerators (such as GPUs), and would also open the door towards easier, model-based experimentation with multi-level parallelism and autotuning, leading towards more efficient parallelization of streaming applications and their mapping onto heterogeneous platforms.

1.3 Approach and Contributions

To address the challenges presented in Section 1.2, we present a novel compile-time approach for the transformation of sequential streaming applications into multi-level parallel programs that can exploit task, data and pipeline parallelism on heterogeneous platforms with GPUs. Figure 1.4 illustrates the contributions of this thesis

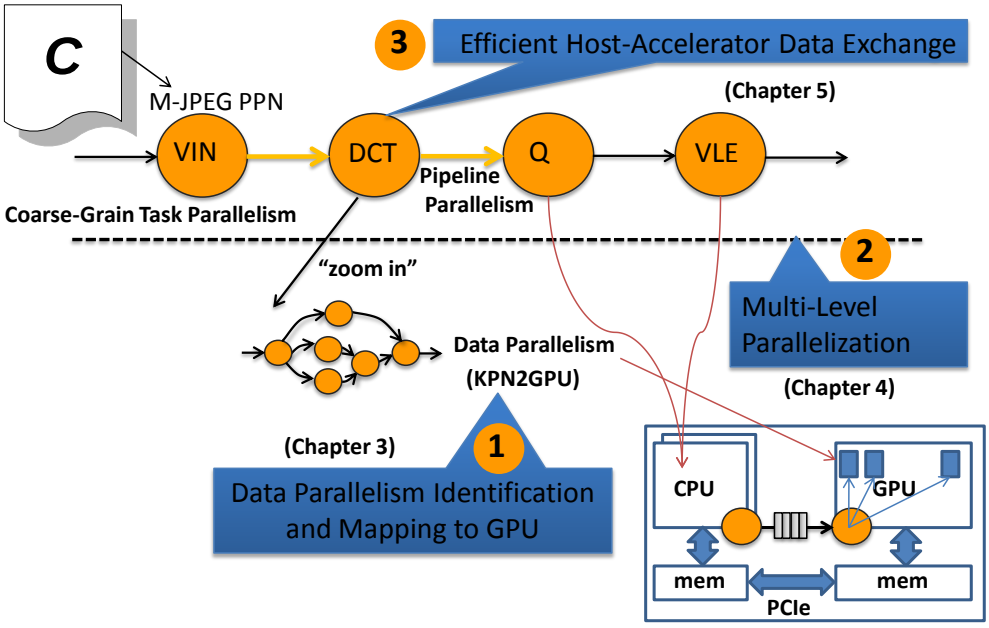

Figure 1.4: Parallelization and Mapping of Streaming Applications onto Heterogeneous Platform: A Compiler-assisted Approach for Generation of Multi-Level Programs, GPU Acceleration and Efficient Host-Accelerator Data Exchange.

on the example of multi-level parallelization and mapping of a sample streaming application (the M-JPEG encoder) onto a heterogeneous platform featuring a quadcore CPU and a GPU accelerator.

- **Contribution I [15, 18, 19]**: We provide a novel method for generation of data and task parallel kernels for massively parallel computing accelerators. We propose a compilation flow that consists of identifying data parallelism in the PPN specification, capturing the data parallelism in an intermediate model called Data Parallel View (DPV) (see Section 3.5), and generation of task and data parallel code from the DPV model. To validate our approach, we developed the KPN2GPU compiler targeting massively parallel GPU accelerators that transforms the PPN specification into CUDA host and kernel code using

the proposed approach. In addition, we leverage the task-parallel nature of the PPN specification to exploit task parallelism on the second generation Fermi-architecture GPUs.

- **Contribution II [16, 21]**: We propose novel transformations and concepts for capturing the notions of program structure and hierarchy in the polyhedral model. We first introduce support for a hierarchical intermediate representation in the polyhedral model, which we call *Hierarchical Polyhedral Reduced Dependence Graph* (HiPRDG) (see Section 4.4). Second, we present the concept of the *slicing* (see Section 4.5) for transformation of the standard polyhedral model of an application into its HiPRDG. The slicing allows the designer to select the desired granularity of tokens that are communicated between program modules and have consistent code and data structures generated at compile-time. Once a HiPRDG is obtained, we present a method for automatic derivation of a multi-level program (see Section 4.6). Each node of a HiPRDG is transformed into an independent program module, making it possible to derive a multi-level parallel program using state of the art polyhedral techniques and tools.

- **Contribution III [17, 20]**: We also propose a novel *stream buffer* design (see Section 5.4.1) to improve the efficiency of the communication between host and accelerator(s). Leveraging the stream buffer design, we introduce support for *asynchronous kernel offloading* (see Section 5.4) in a PPN, and provide a model-based approach for data-driven execution with overlapping of communication and computation on host and accelerator(s).

All these novel methods and techniques contribute significantly to the efficient parallelization and mapping of streaming applications using the PPN model onto heterogeneous platforms. Our approach enables model-based generation of task and data parallel kernels for accelerators and provides improved host-accelerator communication, ultimately leading to improved performance. Moreover, it also extends the polyhedral model in such a way that it makes possible to derive multi-level programs with desired type and granularity of parallelism at each level.

1.4 Related Work

In Section 1, we presented the challenges of parallel computing for heterogeneous platforms. We classified the parallelization approaches according to the degree of automation in three main categories. In Figure 1.5 we illustrate the three categories in the landscape of parallel computing. We will first position our research work in the

landscape of parallel computing and then address related work with respect to each of three main challenges addressed in this thesis.

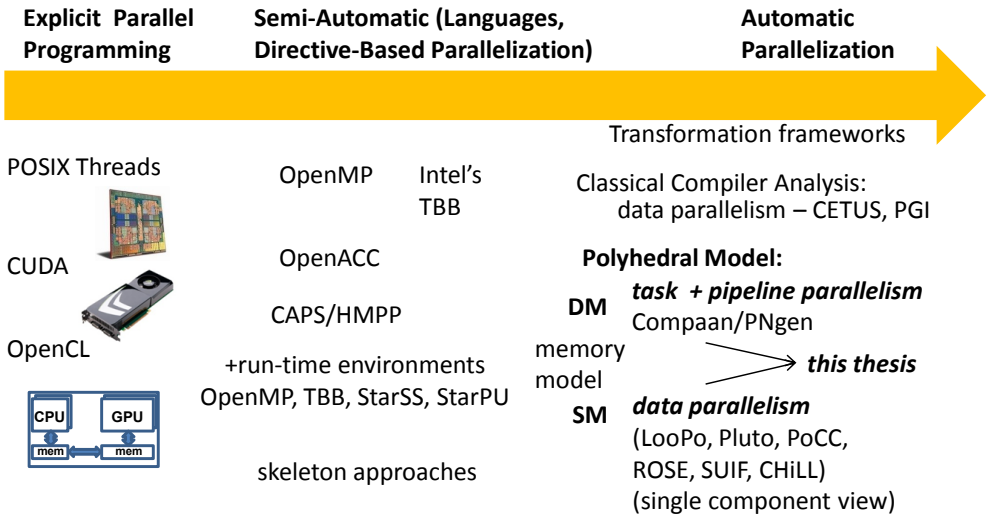

Figure 1.5: The Landscape of Parallel Programming

In the field of explicit parallel programming, the prominent examples include multi-threaded programming using POSIX PThreads, Win32MT, and Boost libraries for programming multicore CPUs, and CUDA for programming GPUs. The CUDA programming model has been generalized by industry into the OpenCL standard targeting portable programming of heterogeneous platforms with accelerators. OpenCL is a promising standard for future work since it introduces supports for programming microprocessors, graphics processing units, and other future accelerators in a portable manner. We leverage the parallel programming APIs and libraries to automatically generate the compilable source code for the target platform.

In the field of the semi-automatic parallelization, the emphasis is on directive-based parallelization. The most widely-adopted semi-automatic approach for parallel programming is the Open Multiprocessing (OpenMP) programming. OpenMP is a shared-memory, parallel programming approach for C, C++ and Fortran, which enables incremental directive-based parallelization. OpenMP is a collection of compiler directives (pragmas), runtime libraries and compiler extensions. The OpenMP API specifies a set of parallel constructs which are used as annotations in the form of compiler directives (pragmas) to guide the compiler which instantiates parallel threads. Support for accelerators, such as a mechanism to describe regions of code where data and/or computation should be moved on a wide variety of compute-capable devices, is planned to be introduced in the next OpenMP standard. Inspired by OpenMP, a novel directive-based standard for acceleration on GPUs called OpenACC was re-

cently introduced by Cray and NVIDIA. The OpenACC API uses directives and compiler analysis to compile regular C and Fortran for the GPU. The OpenACC standard is introduced not only to make GPU programming easier, but also to allow the programmer to maintain a single source version. Ignoring the OpenACC directives will compile the program for the CPU. In C++, support for multi-threaded constructs was introduced via Intel Thread Building Blocks (TBB) and Microsoft Parallel Patterns library (PPL). Both Intel's TBB and Microsoft's PPL use C++ templates and run-time threading support. The TBB provides loop parallelization constructs and parallel programming skeletons (templates) as a part of the language syntax, concurrent data structures, locks, and support task based programming, but requires from the programmer to apply them appropriately. There is also an increasing number of environments that combine multiple features of parallel programming, such as for example CAPS' HMPP, StarSs, and CHPS. CAPS' HMPP is a directive-based compiler targeting multi and many-core architectures with accelerators. HMPP enables offloading of functions or regions of code on GPUs and many-core accelerators as well as the transfer of data to and from the target device memory. StarSs (OMPSs, OpenMPT) is a task based programming model that also provides pragmas to annotate tasks in source code, and then performs computation of dataflow dependencies between tasks, and provides a runtime system supporting different platforms [61, 92, 107]. CHPS [73] is a collaborative execution environment that allows to cooperatively execute a single application on a heterogeneous desktop platform with a GPU, and to do so relies on its own task description scheme. Explicit and semi-automatic parallelization approaches can also be combined with run-time task scheduling frameworks, such as StarPU [9, 10]. In addition, there is an increasing number of skeleton based programming approaches that provide template libraries for common parallel programming patterns [41, 51, 87]. However, the use of a semi-automatic approach still requires an experienced parallel programmer aware of different parallel programming patterns, methods, and mapping mechanisms to identify parallelism in the application and provide parallelization directives in the framework-specific format.

Automatic parallelization frameworks analyze the sequential code, convert it into some intermediate representation, and automatically generate parallel code for the given target architecture. Within the field of automatic parallelization frameworks, most work is done using classical compiler analysis with major players including compiler frameworks such as CETUS [12, 85] and PGI [126].

The polyhedral model is emerging as the most advanced internal representation for manipulation and transformation of programs, due to its strong mathematical foundation. Feautrier significantly contributed to the polyhedral model analysis with his work on program representation and static dataflow analysis which models dependence between operations in program as a system of linear inequalities and equalities which can be then solved using integer linear programming solvers, such as PIP

[54, 55, 58]. Once the dependences are known, it is possible to apply various transformations, such as scheduling [42,56,57,67,86]. Recent advances in the polyhedral model include tiling and fusion [32], vectorization [129], parametric tiling [23], and iterative optimizations [110, 111]. The breakthrough in code generation in the polyhedral model by the CLooG tool [28], made the polyhedral model more applicable to real world problems. The polyhedral model is gradually being adopted by leading edge research and commercial compiler tool-flows. This highly active area of research resulted in several polyhedral frameworks, such as Stanford University's SUIF [136], University of Passau's LooPo [68, 86] University of Ohio's PLuTo [33], joint open source effort PoCC [1], University of Utah's CHiLL [36, 70], gcc GRAPHITE [108], and the commercial R-stream compiler [117], to name a few. We classify polyhedral frameworks in two main categories based on the memory model used for communication into shared memory (SM) and distributed memory (DM) frameworks. While most of the compiler frameworks presented above belong to the first category, the pn and COMPAAN compilers based on the long line of research on dataflow and process network models of computation [46, 77, 79, 81, 114, 116, 120, 131] belong to the second category. These compilers assume a distributed memory model in which each autonomous process communicates with other processes exclusively via tokens. The research work presented in this thesis is highly influenced by the work done on the Compaan [81] and the DAEDALUS frameworks [2, 97, 98, 128], the work of Rijpkema on deriving process networks from nested loop algorithms [46, 114, 115], the work of Stefanov on code transformations like skewing and unfolding [119, 120], the work of Meijer on node splitting for asynchronous data parallelism [94], the work of Turjan on deriving and characterizing process networks [131], the work of Zissulescu on Read/Execute/Write code generation format [145], and the modelling work and initial GPU experiments done by Nikolov [99]. The research in this dissertation makes a step towards combining the two directions in the parallelizing compiler research based on the polyhedral model as indicated in Figure 1.5. Inspired by the Y-Chart paradigm [80] that promotes matching between the application and the architecture specification, we aim to make it possible to combine tools develop in SM and DM research by generating multi-level programs (MLPs), in which each program module can be parallelized using some of the polyhedral frameworks above, and mapped onto the desired component on the heterogeneous platform. Inspired by the distributed memory model adopted by dataflow approaches, we make independent execution of program modules on diverse architectural components possible using private memory within each module and token-based communication between program modules.

Next, we discuss related work in the context of three main challenges addressed in this thesis:

Compiler-based identification of data parallelism has been an active research area for decades. The parallelization techniques developed already in the 80s and 90s

proposed partitioning (also known as tiling) [74, 75, 112, 123, 125, 137, 138] of iteration domain into tiles that are assigned to different processors for execution. Bondhugula [32, 33] was the first to integrate the tiling transformation in the polyhedral model. This work resulted in the polyhedral framework PLuTo targeting coarse-grain parallelization on chip multicore processors and locality optimization. Further, Baskaran adopted the tiling approach in PLuTo for generation of data parallel CUDA kernels for GPUs [24–26]. Semi-automatic approaches for GPU parallelization are for example, extensions of the CETUS research compiler [12, 85] for automatic conversion of OpenMP-annotated code into GPU code [84], the HMPP codelet approach [47], and OpenACC directive-based GPU parallelization supported by the PGI compiler [126].

Our approach to model-drive code generation for GPUs was inspired by the structured scheduling approach proposed by Feautrier [60]. Instead of specifying the graph of independent tasks manually, we automatically obtain a task-graph structure in form of a PPN automatically from the application SANLP using the Compaan compiler. We then leverage the task-graph structure of the PPN model to generate independent tasks for processing on the GPU or CPU. We obtain data parallel CUDA kernels by applying scheduling transforms on each PPN node separately. We use Feautrier's time-optimal scheduling algorithm to illustrate identification of data parallel operations within the nodes [56]. The result of parallelizing each PPN process is a CUDA kernel featuring maximal data parallelism. Such kernels could be further optimized using CUDA auto-tuning tools, such as e.g., [144]. Our methodology for data parallelism identifcation and intermediate representation is however not tied to a specific scheduling algorithm, which makes it possible to combine the work done in this thesis with advanced scheduling and tiling techniques that are being developed in the compiler community. Recently, NVIDIA introduced support in CUDA for concurrent kernel execution [105] on GPU. As a natural application of the task-parallel PPN model which is used as the basis for accelerator mapping, we also provide support for exploiting task parallelism on accelerators.

When executing parts of a program on a GPU accelerator, the overall performance benefits can be seriously affected by data transfers to/from the GPU. The data transfers are one of the major bottlenecks and can possibly diminish benefits from the GPU acceleration [66]. This makes efficient orchestration of data transfers a highly relevant problem. NVIDIA introduced the concept of *CUDA streams* and asynchronous data transfers to mitigate this issue. In case of a streaming application, data transfers to/from the GPU can potentially be overlapped with the GPU kernels following the code pattern for asynchronous data transfers introduced in [100]. Although powerful, this approach requires a custom-made solution for each application. In line with the proposed approach, task-scheduling frameworks such as StarPU [10] make use of asynchronous data transfers to the GPU to minimize the impact of data transfers. In

the context of PPN mapping on heterogeneous platforms, Nikolov [99] experimented with synchronous kernel offloading via replacement of sequential code within a PPN node with synchronous GPU host code. We advance execution of PPNs on platforms with GPU accelerators, by introducing the asynchronous stream buffer design for more efficient implementation of host-accelerator PPN channels. By combining the dataflow nature of the PPN model with the concepts for asynchronous transfers, we present a model-driven solution for generation of asynchronous code for overlapping computation and communication.

Hierarchical decomposition of the program structure has been extensively studied in the context of dataflow models. The Ptolemy [34] environment enables modelling, prototyping and simulation of heterogeneous systems using object-oriented software technology to model each subsystem and to integrate these subsystems into a whole. Ptolemy II [45] provides support for hierarchically combining a large variety of models of computation [64]. Its modelling language allows hierarchical nesting of the models, leading to a more structured approach to heterogeneity [50, 90, 95]. Autopipe [35] provides an application development environment that allows designer to map streaming applications for execution on architecturally diverse computing platforms. The StreamIt language [127] for parallelization of streaming applications provides programming constructs that allow a designer to construct parallel programs with multiple levels of nested parallelism [65]. The StarSs framework [107] provides an environment for hierarchical task-based programming of heterogeneous platforms. However, when using a polyhedral compiler, multi-level (hierarchical) parallelization still needs to be performed by the designer. To derive a parallel program with two levels of parallelism, the designer first needs to manually restructure the input application and then to re-run the compiler on each of the program modules separately. The result of running a polyhedral compiler on each program module separately is a set of unrelated polyhedral models. Recent work on hierarchical parallelization in the Compaan compiler [81] enables a designer to indicate to the compiler that some functions need to be further analyzed, which results in the compile framework to automatically re-run the compiler toolchain on each of the functions separately. However, the transformations involved in restructuring the program for hierarchical modelling, such as outlining of functions and creation of composite data structures, must be first manually performed by the designer. Our approach for multi-level parallelization aims to eliminate the manual restructuring by performing all transforms directly on the polyhedral model. Moreover, due to the particular way in which we approach the transformation, the resulting hierarchical polyhedral intermediate representation is a graph in which each node is annotated with a fully fledged polyhedral specification. As a result, our approach enables structured derivation of multi-level programs, in which each program module can be independently parallelized to obtain a desired form of parallelism.

1.5 Thesis Outline

The remainder of this thesis is organized as follows:

In Chapter 2, we list the requirements for representing programs in the polyhedral model, give an overview of compiler concepts and techniques used in this thesis, and briefly present the architecture and programming model of compute-capable GPUs.

In Chapter 3, we present a three-step transformation approach for identification and exploitation of data parallelism in PPN representation for mapping onto massively parallel accelerators, such as GPUs. Furthermore, we present several memory-related optimization techniques and show how to exploit task-parallelism on accelerators.

In Chapter 4, we present our novel hierarchical internal representation in the polyhedral model, i.e. Hierarchical Polyhedral Reduced Dependence Graph (HiPRDG) and describe a method for derivation of the HiPRDG representation from the standard application specification in the polyhedral model. Furthermore, we present a novel approach for hybrid generation of structured, multi-level programs featuring multiple forms of parallelism.

In Chapter 5, we present an approach for reduction of host-accelerator overhead which makes use of a novel stream buffer design for model-based overlapping of host-accelerator communication and computation leading to asynchronous data-driven execution of PPNs on heterogeneous platforms with accelerators.

In Chapter 6, to evaluate the concepts and techniques presented in Chapters 3, 4, and Chapter 5, we perform an extensive parallelization case study on an example streaming multimedia application (the M-JPEG encoder). We show the benefits achieved through exploiting data parallelism on a GPU accelerator, token adjustment and multi-level parallelization, and wrap up by discussing the overall performance gains.

Finally, in Chapter 7, we conclude the thesis by presenting the summary of the research work along with concluding remarks on prerequisites for further progress and a proposal of future research directions.