



Universiteit
Leiden
The Netherlands

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

Balevic, A.

Citation

Balevic, A. (2013, June 26). *Exploiting Multi-Level Parallelism in Streaming Applications for Heterogeneous Platforms with GPUs*. *ASCI dissertation series*. Retrieved from <https://hdl.handle.net/1887/21017>

Version: Corrected Publisher's Version

License: [Licence agreement concerning inclusion of doctoral thesis in the Institutional Repository of the University of Leiden](#)

Downloaded from: <https://hdl.handle.net/1887/21017>

Note: To cite this publication please use the final published version (if applicable).

Cover Page



Universiteit Leiden



The handle <http://hdl.handle.net/1887/21017> holds various files of this Leiden University dissertation

Author: Balevic, Ana

Title: Exploiting multi-level parallelism in streaming applications for heterogeneous platforms with GPUs

Issue Date: 2013-06-26

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

Ana Balevic

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

Proefschrift

ter verkrijging van

de graad van Doctor aan de Universiteit Leiden,

op gezag van Rector Magnificus Prof.mr.dr. C.J.J.M. Stolker,

volgens besluit van het College voor Promoties

te verdedigen op woensdag 26 juni 2013

klokke 12:30 uur

door

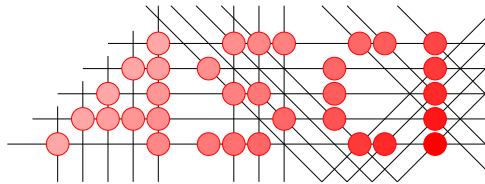
Ana Balevic

geboren te Belgrado, Yugoslavia

in 1980.

Samenstelling promotiecommissie:

promotor	Prof. Dr. Ed F. Deprettere (Universiteit Leiden)
co-promotor	Dr. Bart Kienhuis (Universiteit Leiden)
overige leden:	Prof. Dr. Joost Kok (Universiteit Leiden) Prof. Dr. Harry Wijshoff (Universiteit Leiden) Prof. Dr. Simon Portegies Zwart (Leiden Observatory) Prof. Dr. K. Joost Batenburg (Universiteit Leiden) Dr. Ana Lucia Varbanescu (Universiteit van Amsterdam) Dr. Rob van Nieuwpoort (Netherlands eScience Center)



Advanced School for Computing and Imaging

This work was carried out in the ASCI graduate school.
ASCI dissertation series number 283.

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

Ana Balevic .-

PhD Thesis, Universiteit Leiden. - With index, ref. - With summary in Dutch.

© 2013 by Ana Balevic, The Netherlands.

All rights reserved. No part of the material protected by this copyright notice may be reproduced or utilized in any form or by any means, electronic or mechanical, including photocopying, recording or by any information storage and retrieval system, without permission from the author.

For my mom and dad,
with all my love

Contents

1	Introduction	1
1.1	Research Framework	4
1.2	Problem Statement	6
1.3	Approach and Contributions	8
1.4	Related Work	9
1.5	Thesis Outline	15
2	Preliminaries	17
2.1	Compiler Techniques for Automatic Parallelization	17
2.2	Polyhedral Process Networks	26
2.3	Parallel Computing with GPU Accelerators	31
3	Identification and Exploitation of Data Parallelism in PPNs	37
3.1	Introduction	37
3.2	Problem Statement	38
3.3	Overview of the Solution Approach	40
3.4	Data Parallelism Identification	42
3.4.1	Data Parallelism	42
3.4.2	Space-Time Mapping of PPN Processes	43
3.4.3	Target Domain Characterization	46
3.5	Intermediate Model: A Data Parallel View (DPV)	47
3.5.1	Data Parallel Process	49
3.5.2	Data Parallel Channel	51
3.5.3	Synchronous Data Parallel Execution	53
3.6	Mapping and Code Generation for GPU Accelerators	54
3.6.1	Introduction	54
3.6.2	CUDA Code Generation	55

3.7	Scaling	59
3.7.1	Introduction	59
3.7.2	Tiling for Coarse-Grain Data Parallelism	60
3.7.3	Consequences for GPU Mapping	61
3.8	Extensions and Optimizations	65
3.8.1	Memory Optimizations	65
3.8.2	Task Parallelism	68
3.8.3	Token Composition and Reuse	71
3.9	Results	73
3.10	Conclusions	77
4	Multi-Level Parallelization for Heterogeneous Platforms	79
4.1	Introduction	79
4.2	Solution Approach	83
4.3	Preliminaries	87
4.3.1	Data Space Representation in Polyhedral Model	87
4.3.2	Encapsulation Support: Composite Tokens	89
4.3.3	Introducing Concepts of Depth (Level) and Derived Statements	90
4.4	Hierarchical Polyhedral Reduced Graph (HiPRDG)	94
4.5	The Slicing Transformation	98
4.5.1	Node Splitting	99
4.5.2	Dependence Placement	100
4.6	Construction of a Multi-Level Program (MLP)	100
4.6.1	Preparatory Step	103
4.6.2	Program Module Body Generation	104
4.6.3	Encapsulation/Interface Generation	106
4.6.4	Automatic Type Conversion	107
4.7	Results of MLP Construction	108
4.8	Conclusions and Future Work	110
5	PPN Execution on Heterogeneous Platforms with GPUs	111
5.1	Introduction	111
5.2	Approach	113
5.3	Model-Driven Communication Design	116
5.3.1	Classification of PPN Channels	116
5.3.2	SMC Design	116
5.3.3	Synchronous DMC Design	117
5.4	Asynchronous Offloading of Kernels (AOK)	119
5.4.1	Asynchronous Stream Buffer Design	119
5.4.2	Application in PPN Execution	121

5.5	Experimental Results	123
5.5.1	Platform Specification	124
5.5.2	Platform Micro-Benchmarks	124
5.5.3	Stream Buffer	127
5.5.4	PPN Execution with Asynchronous Kernel Offloading	129
5.6	Conclusions	129
6	M-JPEG Case Study	131
6.1	Introduction	131
6.2	M-JPEG Encoder and its PPN	131
6.3	Experimental Setup	132
6.3.1	Application Configuration	132
6.3.2	Platform	132
6.3.3	Experiments	133
6.4	The Performance of Default M-JPEG PPN	133
6.5	Adjusting Token Granularity by Encapsulation	134
6.6	Leveraging Data Parallelism for GPU Acceleration	136
6.6.1	DCT Kernel Execution	136
6.6.2	Offloading Computation to the GPU	138
6.7	Overall Performance Results with GPU Acceleration	140
6.8	Conclusions	144
7	Conclusion	147
7.1	Summary of Work and Contributions	147
7.2	Prerequisites for Further Progress	150
7.3	Directions for Future Work	150
Appendix A	Application Source Codes	153
A.1	Predictor	153
A.2	Grid	153
A.3	Sobel	154
Appendix B	KPN2GPU	157
B.1	Predictor	157
B.1.1	PPN Model	157
B.1.2	Node P_2 : Space-Time Mapping	157
B.1.3	Node P_2 : DPV Components	158
B.1.4	Predictor: Host Code	161
B.1.5	Node P'_2 : CUDA Kernel (Default)	161
B.1.6	Node P'_2 : CUDA Kernel (Optimized)	163
B.1.7	Scaling	165

Appendix C M-JPEG Encoder	169
C.1 Overview	169
C.2 Code Listings	170
C.3 M-JPEG PPN	172
Bibliography	172
Index	185
Samenvatting	188
Acknowledgments	191
Curriculum Vitae	194

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])

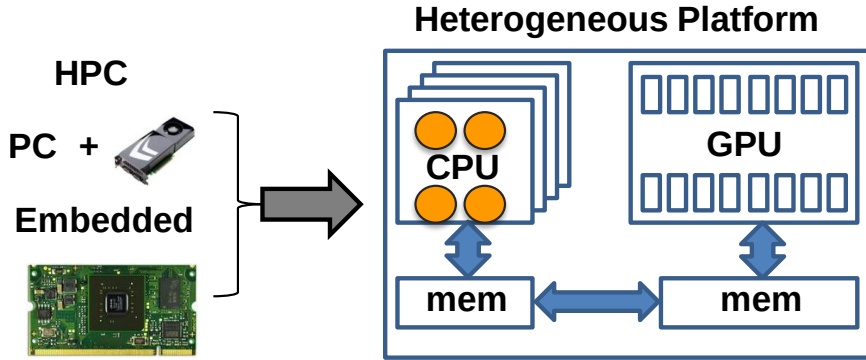


Figure 1.1: A heterogeneous platform featuring a quadcore CPU and a GPU.

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 relieve 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 COMPAAAN compiler. A simple two-node PPN is illustrated in

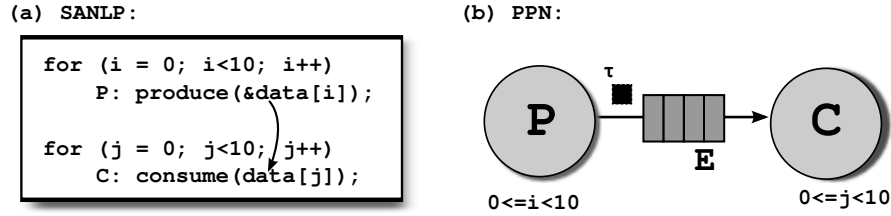


Figure 1.2: SANLP containing two loop nests surrounding statements P and C , and the corresponding PPN model.

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 -loop, 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 auto-tuning, 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

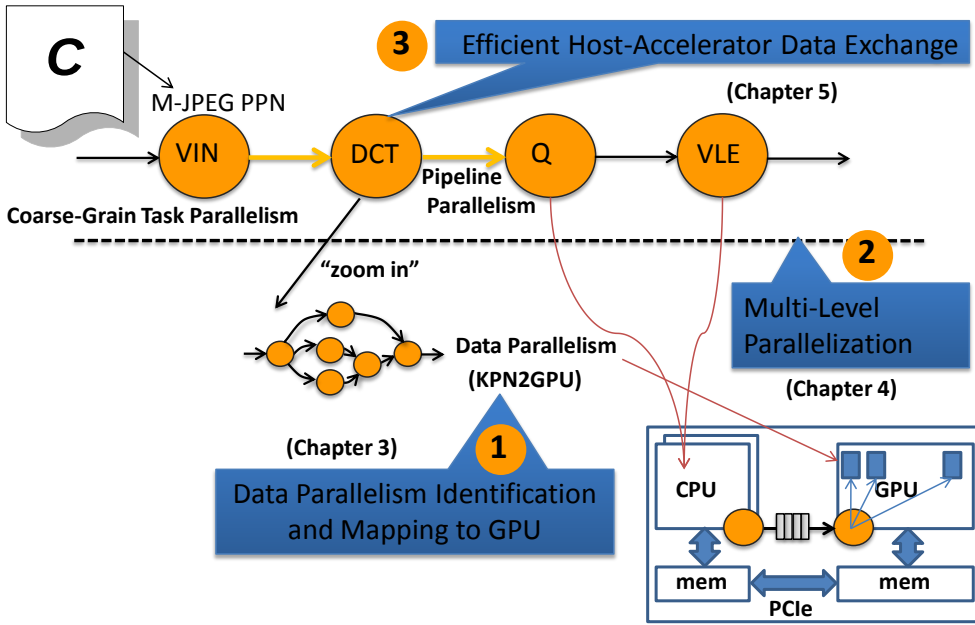


Figure 1.4: Parallelization and Mapping of Streaming Applications onto Heterogeneous Platform: A Compiler-assisted Approach for Generation of Multi-Level Programs, GPU Acceleration and Efficient Host-Accelerator Data Exchange.

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

1.4. RELATED WORK

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

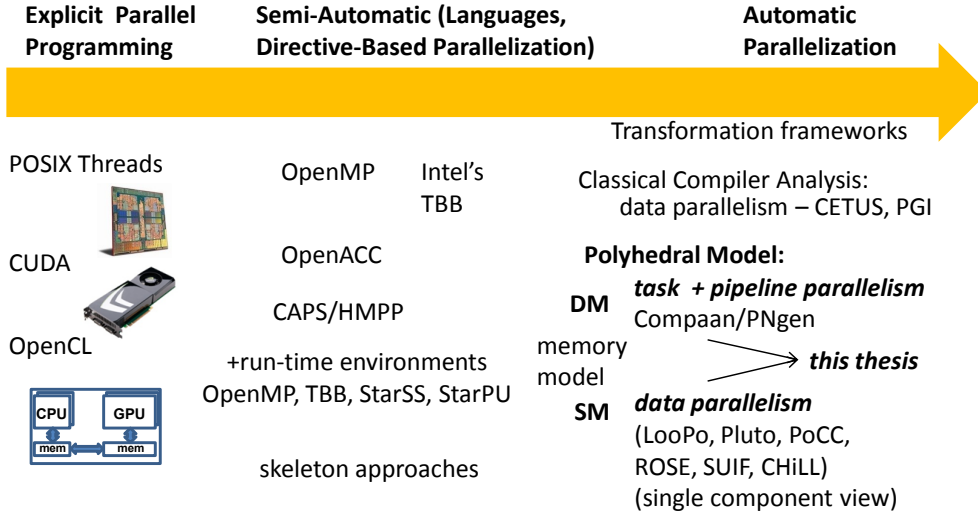


Figure 1.5: The Landscape of Parallel Programming

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

1.4. RELATED WORK

[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 PLoTo [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 P_{Lu}To targeting coarse-grain parallelization on chip multicore processors and locality optimization. Further, Baskaran adopted the tiling approach in P_{Lu}To 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 COMPAN 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 identification 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

1.4. RELATED WORK

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]. Auto-pipe [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.

1.5. THESIS OUTLINE

Preliminaries

2.1 Compiler Techniques for Automatic Parallelization

The polyhedral model is an appealing model to represent and manipulate program statements enclosed in loop nest structures found in static affine nested loop programs (SANLP) [86, 109]. Performance hot-spots in many application domains are naturally expressed in this form of static affine nested loop programs. Numerous examples can be found in multimedia streaming applications in consumer electronics, modeling and simulation applications in high performance computing, molecular biology, radio astronomy, medical imaging, and high energy physics. This thesis considers programs which are (or can be transformed) into the form of a SANLP, and can thus be represented in the polyhedral model. Once the polyhedral model is extracted from a SANLP, data dependence analysis and different loop restructuring transformations such as, e.g., loop fusion, loop fission, and strip-mining can be applied. Before we introduce mathematical concepts and notation required for compile-time program analysis in the polyhedral model, let us define which requirements a program needs to satisfy to be in the form of a SANLP. The definitions in this section have been compiled from the compiler literature [3, 4, 42, 86, 93, 114, 130, 134, 142].

Definition 1 (Static affine nested loop program (SANLP))

A static affine nested loop program (SANLP) is a program in which each program statement is enclosed by one or more loops and if-statements, and where the following conditions hold:

- *for-loop bounds are affine expressions of the enclosing loop iterators, static program parameters, and constants,*
- *for-loop iterators have a constant step size,*

2.1. COMPILER TECHNIQUES FOR AUTOMATIC PARALLELIZATION

- *if-statements have affine conditions in terms of the enclosing loop iterators, constants, and static program parameters,*
- *index expressions of array references are affine expressions of the enclosing loop iterators, constants, and static program parameters,*
- *data flow between statements is explicitly defined.*

An example of a static affine nested-loop program is given in Listing 2.1.

```
1      #define M 4
2      #define N 3
3      int i, j;
4
5      for (int i = 1; i <= M; i++) {
6          for (int j = i; j <= N; j++) {
7              f(in[i][j], &tmp[i][j]); // statement S
8              g(tmp[i][j], &out[i][j]); // statement T
9          }
10     }
11 }
```

Listing 2.1: Example of a regular C program (SANLP).

A loop in an imperative language such as C can be represented using an n -entry column vector called its iteration vector:

$$\vec{x} = [x_1, x_2, \dots, x_n]^T$$

where x_k denotes the k -th loop index and n denotes the innermost loop. The surrounding loops and conditionals of a statement define its iteration domain. The statement is executed once for each element of the iteration domain. When loop bounds and conditionals depend only on surrounding loop iterators, static program parameters and constants, the iteration domain can be specified by a set of linear inequalities defining a polyhedron.

The static program parameters M and N in the program above are defined at lines 1-2. Static program parameters can be also used in expressions that define for-loop bounds and array indices. For example, the upper bound of the for-loop with iterator i at line 5 is given by the program parameter M and the lower bound is given by the constant 1. Thus, the domain of for-loop iterator i can be described by linear inequality $1 \leq i \leq M$. At lines 5-6, there is a doubly-nested for-loop surrounding the program *statement S* at line 7. A program statement can take different forms; a program statement could be for example an assign statement, if condition, or a function call. The statement *S* at line 7 corresponds to the function call *f* with input

2.1. COMPILER TECHNIQUES FOR AUTOMATIC PARALLELIZATION

argument `in[i][j]` and output argument `tmp[i][j]`, and the statement T at line 8 corresponds to the function call `g` with input argument `tmp[i][j]` and output argument `out[i][j]`. The indexes of function arguments `in`, `tmp`, and `out` are affine forms of loop indices i and j . Data flow between statements S and T is made explicit through array variable `tmp`. Below, we define several concepts required for program modeling and compile-time analysis:

Definition 2 (Iteration vector)

The **iteration vector** of a statement is a vector consisting of values of all indices of the loops surrounding the statement, from the outermost to the innermost. With $\vec{x}_S = [x_1, x_2, \dots, x_m]^T$, we denote the iteration vector $\vec{x}_S$ of a statement S surrounded by m nested loops.

Definition 3 (Program parameters vector)

The **program parameters vector** is a vector of static program parameters.

While there is an iteration vector for each statement, there is only a single program parameters vector per program. Together, the iteration vector and the vector of program parameters form the *index vector* of a statement. For simplicity, we often use the terms index vector and iteration vector interchangeably.

Definition 4 (Iteration domain)

The **iteration domain** (*domain*) of a statement is a set of all values of its iteration vector $\vec{x}_S$.

In the polyhedral model, the iteration domain of a statement S is represented by a polyhedron $\mathcal{D}_S$.

Definition 5 (Polyhedron)

A **rational polyhedron** $\mathcal{P}$ is a subspace of $\mathbb{Q}^d$ bounded by a finite number of linear inequalities (affine hyperplanes) i.e.,

$$\mathcal{P} = \{\mathbf{x} \in \mathbb{Q}^d \mid \mathbf{A}\mathbf{x} \geq \mathbf{b}\} \quad (2.1)$$

where $\mathbf{A}$ is an integral $m \times d$ matrix and $\mathbf{b}$ is an integral vector of size m .

A $\mathbb{Z}$ -polyhedron is the set of integer points $\mathcal{P} \cap \mathbb{Z}^d$ (a lattice), i.e. the intersection of the rational polyhedron $\mathcal{P} \subseteq \mathbb{Q}^d$ with a lattice. A $\mathbb{Z}$ -polyhedron is also referred to as integer polyhedron. A **polytope** is a bounded polyhedron.

In this thesis, we use both the terms *polyhedron* and *polytope* interchangeably to refer to bounded, parametric $\mathbb{Z}$ -polyhedra. Note that polyhedra can be parametrized in program parameters (Definition 2 in [134]).

2.1. COMPILER TECHNIQUES FOR AUTOMATIC PARALLELIZATION

Let us illustrate the notation introduced above on the simple SANLP code snippet given in Listing 2.1. Figure 2.1 shows the polytope $\mathcal{D}_S$ associated with statement S nested within for-loops at lines 5-7.

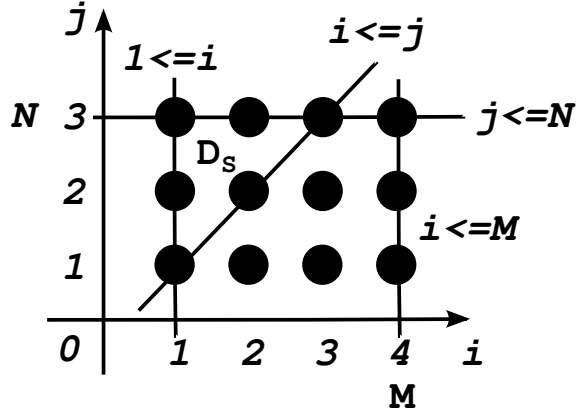


Figure 2.1: Polyhedron $\mathcal{D}_S$ representing the iteration domain of statement S in Listing 2.1 is the closed space bounded by hyperplanes defined by four linear inequalities $1 \leq i$, $i \leq j$, $i \leq M$, and $j \leq N$.

The polytope $\mathcal{D}_S$ lies in a two-dimensional space (i, j) . Each dimension corresponds to the iterators of for-loops within which statement S is nested, i.e. $x_1 = i$ and $x_2 = j$. The iteration domain $\mathcal{D}_S$ is defined by a system of affine inequalities that are derived from the upper and lower bounds of for-loops and if-statements surrounding the statement S :

$$i \geq 0 \quad (2.2)$$

$$i \leq M - 1 \quad (2.3)$$

$$j \geq i \quad (2.4)$$

$$j \leq N - 1. \quad (2.5)$$

The system of (affine) inequalities represented in a homogeneous matrix notation is given below:

$$\begin{bmatrix} 1 & 0 & 0 & 0 & 0 \\ -1 & 0 & 1 & 0 & -1 \\ 0 & 1 & 0 & 0 & 0 \\ 0 & -1 & 0 & 1 & -1 \end{bmatrix} \begin{bmatrix} i \\ j \\ M \\ N \\ 1 \end{bmatrix} \geq \mathbf{0} \quad (2.6)$$

2.1. COMPILER TECHNIQUES FOR AUTOMATIC PARALLELIZATION

This representation is commonly used internally in various polyhedral compiler frameworks, such as [1, 81].

The iteration vector $\vec{x}_S$ takes the values in set $\{(1, 1), (1, 2), (1, 3), (2, 2), (2, 3), (3, 3)\}$. The order in which the iteration vector takes the values in set $\mathcal{D}_S$ corresponds to the order in which the sequential program is executed.

Below, we introduce concepts related to SANLP execution:

A fundamental concept required to discuss program execution is the notion of an *operation*. As an operation we consider a single invocation of a program statement, i.e. the operation is a statement instance. For example, the for loop with iterator i in Listing 2.2 executes three times statement S and three times statement T , resulting in 6 operations: $S(0)$, $S(1)$, $S(2)$, $T(0)$, $T(1)$, and $T(2)$.

```

1
2      for (int i = 0; i < 3; i++) {
3          S: ....
4          T: ....
5      }
```

Listing 2.2: There is an operation for each invocation of a statement in the program code. The for loop executes 6 operations: $S(0)$, $S(1)$, $S(2)$, $T(0)$, $T(1)$, and $T(2)$.

Definition 6 (Operation)

A statement S is executed for each value of the iteration vector $\vec{x}_S$ defined by the for-loops surrounding S . Each execution instance of a statement S , is called an *operation*. An operation is uniquely identified by statement S and the value of iteration vector $\vec{x}_S$, and denoted as $S(\vec{x}_S)$.

A notation also commonly found in literature for an operation of statement S is also $(S, \vec{x}_S)$. We will use the two notations interchangeably.

The operations are carried out in a predefined order, which is called the sequential order. We denote the sequential order as $<_{seq}$. The sequential execution order is always assumed to stem from the lexicographical order of the program source code.

Definition 7 (Lexicographical order)

We say that $\vec{a}$ is lexicographically smaller than $\vec{b}$, i.e. $\vec{a} < \vec{b}$, if it holds $a(i) < b(i)$ for the first position i in which the vectors are different, where $a(i)$ is the i^{th} element of $\vec{a}$ and $b(i)$ is the i^{th} element of $\vec{b}$. Formally, lexicographical order is defined as a disjunction of affine inequalities or equalities. If $\vec{a}$ and $\vec{b}$ are two vectors of size n , then $\vec{a} < \vec{b}$ iff:

$$\vec{a} < \vec{b} \equiv \bigvee_{i=1}^n (a(i) < b(i) \wedge \bigwedge_{j=1}^{i-1} a(j) = b(j)) \quad (2.7)$$

2.1. COMPILER TECHNIQUES FOR AUTOMATIC PARALLELIZATION

In addition, different statements S and T , s.t. $S! = T$ which are enclosed by a common set of loops are executed in the order in which they appear in the text. This order is known as *textual order*, and denoted as $<_{text}$.

The *sequential order* is determined by the lexicographic order defined on the iteration vectors components corresponding to the loops which enclose both operations under consideration. In the case of equality for the lexicographical order, sequential order is determined by the textual order, i.e.:

$$(S, \vec{x}_S) <_{seq} (T, \vec{x}_T) \Leftrightarrow (\vec{x}_S <_{lex} \vec{x}_T) \vee (\vec{x}_S = \vec{x}_T \wedge S <_{text} T)$$

where $\vec{x}_S$ and $\vec{x}_T$ denote the components of iteration vectors belonging to common for-loops, i.e. for-loops which enclose both statements S and T .

When execution order is discussed, the sequential execution order is assumed implicitly. Besides the sequential order, different execution orders of operations are possible. A different execution order of operations can be imposed by specifying a *schedule*, which is typically stored in form of a *scattering matrix*. Code generation tools, such as CLoog [27, 29], generate code from the polyhedral model of the program according to the specification of the original statement domain $\mathcal{D}_S$ and its schedule.

While different execution orders exist, not all possible execution orders are legal. An execution order is legal only if it preserves *data dependences* between operations. According to Bernstein [30], two operations are data dependent if they share some written variable and if at least one access is a write access. A dependence between two operations $S(\vec{x}_S)$ and $T(\vec{x}_T)$, denoted as

$$S(\vec{x}_S) \Rightarrow T(\vec{x}_T)$$

is *loop-independent* if it occurs for a given iteration of all loops that surround both statements S and T , i.e. $\vec{x}_S = \vec{x}_T$, where $\vec{x}_S$ and $\vec{x}_T$ are the vector components up to the common nesting level. Otherwise, a dependence is *loop-carried*, which means that the dependence occurs between operations with different values of the loop counter, i.e. $\vec{x}_S <> \vec{x}_T$.

Definition 8 (Distance vector)

A dependence $S(\vec{x}_S) \Rightarrow T(\vec{x}_T)$ has a **distance vector** $\vec{x}_T - \vec{x}_S$, where $\vec{x}_T$ and $\vec{x}_S$ denote components of the vectors $\vec{x}_T$ and $\vec{x}_S$ up to the common nesting level.

As the dependence is always directed according to the sequential order [42], the dependence vector is always lexicographically non-negative, i.e. $\vec{x}_T - \vec{x}_S \geq_{lex} \vec{0}$.

We classify data dependences in four types. The first three types of dependences, i.e. *dataflow dependences*, *output dependences*, and *anti-dependences* are dependences

2.1. COMPILER TECHNIQUES FOR AUTOMATIC PARALLELIZATION

based on a write to a shared variable. These three dependence types are called *write dependences*. A read dependence is not truly a data dependence, however read-after-read accesses are significant for reuse analysis and data locality optimization.

- **Dataflow dependence: Read-after-write (RAW).** If a write operation to a shared variable is followed by a read operation from the same location, the value read depends on the value written. This type of dependence is also known as a *true* dependence, and it is the main dependence type considered in PPN construction.
- **Output dependence: Write-after-write (WAW).** If two operations write to the same location, the value of memory location will have the wrong value after both operations are performed if the operations are permuted.
- **Anti-dependence: Write-after-read (WAR).** If a read operation from a shared variable is followed by a write operation to the same location, the value read from the memory location will be wrong if the operations are permuted.
- **Read dependence: Read-after-read (RAR).** A read dependence exists if one operation reads the same location as the other.

As explained in [4], anti dependences are a byproduct of using a shared memory model, where the same memory location can be used to write data multiple times. The same holds for output dependencies. Anti-dependences and output dependencies are referred to as *storage dependences*. Storage dependences are not true data dependencies, and they can be eliminated by using different locations for each write.

The COMPAAN compiler used in this thesis considers only dataflow dependences. This is legal, because all storage dependences are eliminated first by conversion of the program into *single assignment code* (SAC) [81]. In SAC, each variable (storage location) can be read many times, but it can be written (assigned) only once. For more information on static single assignment form, interested reader is referred to the seminal paper of Cytron [39]. The SAC representation of a program corresponds to its *dataflow dependence graph*, or simply dataflow graph. This fact is used during PPN construction by the COMPAAN compiler.

In PPN construction, only dataflow dependences are considered, as there is no logical re-use of memory cells for writing. More specifically, PPN dependence edges are annotated with the *exact dependence specification*, which for each dataflow dependence directed to operation $T(\vec{j})$ provides the iteration vector of *the last operation* that writes to a memory location read by $T(\vec{j})$. The exact dependence specification is determined using linear programming techniques for finding lexicographical the maximum [54] as described in [55] and implemented in the PIPLIB library [59]. In

2.1. COMPILER TECHNIQUES FOR AUTOMATIC PARALLELIZATION

addition, read dependences are considered through multiplicity analysis introduced by Turjan [130, 132].

Different abstractions for representing data dependences have been introduced in compiler literature [43]. Some of the widely used dependence abstractions are distance vectors (Definition 8), then level of dependence introduced by Allen and Kennedy [6], and used in their parallelism detection algorithm [5], direction vector introduced by Wolfe [141] and used in Wolf and Lam's parallelism detection and locality optimization algorithm [137, 138], and dependence polyhedra introduced by Irigoin and Triolet [74] and used in their supernode partitioning algorithm [75]. For a detailed overview and comparison of different dependence abstractions and algorithms that use them, interested reader is referred to the book by Darte et al. [42].

Here we give only a brief overview of selected dependence abstractions and related concepts:

Definition 9 (Loop nest depth)

*The **depth of a loop nest** $d_{\mathcal{L}}$ is the number of for-loops in the given loop nest $\mathcal{L}$.*

Definition 10 (Loop nesting level, Loop depth)

*The **nesting level** l_i (**loop depth**) of loop i in loop nest $\mathcal{L}$ is one plus the number of the loops in the surrounding loop nest. The nesting level of the outermost loop thus equals one, i.e.*

$$1 \leq l_i \leq d_{\mathcal{L}}$$

Definition 11 (Common nesting level)

*We define the **common nesting level** $n_{S,T}$ of two statements S and T as the number of for-loops that surround both S and T . Vectors $\vec{x}_S$ and $\vec{x}_T$ are the vectors formed by the first $n_{S,T}$ components of iteration vectors $\vec{x}_S$, and $\vec{x}_T$ respectively. If statements S and T do not share any enclosing loops, then $n_{S,T} = 0$.*

Definition 12 (Level of dependence)

Level of dependence $l(e)$ is the depth (one plus the number of surrounding loops) of the outermost loop for which the loop counters of two dependent operations are different.

This definition of level of dependence follows Allen and Kennedy's seminal paper [6]. The dependence level is always below or at the common nesting level of dependent operations from the statements S and T , i.e. $l(e) \leq n_{S,T}$. Allen and Kennedy allow values between $[1..n_{S,T}] \cup \{\inf\}$ for the dependence level. The dependence level $l(e)$ is determined according to the two cases:

- *loop-independent dependence*: $l(e) = \inf$ if $S(\vec{x}_S) \Rightarrow T(\vec{x}_T)$ with $\vec{x}_S = \vec{x}_T$. This case corresponds to a loop-independent dependence, where statements

2.1. COMPILER TECHNIQUES FOR AUTOMATIC PARALLELIZATION

are nested at the same level $n_{S,T}$ and all vector components equal, i.e. textual order between operations of statements S and T needs to be respected.

- *loop-carried dependence*: $l(e) \in [1..n_{S,T}]$ if $S(\vec{x}_S) \Rightarrow T(\vec{x}_T)$ and the first non-zero component of the vector $\vec{x}_S - \vec{x}_T$ is the $l(e)$ -th component.

This definition of the level of dependence follows Allen and Kennedy's original paper [6], however we will further refine it in Chapter 4 for the needs of our hierarchical parallelization algorithm.

A data dependence is said to be *carried at level k* if dependent operations accessing the same location belong to the same iteration of the first $k - 1$ outermost loops but not to the same iteration of the k -th loop [7]. The level at which dependence is carried can be detected by examining the components of the distance vector. If the first $k - 1$ components of the distance vector are zero and the k -th component is strictly positive, then the dependence is carried at level k . A *dependence polyhedron* $\mathcal{P}_{S,T}$ represents the set of dependence distance vectors (Definition 8) for all dependences between statements S and T .

Dependences between operations define a set of precedence constraints. The precedence constraints can be represented in form of a graph, called the *dependence graph*. Let us define a dependence graph $G = (V, E)$, as a directed multigraph that consists of a set of vertices V (or nodes) and a set of directed edges E (or arcs). In an expanded dependence graph, the nodes are defined as the set of all program operations, i.e.:

$$V = \{S_i(\vec{x}_{S_i}) \mid \forall i, S_i \in \mathcal{P} \wedge \vec{x}_{S_i} \in \mathcal{D}_{S_i} \wedge \bigcup_{S_i} = \mathcal{P}\}$$

where S_i is a program statement with iteration domain $\mathcal{D}_{S_i}$ and $\mathcal{P}$ is the set of all statements in the program. In the dependence graph, there is an edge $e \in E$ for each pair of dependent operations $S(\vec{x}_S) \Rightarrow S(\vec{x}_T)$. A dependence graph with dataflow dependences only is called the dataflow graph. The dataflow graph corresponds to the SAC formulation of the program.

The dependence graph can be subsumed by a more compact representation called *reduced dependence graph (RDG)*. An RDG is a statement-level dependence graph. The set of vertices V of an RDG contains only s nodes, where s is the number of statements in program, with each node corresponding to a different program statement. In this graph, a single edge e between statements S and T represents one or more dependences in the set $R_{S,T}$, where $R_{S,T}$ is the set of all dependence pairs between operations of statements S and T .

In this thesis, we consider the reduced dependence graphs of *SANLPs*. The reduced dependence graph of a SANLP is obtained as the result of exact dependence analysis introduced by Feautrier [55]. We annotate each edge of an RDG using a mapping

2.2. POLYHEDRAL PROCESS NETWORKS

which describes the exact dependence relationships between dependent operations that is obtained as a result of dataflow analysis. The dependence relations are given as affine forms of for-loop iterators, program parameters, and constants.

Definition 13 (Polyhedral Reduced Dependence Graph (PRDG))

A statement-level Polyhedral Reduced Dependence Graph $G = (V, E)$ is a directed multigraph that consists of a set of vertices V (nodes) and a set of directed edges E , where:

- The set of vertices V of a PRDG contains s nodes, where s is the number of statements in program $\mathcal{P}$, i.e. $|\mathcal{P}| = s$. Each node N_{S_i} in the PRDG is defined by the program statement S_i , dimensionality of the statement ds_i (which corresponds to the number of surrounding for-loops), its iteration vector, and its iteration domain $\mathcal{D}_{S_i} \in \mathbb{Z}^{ds_i}$.
- For each set $R_{S,T}$, where $R_{S,T}$ is a non-empty set of dependence pairs between operations of statements S and T , there is a (directed) edge $e_{S,T} \in E$ from node S to node T in the graph.
- Each edge $e_{S,T} \in E$ is annotated with the exact mapping between dependent operations. The mapping between producer (write) and consumer (read) operations is an affine function in iterators, program parameters, and constants.

2.2 Polyhedral Process Networks

A Polyhedral Process Network (PPN) [134] is a variation of the Kahn Process Networks model of computation [76], which describes an application as a network of concurrent autonomous data-driven processes that communicate via channels using a blocking read. Tokens are used as means of communication between processes. The KPN processes pass tokens via unidirectional communication channels. Each communication channel has one writer (producer) and one reader (consumer). We refer to two processes connected by a channel as a producer-consumer (P/C) pair of processes.

In a PPN, program statements, dependence edges, and the input and output arguments of the statements are described as polytopes obtained by polyhedral analysis of static affine nested loop programs (SANLPs) [3, 134]. Extraction of a SANLP's polyhedral model enables exact dataflow analysis of scalar and array references, which is the fundamental requirement for derivation of a Polyhedral Process Network (PPN) model [93]. For the details on analytical derivation of a PPN model, interested reader is referred to [130, 131, 135]. The PPN processes are structured in a particular way,

which is described below. The producer-consumer relationships between processes are also described by polyhedra. In this section, we give fundamental PPN definitions and explain its execution model on a simple example.

During PPN derivation as described in [130, 131, 135], an autonomous process is created for each program statement in the SANLP. The simple SANLP code snippet presented in Figure 1.2(a) (Section 1.1) shows two statements, named P (as in producer), and C (as in consumer), which results in the simple PPN shown in Figure 1.2(b) with two nodes corresponding to these statements.

Each PPN process has a **read-execute-write** (R/E/W) structure, i.e. internally it is structured into three phases:

1. **Read (R)** - in this phase, a process reads input data from incoming channels into local variables. If input data is not available, the process blocks.
2. **Execute (E)** - in this phase, a process executes the process function (program statement) on the input data in local variables, and produces output data.
3. **Write (W)** - in this phase, a process writes the output data from local variable into outgoing channels.

Together, the three phases form the body of a process. The process iteration domain, or simply node domain, specifies a set of iteration points for which the process body is executed. The node domain typically corresponds to the iteration domain of the program statement represented by the given process. For example, the node domain of process P in Figure 1.2(b) is defined by linear inequalities $0 \leq i$ and $i \leq 9$. PPN processes are informally classified into three categories: **producers**, **transformers**, and **consumers**. As the name says, the producer process, e.g. process P in Figure 1.2(b), only produces output data, and as such it has only the execute and write phases. The transformer process reads input data, transforms input into outputs by executing process function, and writes output data. It has all three phases. Finally, the consumer process, such as process C in Figure 1.2(b), just reads in the data and consumes it, thus it has only the read and execute phases.

The definitions that follow are adopted from Meijer [93].

Definition 14 (PPN Process)

A PPN process is an autonomous execution entity specified by:

1. A **process domain** that specifies all process iterations,
2. A specification of **input port domains** to read all the function input arguments from the corresponding input channels,

2.2. POLYHEDRAL PROCESS NETWORKS

3. A **process function** to processes the input arguments and to produce output arguments, and
4. A specification of **output port domains** to write the function output arguments to the corresponding output channels.

The body of the PPN process follows the **read-execute-write** structure.

Definition 15 (Process function)

A **process function** represents the computational part of a process. It corresponds to a function call statement in the sequential application that is a pure function without side-effects which only reads/writes through its input/output arguments.

In the example given in Figure 1.2, the process function of process P is the program function `produce(...)`, while the process function of process C is the program function `consume(...)`.

Definition 16 (Process body)

The body of a process (**process body**) includes communication and computation parts of the process. The process body is structured in three phases: read phase, execute phase, and write phase. First all input data is read from incoming channels (read phase), the process function is executed (execute phase), and subsequently all output data is written to outgoing channels (write phase).

Definition 17 (Process iteration)

A **process iteration** of a process P is defined as a (single) execution of the process body, i.e. the evaluation of the process function on a single set of input and output arguments.

All iterations of a process are described by a process iteration domain. Thus, process P executes 10 times.

Definition 18 (Process iteration domain (Node domain))

The **process iteration domain** (also known as **node domain**) of a process P , denoted by D_P , is defined as a set of all process iterations of process P .

The process iteration domain of a PPN node corresponds to the iteration domain of the matching statement in SANLP.

The PPN processes communicate via single-producer single-consumer channels, typically implemented as FIFO buffers. The PPN model imposes blocking read of input data from the channel, and blocking write of output data to the channel. This means that the process stays in blocked state in the read phase, as long as there is no input data on the incoming channels. As soon as the input arguments of the function

appear on the input ports of the process, the process executes the process function. Then it needs to write the output data from a local variable into the outgoing PPN channel connected to the relevant consumer node. The write can also block if there is no space in the PPN channel.

During PPN construction, a PPN channel is derived for each dependence edge between a pair of nodes. Each PPN channel is associated with a dependence polyhedra that specifies the affine mapping between the iterations of the producer node and the dependent iterations of the consumer node. We refer to a PPN channel that has the same producer node and the same consumer node as a *self-link*. Definitions of PPN properties relevant for realizing communication in polyhedral process networks are given below:

Definition 19 (Input Port Domain (IPD))

The k -th **input port domain (IPD)** of process P_i , denoted by $IPD_{P_i,k}$, is a subset of the node domain where input data is read from the k -th incoming channel, i.e., $IPD_{P_i,k} \subseteq D_{P_i}$.

Similarly, we define an output port domain which represents the process iterations for which output data is written to a given PPN channel.

Definition 20 (Output Port Domain (OPD))

The k -th **output port domain (OPD)** of process P_i , denoted by $OPD_{P_i,k}$, is a subset of the node domain where output data is written to the k -th outgoing channel, i.e., $OPD_{P_i,k} \subseteq D_{P_i}$.

Finally, each channel is annotated with a mapping function that for each pair of producer-consumer (P/C) nodes specifies the relationship between producer iterations and consumer iteration. Using the mapping function, for each input argument of the consumer node it is possible to determine the identity of the node that produced it and at which iteration it was produced.

Definition 21 (Mapping)

An affine **mapping** M^k is a function that specifies producer-consumer relationship between two processes. The mapping M^k maps the iteration points from the k -th input port domain of a consumer process P_C to the corresponding iteration points of its producer process P_P , i.e.,

$$OPD_{P_P,l} = M^k(IPD_{P_C,k}).$$

The affine mapping the specifying producer-consumer relationship is the result of exact dataflow analysis. It can be derived from the dependence polyhedra of a dependence edge between producer and consumer nodes based on the computation of

2.2. POLYHEDRAL PROCESS NETWORKS

the *last write access* [55, 78, 135]. This is relevant because in shared memory model, multiple iterations could write to the same memory location. However, only the last iteration to write before a read occurs is the actual source of the dependence. In the simple P/C PPN example, the IPDs and OPDs are equal to the process domains, but in general this is not the case. The mapping function specifies how consumer's iteration vector $\vec{x}_C = [j]$ maps into producer's iteration vector $\vec{x}_P = [i]$, i.e. given a value of iterator j in producer node, we need to find out what was the value of iterator i in the consumer node for the dependence edge E . In this case, the mapping function is the simply equality $i = j$, which means that input data of, for example the 7th iteration of producer node are provided by the 7th iteration of the consumer node.

The communication between nodes in a PPN is realized via tokens. The specification of token data types is derived directly from the specification of program variables, i.e. there is a one-to-one mapping between data types in the program and token types. In the running example (Figure 1.2(a) in Section 1.1), the tokens correspond to elements of the data array.

2.3 Parallel Computing with GPU Accelerators

Heterogeneous Platforms with GPU Accelerators Over the years, GPUs evolved from hard-wired VGA controllers to programmable parallel processors. On the hardware side, the fixed logic on early graphics cards dedicated to graphics processing was replaced by programmable processors. On the software side, a programming environment was created to allow GPUs to be programmed using the familiar C/C++ programming language with minimal extensions for supporting parallelism. This innovation made a GPU a fully general-purpose, programmable, manycore processor, and enabled the use of what were previously gaming devices for high performance computing. As such, the modern GPUs are frequently used as data parallel accelerators on heterogeneous platforms. Since the CPU and the GPU architecture have a different design point, the best results are achieved when each is used for processing certain types of computation. The unified GPU architecture that evolved in the last decade excels at acceleration of data parallel computations.

Below we give an overview of the wide-spread GPU computing architecture and programming model introduced by NVIDIA, called Compute Unified Device Architecture (CUDA). We compiled this short overview from several resources [38, 88, 96, 105, 106]. The compute device architecture and programming model introduced by CUDA inspired the Open Compute Language (OpenCL) standard for heterogeneous computing with accelerators. OpenCL is an open, royalty-free standard for programming parallel, compute intensive applications on heterogeneous platforms with accelerators. Due to the maturity of tools supporting OpenCL, majority of GPU developers still use CUDA as the GPU API of choice. In this thesis, we use CUDA as a model of choice for GPU, but translation to OpenCL is straightforward.

Architecture The GPU architecture is based on a parallel array of programmable processors [89]. The GPU processor array contains many simple processors, which are called CUDA cores or *streaming processors* (SPs). The SPs are organized into multi-threaded multiprocessors, which are also known as *streaming multiprocessors* (SMPs). Figure 2.2 shows a GPU device composed of 16 SMPs. Each SMP in Figure 2.2 contains 32 SPs, a common instruction unit, on-chip memory (shared memory and register files), and special function units. The number of SPs and their organization into SMPs varies with the architecture version. The first CUDA-programmable GPU, the NVIDIA GeForce 8800 with the Tesla architecture, features 128 SPs cores, structured as 16 SMPs with 8 SPs each. In this thesis, we made experiments using a second-generation GPU Tesla C2050 with the Fermi architecture. The Tesla C2050 GPU contains 448 SPs, structured as 14 SMs with 32 SPs each.

A multiprocessor is designed to execute hundreds of CUDA threads concurrently. To manage such a large amount of threads, it employs the *single-instruction, multiple-*

2.3. PARALLEL COMPUTING WITH GPU ACCELERATORS

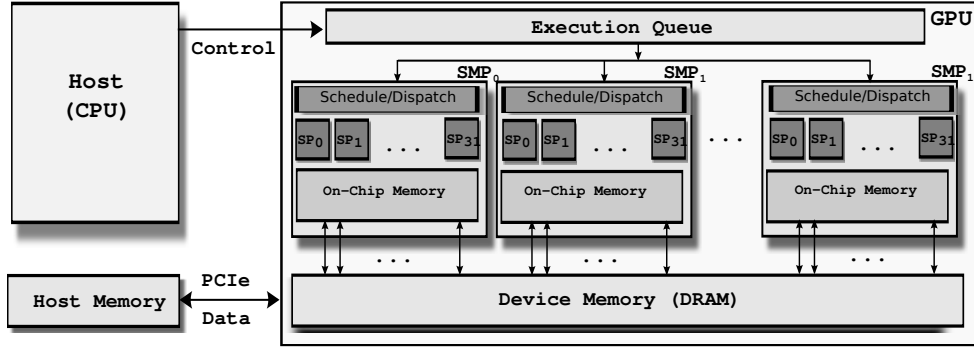


Figure 2.2: High-Level Overview of The GPU Architecture.

thread (SIMT) paradigm introduced by NVIDIA. The SMP creates, manages, schedules, and executes threads in groups of 32 parallel threads called *warps*. The GPU overlaps execution of instructions from warps assigned to the same SMP. A warp of 32 threads executes one common instruction at a time across 32 SPs in the given SMP. Individual threads composing a warp start together at the same program address, but they have their own instruction address counter and register state. Threads within a warp are free to branch and execute independently. However, full efficiency is realized with SIMD-like execution of threads that compose a warp, i.e. when all 32 threads in a warp follow the same execution path.

The GPU memory hierarchy has several layers and it is programmer managed. Physically, there is a *device memory* on the GPU and *on-chip memory* in each of SMPs. Device memory realized as DRAM is the main interface between the GPU and the rest of the heterogeneous platform, as shown in Figure 2.2. Device memory is accessible by all threads executing on the GPU. It is structured into several address spaces: *global memory*, *texture memory*, and *constant memory*, shown within device memory in GPU memory model in Figure 2.3(a). The SMP on-chip memory is partitioned into thread-private memory (*registers*), and so called *shared memory*, which is used for sharing data between threads running on multiprocessor's SPs. To run a program on a GPU, it is necessary to first transfer the input data (if any) from the host memory to the device memory on the GPU via a host-accelerator interconnection link. The host-accelerator link is realized as a PCIe bus. After the GPU has finished the computation, the results are typically transferred back to the host for further processing.

Programming Model The Compute Unified Device Architecture (CUDA) is a programming model and software architecture for the GPU accelerators that allows

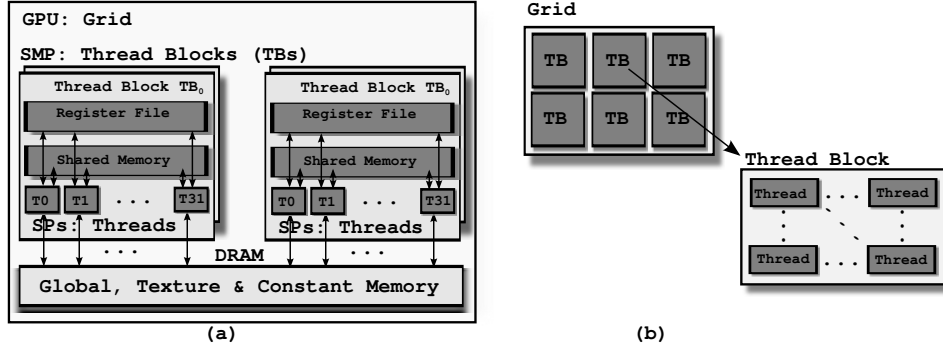


Figure 2.3: GPU Programming Model

programmers to bypass the graphics API and simply program the GPU in C/C++. The GPU programming model is fundamentally different from programming models used for multicore CPUs. The CUDA programming model exposes GPU's parallel processing resources to the programmer as a structured hierarchy of programmable, light-weight CUDA threads, as shown in Figure 2.3(b). There are two main levels of parallelism: coarse-grain data parallelism and fine-grain data parallelism. In addition, starting with the second-generation of GPUs (Fermi architecture), there is a support for task parallelism on the GPUs [105].

The CUDA programming model follows a single-program multiple-data (SPMD) software style. The SPMD is a software style for programming a multiple-instruction multiple-data (MIMD) parallel architecture in Flynn's taxonomy [22, 62]. In SPMD style, a single program is written to run on all processors of a MIMD computer, using conditional statements when different processors should execute different code. A CUDA program (kernel) is written for an independent thread. However, the kernel is instantiated for N threads, and the computation within the CUDA kernel is executed by N threads on the GPU in parallel. Each kernel instance is parametrized by using the thread's unique identifier which is provided by the CUDA environment.

The key programming abstractions that are introduced for mapping onto GPU hardware are the following: (CUDA) threads, shared memories, and barrier synchronization. The abstractions are simply exposed to the programmer as a set of C language extensions. CUDA threads executing a kernel are structured into *grid of thread blocks*, which is shown in Figure 2.3(b). The GPU executes a grid of thread blocks, as indicated in Figure 2.3(a). An SMP executes one or more thread blocks from the grid, as indicated in Figure 2.3(a). Each thread block (TB) (or simply a *block*) consists of multiple CUDA threads. CUDA threads are executed by SPs on the SMP. Threads are organized in one, two or three-dimensional blocks. Threads have private

2.3. PARALLEL COMPUTING WITH GPU ACCELERATORS

memory, which is implemented as (partitioned) register file. Threads within a thread block can access a shared partition of on-chip memory, which is in CUDA terminology called shared memory. All threads can access shared global memory, which is realized as device memory (DRAM). The thread blocks in a grid express coarse-grain data parallelism. Thread blocks must be independent. The concurrent threads in a thread block express fine-grain data parallelism. Threads within a block are not necessarily independent. Independent grids express coarse-grain task parallelism. As a reference, the mapping between CUDA and OpenCL concepts is given in Table 2.3.

CUDA Concepts	OpenCL Concepts
Thread	Work-Item
(Thread) Block	Work-Group
Grid	NDRange
global memory	global memory
shared memory (scratchpad)	local memory (scratchpad)
local memory	-
registers	private memory

Table 2.1: CUDA and OpenCL programming models

Mapping of an application to a GPU requires partitioning of a problem into sub-problems. Sub-problems are assigned to thread blocks, which are required to be executed independently in parallel [37]. Threads within a block communicate with each other through shared memory. The sub-problems can be solved either *independently* or *cooperatively* by all threads in a block. Cooperative execution is realized by using GPU's shared memory for communication and barrier synchronization for coordination between threads.

Execution Model The GPU architecture implements hardware management and scheduling of threads and thread blocks. GPUs have an efficient mechanism for fine-grain hardware multi-threading. The GPU scheduler time-slices execution of threads on an SMP. The scheduling unit is a warp of 32 threads. The execution context (program counters, registers, etc) for each warp processed on an SMP is maintained on-chip during the entire lifetime of the warp. The multiprocessors tracks which warps are ready to execute (i.e. which warps have all operands available for the next instruction) using a mechanism called *scoreboarding*. The warps executed on a single SMP do not have to belong to the same thread block, but they can belong to any thread block scheduled for execution on that particular multiprocessors. The advantage of this approach is that after issuing a long latency instruction, such as memory

2.3. PARALLEL COMPUTING WITH GPU ACCELERATORS

load or store, the GPU does not need to wait idle until that particular instruction is completed. Instead, at every instruction issue time, a warp scheduler selects a warp that has threads ready to execute its next instruction and issues the instruction to those threads. Having a large number of independent instructions in flight is required to hide latencies on the GPU. The common way to achieve this is by finding fine-grain data parallelism in the problem and exposing it to the GPU as large number of light-weight CUDA threads.

Comparison Multicore CPUs and manycore GPUs do not only differ in the number of processing cores, but they also have a different architectural design style [106]. While CPUs rely on multilevel caches to overcome the long memory latencies, GPUs rely on having enough instructions to execute in parallel. Between the time a GPU issues a memory request and the time that data arrives, the GPU may execute instructions from many (thousands) other threads in between. This particular difference also dictates the GPU programming style, which requires the designer to make the parallelism explicit in form of parallel threads. The GPU architectural design is focused around efficient execution of large number of parallel threads on large number of simple, but multi-threaded processors. On a GPU, a larger portion of on-chip transistor budget is devoted to the computation units, and less to on-chip caches and control logic as in the case of a CPU architecture.

2.3. PARALLEL COMPUTING WITH GPU ACCELERATORS

Identification and Exploitation of Data Parallelism in PPNs

3.1 Introduction

Starting with a Polyhedral Process Networks (PPN) [134] specification of a sequential program as produced by the COMPAAN compiler [81], we present in this chapter a method for systematic transformation and mapping of the PPN specification onto a massively data parallel architecture, such as the GPU architecture presented in Section 2.3. Our approach includes techniques for the discovery of data parallelism in the PPN specification, methods for capturing data parallelism in an intermediate model, matching between the intermediate model and the architectural features, and finally code generation for the GPU accelerator. These methods and techniques bridge in a particular way the gap between the Polyhedral Process Network specification and the data parallel nature of processing on modern GPUs. The approach is prototyped in a tool, called KPN2GPU (Kahn Process Network mapper to Graphics Processing Units), which produces fine-grain data parallel kernels targeting the Compute Unified Device Architecture (CUDA) architecture and programming model for GPUs, and host code required to offload accelerated processes onto the GPU. The KPN2GPU tool is designed and implemented as an extension to the COMPAAN compiler [81]. After having presented the basic steps to map a PPN to a GPU, we present several techniques to further improve the mapping of a PPN to GPU (see Section 3.8) including memory related optimizations and model-driven exploitation of task parallelism on second-generation GPUs [105].

3.2 Problem Statement

In the context of the TSAR project, the Polyhedral Process Network (PPN) specification produced by the COMPAAN compiler is used as a basis for the research on the GPU parallelization. The PPN model has been used at LERC as an intermediate model for representation of SANLPs, and automatic generation of pipeline and task parallel programs for a given target architecture (e.g. x86, FPGAs) and programming model (e.g. PThreads, SystemC, etc). The key question is whether the PPN model can be also used for mapping onto data parallel accelerators, and what is required to complete this goal.

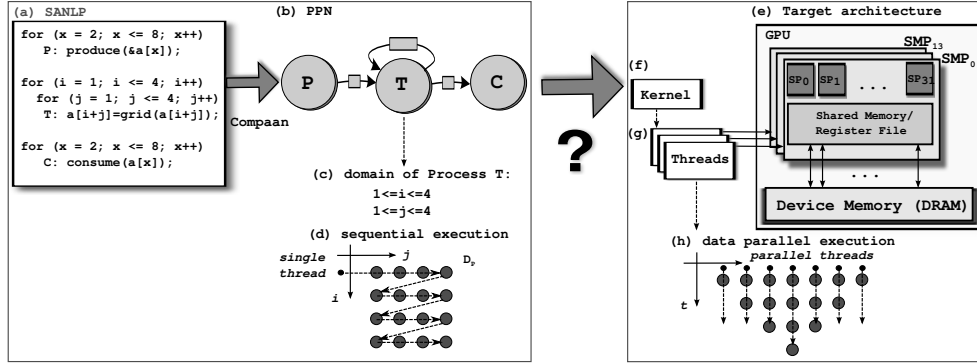


Figure 3.1: Sequential processing of a process network node vs. data parallel processing on GPU.

The GPU architecture is based on a parallel array of simple programmable processors (also known as streaming processors, or CUDA cores) [89]. The streaming processors are organized into multithreaded multiprocessors. A block diagram of the GPU architecture is given in Figure 3.1(e). Each multiprocessor (SMP) contains 32 streaming processors (SPs), a common instruction unit, and shared memory and register files. To port a sequential program for execution on a GPU, it is necessary to express the program as a kernel that follows the single-program multiple-data (SPMD) paradigm [63], which means that each thread runs the same program but on a different data set. The kernel, depicted in Figure 3.1(f), is executed by multiple threads on the GPU (Figure 3.1(g)) in parallel. The hardware executes threads in a data parallel manner, as depicted in Figure 3.1(h). Figure 3.1(h) depicts 7 CUDA threads that go step by step through the node domain and process operations in parallel. In the first time step this batch of threads (thread block) processes 7 data parallel operations, in the second time step 5 data parallel operations, and so on. The kernel code is typically parametrized in thread identifiers. The parametrization of the kernel enables different threads to evaluate different conditions and load/store data from different memory ad-

addresses. The communication on the GPU follows the shared memory programming model, i.e. a location in memory can be read and written by multiple threads. It is the role of the programmer to manage communication, synchronization, and ensure data consistency.

A PPN model of a sequential program expressed in the form of a SANLP can be automatically derived using, for example, tools such as the COMPAAN compiler [81] and DAEDALUS [2]. An example of a PPN that was automatically generated by the COMPAAN compiler from the sequential C program consisting of three nested loops in Figure 3.1(a) is depicted in Figure 3.1(b). The PPN model of the program consists of three processes, namely producer P , transformer T and consumer C , which communicate via channels. The PPN processes execute operations in their node domains sequentially. The node domain of process P is specified as a set of linear inequalities in Figure 3.1(c). Each operation corresponds to the evaluation of the `transform(...)` function in statement T of the SANLP, which has one input and one output argument. The function is evaluated for all iteration points in the node domain of the process P , following the sequential execution order depicted in Figure 3.1(d).

When looking at the sequential iteration domain execution in Figure 3.1(d) and the parallel iteration domain execution in Figure 3.1(h), we can observe a significant difference in the execution style. Although the data dependence analysis shows that Figure 3.1(d) contains plenty of data parallel operations, this code can not be immediately executed on the GPU, because the data parallelism is not exposed in an explicit manner. To be executed on a GPU, the domain in Figure 3.1(d) needs to be transformed in some way to get the data parallel execution style depicted in Figure 3.1(h). To transform this program representation for execution on a data parallel accelerator, such as a GPU, we need to address the following topics:

Computation The PPN model exposes task parallelism in an SANLP. Using the techniques presented in [94, 120], the PPN can be transformed to expose data parallelism in a specific way, i.e., via replication of PPN tasks. This approach maps very well to multicore CPUs and MPSoCs where a small number of coarse-grain tasks is required. However, this form of data parallelism does not scale enough to be mapped onto a GPU. Modern data-parallel architectures, such as GPUs, have hundreds of processing elements and require large amounts of data parallel operations to utilize them. The CUDA programming model virtualizes the GPU architecture and exposes it to the programmer as a hierarchy of light-weight threads. While each PPN node is processed by a single thread that sequentially executes iteration points in the node domain as depicted in Figure 3.1(d), the GPU is designed to execute many threads in parallel as for example in Figure 3.1(h). The key question is how to identify data parallel operations within the PPN and expose them as a Single Program Multiple Data (SPMD) kernel that is executed by a large number of GPU threads in parallel.

3.3. OVERVIEW OF THE SOLUTION APPROACH

Communication The GPU architecture makes use of the shared memory model for communication between threads. In addition, GPUs feature a rich, multi-level shared memory hierarchy that is managed by the programmer. Similar multi-level memory hierarchies can also be found in high performance embedded systems. On the other hand, PPN nodes communicate via channels. The channels are typically implemented as FIFO buffers that do not have an efficient implementation on the GPU. To execute a PPN on such a shared memory architecture, it is necessary to find an architecture-friendly realisation of the communication channels. In addition, to support data parallel execution it is necessary to enable PPN ports and channels to read/write multiple data values in parallel. To achieve higher performance of the solution, the rich memory hierarchy of the GPU architecture should be taken into account.

Synchronization The PPN semantics requires a blocking read primitive, which is not supported on the GPU architecture. Instead, the GPU provides a *barrier* synchronization primitive. In addition, the CUDA programming model offers concepts of *events* and *streams*, which can be used to coordinate launches of *different kernels*. An important constraint of the CUDA synchronization model is that fine-grain synchronization between running kernels is not possible, i.e. each kernel must run to completion. To map a PPN model onto a GPU, we need to determine how to realize the synchronization on a GPU: (a) within a PPN node, and (b) between different PPN nodes.

3.3 Overview of the Solution Approach

To address the parallelization challenges presented in Section 3.2, we propose a novel method for discovery and exploitation of data parallelism in PPNs for mapping onto data parallel accelerators, e.g. GPUs, which is represented as a three-phase workflow shown in Figure 3.2. The first phase of the workflow consists of the identification of data parallelism. The second phase of the workflow consists of the construction of an intermediate model called *Data Parallel View* for capturing the identified data parallelism in an explicit manner on top of the PPN specification. The third phase consists of the mapping of the intermediate model to the accelerator programming model, e.g. CUDA for GPUs and code generation. This three-phase approach is implemented in the tool called KPN2GPU, which was developed during the TSAR project as an extension to the COMPAAAN compiler. The KPN2GPU takes as input the PPN specification generated by the COMPAAAN compiler from the application sequential C source code in the form of a SANLP, and generates host and kernel code for compute-capable GPU accelerators.

3.3. OVERVIEW OF THE SOLUTION APPROACH

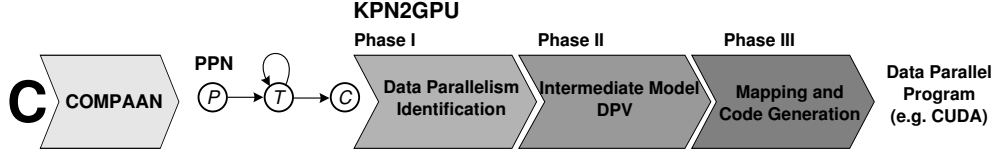


Figure 3.2: KPN2GPU Workflow.

Data Parallelism Identification In the first phase of the KPN2GPU workflow, we analyze the PPN specification for data parallelism within the PPN nodes. For the purpose of data parallelism identification, we use well known dataflow analysis and scheduling techniques. The result of the data parallelism identification phase is a per-node space-time mapping, which for each PPN node specifies its node domain in terms of data parallel operations. The details of the data parallelism identification phases are presented in Section 3.4.

Intermediate Model: Data Parallel View (DPV) In the second phase, we introduce an intermediate model to capture data parallelism within a PPN process, called a Data Parallel View (DPV) on PPN. In DPV, we provide a minimal set of extensions on top of the PPN model used by the COMPAAN compiler that enable data parallel execution. These extensions are per-node space-time mappings and additional control required for synchronous data parallel execution (see Section 3.5.3). The consequence of the space-time mapping is that communication patterns between nodes may be affected as presented in [69]. In the DPV model, we address data parallel communication by introducing data parallel channel (DPCs). The operational semantics of the process network components under DPV is explained on several running examples. Finally, we conclude the section by classification of two data parallel execution types, namely cooperative and independent data parallelism, and present consequences for the execution of processes. We present the intermediate model in Section 3.5.

Mapping the Intermediate Model (DPV) onto a CUDA/GPU In the third phase, we show how to map the DPV model onto the CUDA architecture and programming model for GPUs. We use the DPV components to generate CUDA kernels, communication channels, and host code for accelerator offloading. The details of the mapping and code generation for CUDA are given in Section 3.6.

The result of the three steps above is a structured approach for mapping the PPN specification onto second-generation of GPU accelerators, which enables automatic generation of task and data parallel kernels for accelerators, as well as host code that facilitates kernel offloading to the accelerator.

3.4 Data Parallelism Identification

3.4.1 Data Parallelism

Each execution instance of a statement in a program defines an operation as indicated in Section 2.1 (Definition 6). An operation, denoted as $S(\vec{x}_S)$, is uniquely identified by its statement S and the value of iteration vector $\vec{x}_S$. Two operations can be dependent or independent. Data parallel operations are considered to be *independent* instances of the same statement that process different data.

Definition 22 (Data Parallelism) *Let us define **data parallelism** as the execution of a set of independent operations on different data elements, where all operations are instances of the same program statement.*

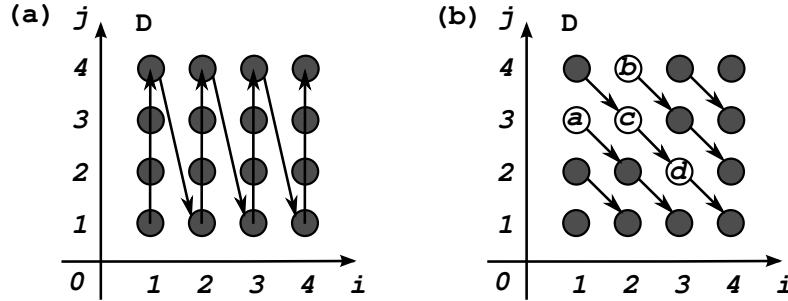


Figure 3.3: Example: Independent and Dependent Operations.

Let us illustrate the concept of data parallel operations on the example in Figure 3.3. Figure 3.3(a) depicts the two-dimensional iteration domain of some statement S with the iteration vector $\vec{x}_S = [i, j]^T$. The arrows indicate the sequential (lexicographic) execution order of operations in $\mathcal{D}$. Each iteration point in the domain corresponds to the execution of an operation $S(\vec{x}_S)$ for a different value of iteration vector $\vec{x}_S$. Figure 3.3(b) illustrates the data dependences between operations. Operations $a = S(1, 3)$ and $b = S(2, 4)$ are independent, since they are not connected with a dependence vector in Figure 3.3(b). Since operations a and b are instances of the same statement S , they represent data parallel operations. The operation $d = S(3, 2)$ is dependent on the operation $c = S(2, 3)$, denoted as $c \Rightarrow d$.

Data dependences between operations are the only source of causal constraints on the execution order. To preserve the semantics, the operation c , which is the source of the data dependence, must execute before operation d , which is the destination of the data dependence.

Let us define a function t that assigns a time step to each iteration point of the iteration domain $\mathcal{D}$ as follows:

Definition 23 (Schedule) A *schedule* is a function t that assigns a time step t_k to each operation x in an iteration domain while preserving data dependencies:

$$\forall x, y \in \mathcal{D} \wedge (x \Rightarrow y) : t(x) < t(y).$$

where $x \Rightarrow y$ denotes data dependence between operations x and y .

The scheduling function assigns a discrete time step to each operation in a domain. The operations of statement S assigned to the same time step t_k form a set of data parallel operations, denoted as S_{t_k} . All operations in set S_{t_k} can execute concurrently provided that there are sufficient parallel processing resources. In the example above, operations a and b belong to the set $S_{t_1} = \{a, b\}$.

3.4.2 Space-Time Mapping of PPN Processes

A schedule is a function that assigns a logical time step to each iteration point (i, j) in a domain while preserving data dependencies. Feautrier's seminal work on scheduling [56, 57] presents an algorithm for finding a minimal latency schedule as a *piece-wise affine function* of iteration vectors, program parameters, and constants. Finding a legal schedule requires solving a linear program. The size of the scheduling problem is proportional to the number of dependences, and scales as the sixth power of the program size [53]. Hence, scheduling is not scalable. In [60], Feautrier acknowledges the problem of the high computational complexity, and outlines a more scalable approach to program scheduling that is based on splitting the program into smaller units (modules), which can be scheduled separately. In this thesis, we apply the modular scheduling approach of Feautrier on the PPN model.

The PPN model is a very convenient model for modular processing, since the distinguishing property of the PPN model is the clear separation of communication and computation. Each PPN process can be treated as an independent module with a well-defined R/E/W structure [145]. The communication between processes is decoupled from the computation. By generating the PPN model of a program and treating each PPN node as a separate module, we obtain a set of smaller scheduling problems. Since each PPN node represents the iteration domain of a single program statement, all operations in the PPN node domain evaluate the same function. As a consequence, scheduling of a node domain reveals the data parallelism within the process.

Numerous algorithms exist that can be used to identify sets of concurrent operations and find their partial order, i.e. a *schedule*. A detailed review and comparison of scheduling algorithms is given in [42]. We use Feautrier's scheduling algorithm [56]

3.4. DATA PARALLELISM IDENTIFICATION

to find the time-optimal schedule for each PPN node. The time-optimal schedule minimizes latency (number of time steps), and maximizes data parallelism.

Given a schedule, it is possible to find an *allocation* function $p : \mathcal{D} \rightarrow \mathbb{Z}$ that assigns each data parallel operation for execution on a different processing element. A schedule of a PPN process and its matching allocation together form a *space-time mapping* T . The space-time mapping is the concept well-known from the systolic array research [82], which was later adopted for the compiler-based parallelization in the polyhedral model [86]. Instead of interpreting values in the space dimension as hardware processors like systolic array community, we will interpret values in the *space dimension* as GPU threads. The *time dimension* specifies the execution order of operations. Following [86], we construct the time-optimal schedule and the allocation for maximal data parallelism, and combine them into a space-time mapping T . The application of a space-time mapping on a domain results in what we call a data parallel *target domain*, or simply a target domain.

Let us illustrate the space-time mapping of three different PPN processes. For this purpose, we selected the PPN processes with data dependence patterns that are representative for a large class of imaging, simulation, and scientific applications:

- Predictor
- Grid
- Parallel2D

We now discuss each pattern in detail.

Predictor The source code of the predictor example is given in Appendix A.1. Let us analyze the transformer node T , which evaluates the function `predict`. The

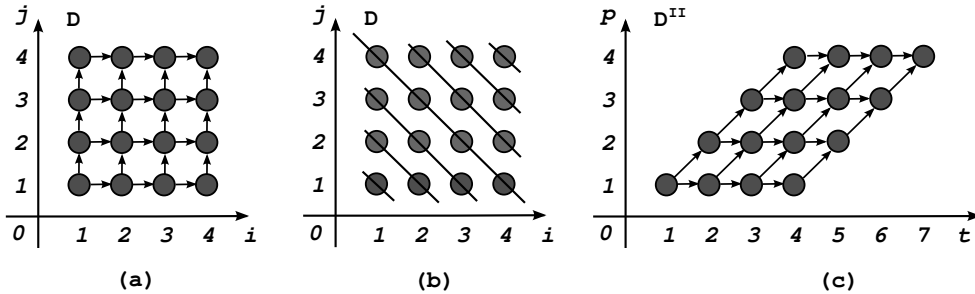


Figure 3.4: Predictor Node T with domain D : (a) Data dependences, (b) Sets of independent operations, (c) Data Parallel Target Domain (D^{II}).

transformer process computes a new pixel value in a 2D-image on the basis of its two

3.4. DATA PARALLELISM IDENTIFICATION

neighbouring pixels. The statement executed by process T is $A[i][j] = \text{pred}(A[i-1][j], A[i][j-1])$. This statement is executed by all operations of the iteration domain shown in Figure 3.4(a). As a result of data flow analysis, the PPN process T contains two self-links representing the two data flow dependences:

- (i) from write access $a[i, j]$ to read access $a[i-1, j]$
- (ii) from write access $a[i, j]$ to read access $a[i, j-1]$

The arrows in Figure 3.4(a) show these data flow dependences between operations. Following the approach described in Section 3.4, we identify sets of data parallel operations, and their associated time stamps. In this case, a similar result can be obtained by skewing the iteration domain [120]. In Figure 3.4(b), we illustrate the operations that can be executed in a data parallel manner by representing them on the same line. At $t = 0$, only operation $(1, 1)$ can be executed, while at $t = 1$, operations $(2, 1)$ and $(1, 2)$ can be executed in parallel. At $t = 2$, operations $(3, 1)$, $(2, 2)$, and $(1, 3)$ can be executed in parallel. Using Feautrier's scheduling algorithm, we obtain an affine 1-dimensional schedule $t(i, j) = i + j - 2$, and a 1-dimensional allocation function: $p(i, j) = j$, which together form a space-time mapping T . Transformation of process P with space-time mapping T results in a data parallel target domain depicted in Figure 3.4(c). All operations with the same value of t can be executed in a data parallel manner.

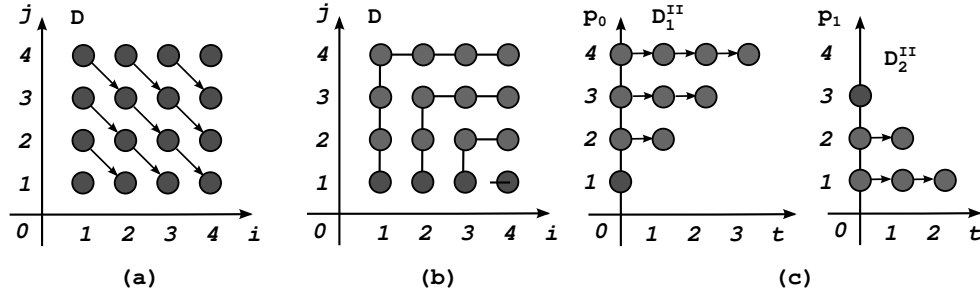


Figure 3.5: Grid node T : (a) Data dependences, (b) Sets of independent operations, (c1) First target domain D_1^{II} , (c2) Second target domain D_2^{II} .

Grid The source code of the grid example is given in Appendix A.2. Let us analyze the transformer node T , which evaluates the function `grid`. The transformer process computes a new pixel value in a 2D-image on the basis of its two neighbouring pixels. The statement executed by process T on the iteration domain in Figure 3.5(a) is $a[i+j] = \text{grid}(a[i+j])$. The PPN process T has a single self-link representing the data flow

3.4. DATA PARALLELISM IDENTIFICATION

dependence from write access in iteration $(i - 1, j + 1)$ to read access in iteration (i, j) . The arrows in Figure 3.5(a) show these data flow dependences between operations. If we look at the direction vector of data dependences, we can see that in Figure 3.5(b), all operations on the same line are independent and can be executed at the same time step in parallel. The schedule obtained using Feautrier's scheduling algorithm, is a piece-wise affine function, which splits the iteration domain according to the condition $i + j \leq 5$:

$$t(i, j) = \begin{cases} i - 1, & \text{if } i + j \leq 5; \\ -j + 4, & \text{otherwise.} \end{cases}$$

The iteration domain of process T is first split into two source domains according to the condition. Each sub-domain is transformed using a different space-time mapping which results in two independent target domains D_1^{II} and D_2^{II} shown in Figure 3.5(c). Each target domain can be processed concurrently as a parallel task.

Parallel2D As a special case, let us consider a process featuring an *absolutely parallel* node domain. An absolutely parallel node domain means that the process has no data dependences. As a consequence, all operations in the domain can execute in parallel. This is a very simple, but significant case as it frequently occurs in image processing applications. A typical example of a PPN node with a *parallel2D* pattern can be found in the `sobel` edge detection algorithm. The source code of the `sobel` example which has the *parallel2D* pattern is given in Appendix A.3. The space-time mapping obtained by Feautrier's scheduling algorithm is a singular matrix, and the space-time mapping can not be inverted. In this case, we perform one-to-one mapping of the source domain into a target domain that has only spatial dimensions. All operations are executed at $t = 0$.

3.4.3 Target Domain Characterization

Section 3.4.2 shows how to identify data parallel operations within PPN processes. Our method for data parallelism identification is based on the construction of a space-time mapping for each PPN node domain individually.

Once we obtain a target domain, we can quantify the amount of data parallelism. We characterize this with two parameters, the maximal parallel *width* W of the target domain, and the *depth* D of the target domain. The width W of a target domain corresponds to the maximal amount of data parallelism within the target domain. The depth D equals the number of sequential time steps. The (W, D) parameters for the *predictor* and the *grid* examples are illustrated in Figure 3.6.

Definition 24 ((W,D) parameters) Given a target domain $\mathcal{D}_p^{\parallel}$ that is the image of the node P 's domain $\mathcal{D}_P$ with index vector $\vec{x}_P$ under a space-time mapping T , which

3.5. INTERMEDIATE MODEL: A DATA PARALLEL VIEW (DPV)

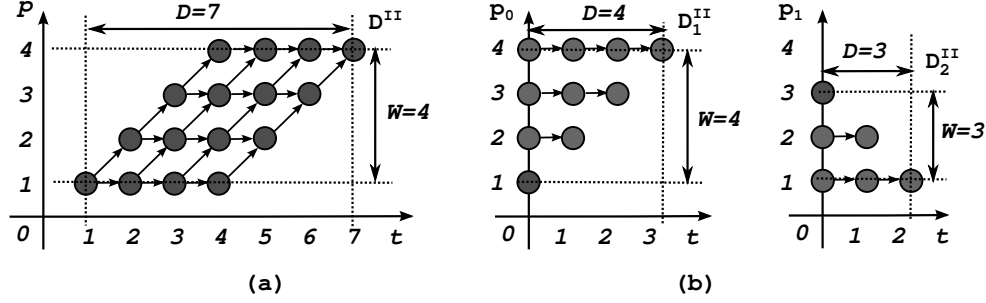


Figure 3.6: (W, D) Parameters of Target Domains: (a) predictor, (b) grid.

is composed of the schedule $t(P, \vec{x}_P)$ and the allocation $p(P, \vec{x}_P)$, we define (W, D) parameters as follows:

$$D(\mathcal{D}_P^{\parallel}) = \text{lexmax}(t(P, \vec{x}_P)) - \text{lexmin}(t(P, \vec{x}_P)) + 1, \quad (3.1)$$

$$W(\mathcal{D}_P^{\parallel}) = \text{lexmax}(p(P, \vec{x}_P)) - \text{lexmin}(p(P, \vec{x}_P)) + 1. \quad (3.2)$$

The lexicographic minimum lexmin and maximum lexmax of parametric polytopes are computed using the *piplib* library [59]. The (W, D) parameters obtained in this way are affine functions in the program parameters and constants.

Once a target domain is characterized with the (W, D) parameters, we use its width W to determine the maximal number of independent processing entities that can process the target domain in a data parallel manner, and its depth D to determine the number of time steps required to process the target domain.

3.5 Intermediate Model: A Data Parallel View (DPV)

In Section 3.4, we presented a method for discovery of data parallelism within PPN processes. However, the PPN model used in the *COMPAA*N compiler does not provide means of capturing data parallel operations. Even if after the data parallelism analysis stage we know that the transformer process in the *predictor* example can safely execute 4 operations of its node domain at each time step in parallel, the PPN process will still execute the operations one by one sequentially. To capture the data parallelism within PPN nodes and exploit it for mapping onto data parallel accelerators, we introduce an intermediate model called the *Data Parallel View (DPV)* on Polyhedral Process Networks. Let us explain how we extend PPN components for data parallelism.

The PPN processes execute process iterations one by one. The operations are executed following the lexicographic order in the source code from which the PPN node

3.5. INTERMEDIATE MODEL: A DATA PARALLEL VIEW (DPV)

is derived. In DPV, we make the execution order explicit by associating a space-time mapping (in form of a scattering function) with each PPN node. As a result, we obtain a target node domain. Some dimensions of the target domain are marked as space dimensions (p) and some dimensions are marked as time dimensions (t), in line with state of the art polyhedral literature [32]. We call such node domain a *parallel node domain* (PND). The execution order in PND is explicit, i.e. the execution follows the time dimension. A transformed process is called a *Data Parallel Process* (DPP). A Data Parallel Process (DPP) can be executed by multiple active processing entities (e.g. threads) in parallel. To guarantee correctness of the results, we introduce a control unit in DPP. The DPP controller is responsible for implementing transitions between time steps and synchronization in the time dimension.

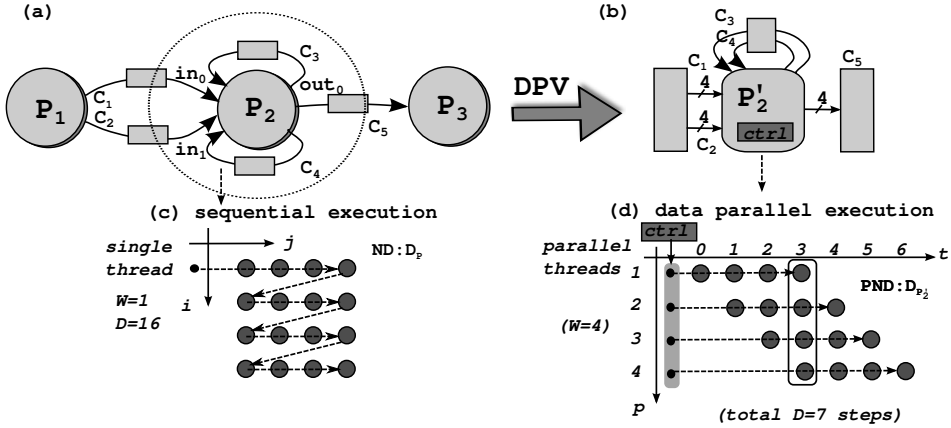


Figure 3.7: (a) Sequential processing of PPN node P_2 with a single processor, (b) data parallel processing of DPP node P'_2 with 4 parallel processors (threads).

Let us illustrate the DPV model on the *predictor* example. Figure 3.7(a) shows the PPN of the *predictor* example, and Figure 3.7(b) shows the transformer node P_2 under the DPV, and its incoming and outgoing channels. Instead of processing the node domain one operation at a time, the DPP process P'_2 at $t = 3$ executes $W = 4$ operations in parallel. By increasing the processing width from $W = 1$ to $W = 4$, we reduce the number of time steps required to process the iteration domain from $D = 16$ to $D = 7$.

In a PPN, the processes read and write single tokens into the channels. To support data parallel execution, we introduce a *Data Parallel Channel* (DPC). A data parallel channel supports concurrent accesses by parallel processing elements, i.e. a process can read multiple tokens at once, or write multiple tokens at once. The ports are transformed in a similar manner, thus instead of being one token wide, now they are W tokens wide, where W corresponds to the amount of data parallelism in the process

3.5. INTERMEDIATE MODEL: A DATA PARALLEL VIEW (DPV)

domain. For example, the DPP process P'_2 at time step $t = 3$ in the *predictor* example above reads 4 tokens in the *read* phase, evaluates the *predict* function 4 times in the *execute* phase, and writes 4 results to the output channels in the *write* phase.

In this thesis, we consider only coarse-grain synchronization between data parallel processes. In a P/C pair of DPP nodes, the consumer DPP starts after the producer DPP finishes its execution. Fine-grain synchronization of processes is also possible, e.g. by associating an availability flag with each cell of the channel as proposed by Feautrier [60]. Following this approach, the role of the DPP controller would be extended to include checking whether input arguments of all operations in a data parallel iteration are available. Since contemporary GPUs do not offer architectural support for communication and synchronization between two running kernels, such fine-grain synchronization and its mapping on the GPU architecture is currently not considered.

Now that we have explained the basic principles of a DPV, let us give definitions for different PPN components, i.e. nodes, edges and domains, under DPV.

3.5.1 Data Parallel Process

A PPN process iteration represents a single execution of the process body (Definition 17). In the *write* phase of the process iteration, a single *operation* (Definition 6) of the node domain is processed, i.e. the PPN process evaluates the process function for a single set of input-output arguments. Under DPV, the execution of a process iteration includes the execution of *multiple operations* from the node domain in a data parallel manner. Let us define a *parallel process iteration*, as:

Definition 25 (Parallel Process Iteration (PPI))

A *parallel process iteration* of a DPP P' is a set of data parallel operations.

In each process iteration, a PPN node performs a functional evaluation, such as $out_0 = f(in_0)$. A data parallel process iteration evaluates multiple function instances at once. As an illustration, the statement below shows 32 elements of the array in_0 being transformed in parallel and stored into the 32 elements of array out_0

$$out_0[0..31] = f(in_0[0..31]),$$

where we informally used the notation $0..31$ to denote 32 elements of an array. A PPI is typically executed by parallel processing elements. The number of processing elements to process a PPI corresponds to the width W of the target domain (Section 3.4.3). The largest parallel process iteration in the *predictor* example above has the width $W = 4$.

3.5. INTERMEDIATE MODEL: A DATA PARALLEL VIEW (DPV)

Definition 26 (Parallel Node Domain (PND))

The **parallel node domain** of a data parallel process P' , denoted by $D_{P'}$, is defined as a set of all parallel process iterations of process P' .

The parallel node domain $D_{P'}$ is obtained as the image of the PPN node P 's domain under the space-time mapping. In Section 3.4.2, we discussed how to construct a space-time mapping, and demonstrated the results of its application on three different domains. Typically (as demonstrated in the *predictor* example), there is a single target domain as a result of applying an (elementary) schedule. However, in some cases (as demonstrated in the *grid* example), the resulting schedule may be a piece-wise affine function (composite schedule), and contain a scheduling condition. In this case, we first perform splitting of the PPN node T according to the scheduling condition, e.g. $i + j \geq 5$. The mechanics of the PPN node splitting is described in detail in [93, 119]. This results in two PPN nodes T_1 and T_2 , with domains being complementary subsets of T 's domain, i.e. we obtain a domain $D_{T_1} = D_T \cap \{(i, j) \mid i + j \geq 5\}$, and a domain $D_{T_2} = D_T \cap \{(i, j) \mid i + j < 5\}$. This is considered to be a preprocessing step in the derivation of a DPV. After preprocessing, each of the derived PPN nodes can be transformed using a single space-time mapping.

A PPN process has a set of input ports, and output ports. The domains of all ports are transformed in the same way as the node domain with the space-time mapping.

Definition 27 (DPP Process)

A DPP process is an autonomous execution entity specified by:

1. A **parallel node domain** (PND) in space-time coordinates that specifies all data parallel process iterations,
2. (W, D) parameters of the PND,
3. A specification of **input and output port domains** in space-time coordinates.
4. A **process function** processing input arguments and producing function output arguments,
5. A control unit that is responsible for transitions between time steps and synchronization between data parallel process iterations.

where (W, D) parameters introduced in Definition 24 specify the maximal number of parallel processing entities¹ that can process the PND, and the number of time steps required to process the PND with W processing entities.

¹A processing entity can be implemented, for example, as a process or a thread.

The Data Parallel Process executes three phases in each iteration, namely `read`, `execute`, and `write`, similar to a PPN process. Each phase of a DPP is executed by multiple processing elements in parallel, which results in the evaluation of a process function multiple times in the same time step.

3.5.2 Data Parallel Channel

Each channel in a PPN is defined by a port of its producer node (edge source), and the corresponding port of its consumer node (edge destination). A DPP can write multiple tokens to the channel in a single data parallel write access. To allow concurrent accesses to the channel, we represent a Data Parallel Channel (DPC) as an array in random access memory (RAM). Each element of the array has a unique index. A DPP producer can write one or more tokens to an outgoing DPC in a single time step. Similarly, a DPP consumer can read one or more tokens from an incoming DPC in a single time step. In general, it is not required that all tokens read in one data parallel read access come from adjacent location in the memory, since it is possible to address the cells of a DPC. For each read/write access to the DPC, we construct an address function on the basis of the DPP's iteration vector, space-time mapping, and the channel's mapping. The details of the DPC construction are worked out for the *predictor* example in Appendix B.1.3.

Let us explain the construction of the DPC address function step by step on the *predictor* example's node P_2 in Figure B.1. For the purpose of the explanation, let us suppose that a PPN channel C_2 is represented as a memory array a_1 as illustrated in Figure 3.8. At each process iteration, producer P_1 writes an element to the location that corresponds to the current value of its iteration vector. A PPN channel is annotated with an affine mapping that specifies the exact producer-consumer relation between the nodes, i.e. given a specific value of the iteration vector of the consumer, it is possible to trace back the exact iteration at which the token was produced by P_1 . To read a token into process P_2 , we construct the array index as the read vector $\vec{r}$ using the current value of the process iteration vector and the channel mapping specification as $\vec{r} = \vec{x}_P = M\vec{x}_C$. In the case of a read access from P_2 to channel C_2 with the mapping $M_{C_2} = \begin{bmatrix} 1 & 0 & -1 \\ 0 & 1 & 0 \end{bmatrix}$, we obtain read vector $\vec{r}_{P_2} = M_{C_2}\vec{x}_{P_2} = [i-1, j]^T$, since $\vec{x}_C = \vec{x}_{P_2} = [i, j]^T$. This means that at iteration $i = 3, j = 1$, the process P_2 reads input argument in_0 from channel C_2 from the location $a_1[2, 1]$.

Since a DPP node can read/write multiple tokens at once to a DPC, at each parallel process iteration we generate *multiple* read/write vectors for access to the DPC memory, i.e. we generate one access vector for each operation. Since DPP processes are PPN nodes under a space-time mapping, when computing accesses to a data parallel channel we need to additionally consider space-time mappings of each node. Let us

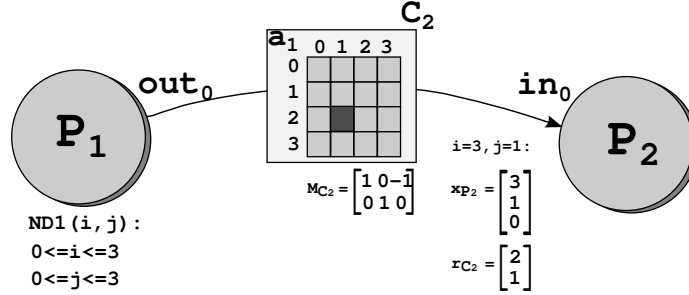


Figure 3.8: Computation of the read index for reading from the array.

denote space-time mapping of the producer node as T_P , and space-time mapping of the consumer node as T_C . The corresponding DPP nodes execute operations according to these two space-time mappings. At each execution, the producer DPP writes one or more values into the channel represented by array a'_1 , and the consumer reads one or more values from the array a'_1 . The write address $\vec{w}$ corresponds to the iteration vector $\vec{x}_P = (t_P, p_P)$ of the operation processed at time step t_P by processing element p_P . If a DPP producer is processed by 4 processing elements and generates 4 output values, there will be 4 write addresses $\vec{w}[0..3]$. The read address $\vec{r}$ for each processing element is obtained as a composition of mappings:

$$\vec{x}_P = T_P \vec{x}_P \wedge \vec{x}_C = T_C \vec{x}_C \wedge \vec{x}_P = M \vec{x}_C \Rightarrow \vec{x}_P = T_P M T_C^{-1} \vec{x}_C.$$

The resulting mapping M' that is associated with a DPC is a composite function $M' = T_P M T_C^{-1}$. To conclude:

Definition 28 (DPC Mapping)

An affine **mapping** M' of a Data Parallel Channel (DPC) is a function that specifies producer-consumer relationship between two DPP processes. The mapping M' maps the iteration points from the k -th input port domain of a DPP consumer process C' to the corresponding iteration points of its DPP producer process P' :

$$OPD_{P',i} = M'(IPD_{C',k}).$$

The mapping M' is a composite function:

$$M' = T_P M T_C^{-1}.$$

where T_P corresponds to the space-time mapping of the producer node P , M corresponds to the affine mapping of the PPN channel, and T_C is the space-time mapping

of the consumer node C .

Finally, let us discuss the special case of a PPN node with an absolutely parallel domain. An absolutely parallel domain has a zero schedule $t = 0$, i.e. all operations can be executed in parallel. The zero schedule is not included in the space-time mapping, since it would cause a singular matrix. The space-time mapping in this case is an identity matrix with all dimensions marked as space. This means that the parallel node domain is in one-to-one correspondence to the original node domain, but each of node domain dimensions is treated as a processor dimension. Being an identity matrix, such space-time mapping does not influence the channel address calculation and can be simply left out.

For each write operation of a producer there is a location in the channel to hold the result. Furthermore, each PPN channel is by default converted into a DPC. The size of the channel is by default determined by the size of the producer's PND. This may result in unnecessary memory size explosion. We present some possible optimizations to alleviate this problem, such as channel merging and buffer size optimization, in Section 3.8.1.

3.5.3 Synchronous Data Parallel Execution

Since we can not rely on blocking read semantics for synchronization between processes on a GPU, we need to address the synchronization ourselves. Let us now consider synchronization within a DPP required to support data parallel execution. In a PPN process there is an implicit synchronization between the execution of each two operations due to the blocking read/write operations. The blocking read/write primitives are not supported by the GPU architecture. Instead, we use explicit synchronization. For this purpose, we introduce a controller in each DPP that manages transitions between time steps and synchronization within a DPP.

We distinguish two classes of parallelism within a DPP: **cooperative parallelism** and **independent parallelism**. Cooperative parallelism is the general form of data parallelism in a DPP. The data produced by a single active processing entity is read by another processing entity in the next time step, and thus it is necessary to provide a mechanism that ensures that correct values are read. Processing of the parallel node domain in Figure 3.9(a) (the *predictor* example) is an example of cooperative parallel execution. Dependence vectors that are indicated by arrows between operations in Figure 3.9(a) show that to execute the operation at $t = 3$ by active processing entity $p = 3$, the results of the previous step produced by both processing entity $p = 3$ and processing entity $p = 2$ must be available. We realize this with synchronous data parallel execution, which works as follows: After the processing entities $p = 2$ and $p = 3$ compute their output arguments at $t = 2$ and write them to the DPP self-link. At the end of the write stage there is a synchronization barrier, that guarantees that

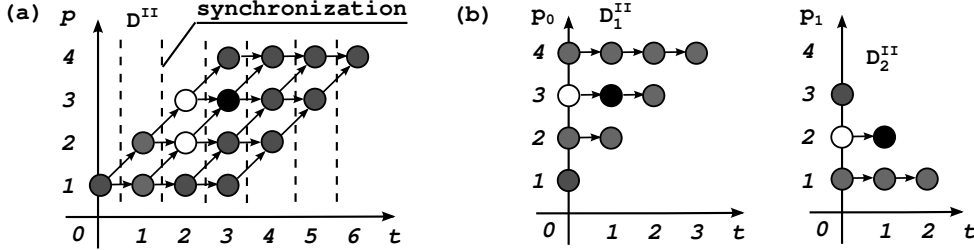


Figure 3.9: (a) the *predictor* example: Synchronous (cooperative) parallelism, (b) the *grid* example: Synchronization-free (independent) parallelism.

the results of all processing entities are available in the channel, and can be used in the next data parallel process iteration. In the general case, a synchronization barrier is required after each time step of DPP processing.

In some cases, this approach is overly pessimistic. For example, let us consider processing of the two parallel node domains resulting from the *grid* example that are shown in Figure 3.9(b). The dependence vectors point from the operation executed by some processing entity to the operation executed by the same processing entity in the next time step. As a consequence each processing entity can progress independently of the other processing entities. This is the case of *independent parallelism*, which is also known as *synchronization-free* parallelism. In this case, there is no need for communication between processing entities, and synchronization between processing entities at each time step is optimized out.

3.6 Mapping and Code Generation for GPU Accelerators

3.6.1 Introduction

In Section 3.5, we introduced concepts for capturing and exploiting data parallelism which facilitate mapping of a PPN onto data parallel accelerators. The most widely used accelerators for data parallel computations are today programmable GPUs. An overview of the CUDA architecture and programming model which exposes the GPU as a general purpose device to the programmer is given in Section 2.3.

In this section, we provide a structured way of mapping a process under DPV (i.e. a DPP) onto a CUDA kernel. We show how to generate all necessary components of a CUDA kernel from the DPV model. A simple, functionally correct CUDA kernel that exposes maximal data parallelism is obtained by traversing the data model behind the Data Parallel View in a straight-forward manner. This approach is implemented in the KPN2GPU tool, which is a Java-based extension of the COMPAAN compiler.

The same mapping approach can be applied for obtaining parallel programs according to the OpenCL standard. OpenCL kernels can easily be obtained by writing an OpenCL backend according to CUDA to OpenCL mapping scheme, or by using source-to-source code conversion tools [71, 91].

3.6.2 CUDA Code Generation

The CUDA programming model allows a programmer to define C functions, called kernels, that are executed N times in parallel by N different CUDA threads [37]. The kernel execution follows the SPMD model. This means that each thread executes the same kernel code (i.e. program) but can access different data. The kernel code can be parametrized in thread identifiers and global parameters.

Each DPP node is mapped to a single CUDA kernel ². Each kernel is parametrized in the CUDA thread identifier. Let us now explain how space-time mapping concepts map to CUDA kernel execution. A CUDA thread is uniquely identified with a thread identifier, which is provided by the CUDA runtime environment to each thread as a three-dimensional variable `threadIdx`. Each dimension of the `threadIdx` variable corresponds to one dimension of the thread block. Thus, the threads belonging to a one-dimensional thread block are identified using `threadIdx.x` only. In Figure 3.10(c) we see 4 CUDA threads processing the PND in Figure 3.10(f). We simply use the one-to-one mapping of the unique thread idx `threadIdx` to the space dimension p_0 of our PND as follows ³:

$$p_0 = \text{threadIdx.x} + 1.$$

Since all threads execute the same kernel code, the answer on the question *who am I?* obtained via `threadIdx` variable determines which operations the thread is processing. The time dimension of the PND, i.e. t_0 , is used to generate logical time steps within the thread body. This is simply realized by transforming the t_0 dimension into a for loop as follows:

```
for( $t_0 = 0$ ;  $t_0 < ND\_2\_D$ ;  $t_0++$ )....
```

For each thread we can answer the question *where am I* in the domain by combining the information on the thread identifier with the current time step t_0 . The answer determines if the thread is (1) active and should execute the next operation in the PND, and (2) what that operation exactly is.

²The PPN nodes with composite schedules are split into independent processes during preprocessing stage, and transformed into independent DPPs that can be executed concurrently in task parallel manner.

³The offset 1 is necessary only out of technical reasons. The scheduler returns t values starting from 0, but the allocator tool returns p values starting from 1.

3.6. MAPPING AND CODE GENERATION FOR GPU ACCELERATORS

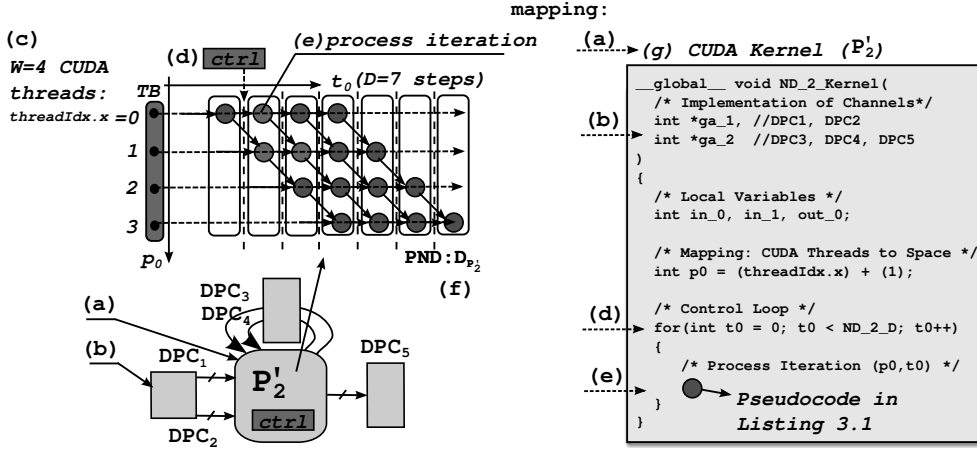


Figure 3.10: Generation of a CUDA kernel from DPV specification.

Now that we know how to parametrize kernel execution, let us show on the *predictor* example how to automatically generate a CUDA kernel from the specification of a DPP. A detailed specification of the DPP and the DPCs used to generate `ND_2_Kernel` from the DPP node P'_2 can be found in Appendix B.1.3, and its complete CUDA kernel can be found in Listing B.2. For reference, we also illustrate the mapping between DPP components for the transformer node P'_2 in the *predictor* example and the programming constructs in the CUDA kernel in Figure 3.10.

We generate the number of CUDA threads W executing the kernel (i.e. 4 threads shown in Figure 3.10(c)) from the W value in the (W, D) characterization of DPP P'_2 . We generate the number of time steps executed by the DPP from the D value in the (W, D) characterization of DPP P'_2 . As a result, the kernel `ND_2_Kernel` that implements DPP P'_2 is executed by $W = 4$ parallel threads in $D = 7$ synchronous time steps. The 4 threads are organized in one thread block (TB), as shown in Figure 3.10(c).

We transform the DPP control unit `ctrl` into the kernel control code. The kernel control code consists of the for-loop that goes through all instances of the time-dimension t_0 and invokes process iterations, and the synchronization to control transitions between the R/E/W phases of process iterations. To guide the execution of 4 CUDA threads through 7 time steps, we introduce the control loop in the `ND_2_Kernel` as illustrated in Figure 3.10(g). The upper bound of the control loop `ND_2_D` corresponds to the number of time steps executed by the DPP, i.e. the value of the D parameter. At each iteration of the loop, one process iteration (See Figure 3.10(e)) is executed by each thread. Since all threads execute the kernel body in parallel, this means that at each time step t_0 multiple process iterations are executed (one per each active thread). As in the PPN process, each process iteration contains the three

3.6. MAPPING AND CODE GENERATION FOR GPU ACCELERATORS

R/E/W phases. Since the results of one thread's write phase are used as input to other thread's read phase, we need to take care that no race conditions can occur ⁴.

```
1  ////////////////////////////////////////////
2  // Process Iteration (Executed by W CUDA Threads)
3  // Parametrized in (p0, t0)
4  ////////////////////////////////////////////
5  // Phase I: READ
6  // Multiplex load of argument in_0
7  if (ND_2IP_1)
8      in_0 = DPC1(t0, p0);
9  if (ND_2IP_2)
10     in_0 = DPC2(t0, p0);
11
12 // Multiplex load of argument in_1
13 if (ND_2IP_3)
14     in_1 = DPC3(t0, p0);
15 if (ND_2IP_4)
16     in_1 = DPC4(t0, p0);
17
18 // Phase II: EXECUTE
19 if (ACTIVE_ND_2)
20     out_0 = predictor(in_0, in_1);
21
22 // Phase III: WRITE
23 // Write result out_0 to output channels
24 if (ND_2OP_1)
25     DPC1(t0, p0) = out_0;
26 if (ND_2OP_1_d1)
27     DPC3(t0, p0) = out_0;
28 if (ND_2OP_1_d2)
29     DPC5(t0, p0) = out_0;
30 __syncthreads();
```

Listing 3.1: CUDA Kernel: Process Iteration

To avoid race conditions, we must ensure that all threads complete their write phase before proceeding to the read phase of the next iteration. We realize the synchronization using the CUDA primitive (`syncthreads()`) The synchronization primitives `syncthreads()` synchronizes execution of the threads within the same thread block by inserting a synchronization barrier. Only after all threads have reached the barrier, the threads can proceed to execute the next instruction.

As mentioned before, each process iteration consists of the three R/E/W phases. Each CUDA thread associated with p_0 executes the complete R/E/W cycle at each

⁴A race condition occurs when two threads try to access the same memory location at the same time and one of the accesses is a write access.

3.6. MAPPING AND CODE GENERATION FOR GPU ACCELERATORS

instance of t_0 . For reference, we show the parametrized CUDA code that implements the process iteration from Figure 3.10(e) also in Listing 3.1.

In the `read` phase, each active thread reads the input arguments in_0 and in_1 from incoming channels into the local variables. In the *predictor* example, the actual values of each argument can be provided by two channels, e.g. they could be read from either Data Parallel Channel *DPC1* or *DPC2* for the input argument in_0 . From which of the two channels the thread gets the value at a given iteration is determined by evaluation of the guard conditions at lines 7 and 9 for the input argument in_0 and by evaluation of the guard conditions at lines 13 and 15 for the input argument in_1 . The CUDA definition of the guard conditions is generated automatically from the polytopes describing the input port domains (IPDs) which the DPP uses to read data from the channels *DPC1* and *DPC3*. The resulting guard conditions are parametrized in (p_0, t_0) , i.e. thread index and control-loop counter. For more details, see the guard condition definitions on lines 5-14 in Listing B.2.

In the `execute` phase, each active thread evaluates an instance of process function. We obtain the guard condition at line 19 of Listing 3.1 from the specification of the PND polytope (i.e. node domain in space-time coordinates). Similarly to the read conditions discussed above, the guard condition is parametrized in (p_0, t_0) coordinates. This makes it possible for each CUDA thread assigned to p_0 to determine if it is active at the current time step t_0 , i.e. whether it is within the bounds of the PND polytope. As a consequence, the parametrized guard condition allows only threads active in a given time step t_0 , to perform the function evaluation at line 20, while other threads idle. For example, at time step $t_0 = 0$ in Figure 3.10(f) only a single thread (`threadIdx.x = 0`) evaluates the process function, while at time step $t_0 = 3$ all 4 threads evaluate the process function. For the full specification of the guard condition, see line 5 in Listing B.2.

In the `write` phase, all active threads write the results of the function evaluation to output channels. The guard conditions for the write accesses are generated from DPP's OPDs. Due to parallel processing of the PND, a synchronization barrier is inserted at the end of the `write` phase (line 30) to ensure that none of the threads executing the CUDA kernel proceeds to the `read` phase of the next iteration before all other threads have completed the current process iteration. In case of synchronization-free parallelism, this barrier can be safely omitted from the kernel.

Accesses to DPCs are implemented as read/write accesses to linear arrays in the global memory of the GPU. We generate the address function for each DPC access from the definition of the DPC mapping M' (see Definition 28). The mappings for all 5 DPC channels are derived step by step in Appendix B.1.3. These mappings translate into channel access code on lines 18-22 in Listing B.2. Together, these components result in a fully functional CUDA kernel code given in Listing B.2.

So far, we focused on showing how to generate the CUDA kernel code for a single

DPP. The steps described above are repeated for each individual node in a PPN. The result is a collection of CUDA kernels. Once the kernels are obtained it is necessary to construct the CUDA host code to run the whole network. The complete CUDA host code for the *predictor* example is given in Listing B.1. A snippet from the CUDA host code illustrating the kernel launches is given below:

```
/* Execution of node 1 */
ND_1_Kernel<<< 1, 16 >>>(ga_1);
/* Execution of node 2 */
ND_2_Kernel<<< 1, 4 >>>(ga_1, ga_2);
/* Execution of node 3 */
ND_3_Kernel<<< 1, 16 >>>(ga_2);
```

This host code launches a CUDA kernel for each DPP node. The 4×4 node domains of processes P and C are absolutely parallel, which means that their kernels can be executed by 16 threads in parallel. The total number of the threads executing the kernel is determined by setting the size of a CUDA thread block in the CUDA kernel launch configuration specified by CUDA specific notation `<<< blocks, threads >>>>`. Setting the first parameter (`blocks`) will be discussed in the next section. The second parameter (`threads`) determines to the number of threads in a thread block, i.e. there are 16 threads processing `ND_1_Kernel` and 4 threads processing `ND_2_Kernel`. Each of the kernels runs to completion before the next kernel is launched. All edges in the network, i.e. the DPCs, are mapped onto memory regions in the global memory space. As a consequence, the CUDA kernels implementing the DPPs communicate data through global memory arrays on the GPU. So, `ND_1_Kernel` communicates via memory array `ga_1` with `ND_2_Kernel`. Since `ND_1_Kernel` runs to completion, all data is available in `ga_1` when `ND_2_Kernel` starts to execute. Using this principle, each PPN under DPV is executed correctly as a topologically sorted network of kernels.

3.7 Scaling

3.7.1 Introduction

In Section 3.6 we presented an approach for mapping a DPV network onto a GPU. The parallelism in a DPP conceptually matches very well a single CUDA thread block. Let us see how this approach can be scaled up. In the previous sections, we have shown how to convert a 4×4 predictor domain into a DPP processed by 4 threads in parallel. If we increase the size to 512 threads, the code can be still obtained and processed in the same manner as explained earlier. By further increasing the domain size to 4000×4000 , we exceed the number of threads supported by CUDA for a single thread block. Taking care of this problem is important, since to fully fill up the GPU,

3.7. SCALING

it is necessary to provide work for all of its streaming multiprocessors (SMPs). For example, our Tesla C2050 GPU with 14 SMPs needs more than 14 thread blocks to make all SMPs busy. Furthermore, even larger number of thread blocks is desired in order to process work efficiently on the GPU. Having a larger number of thread blocks enables the scheduler on each SMP to overlap independent instructions from different thread blocks. It also enables better load balancing on the GPU, since different blocks can take different time to complete.

Scaling is typically solved by partitioning (tiling) the iteration domain into smaller blocks (tiles), each of which can be processed by a single thread block. Legal and efficient tiling is a topic which has received much research attention in the compiler community, see e.g., [23, 32, 75, 112, 113, 123–125, 138, 140]. In this section, we sketch how mapping of DPP onto GPU could be made more scalable by tiling the node domain to generate coarse-grain independent data parallel tasks. After tiling the DPP node domain on the *predictor* example, we introduce a set of extensions for mapping a DPP on multiple CUDA thread blocks and CUDA code generation, such as parametrization of CUDA kernel code, mapping two-level CUDA thread hierarchy to DPP's space-time coordinates, adjustment of DPP's node and port domains, and the extension of the channel addressing scheme. The result is SPMD CUDA kernel code that is parametrized in (1) static parameters, (2) (W, D) parameters of the target domain, (3) tile sizes, and (4) run-time CUDA parameters of each thread, such as thread index and thread block index. In addition, we show the host-side code that is required to execute tiles in the correct order.

3.7.2 Tiling for Coarse-Grain Data Parallelism

To partition the problem, we create node domain tiles such that they can be each processed with one CUDA thread block. Next step is to schedule the tiles to make sure that dependences are preserved.

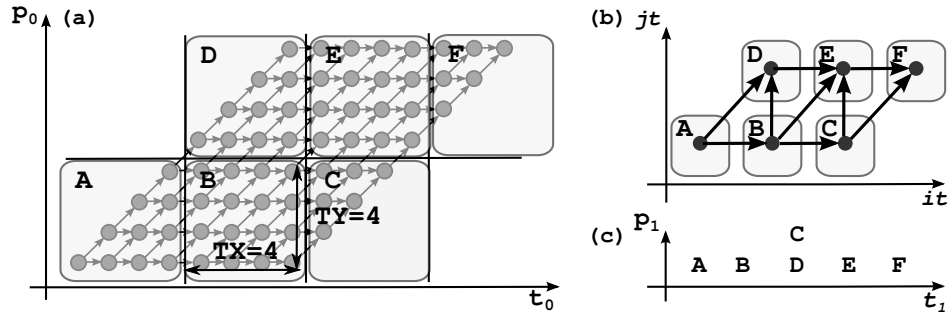


Figure 3.11: (a) Partitioning (tiling) of the target domain, (b) Tile domain, (c) Execution order of the tiles.

Let us discuss how to scale the *predictor* example, which has the most complex dependence pattern of the three running examples. Figure 3.11(a) shows the target domain of the *predictor* example overlayed with $TX \times TY$ tiles. Operations from the node domain in (a) are assigned to tiles according to the tiling conditions:

$$\begin{aligned} TY \cdot jt &\leq p_0 \leq TY \cdot jt + TY - 1, \\ TX \cdot it &\leq t_0 \leq TX \cdot it + TX - 1. \end{aligned} \quad (3.3)$$

where p_0 and t_0 are PND coordinates, TX and TY denote tiles sizes in each dimension of the domain, i.e. $TX = 4$ is the width of the tile in t_0 -dimension, and $TY = 4$ is the width of the tile in p_0 dimension, and it and jt represent tile coordinates. By representing all operations encapsulated in a single tile with a single iteration point in tile space (it, jt) , we obtain the tile domain with tile iteration vector $\vec{x}^T = [it, jt]^T$ shown in Figure 3.11(b), which is described as a polytope as follows:

$$\begin{aligned} 0 &\leq it \leq 3, \\ 0 &\leq jt \leq 1, \\ it - 2 &\leq jt, \\ jt &\leq it. \end{aligned} \quad (3.4)$$

Each tile is considered to be atomic unit of workload. If there is any dependence relation in the original target domain between operations mapped to different tiles, it results in a dependence between tiles in the tile domain. For example, the operations from tile B in Figure 3.11(a) are the sources of dependences to the operations in: (i) tile C, (ii) tile D, and tile E.

Dependences between tiles must be satisfied during the execution. In the example above, this means that tiles C, D, and E must be executed after tile B. Since there are no dependences between tiles C and D, these two tiles can be executed in parallel on a GPU. Finding a valid execution order of the tiles is the same problem as finding data parallelism in a domain, and it can be solved following the approach in Section 3.4.2. We optimize for maximal data parallelism in order to maximize the number of independent tiles in each time step t_1 . For the example above, the tile scheduling function is $t_1(it, jt) = it + jt$, and the tile allocation function is $p_1(it, jt) = jt$.

3.7.3 Consequences for GPU Mapping

Recall from Section 2.3 that CUDA specifies a two-level architecture: an array of simple streaming processors organized in streaming multiprocessors (SMPs). The CUDA programming model provides the concept of a thread block as a unit of execution on an SMP, and the concept of a thread as a unit of execution on a single

3.7. SCALING

streaming processor (SPs). Section 3.6, illustrates mapping of a Data Parallel Process to fine-grain threads within a single CUDA thread block. Using the approach in Section 3.7.2 to tile the node domain, we obtain the coarse-grain parallelism which enables the generation of a CUDA grid of blocks, and scaling the DPP to multiple GPU SMPs.

Grid Specification

The amount of coarse-grain data parallelism that directly corresponds to the number of thread blocks mapped onto SMPs, is specified as a parameter of the CUDA grid using the `<<< GridDim, BlockDim >>>` notation, where the `GridDim` variable corresponds to the number of thread blocks, and the `BlockDim` variable corresponds to the number of threads within a thread block. In the ideal case, all tiles are independent, and can be processed by different thread blocks during a single kernel execution. In the general case, there can be dependences between tiles, as in the *predictor* example depicted in Figure 3.11(b). The inter-tile dependences determine the precedence order of the tiles. We satisfy the precedence order by processing the tiles according to the tile schedule and allocation function. At each time step t_1 , we launch a kernel which processes up to W_2 independent tiles in parallel. In the example above, the maximal width $W_2 = width(p_1) = 2$, since only the tiles *C* and *D* can be processed in parallel. The domain in Figure 3.11(a) is only given for the illustration. In practice, domains offloaded to the GPU for acceleration are much larger. As a consequence the starting domain can be partitioned into larger number of tiles that can be executed in parallel. For example, a 8000×8000 *predictor* could be processed with 32 thread blocks of 256 threads, resulting in much higher utilization of a Tesla C2050 GPU than with a single thread block. The 32 thread blocks are distributed to 14 GPU SMPs by the GPU block scheduler. The parallelism on the tile level is specified through the grid size `ND_2_GridDim` parameter, which corresponds to the number of CUDA thread blocks. The scalable processing of the *predictor* example on GPU is realized with the following CUDA host code:

```
/* t1=0: Execution of 1 tile A */
ND_2_GridDim = 1;
ND_2_BlockDim = 4;
GridTimeStep = 0;
ND_2_Kernel<<< ND_2_GRIDDim, ND_2_BlockDim>>>(GridTimeStep, ga_1, ga_2);
/* t1=1: Execution of 1 tile B */
ND_2_GridDim = 1;
ND_2_BlockDim = 4;
GridTimeStep = 1;
ND_2_Kernel<<< ND_2_GRIDDim, ND_2_BlockDim>>>(GridTimeStep, ga_1, ga_2);
/* t1=2: Execution of 2 tiles C and D in parallel */
ND_2_GridDim = 2;
```

```

ND_2_BlockDim = 4;
GridTimeStep = 2;
ND_2_Kernel<<< ND_2_GRIDDim, ND_2_BlockDim>>>(GridTimeStep, ga_1, ga_2);
...

```

The result is synchronous data parallel execution of the parallel node domain at two levels (the level of single operations, and the tile level). A CUDA host template for the execution of tiled domains is given in Section B.1.7.

Parametrized SPMD Code Generation

Once launched, a CUDA kernel is executed in the SPMD manner by the CUDA thread hierarchy. Threads are organized in thread blocks which execute independently. We assume that each CUDA thread block processes one tile of a DPP's Parallel Node Domain. Threads of each thread block process different iteration points enclosed in a tile. According to the CUDA programming model, each CUDA thread executes the complete body of the kernel function. We started writing CUDA kernel code with a single level of parametrization via `threadIdx`, which is local to a thread block. To obtain a unique thread identifier in the CUDA thread hierarchy, we introduce a second level of parametrization via block identifier `blockIdx`. For all threads to process different iteration points of the DPP's PND, the body of the kernel is now parametrized in thread (`threadIdx`), and thread block identifiers (`blockIdx`). This is achieved by the following modifications to the dpp-to-cuda mapping approach:

- Mapping of CUDA two-level thread hierarchy to PND iterations
- Augmenting R/E/W conditions (DPP's node and port domains)

Mapping Two-Level Thread Hierarchy Let us illustrate the mapping of the CUDA thread hierarchy on PND with the *predictor* example. Let us consider the third kernel launch that processes tiles *C* and *D* on the GPU. The third kernel call is executed by two CUDA thread blocks with unique identifiers $blockIdx.x = 0$ and $blockIdx.x = 1$. Each of the thread blocks contains 4 CUDA threads with thread identifiers $threadIdx.x = 0$ to 4. The first thread block $blockIdx.x = 0$ processes tile *C* with tile index vector $(it, jt) = (2, 0)$. The threads of the first block map to processing entities $p_0 = 1.4$ in space-time coordinates, and execute time steps $t_0 = 8..11$ of the DPP's PND. The second thread block $blockIdx.x = 1$ processes tile *D* with tile index vector $(it, jt) = (1, 1)$. The threads of the second block map to processing entities $p_0 = 5..8$ in space-time coordinates, and execute time steps $t_0 = 8..11$ of the DPP's PND. There is a one-to-one mapping between the data parallel operations in the transformed tile domain and thread blocks, i.e. $p_1 = blockIdx.x$. This enables us to calculate which tiles in the original PND are processed by which thread block

3.7. SCALING

at each time step t_1 by using the inverse space-time mapping to calculate the vectors. Using the schedule t_1 and the allocation p_1 for the *predictor* example, we obtain the tile index components $it = t_1 - p_1$ and $jt = p_1$. As a result, for each thread block launched at some time step t_1 , we can determine which tile (it, jt) is processed as $(t_1 - \text{blockIdx.x}, \text{blockIdx.x})$.

This allows us to construct the mapping of CUDA thread hierarchy onto the Parallel Node Domain of the DPP. Following the tiling approach depicted in (a), we can express each p_0 value as the number of tiles in dimension p_0 plus some offset from the start of the tile. Since each tile is processed by a single thread block, the offsets within the tile correspond to thread indexes within a thread block (threadIdx.x). The beginning of each tile in p -dimension can be expressed as $TY * jt$, where jt is the tile index across p -dimension, and TY is the tile width in p -dimension. Applying the results from previous paragraph, we obtain the following mapping:

$$p_0 = 4 * \text{blockIdx.x} + \text{threadIdx.x}.$$

Similarly, the current time step for each thread is calculated as a function of tile index and tile width across t_0 dimension. The lower bound lb for t_0 in each tile is found at $TX * it$, i.e. $lb(t_0) = 4 * (t_1 - \text{blockIdx.x})$.

Augmenting R/E/W Conditions Since now each CUDA thread processes only iteration points within one tile, the guard conditions for all three R/E/W phases of the process execution must be augmented with additional constraints. The additional constraints ensure that the threads of one thread block execute only operations of the PND that are within the tile assigned to the given thread block. After substitution of the thread block identifiers into $it = t_1 - \text{blockIdx.x}$ and $jt = \text{blockIdx.x}$, we obtain constraints in Definition 3.3 and Definition 3.4 as a function of space-time coordinates, global parameters, and CUDA variables:

$$\begin{aligned} 4 \cdot \text{blockIdx.x} \leq p_0 &\leq 4 \cdot \text{blockIdx.x} + 3, \\ 4 \cdot (t_1 - 1 - \text{blockIdx.x}) \leq t_0 &\leq 4 \cdot (t_1 - \text{blockIdx.x}) + 3, \\ 0 \leq (t_1 - \text{blockIdx.x}) &\leq 3, \\ 0 \leq \text{blockIdx.x} &\leq 1, \\ (t_1 - \text{blockIdx.x}) - 2 &\leq \text{blockIdx.x}, \\ \text{blockIdx.x} \leq (t_1 - \text{blockIdx.x}). \end{aligned} \tag{3.5}$$

We use the constraints in 3.5 to augment the conditions for DPP's node domain, IPDs, and OPDs. We have parametrized the CUDA code in such a way that the same code can be executed by multiple thread blocks. As a result, the CUDA kernel can

be processed by multiple thread blocks, with each thread block now processing the iterations within one tile of the DPP's PND.

For reference, we give the functional CUDA kernel code for the scaled-up predictor example in Appendix B.1.7.

3.8 Extensions and Optimizations

So far, we have presented a structured approach showing how to generate CUDA code for each PPN process. There are ample opportunities to further improve and optimize the CUDA code obtained in this way. In this section, we will present and discuss some of the extensions and optimizations.

3.8.1 Memory Optimizations

DPV Channel Merging

Under DPV, we transform each PPN channel by default into a DPC. For each dependence relationship between a consumer node and a producer node in a PPN, there is one PPN channel. The result of a function evaluation is written to one or more output channels by a produced process. The separation of channels in a PPN enables task-parallel distributed memory style of processing. However, the separation of channels also has three disadvantages for performance. It results in increased total memory size requirements, may increase the number of write accesses per output argument, and it limits data reuse. Thus, the separation of channels should be used only when necessary.

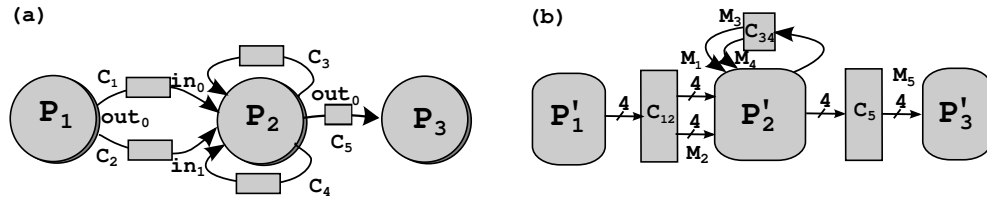


Figure 3.12: DPV: Channel merging optimization.

To alleviate the impact of these issues on DPV, we propose a selective channel merging strategy. Let us consider the *predictor* example in Figure 3.12(a). By default, each of the channels C_1 and C_2 connecting nodes P_1 and P_2 , is in DPV represented with an indexable memory array corresponding to the producer's parallel node domain. To optimize the memory requirements, we merge the arrays representing two channels into a single array depicted in Figure 3.12(b) as memory block

3.8. EXTENSIONS AND OPTIMIZATIONS

C_{12} . Although the memory space for channels is merged into a single array, we preserve separate channel mappings M_1 and M_2 . We use the mappings to reconstruct exact values of the read vectors for each input argument to the consumer process. Similarly, the self-links C_3 and C_4 are merged into C_{34} .

We call our channel merging transformation selective, because we apply it only to channels connecting the same producer and the same consumer nodes, and thus using the same memory space. Based on this property, we defined a *channel merge test*. The channels connecting a producer with different consumer nodes are not merged in order to preserve task-parallelism.

Buffer Size Optimization

A Data Parallel Channel is represented with an array in memory, i.e. a buffer. The default channel size is determined by the size of the producer node that writes data to the channel. The size of the buffer can be optimized by means of *lifetime analysis*⁵.

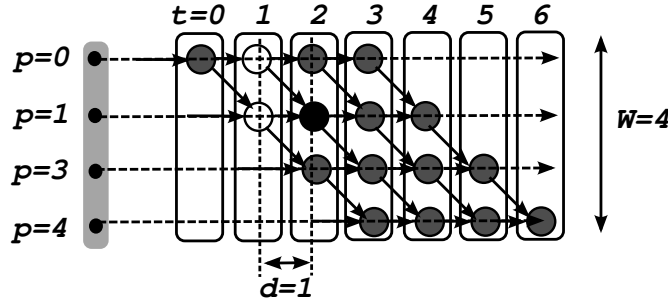


Figure 3.13: The input arguments to the selected operation (black) at $t = 2$ are two values produced by process elements at $t = 1$ (white). The lifetime of the values produced by the node is one time step ($d = 1$).

Let us analyze how the transformer node in the *predictor* example executes. The execution of the transformer node is depicted in Figure 3.13. The transformer node P'_2 writes up to 4 results in parallel to the channels C_{34} and C_5 , and reads up to 4 tokens per input argument in parallel from channels C_{12} and C_{34} . The guard conditions constructed from IPDs determine for each input argument which channel needs to be accessed. At time step $t = 0$, only a single value is read from the incoming channel C_{12} . At time step $t = 1$, one value is read from the incoming channel C_{12} , and one value from the self-link C_{34} . This value has been produced in the previous data parallel iteration of process P'_2 . Each of the values produced by the process P'_2

⁵Lifetime analysis is a compiler techniques that determines how long a value is actually used and thus how long it needs to be kept alive.

is used as an input argument only in the first subsequent time step. After the `read` phase of the next step is complete, the value is not needed any more, and it is safe to overwrite the memory location. This observation can be used to reduce the size of the memory buffer for C_{34} . Instead of reserving $D \times W$ locations for the channel, only W locations are necessary.

Thus, the buffer size can be significantly reduced by considering the lifetime of values. The life time of a value is the number of time steps for which that value needs to be preserved in memory, so that it can be used, i.e. the distance between the time step in which it is produced and the time step in which it is consumed. This distance corresponds to the length of the direction vector in target space time steps. Following this approach, the buffer size BS of channel C' is computed as follows:

$$BS(C') = d'_p \times W'_p$$

where W'_p represents the width of the spatial dimension of process P' PND, and d'_p is the length of direction vector in target domain projected onto the time dimension t . Domains with spatial dimensions only are considered to have $d'_p = 1$, i.e. their size corresponds to the total width of the PND.

After resizing the buffer according to the calculation above, the channel address function needs to be normalized. We realize this by using the modulo function to calculate address coordinates in the time dimension. In the *predictor* example above, $d_{p_2} = 1$ which results in buffer size $BS(C_{34}) = 1 \times 4$ locations. The results are written to some array `a'[1][W]`. In case of a merged channel, the number of W -wide buffer cells is determined by the length of the longest direction vector projection on the time dimension.

After the buffer size optimization, the buffer cells can be reused for writing output data. In order not to overwrite the values produced in one step before they are read by all processing elements in the subsequent step, additional synchronization is needed. We realize this by adding an additional synchronization barrier between `read` and `execute`. This barrier ensures that no thread proceeds to the `execute` and `write` phases, before all the threads have completed the `read` phase and loaded the input arguments from the shared memory into thread-private variables. For implementation details, see the optimized CUDA kernel for the *predictor* example in Listing B.3.

Channel Mapping Optimizations

In Section 3.6, we demonstrated the mapping of DPV channels to the GPU device memory, and explained the addressing scheme. All data parallel channels are by default mapped to the global memory of the GPU. The GPU global memory also serves as the main communication interface between the CPU and the GPU. The GPU ar-

3.8. EXTENSIONS AND OPTIMIZATIONS

chitecture has a rich memory hierarchy, as described in Section 2.3. GPU memory includes the global memory space implemented in device DRAM, shared memory implemented as on-chip scratchpad memory, and registers. The shared memory and the register file have much smaller latency (on the order of 10s of cycles) than the global memory (100s of cycles).

We classify the channels according to their source and destination nodes into two categories:

- inter-node channels - source and destination processes are two different nodes,
- intra-node channels (self-links) - the source and the destination is the same node.

The inter-node channels must be mapped to the global memory space, since it is the only memory that is guaranteed to preserve the data between kernel invocations, and thus executions of different DPPs. However, we can leverage the rich GPU memory hierarchy to improve the mapping of self-links. In most cases, self-links can be mapped onto the shared memory of the GPU (i.e. fast on-chip scratchpad). The scratchpad can be accessed by multiple threads in parallel, and it is also frequently used for inter-thread communication. However, the on-chip shared memory is a limited resource. The size of the shared memory available on each streaming multiprocessor is typically several orders of magnitude smaller than the size of the device's global memory. Thus, an important consideration is whether the buffer allocated for the channel can fit into the shared memory of the GPU. Before channel mapping optimizations, the buffer sizes should be optimized, e.g. using techniques described in Section 3.8.1. The results of the channel mapping optimizations are given for the *predictor* example in Listing B.3. The lines 10-14 of Listing B.3 illustrate mapping of self-links *DPC1*, *DPC3*, and *DPC13* into shared memory array, which is defined at line 20 of Listing B.3.

In case of independent parallelism, the mapping of self-links can be further optimized. Since there is no communication between threads, all output and input arguments of a thread can be stored in thread-private memory, i.e. which is implemented as a partition of the low-latency register file on the GPU.

3.8.2 Task Parallelism

Introduction of concurrent kernel execution by NVIDIA in second-generation of GPUs for general purpose processing [105] creates an opportunity for exploiting both data and task parallelism on GPUs. The PPN model inherently exposes task parallelism. An application is specified as a network of communicating processes, i.e. autonomous tasks. In this section, we show how to leverage the task-parallel nature of PPNs to take advantage of the concurrent execution on the second-generation

GPUs. Let us describe concepts required for mapping an arbitrary direct acyclic PPN graph⁶ onto a GPU in task-parallel manner.

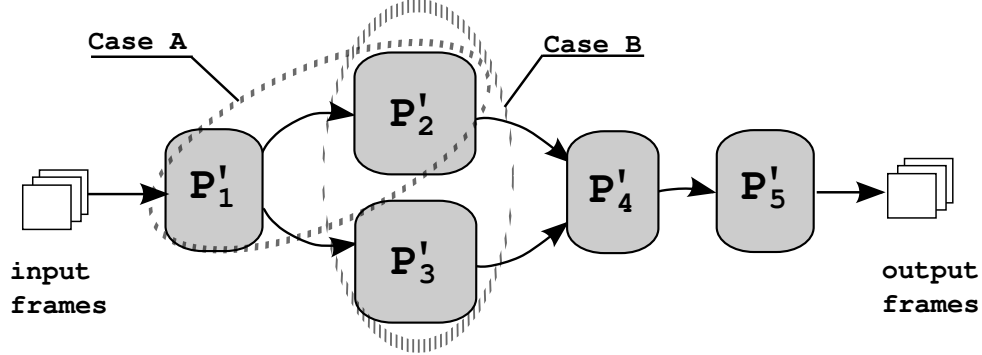


Figure 3.14: A PPN under DPV of a simple streaming application. Case A depicts two dependent tasks. Case B depicts two independent tasks.

As an illustrative example, let us consider the mapping of a simple streaming application (Sobel edge detecting algorithm) onto a second-generation GPU. The PPN that was obtained by the COMPAAAN compiler is shown in Figure 3.14. In the given application, all processes feature absolutely parallel node domains. This simple PPN contains two representative cases: (a) dependent processes, (b) independent processes. Solving these two cases enables us to map an arbitrary PPN without feedback cycles on a GPU.

To map a PPN on the GPU, we leverage several advanced concepts in the CUDA model: *CUDA streams*, *CUDA events*, and *event synchronization mechanisms*. Let us first explain the concept of a CUDA stream [37]. A CUDA stream is a sequence of GPU operations. By GPU operation, we refer to operations such as a kernel launch, data transfer, or a GPU event. GPU operations within a CUDA stream execute in-order, as specified in the stream source code. Conceptually, the CUDA stream resembles an independent thread of execution on a CPU. Different CUDA streams can be executed on a GPU in parallel. The operations in streams are assumed to be independent. However, coordination between operations in different streams can still be achieved by means of GPU events. We leverage this model of execution to implement task-parallel DPV execution on the GPU. Exploiting task parallelism may be beneficial in situations when GPU tasks do not contain sufficient coarse-grain data parallelism to fully utilize all GPU resources, e.g. when there are not enough thread blocks to fill up the GPU multiprocessors.

As explained in Section 3.6, we generate a CUDA kernel for each DPP. We model

⁶Feedback cycles are undefined on GPU and cause a GPU to deadlock.

3.8. EXTENSIONS AND OPTIMIZATIONS

a DPP as a data-driven GPU task containing synchronization and kernel launch capabilities. We designate an independent CUDA stream to each DPP in the process network. Given that the CUDA programming model allows GPU operations in different streams to execute concurrently, this makes it possible for two independent DPPs P'_2 and P'_3 in Figure 3.14 (Case B) to run in task-parallel manner on a single GPU. We realize the inter-task dependences (Case A) with an event-based synchronization mechanism.

The event-based synchronization works as follows. Each DPP is implemented as a GPU task that is structured as a finite state machine (FSM) with three states:

- **WAIT(W)** - blocking wait on input arguments/data to become available
- **EXECUTE(E)** - execution of CUDA kernel implementing a given DPP
- **SIGNAL(S)** - signaling that the kernel has finished and that the output data is now available

The PPN blocking read is realized via **WAIT** state. A GPU task is in the **WAIT** state, if the prerequisites for the kernel execution are not yet met, i.e. as long as the input arguments are not available. Once the input arguments become available, the FSM goes into the **EXECUTE** state, in which the kernel generated from the DPP is launched. Once the kernel execution is completed, the output arguments of the given task are fully available in the channel, and other GPU tasks waiting on this data to become available can proceed with execution. To signal data availability to other waiting GPU tasks, a kernel termination event is recorded in the **SIGNAL** state. All GPU tasks waiting on this event can then proceed with execution.

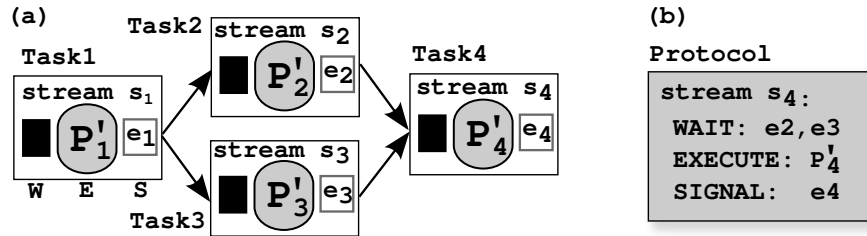


Figure 3.15: An event-based protocol for data-driven execution on the GPU.

The concept of event-based synchronization between DPP nodes P'_1 , P'_2 , P'_3 , and P'_4 is illustrated in Figure 3.15(a). First, *Task1* launches the kernel that implement the DPP node P'_1 . After the kernel finishes, *Task1* issues event e_1 to signal that the results of P'_1 are available. Until a data available event e_1 is recorded, the DPP processes P'_2 and P'_3 are in the blocked (**WAIT**) state. Observing event e_1 indicates to DPP processes

P'_2 and P'_3 that the output of process P'_1 is ready to be consumed. Tasks implementing processes P'_2 and P'_3 transition into the EXECUTE state. Each process then launches its kernel. The two kernels are free to execute in parallel on the GPU. Upon completion of each kernel, the GPU task issues the kernel completion event. A GPU task may block on one or more data events. Links between DPP processes denote dataflow dependencies. The event sensitivity list of each GPU task is derived from the list of incoming channels. A GPU task can be blocked waiting on one or more events. An example of a GPU task with multiple events in the sensitivity list is Task4. Task4 must wait on event e_2 and e_3 before it is allowed to launch kernel P'_4 . When all events in the sensitivity list of a process are set, a kernel can start its execution. The protocol according to which Task4 executes is given in Figure 3.15(b).

As a result, a GPU can leverage not only data parallelism within DPPs, but also task parallelism between independent DPPs, such as P'_2 and P'_3 , as illustrated in Figure 3.16. We did some experiments with this technique on the sobel example, but

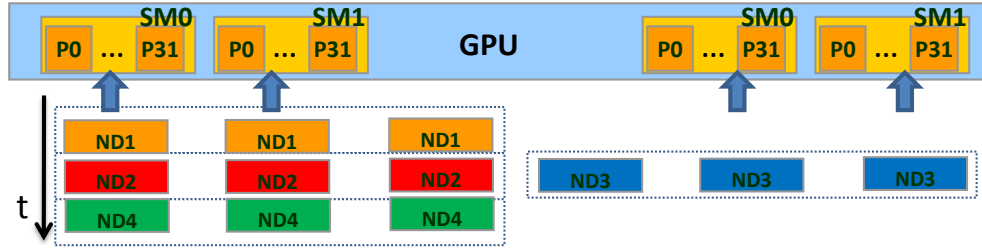


Figure 3.16: Task-parallel execution of DPP nodes on a Fermi-architecture GPU.

were not able to achieve significant benefits. This is primarily due to the short processing time of the sobel tasks and already sufficient amount of data parallelism in each task. To benefit from the task-parallelism on the GPU, the tasks must have sufficiently long running time and be sufficiently intensive to amortize overheads incurred by kernel launches, the use of CUDA streams, and event-based synchronization. Further research is needed to determine the threshold and application characteristics for efficient task parallel execution.

3.8.3 Token Composition and Reuse

To further extend the applicability of the KPN2GPU compiler, we introduce in this section two other techniques that we can exploit in the context of GPU parallelization: token composition and multiplicity. Token composition is a technique that allows composition of a number of tokens into a larger data structure. Multiplicity is a technique for detection of data reuse which enables data locality optimizations.

3.8. EXTENSIONS AND OPTIMIZATIONS

So far, we have considered only PPN communication, in which the producer writes and the consumer reads tokens of the same data type, e.g. a pixel. However, there are situations possible where the producer writes a token of data type pixel while the consumer reads a collection of these tokens, for example a block of pixels. In this case, we introduce the notion of a *composite token* to designate the collection of the tokens. The concept of the composite token is similar to the multirate concept found in dataflow formalisms, where a process (actor) can produce and consume multiple tokens from a stream on a single firing [83, 104]. A more extensive and formal discussion on the concept of the composite token will be given in Section 4.3.2.

The use of composite tokens leads to the synchronization on a much larger data structure instead of performing the synchronization on each pixel. For example, let us assume that a producer generates a stream of pixels, while the consumer wants to read in blocks of 64 tokens. The consumer synchronizes on 64 pixels instead of single tokens. If 64 tokens are available, the consumer performs a computation. Since 64 tokens are immediately present in the memory, a GPU can execute 64 threads in parallel.

An important GPU optimization is related to the concept of multiplicity introduced by Turjan [132]. In the DCT, each time `dotProduct1` executes, which happens 64 times, it reads the value of the complete `blockIn` variable as the input argument. Since the `blockIn` data does not change in the 64 times, it is necessary to read `blockIn` only once and reuse that value 64 times. This concept is called *multiplicity*.

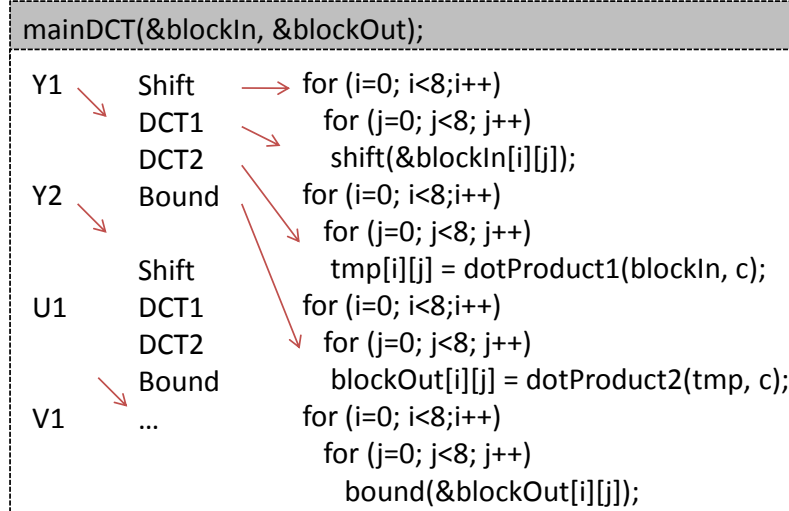


Figure 3.17: Pseudocode of `mainDCT`. The DCT-related computations (Shift, 2D Separable DCT, and Bound) are repeated on 8×8 blocks for each of the four color components in the YUV color model.

To see how token composition and multiplicity help to parallelize the DCT block in the M-JPEG code, let us have a look at the pseudocode for this calculation (the function `mainDCT`) shown in Figure 3.17 (See Appendix C for more a complete description of the MJPEG application). The pseudo code shows the function `DCT` that obtains an 8×8 block `blockIn` and which produces an 8×8 `blockOut`. On the incoming block of data the DCT function is performed in a sequence of computations: a normalization of values (`shift`), two integer arithmetic passes: (`DCT1`) and (`DCT2`), and bounding of values (`bound`).

The pseudo code shows that the shift operation is performed on each element of `blockIn`, pixel by pixel. On the `dotProduct1`, the pseudocode indicates that a complete 8×8 `blockIn` data structure is read. A consequence is that when function `dotProduct1` executes, it knows that all 64 tokens of `blockIn` are present. The function `dotProduct1` can be executed on a 8×8 block also using 64 parallel threads on a GPU.

Using the concept of multiplicity, the GPU code for DCT can be optimized for data locality. Each execution of the `dotProduct1` function requires the entire `blockIn`. Since each thread needs the same information, we introduce a small memory buffer that is shared by all 64 threads. This way, we can optimize for locality on variable `blockIn`. Instead of reading the entire `blockIn` from the channel by each thread, we use the value read into the buffer. We determine the fact that `blockIn` is reused by the 64 threads from the multiplicity property of the channel between the `shift` and `dotProduct1` functions. The multiplicity property is calculated using the techniques developed by Turjan [130, 132]. As a further optimization, we exploit the fast shared memory on the GPU to hold the reused data. In this section, we have shown how the concepts of multiplicity and token composition can be exploited on the GPU. With the composite token as an enabler concept, we are able to port DCT to the GPU resulting in in Figure 3.18. The results show substantial performance improvements with multiplicity, since it improves data locality and enables data reuse. Using multiplicity the DCT performance jumps from 535MB/s to 7445MB/s.

3.9 Results

We implemented the three-phase approach presented in this Chapter as an extension to the `COMPAA` compiler, called `KPN2GPU`. The `KPN2GPU` takes as input a PPN, transforms it into an intermediate model with data parallelism, and generates CUDA code for GPUs. As a result, we present the execution time for kernels generated by `KPN2GPU` from the PPNs of the two running examples: the *predictor* example and the *grid* example. For reference, we also add the execution time of an absolutely parallel code captured within the *parallel2d* example. The main computational ker-

3.9. RESULTS

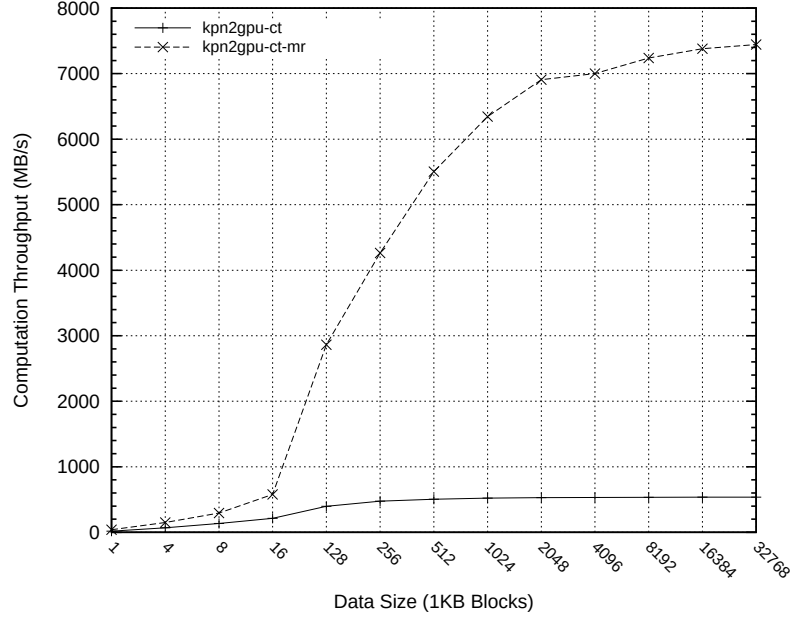


Figure 3.18: GPU Accelerated DCT Computation: Composite Tokenes and Multi-plicity.

nels of these three test cases have dependencies which are characteristic for many imaging, simulation, and scientific applications:

- predictor - synchronous (cooperative) data parallelism
- grid - synchronization-free (independent) data parallelism
- parallel2D - absolutely parallel

The absolutely parallel code, captured in the *parallel2d* example, is representative for application fields such as image processing, which have shown significant benefits from acceleration on the GPU architecture. Each operation, e.g. such as computation of a new pixel value, is executed independently. The *predictor* example is an example of synchronous data parallelism. The two-way dependencies between iteration points impose fine-grain synchronization requirements between time steps as discussed in Section 3.5.3 The *grid* example features synchronization-free data parallelism. As a consequence, no synchronization between time steps of the *grid* example is needed. We parallelized the PPNs of the three test cases using the approach presented in the previous sections. The generated CUDA code was executed and benchmarked on a PC with an AMD Processor and Tesla C2050 GPU.

3.9. RESULTS

Case Name	Target Domain(s)	(Space,Time) Dim	W	D
Predictor	1xPND	(1D, 1D)	N	$M + N - 1$
Grid	2xPND (merged)	(2*1D, 1D)	$M + N - 1$	$\max(N, M - 1)$
Parallel2D	1xAPND	(2D, t=0)	$M \times N$	1

Table 3.1: Data Parallel View on The Transformer Node

The statistics of the nodes under DPV are given in Table 3.1. The form of data parallelism determines the type and the shape of the target domain. In Table 3.1, column W corresponds to the number of threads used for processing the target domain, and column D represents the number of sequential steps executed by each thread. The general case of data parallel processing is captured within *predictor*, which shows how a 2D node domain is transformed into a target domain that has one space dimension (i.e. it maps to a 1D thread block) and 1D time. The time dimensions is reflected in the for loop over time steps in the kernel code. Scaling up the *predictor* example requires additional synchronization between tiles. In the grid case, the maximum data parallelism is obtained by a piecewise-affine schedule that splits the node domain into two independent sub-domains. For each sub-domain, a CUDA kernel is created. A special case is the target domain of the *parallel2D* transformer node. Each iteration point of this absolutely parallel node domain (APND) is processed by a different CUDA thread.

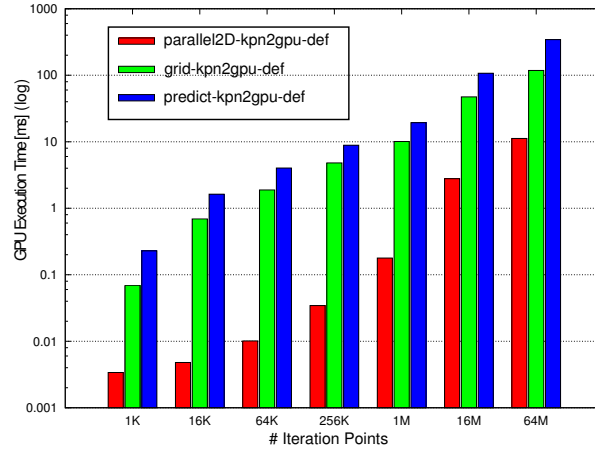


Figure 3.19: Execution time on Tesla C2050 GPU GPU for computationally intensive transformer nodes of the three test cases.

Figure 3.19 shows a comparison of the GPU execution time for the selected test cases as a function of iteration domain size. All channels are mapped onto the global memory arrays in GPU DRAM memory. The scaled-up iteration domains are first manually tiled and then executed in one or more kernel invocations. The parallel2D

3.9. RESULTS

represents an application that perfectly matches the GPU architecture, and can be executed by all threads in parallel within a single kernel invocation. Codes with synchronous data parallelism, such as the *predictor* example, map well onto the GPU only if the size of the target node domain fits a single thread block, since their dependence patterns impose inter-tile communication. The CUDA architecture requires that all thread blocks of a single kernel invocation execute independently. Since it is necessary to preserve dataflow dependencies between tiles, multiple kernel invocations are required to process the *predictor* example. The global synchronization points introduce additional overheads and significantly impact the processing time, as illustrated by an order of magnitude larger run time of the predictor kernel compared to parallel2D in Figure 3.19 (time is given in logarithmic scale).

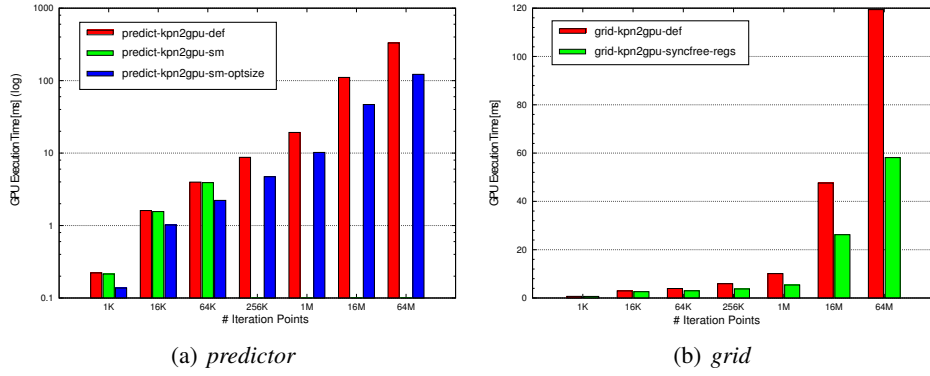


Figure 3.20: KPN2GPU optimizations. (a) Execution time (log-scale) of the *predictor* example. Comparison of default KPN2GPU code, and optimizations concerning channel mapping and buffer size reduction. (b) Execution time of the *grid* example. Comparison of default KPN2GPU code, and optimizations concerning synchronization-free parallelism and improved channel mapping (to registers).

The measurements in Figure 3.19 are the result of a default PPN mapping onto the CUDA architecture. However, the channel classification in DPV can be used to better exploit features of the rich CUDA memory hierarchy. CUDA provides not only global memory, but also low latency on-chip shared memory (SM), and on-chip register file. Figure 3.20 shows performance improvements achieved by channel mapping optimizations introduced in Section 3.8.1. As part of the *predictor* example optimizations, we mapped intra-node channels to the shared-memory of the GPU as proposed in Section 3.8.1. However, this approach (illustrated by *pred-kpngpu-sm*) does not scale, since the buffer sizes quickly exceed the rather small shared memory size (16 KB). Moreover, the large shared memory requirements limit the number of thread blocks processed in parallel. We alleviate both issues by combining the optimized channel mapping with lifetime analysis presented in Section 3.8.1 to re-

duce buffer sizes. As illustrated in Figure 3.20(a), combining these two optimizations results not only in performance improvements, but also in much better scalability. Although *pred-kpngpu-sm-optsize* requires introduction of an additional barrier to ensure that all threads have finished loading shared memory data before another round of writing starts, it results in a speedup of 3x compared to the baseline version. In the *grid* example the intra-node channels can be implemented as registers, resulting in considerable performance improvements shown in Figure 3.20(b).

The execution time of CUDA codes running on an NVIDIA Tesla C2050 card was also compared to the sequential C code of the three test cases running on AMD Phenom II X4 965 CPU. In these experiments, we observed speedups of up to 7x for the *predictor* example, 30x for the *grid* example, and 150x for the *parallel2d* example.

3.10 Conclusions

In this section, we presented a three-phase compile-time approach for mapping of a sequential application onto a GPU. First, we presented concepts and techniques for identifying data parallelism within the application’s PPN, second, we introduced an intermediate model (DPV) that allows us to capture task and data parallelism, and third, we demonstrated model-based generation of CUDA host and kernel code from the DPV. The structured nature of these three phases enabled us to implement the approach as an automated compiler step in the COMPAAAN compiler. Withing extensions and optimizations topics, we presented several memory-related optimizations and illustrated their impact on three test cases. We also showed how to leverage the PPN specification to exploit task-level parallelism on a second generation GPU. The task-parallel code generation leveraging the PPN model can be in future extended for automated multi-GPU mapping, which is a highly interesting topic with a large application potential in HPC.

3.10. CONCLUSIONS

Multi-Level Parallelization for Heterogeneous Platforms

4.1 Introduction

With the rapid proliferation of mobile multimedia devices, the need for processing of computationally-intensive streaming applications is growing at an increasingly fast pace. More and more computational power is needed to process ever larger data sets, perform signal transforms, and physics calculations on the fly. This poses increasing challenges on parallelization and mapping of streaming applications onto heterogeneous platforms. As a representative example of a streaming multimedia application, we analyze parallelization of the Motion JPEG (M-JPEG) encoder. In this chapter, we improve the performance of the existing task-level parallelization of M-JPEG using PPNs. We show how this parallelization approach can be improved by exploiting multi-level parallelization and token granularity adjustment to take advantage of task, data, and pipeline parallelism on heterogeneous platforms with GPUs.

An overview of the M-JPEG encoding process is given in Appendix C.1 and its pseudocode in Listing C.2. The task-parallel PPN model of the M-JPEG encoder can be easily obtained using the COMPAN compiler, resulting in the four-node PPN illustrated in Figure C.2. Analysis of the M-JPEG encoder shows that the most computationally intensive functional block in M-JPEG is the block performing the discrete cosine transform (DCT) transformation. This makes the DCT computation the bottleneck in the task-parallel PPN processing.

There are several levels of parallelism that we can exploit on a heterogeneous platform. As Figure 1.1 (Chapter 1) shows, the platform contains a number of components, such as a CPU with multiple cores and a GPU. By mapping each task on a

4.1. INTRODUCTION

different platform component we can exploit task-parallelism at the *platform-level*. In the context of PPNs, the platform-level parallelism is exploited by mapping coarse-grain tasks to different platform components for processing. By realizing the communication between concurrent tasks using FIFO buffers, we can also exploit pipeline parallelism at the platform-level. Moreover, since each platform component may contain additional parallel processing capabilities, we can also exploit different forms of parallelism within platform components, at the *component-level*. For example, CPUs contain multiple cores with cores often supporting vector instructions (SSE, AVX). In addition, GPUs offers unprecedented support for data parallel processing. Thus, there are different levels of parallelism and different types of parallelism to be considered (task, pipeline, data parallelism). It has already been demonstrated that finding data parallelism and offloading of the DCT computation to GPU is a worthwhile effort [101]. Let us assume that we want to bring the DCT processing to the GPU. The question that we address in this chapter is how to exploit the GPU to accelerate the DCT process, while still reaping benefits of task and pipeline parallel execution on the platform level.

```
1  TBlock block[VNumBlocks][HNumBlocks];
2
3  for (f = 0; f < NumFrames; f++) {
4      for (is = 0; is < VNumBlocks; is++)
5          ----- Encapsulation Boundary
6          for (js = 0; js < HNumBlocks; js++)
7              S: mainVIN(&block[is][js]);
8
9      for (it = 0; it < VNumBlocks; it++)
10         ----- Encapsulation Boundary
11         for (jt = 0; jt < HNumBlocks; jt++)
12             T: mainDCT(block[it][jt], &block[it][jt]);
13 }
```

Listing 4.1: An example of a P/C pair (M-JPEG Encoder)

Let us have a look at the code snippet in Listing 4.1 showing the first pair of producer-consumer (P/C) statements in M-JPEG. The way in which the program source code is specified leads to the task-parallel processing shown in Figure C.2. The video input process P_1 (the first node in Figure C.2) executes the function `mainVIN` in statement S sequentially on one microporcessor for all iterations of the loops (f, is, js) and the DCT process P_2 (the second node in Figure C.2) executes the function `mainDCT` in statement T sequentially on another microprocessors for all iterations of the loops (f, it, jt) . To offload DCT processing to the GPU, it is necessary to restructure the code by outlining (encapsulating) parts of the code that should be moved to the GPU and introducing novel data structures to send data to the new func-

tions obtained by outlining. Currently, the program restructuring must be done manually by the designer to obtain a multi-level program. As a consequence, creating different program variants requires repetitive manual modifications to the program code.

Instead, let us suppose that we could split the program model automatically in two parts, e.g. one part that can be executed on the platform-level, and the other part that can be executed on the component-level, e.g. on the GPU. We introduce the concept of the *encapsulation boundary* in the program model to support restructuring of program code into multiple levels by the compiler. The introduction of an encapsulation boundary enable us also to create independent program models that can be transformed and parallelized independently using different polyhedral compilers and tools. The encapsulation boundary illustrated by the dashed lines in Listing 4.1 splits the program code into two levels. As a consequence, the part of program code below the boundary that contains the call to `mainDCT` function enclosed in loop *jt* could be accelerated on the GPU. Let us call this part of the code a *loop subnest* $\mathcal{L}(jt)$.

Introduction of an encapsulation boundary would lead to the pseudocode given in Figure 4.1, where the new *main* (here denoted as *main'*) runs on a multicore processor, and where *S'* corresponds to the loop subnest $\mathcal{L}(js)$ and *T'* corresponds to the loop subnest $\mathcal{L}(jt)$ running on the GPU. To achieve this execution pattern, it is

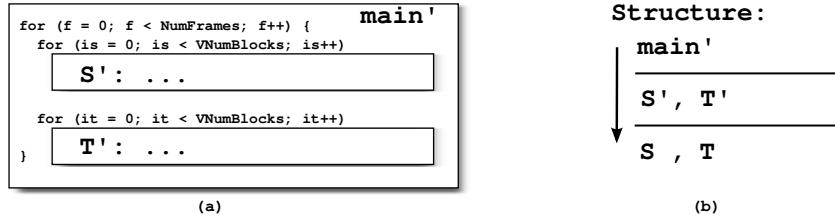


Figure 4.1: Goal: Program structured into two-levels.

essential to understand that the pseudocode in Figure 4.1 is structured into two levels of hierarchy. The new *main'* program invokes parts of the program code that are encapsulated into code boxes *S'* and *T'*. The code boxes correspond to the parts of the code that are outlined into separate function calls. We introduce a novel concept to refer to these code boxes - we call them *derived statements* *S'* and *T'* (see Subsection 4.3.3). The derived statement *S'* corresponds to the statement *S* and all surrounding loops below the encapsulation boundary (here: loop *js*). Similarly, derived statement *T'* corresponds to the statement *T* and all surrounding loops below the encapsulation boundary (here: loop *jt*). By introducing the notion of derived statements we obtain a program that is structured in two levels. At the top level, the main program executes the derived statements *S'* and *T'*. The implementation of derived statement *S'* now invokes the statement *S*, and the implementation of derived

4.1. INTRODUCTION

statement T' now invokes the statement T that corresponds to `mainDCT` function call. Introduction of the encapsulation boundary and restructuring of the code as described above creates the possibility to parallelize and map each derived statement independently, e.g. to derive CUDA code for statement T' executing DCT and offload it to the GPU. It also creates two novel questions: First, how do we define the derived statement T' ? Second, how do we know which data to pass as arguments to T' ?

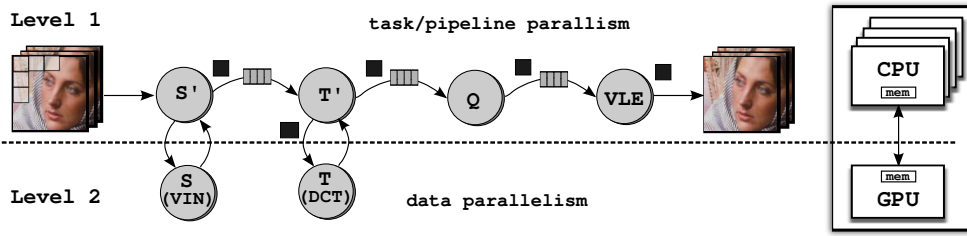


Figure 4.2: Two-level M-JPEG. The nodes at Level 1 execute in a task-parallel manner forming a streaming processing pipeline. The nodes at Level 2, e.g. the computationally intensive DCT, can be now transformed for data parallelism and executed on the GPU.

Using the multi-level parallelization approach, we can generate a PPN in which the node P_2 representing the DCT block now invokes derived statement T' instead of statement T . This is important since for the derived statement T' we have the complete polyhedral model and can map it on the GPU to take advantage of data parallelism, as illustrated in the lower part of Figure 4.2. The process P_2 sequentially executes the loops (f, it) that are above the encapsulation boundary as before, but the insertion of the encapsulation boundary and outlining make it possible to process all iterations of the loop jt below the encapsulation boundary in a data parallel manner and take advantage of the GPU. The result is a parallel program structured into two levels.

To improve the mapping of parallelism-rich multimedia streaming application onto heterogeneous platforms, we need what we call *multi-level programs* (MLPs). To take advantage of different architectural components and different forms of parallelism, we propose to transform and map each program component in a MLP on the desired component of the target platform. Thus, having a two-level MLP for M-JPEG would enable us to exploit task parallelism on the platform-level, and also to take advantage of the GPU for accelerating data parallel computations. The question that we address in this chapter is how to obtain such a multi-level program from the program's polyhedral representation in a structured way.

To generate a MLP without having to perform manual code modifications that include code restructuring, introduction of data structures, outlining (encapsulation),

and subsequent compiler reruns, having an intermediate program representation (IR) that captures the concepts of hierarchy and encapsulation would be highly advantageous. Contemporary state of the art compiler frameworks use the polyhedral model for internal representation of programs, but they lack these notions of hierarchy and encapsulation.

4.2 Solution Approach

To address the multi-level parallelization problem, we introduce a novel intermediate model for multi-level program representation and manipulation in the polyhedral framework. We named this intermediate representation *Hierarchical Polyhedral Reduced Graph* (HiPRDG) (see Section 4.4). The HiPRDG is derived from the Polyhedral Reduced Dependence Graph (PRDG). In addition, it captures the concepts of encapsulation and hierarchy. The HiPRDG provides a basis for compiler-assisted generation of modular, multi-level programs targeting multiple levels and multiple of parallelism on heterogeneous platforms. The workflow for compiler-assisted derivation of an MLP is shown in Figure 4.3.

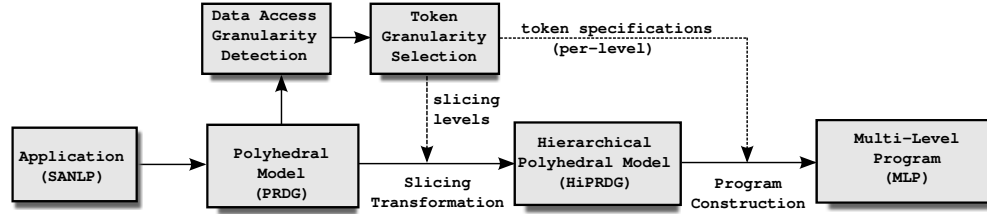


Figure 4.3: Derivation of a MLP from a SANLP.

A standard polyhedral representation is converted into the Hierarchical Polyhedral Reduced Dependence Graph by means of what we call a *Slicing Transformation* (see Section 4.5). The slicing transformation restructures the application’s polyhedral representation in form of the Polyhedral Reduced Dependence Graph into two or more levels. As a result, we obtain one or more components, which we encapsulate into separate HiPRDG nodes. Later on, each HiPRDG node is converted into a separate program module in the transformation and code generation stage.

To perform restructuring of the PRDG, we need to know where to introduce derived statements (See Subsection 4.3.3). The introduction of derived statements is guided by the placement of the encapsulation boundary, which could be for example manually inserted by designer as a compiler directive in program code (*pragma*) or generated by an auto-tuner and passed to the slicing stage. We denote the loop nesting level at which the encapsulation boundary is placed as *slicing level* (See Sub-

4.2. SOLUTION APPROACH

section 4.3.3). The decision at which level to slice the program model is guided by the designer's selection of token granularities processed at each level of the program. To select the token granularity and thus to decide where to slice the program model, it is necessary to find out what are the data access granularities at which various parts of the program operate.

The data access granularity of a part of a program, i.e. a loop subnest, corresponds directly to the memory footprint of that loop subnest. Let us have a look at the M-JPEG code snippet in Figure 4.4, and focus on the granularity of the access reference to the `block` variable. Statement S produces one block, i.e. a single element of the `block` array. Thus, the access granularity of statement S are tokens corresponding to single blocks. However, if we look at loop subnest $\mathcal{L}(js)$ we can see that each iteration of the loop js accesses a block from a row `block[is]` of the `block` array. The memory footprint of the loop subnest $\mathcal{L}(js)$ corresponds to a row of the array `block`. This means that the data access granularity of the loop subnest $\mathcal{L}(js)$ is a one-dimensional block array with `HNumBlocks` elements. Thus, the access granularity of loop subnest $\mathcal{L}(js)$ are tokens corresponding to entire rows. Analogously, the loop subnest $\mathcal{L}(is, js)$ accesses the whole two-dimensional array. Thus, the access granularity of loop subnest $\mathcal{L}(is, js)$ are tokens corresponding to entire frames, as illustrated in Figure 4.4.

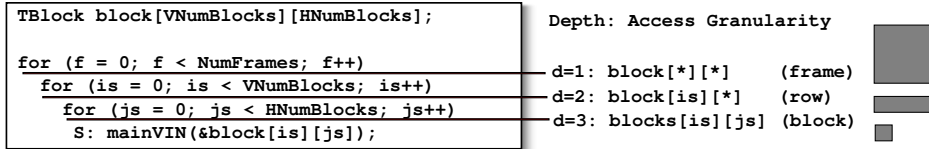


Figure 4.4: Access Granularities in M-JPEG code snippet.

By changing the placement of the encapsulation boundary, we determine the amount of data that is processed on GPU in one T' execution. Instead of being able to process only a single block in one GPU call, restructuring of the program enables us to process a row or a complete frame in a single T' call. This approach results in better GPU utilization because a typical GPU has 14–16 SMPs and processing only a single block per GPU kernel utilizes only a single SMP, while other SMPs idle. In addition, it is not always desirable to parallelize the application using the token granularity implicitly specified by the program source code. The optimal token granularity is highly dependent on the target architecture and the application. For example, by processing a single block on the GPU we would pay a high performance penalty: processing only a single block on a GPU is not efficient. Transferring small data packages over DMA channels also leads to high data transfer overheads. Instead, we need a more flexible token representation.

Knowing the data access granularities enables us to introduce data structures that facilitate structuring the program into multiple levels. This enables us to answer the question from Section 4.1 on which data to pass to derived statement S' that executes code derived from loop subnest $\mathcal{L}(js)$. For example, let us suppose that the program is split into two levels by introducing an encapsulation boundary at level $d = 2$, just above the loop js , and that we outlined the loop subnest $\mathcal{L}(js)$ into a function. The outlined function is invoked by the derived statement S' . The memory

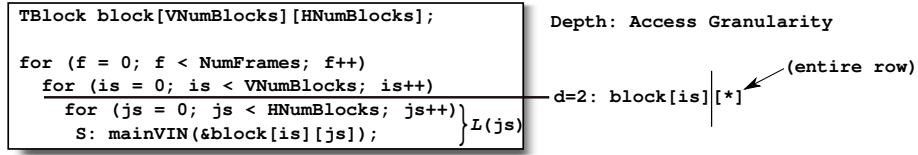


Figure 4.5: Access granularity at depth $d = 2$ corresponds to the entire row (denoted as `block[is][*]`) of the two-dimensional array `block`.

footprint of loop subnest $\mathcal{L}(js)$ corresponds to a row of a frame, i.e. one dimensional array of blocks. Thus, we need to create tokens corresponding to the rows of the frame, and pass them to derived statement S' that executes $\mathcal{L}(js)$. We do this by splitting the original two-dimensional variable `block[VNumBlocks][HNumBlocks]` of data type `TBlock` into two data structures. The first data structure represents the lower dimension of `block` which is indexed by js . and the second data structure represents the higher dimension of `block` which is indexed by is . Splitting of the `block` variable is illustrated it in Figure 4.5. First, to encapsulate the lower dimension of the `block` variable, we generate a new structure `TBlockRow` that represents a row of blocks. Second, we generate a new definition of a program variable `block` to encapsulate the lower dimension of the array. The result is the new variable `block'` that corresponds to a one-dimensional array of rows, i.e. `TBlockRow` elements. The variable `block'` is the *composite* representation of array elements `block[is][js]`. Each element of `block'` is a composite token (see Section 4.3.2) containing multiple elements of the `block` array.

As a result, we obtain the data structures and modified program code using the new data structures shown in Figure 4.6. The top level of the program works on the array of newly defined `TBlockRow` data type. This enables us to pass tokens of type `TBlockRow` to derived statement S' . Since we have not modified the function invoked by the derived statement, it is necessary to copy blocks from the row-token (`TBlockRow`) into the local array of blocks used by the function, and vice versa. We complete the program wiring by inserting `COPYIN` and `COPYOUT` statements. The prototype methods from conversion of composite tokens into data space elements (`COPYIN`) and the other way round (`COPYOUT`) are given in Section 4.6.1. These

4.2. SOLUTION APPROACH

```

//TBlock block[VNumBlocks][HNumBlocks];
struct TBlockRow {TBlock block[HNumBlocks]};
TBlockRow block'[VNumBlocks];

for (f = 0; f < NumFrames; f++)
  for (is = 0; is < VNumBlocks; is++)
    S'(&block'[is]);          d=2: block'[is] (one row)

```

Figure 4.6: Code after granularity selection and data structure generation.

statements do the automatic conversion from one data granularity into another granularity. The COPYIN statement copies the blocks from the input row-token into the local block array, and the COPYOUT statement copies the blocks from the local block array into the output row-token.

<pre> typedef TBlock token_t0_L1; struct token_t0_L0 rowToken{ token_t0_L1 block[HNumBlocks]; }; token_t0_L0 rowToken; main for (f = 0; f < NumFrames; f++) { for (is = 0; is < VNumBlocks; is++) DS: X(&rowToken); for (it = 0; it < VNumBlocks; it++) DT: Y(&rowToken, &rowToken); } </pre> (a)	<pre> program module X: DS //process statement S (mainVIN) in js forall (jt = 0 to HNumBlocks-1) S: mainVIN(&block[jt]); COPYOUT(blocks into rowToken) program module Y: DT COPYIN(rowToken into blocks) //process statement T (mainDCT) in jt forall (jt = 0 to HNumBlocks-1) T: mainDCT(block[jt], &block[jt]) COPYOUT(blocks into rowToken) </pre> (b)
---	---

Figure 4.7: Resulting multi-level program data types and modules.

From the M-JPEG SANLP in Listing 4.1 our goal is to generate a multi-level parallel program, e.g. as the program illustrated in Figure 4.7. In the remainder of this chapter, we do this by introducing the hierarchical intermediate program representation into the polyhedral model, a transformation to derive this hierarchical IR from the standard polyhedral program model, and a novel method for MLP construction from a hierarchical IR. We first address the prerequisites for the hierarchical IR construction, i.e. data space representation, token granularity, and definitions of levels, in the preliminaries (Section 4.3). Second, we introduce our hierarchical IR called Hierarchical Polyhedral Reduced Dependence Graph (HiPRDG) in Section 4.4. Third, we give a structured method for HiPRDG derivation from the standard polyhedral representation in Section 4.5. In Section 4.6, we describe steps in generation of a multi-level program from the HiPRDG. Finally, we demonstrate on the M-JPEG example how to obtain a two-level program featuring task, data, and pipeline parallelism.

4.3 Preliminaries

4.3.1 Data Space Representation in Polyhedral Model

Let us recall the notation introduced in Chapter 2.1. A Static Affined Nested Loop Program (SANLP) program $\mathcal{P}$ is a program composed of s statements $S_1, S_2, \dots, S_s$. In the polyhedral model of program $\mathcal{P}$, each statement S has an associated iteration vector $\vec{x}_S$ which takes values in its iteration domain $\mathcal{D}_S$ by following the sequential (lexicographical) execution order, or the execution order specified by some program transformation (also called scattering). The iteration domain $\mathcal{D}_S$ is an abstract representation of the loops that surround statement S in the program source code. For example, the iteration domain of a statement S surrounded by a doubly nested loops with iterators i and j is represented by a two-dimensional polyhedron with iteration vector $\vec{x}_S = (i, j)^T$.

To reason about data access granularity, we extend the polyhedral model described in Section 2.1 with abstractions for several data-related concepts that occur in a SANLP, such as different array definitions, array indexing functions, and mapping between iteration vector of a statement and index vector of a data space. In line with Anderson et al. [8], we represent an m -dimensional array A as an m -dimensional polytope whose boundaries are given by the array bounds. The polyhedral model of a program variable is the basis for our definition of a data space:

Definition 29 (Data Space)

Data space $\mathcal{D}_A$ is a representation of a program variable, such as an 2-dimensional array A of integer elements, in memory. We define data space $\mathcal{D}_A$ as a tuple $(D_A, \epsilon, \vec{s})$, where

- *D_A is a polyhedral representation corresponding to a program variable A , which can be for example a two-dimensional array of integers `intA[M][N]`. D_A is an m -dimensional polyhedron bounded by linear inequalities derived from the array A dimensions.*
- *$\vec{s}_A$ is a m -dimensional vector representing the sizes of the program variable A in each dimension, e.g. $\vec{s}_A = (M, N)$.*
- *ϵ is the type of a unit element of the program variable A , e.g. `int` data type.*

The dimensionality of data space A corresponds to the dimensionality of the program variable. A scalar variable is treated as a 0-dimensional space, with a single element. The unit element type in the example above is `int`, however it can be as well a composite data type, such as an array.

The values of array indices in a memory reference determine which array *element* is accessed. We make this notion explicit in form of a *data space index vector*.

4.3. PRELIMINARIES

Definition 30 (Index vector of a data space)

The index vector of a data space is an integral vector consisting of all index dimensions of the array represented by the given data space, from the outtermost to the innermost. With $\vec{x}_A = (x_1, x_2, \dots, x_m)^T$, we denote the index vector $\vec{x}_A$ of the data space representing m -dimensional array A .

The index vector of a data space $\mathcal{D}_A$ is used for indexing the elements of array A . An access to array A is denoted as $A[\vec{x}_A]$. Arrays are assumed to be stored in a *row-major*¹ order, i.e. in a two-dimensional array A , index x_1 address a row of array A , and index x_2 address a column in the selected row.

For a reference r to an m -dimensional array A in statement S , we define a *projection* $\mathcal{F}_{S,r}^A : \mathcal{D}_S \rightarrow \mathcal{D}_A$:

$$\vec{x}_S \in \mathcal{D}_S, \vec{x}_A \in \mathcal{D}_A, \vec{x}_A = F_{S,r}^A \vec{x}_S \quad (4.1)$$

where $F_{S,r}^A$ is an $m \times (\dim(\vec{x}_S) + \dim(\vec{n}) + 1)$ matrix representing a homogenous affine mapping from reference r to array A in the operation $S(\vec{x}_S)$ to the element of the data space of array A . The dimensionality of affine mapping $F_{S,r}^A$ corresponds to the dimensionality of array A , with i th row of $F_{S,r}^A$ being used to index the i th-dim of array A , and n being the number of program parameters.

```
int A[4][4];
for (i = 1; i < 4; i++)
  for (j = 1; j < 4; j++)
    S: A[i][j] = A[i][j-1] + 1;
```

Figure 4.8: Example SANLP.

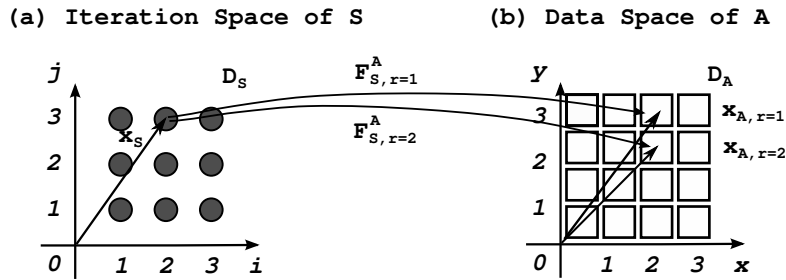


Figure 4.9: Mapping Iteration Space of Statement S onto Data Space of Array A .

¹In the row-major order, an array is stored row-by-row, while in the column-major order an array is stored column-by-column. While FORTRAN uses the column-major order, most of the other programming languages, such as C, assume the row-major order.

illustrates the projection $\mathcal{F}_{S,r}^A$ for the code snippet The projection $\mathcal{F}_{S,r}^A$ for the code snippet in Figure 4.8 is shown in Figure 4.9. The iteration domain $\mathcal{D}_S$ of statement S in Figure (a), and the data space polyhedron $\mathcal{D}_A$ representing array A is shown in Figure (b). Let us consider accesses to array A in statement S for the following values of for-loop iterators: $i = 2, j = 3$, i.e. $\vec{x}_S = (2, 3)^T$. The first reference $r = 1$ to array A in statement S accesses the element $A[2][3]$, and the second reference $r = 2$ to array A in statement S accesses the element $A[2][2]$. The projection of iteration vector $\vec{x}_S = (2, 3)^T$ from data space $\mathcal{D}_S$ is illustrated for two references to A , i.e. $A[i][j]$ representing the first reference $r = 1$ and $A[i][j - 1]$ representing the second reference $r = 2$ via the arrow that maps an iteration point in Figure 4.9(a) to a data space element in Figure 4.9(b).

4.3.2 Encapsulation Support: Composite Tokens

Encapsulation of program parts into independent modules requires a communication mechanism between modules to be in place. In the dataflow and process networks models of computation, the communication between independent modules is realized via tokens. For example, the PPN processes in the M-JPEG PPN shown in Figure C.2 pass through the pipeline tokens corresponding to image blocks. We leverage the concepts of token-based communication developed in process networks to support the encapsulation which is necessary for multi-level program generation.

Nikolov [99] defines a token as a packet of data that can represent any type of information. In practice, the polyhedral tools exclusively perform analysis on the tokens corresponding to the elements of user defined data types in the program code, such as elements of the block array. When M-JPEG is specified as in Listing 4.1, the token type corresponds to the TBlocks data type, i.e. a token is always an image block.

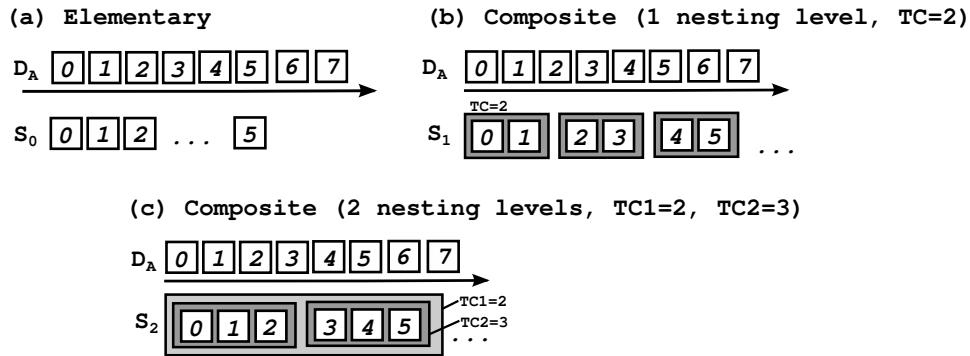


Figure 4.10: Multiple ways of composing tokens from elements of data space D_A

4.3. PRELIMINARIES

An *elementary token*, as used in dataflow analysis [55, 59, 78], and contemporary PPN generation tools [2, 81], is a unit element of a data space $\mathcal{D}_A$. We define a *composite token* as a data token containing multiple unit tokens. A unit token is either an elementary token or a composition of tokens (a *composite token*). We denote the number of unit tokens in a composite token as *token cardinality* (TC). We illustrate the composite token concept in Figure 4.10. Data space D_A containing unit elements with ids from 1 to 7 can be streamed in multiple ways. In Figure 4.10(a), we show the stream S_0 of elementary tokens obtained by streaming elements of data space D_A . Figure 4.10(b) shows the stream S_1 of composite tokens with token cardinality $TC = 2$. Each composite token contains 2 elementary tokens corresponding to elements of data space D_A . There is one nesting level. Figure 4.10(c) shows the stream S_2 of composite tokens with two levels of nesting. Each composite token is composed of two unit tokens. Each unit token is composed of three elementary tokens corresponding to elements of data space D_A . There are two nesting levels with token cardinalities $TC_1 = 2$ and $TC_2 = 3$.

Definition 31 (Token, Composite Token, Token Cardinality)

An *elementary token* corresponds to a unit element of a data space $\mathcal{D}_A$. A *composite token* is a composite d_t -dimensional data structure containing a finite number of unit tokens. A unit token can be an elementary token or a composite token itself. The **token cardinality** (TC) is the number of unit tokens in the composite token. We define a composite token as a tuple $(\epsilon, d, \vec{s}, m)$, where

- ϵ is the data type of unit elements of this token. The unit elements can be data elements of a primitive data type, or composite tokens.
- d is the dimensionality of the token data space, e.g. a composite token can represent a two-dimensional data space.
- $\vec{s} = (s_1, s_2, \dots, s_d)^T$ is a d -dimensional vector representing the token size across different dimensions in terms of unit elements.
- TC is the number of tokens contained in the composite token:

$$TC = \prod_{i=1}^d s_i$$

4.3.3 Introducing Concepts of Depth (Level) and Derived Statements

In software design, programs are structured as a set of program modules that are called from various places in the program. Nesting of function calls is typically allowed in programming languages to structure the program and to foster reuse. For

each function call it is possible to determine its nesting depth by starting at the program *main*, and counting the number of enclosing function calls.

Inspired by this approach, we leverage the notion of *nesting level*, i.e. depth, to derive a hierarchical program model out of the standard polyhedral program model. Instead of counting the number of enclosing calls, we count the number of enclosing for loops. The depth of a for loop i in a loop nest $\mathcal{L}$ takes values in $[1 \dots d_{\mathcal{L}}]$, where $d_{\mathcal{L}}$ is the number of nested loops in $\mathcal{L}$ and corresponds to the number of enclosing loops plus one. In the M-JPEG snippet in Listing 4.1, the first for loop with iterator f is located at depth 1, the for loops with iterators is and it are located at depth 2, and the for loops with iterators js and jt are located at depth 3. The number of enclosing loops in the program code corresponds to the number of iteration space dimensions in the polyhedral model. In traditional compiler analysis, the nesting level l_i (i.e. its depth in the loop nest) of some loop i in loop nest $\mathcal{L}$ is defined as the number of the enclosing loops plus one [6]. The concept of a nesting level (or simply a level) is equal to the concept of depth. We will use these two terms interchangeably through the thesis.

Let us refine this definition for the analysis of imperfectly nested SANLPs in the polyhedral model. A statement in a SANLP with imperfectly nested loops is fully defined by its iteration domain *and* its position (i.e. the textual order) within the loop nest. However, the information on the textual order is not explicitly included in our tools. Currently, the iteration domain of the statement is captured in its iteration vector. Iteration vector components represent for loop counters. In addition, we need to include the position of a statement within the loop nest in our model in order to reason about encapsulation at different levels (depths). In the literature, different ways are used to address the textual order, e.g., by introduction of minor dimensions into iteration vectors of the statements [29]. In order to minimize the modification required to the existing toolset, we opted for the addition of a simple textual order encoding in the representation of the depth, which will be explained below.

Statement level In classical compiler analysis, the statement level denoted as l takes values in the range $l \in [1 \dots n_S]$. In the pseudocode given in Listing 4.11(a), both statements S and statements T are found at level 2.

The question that arises in analysis of programs with imperfectly nested statements is how to represent the statement level in the situations when one or more statement are encapsulated by the same loop. This situation is illustrated in Listing 4.11(b). Both statements S and T are enclosed by the same two nested loops i and j , and following the approach above, they would both be assigned statement level $l = 2$. As a consequence, it is not possible to differentiate between the case A in Listing 4.11(a) and the case B in Listing 4.11(b). Solving this problem requires encoding of the information on the textual order of the statements in the model. We approach this by

4.3. PRELIMINARIES

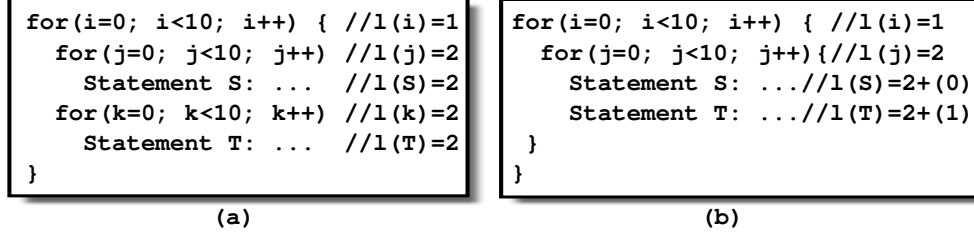


Figure 4.11: (a) Standard statement level $l = 2$, (b) Statement level is extended with information on textual order (line numbers).

extending the range of the statement level to $l \in \{1 \dots n_S\} \cup \{n_S^+\}$. The special value n_S^+ encodes the presence of the textual order at some loop level n . Using the proposed encoding scheme, statements S and T from Listing 4.11(b) are now assigned the level $l = 2+$, which makes it possible to differentiate it from the case in Listing 4.11(a).

Derived statement To describe the notion of encapsulation and hierarchy in the polyhedral model, we introduce the concept of a *derived statement*. Essentially, a derived statement is a "stand-in" for some part of program code. The derived statement stands in for the code in a loop subnest. For example, derived statement S' in Figure 4.1 substitutes the loop subnest that encloses the for loop js and the statement S at lines 6-7 of Listing 4.1.

A derived statement is obtained by outlining some part of the program code and encapsulating it into a function. The details of the outlining and encapsulation are addressed later in Section 4.6. Let us now illustrate the concept of the derived statement on the M-JPEG code snippet.

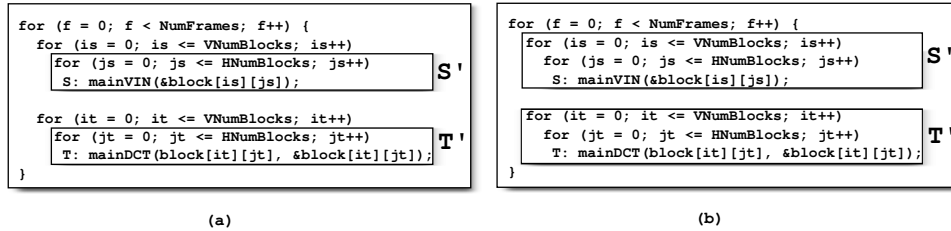


Figure 4.12: Derived statements S' and T' encapsulate parts of program code.

In the M-JPEG snippet in Listing 4.1, we could derive statements S' and T' by encapsulating the statements S and T surrounded by the loops below the encapsulation boundary as illustrated in Figure 4.12(a). As a result, derived statement S' encapsulates statement S , and derived statement T' encapsulates statement T . Moving up the

encapsulation boundary one level up in Listing 4.1 would result in different definition of derived statements S' and T' , as illustrated in Figure 4.12(b).

In both cases, the statement level is computed for derived statements in the same way as for the standard statements. As a result, we assign level $l = 2$ to derived statements S' and T' in Figure 4.12(a), and level $l = 1+$ to in Figure 4.12(b).

Dependence level Allen and Kennedy [6] define the dependence level $dl(e)$ of an edge $e : S(\vec{x}_S) \Rightarrow T(\vec{x}_T)$ as the depth of the first (outermost) loop for which the iterator values are different. The dependence is said to be *carried* at depth k of a SANLP, if its dependence level equals k , i.e. $dl = k$. The dependence level of a *loop-carried* dependence takes a value from 1 up to the total number of common loops for two dependent statements, denoted as $n_{S,T}$.

A special case is loop-independent dependence which stems from the lexicographical order of statements S and T , when they are one below the other at the same loop level. To encode the level of dependence for a dependence imposed by the *textual order* of statements at depth $n_{S,T}$, we use the special value $n_{S,T}^+$. The zero valued dependence level $dl = 0$ is assigned in the case of two dependent operations that belong to statements in different loop nests. In this case, the dependence is carried at the top-level of the program. The valid range of dependence levels is in the set $\{0, 1, \dots, n_{S,T}\} \cup \{n_{S,T}^+\}$. In the M-JPEG example, there is a dataflow dependence between the write access in statement S and the read access in statement T to the shared variable `block`. The common nesting level of the two statements is $n_{S,T} = 1$. The dependence is imposed by the textual order of program blocks denoted as S' and T' in Figure 4.12(b). Thus, the level of the dependence is $dl(e) = 1+$.

Deriving dependence level from mapping The dependence level of a dependence edge can be derived from its mapping. The linear equality in the mapping specifies the affine relation between the pair of P/C operations, i.e. it defines the form of the distance vector $\vec{d} = \vec{x}_T - \vec{x}_S$ between two dependent operations, s.t. $S \Rightarrow T$. In the COMPAAN compiler, the iteration vector of the consumer statement is expressed as a function of the producer statement's iteration vector, i.e.:

$$\begin{aligned} \vec{d} &= \vec{x}_T - x_S \\ \vec{d} &= \vec{x}_T - M\vec{x}_T \end{aligned} \tag{4.2}$$

where M is the affine mapping matrix that maps the iteration vector of dependence's target operation to the iteration vector of dependence's source operation.

The dependence level dl of an edge e corresponds to the first positive component of the distance vector between the pair of P/C statements. Let us illustrate this on a simple example in Listing 4.2.

4.4. HIERARCHICAL POLYHEDRAL REDUCED GRAPH (HIPRDG)

```

1  for (int i=0; i<M; i++)          //l=1
2    for (int j=1; j<N; j++)        //l=2
3      S: A[i][j] = A[i][j-1] + ...;

```

Listing 4.2: Example of a loop-carried dataflow dependence from the write access to A in the previous iteration $(i, j - 1)$ to the read access in the current iteration (i, j) .

Each iteration of the loop j depends on the value producer in the previous iteration of the loop j , i.e. there is a dataflow dependence from each iteration (i, j) to the next iteration of j , i.e. $(i, j + 1)$ via program variable A. The P/C mapping between two dependence operations expressed in the iteration vector of the consumer operation is $S(i, j - 1) \Rightarrow S(i, j)$. The affine mapping corresponds to the distance vector, i.e.

$$\vec{d} = (i, j)^T - (i, j - 1)^T = (0, 1)^T$$

The first positive component of the distance vector is the 2nd component (counting the components from one), which means that the dependence level is $dl = 2$. Thus, the dependence is carried at depth 2 (corresponding to the j loop).

If the statements have at least one common loop, i.e. $n_{S,T} \geq 1$, but the distance vector is 0-valued, the dependence is loop-independent and its direction is determined by the textual order of statements S and T in the code block surrounded by $n_{S,T}$ loops. According to the definition above, we assign dependence level $dl(e) = n_{S,T} +$ to edge $e : S \Rightarrow T$.

Finally, if there is a dependence, but statements S and T do not share any common loops, this is a special case of a top-level dependence, where $dl(e) = 0$.

4.4 Hierarchical Polyhedral Reduced Graph (HiPRDG)

To capture the concept of a **hierarchical intermediate representation** for modelling **structured, multi-level programs** in the polyhedral framework, we introduce an intermediate representation which we called Hierarchical Polyhedral Reduced Dependence Graph (HiPRDG). A HiPRDG is a connected, acyclic graph with a root node, i.e. a tree. A HiPRDG consists of a set of nodes and a set of edges. The root node of the HiPRDG is used to generate the body of the *main* function in a multi-level program, and the nodes of the HiPRDG are used to generate program modules. The generation of a multi-level program from a HiPRDG is covered in Section 4.6. Let us first explain what a HiPRDG is, and illustrate the key concepts on a simple example.

As an extension to the standard polyhedral representation, the HiPRDG provides the possibility to "zoom" into its nodes. Let us explain this fundamental idea behind the HiPRDG on the simple two-statement example shown in Figure 4.13. The SANLP in Figure 4.13(a) is again the code snippet of the selected M-JPEG producer-consumer

4.4. HIERARCHICAL POLYHEDRAL REDUCED GRAPH (HIPRDG)

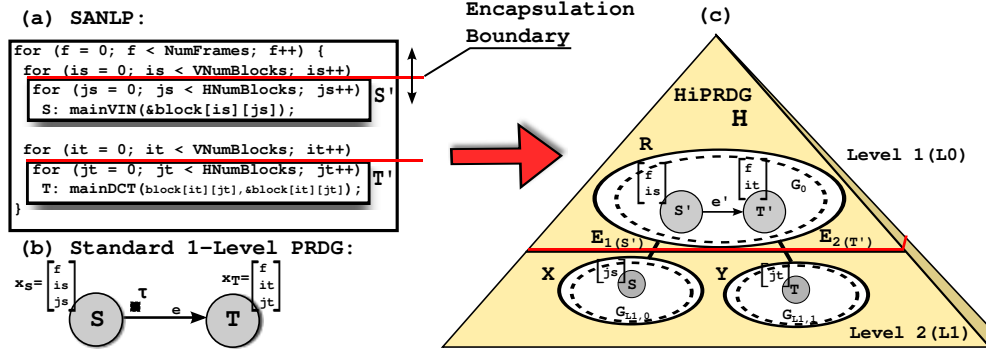


Figure 4.13: (a) The M-JPEG code snippet, (b) Its standard polyhedral model, (c) A two-level HiPRDG (See also Figure 4.2).

pair which we use as the running example through the chapter. Figure 4.13(b) shows its Polyhedral Reduced Dependence Graph (PRDG). It has two nodes with three dimensional iteration domains connected with one edge representing the dataflow between statements S and T . This PRDG is the basis for the construction of the HiPRDG shown in Figure 4.13(c). The HiPRDG H in Figure 4.13(c) came to existence as a hierarchical representation of the program model obtained by introducing derived statements S' and T' , illustrated by overlaid code boxes below the encapsulation boundary at depth $l = 2$ in Figure 4.13(a). Placing of the encapsulation boundary in the standard polyhedral model results in splitting of the model into two layers, which yields a two-level HiPRDG graph H depicted in Figure 4.13(c). Each node of a HiPRDG is annotated with a fully-fledged polyhedral model, i.e. a PRDG. For example, the HiPRDG root node R is annotated with PRDG G_0 , and the HiPRDG leaves X and Y are annotated with graphs $G_{L1,0}$ and $G_{L1,1}$ respectively. The nodes representing derived statements S' and T' are found in PRDG G_0 . The two derived statements are defined by PRDGs $G_{L1,0}$ and $G_{L1,1}$. The PRDG G_0 of the root node R is illustrated in Figure 4.14(b). The two-node PRDG G_0 corresponds to the SANLP in Figure 4.14(a). This SANLP actually corresponds to the original M-JPEG example if we replace all code below the encapsulation boundary with calls to the derived statements S' and T' . As a consequence, the nodes of the PRDG in Figure 4.14(b) have two-dimensional iteration domains, containing only for loop iterators above the encapsulation boundary in Figure 4.13(a).

In Figure 4.13(c), the root node R of the HiPRDG is annotated with a graph G_0 . The PRDG G_0 has two nodes representing derived statements S' and T' in iteration space generated from for loops (f, is) and (f, it) respectively. Zooming into each derived statement of the PRDG G_0 reveals its polyhedral model, i.e. the definition of the derived statement. The derived statements S' and T' encapsulate the innermost loops

4.4. HIERARCHICAL POLYHEDRAL REDUCED GRAPH (HIPRDG)

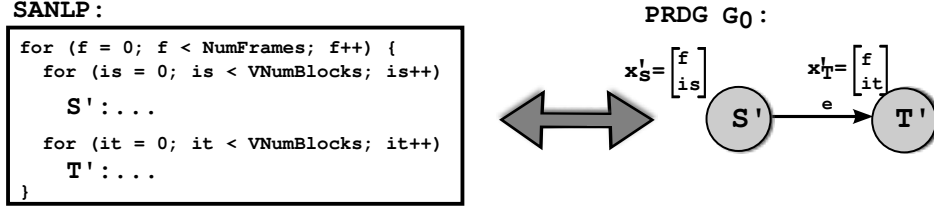


Figure 4.14: (a) The M-JPEG code snippet with code below the encapsulation boundary replaced by derived statements S' and T' , (b) The PRDG representation.

js and jt surrounding the statements S and T from the program source. Since in this example there are no loop-carried dependencies in these loops, the PRDGs that define derived statements S' and T' are rather simple, namely each of them contains a single node with one-dimensional iteration domain specification, but in the general case can be arbitrary PRDGs. The PRDG of derived statement S' is depicted as $G_{L1,0}$ within the HiPRDG node X at level $L1$ in Figure 4.13(c), and it contains a single node that represents the unmodified statement S and has a one-dimensional iteration domain describing the innermost for loop js . Similarly, the PRDG of statement T' is depicted as $G_{L1,1}$ within the HiPRDG node Y at level $L1$ in Figure 4.13(c), and it contains a single node that represents statement T and has one-dimensional iteration domain describing the innermost for loop jt .

In Figure 4.13(c), the HiPRDG edge E_1 connects the HiPRDG nodes R and X . The graph in node X is the definition of the derived statement invoked by PRDG node S' . The edge E_1 thus leads to the definition of the derived statement of S' which is the PRDG in node X . Since node R is not atomic, i.e. it contains the PRDG G_0 which in turn contains PRDG node S' , it is necessary to indicate for each HiPRDG edge which PRDG nodes it is used to define. To express this relation we annotate each HiPRDG edge with the identifier of the PRDG node that it defines, which in this case results into the edge $E_1(S')$. Similarly, the HiPRDG edge $E_2(T')$ represents the definition of derived statement T' within HiPRDG root node R with the PRDG $G_{L1,1}$ that is used to annotate the HiPRDG child node Y . Hence, there is a HiPRDG edge between a parent and a child node of a HiPRDG for each *define* relationship between a statement within a PRDG annotating the HiPRDG parent node and the PRDG of its HiPRDG child node. Let us now formalize this discussion:

Definition 32 (Hierarchical Polyhedral Reduced Dependence Graph (HiPRDG))

*A **Hierarchical Polyhedral Reduced Dependence Graph** $T = (V, E)$ is an acyclic, undirected multigraph that consists of a set of vertices V (nodes) and a set of edges E .*

4.4. HIERARCHICAL POLYHEDRAL REDUCED GRAPH (HIPRDG)

- The HiPRDG has a designated root node R .
- Each HiPRDG node is annotated with a Polyhedral Reduced Dependence Graph (Definition 13). The PRDG consists of a set of nodes and a set of vertices. Each node of a PRDG represents a statement of the original program.
- A HiPRDG edge denotes a define relationship between a statement within the PRDG annotating the HiPRDG parent node at one level of the HiPRDG, and a PRDG annotating the HiPRDG child node at the next level of the HiPRDG. Each HiPRDG edge is annotated with the unique identifier of the statement that is being defined by the child's PRDG,

where a statement could be a simple program statement, or a derived statement introduced in Section 4.3.3.

The number of levels in a multi-level program generated from a HiPRDG directly corresponds to the HiPRDG depth.

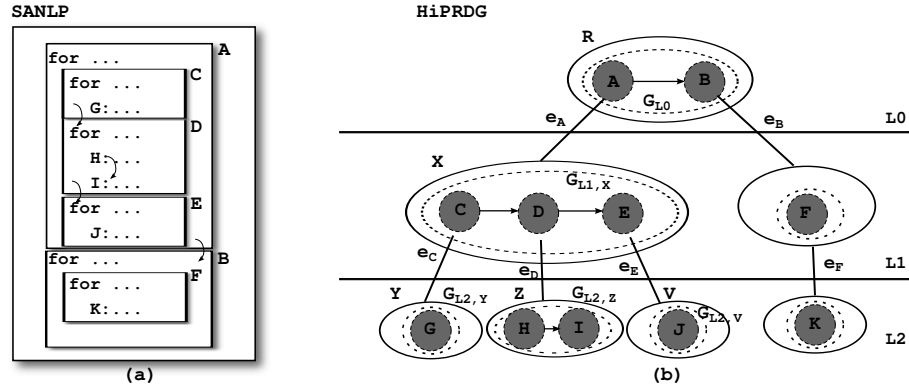


Figure 4.15: A Multi-Level Program and its HiPRDG.

A more complex example of a HiPRDG is depicted in Figure 4.15(b). The HiPRDG R in Figure 4.15(b) represents three levels of a multi-level program: the top-level L_0 that contains the root node R , the middle-level L_1 that contains HiPRDG nodes annotated with definitions of the statements A and B in root node R 's graph G_{L_0} , and the bottom-level L_2 . Let us consider node X at level L_1 , which is annotated with PRDG $G_{L_1,X}$. This PRDG contains three nodes, representing statements C , D , and E . The three statements are defined by PRDGs annotating the nodes Y , Z , and V at HiPRDG level L_2 . The leftmost child of X , i.e. node Y , defines statement C in X 's graph $G_{L_1,X}$. As a consequence, in the code generation phase statement C is substituted with a call to the function that encapsulates code generated from PRDG $G_{L_2,Y}$ which

annotates the child node Y . As an illustration, an abstract pseudocode structure with three nesting levels corresponding to this HiPRDG is shown in Figure 4.15(a).

4.5 The Slicing Transformation

The HiPRDG model can be obtained by a set of structured *transformations in the polyhedral model*. In this section, we present the novel *slicing* transformation for derivation of a HiPRDG from a standard PRDG of an application. To perform the slicing transformation, we need the program's polyhedral model in the form of a PRDG and the information at which depths to insert encapsulation boundaries and generate derived statements. This information is passed to the slicing transformation as a list of slicing levels. In this section, we will show how the slicing transformation works for a single level. The method is general enough to support slicing for multiple levels. In case that we are given multiple slicing levels, we simply repeat the method until a multi-level HiPRDG is derived.

Slicing at *slicing level* sl means that all program model components of the polyhedral program model are assigned to PRDGs at different levels of the hierarchical program model. By *program model components* we refer to statements and their iteration domains, iteration vectors, and dependences. Essentially, loops above the slicing level sl are assigned to a higher level of the hierarchical model denoted here as $\mathbf{L}_h$, whereas loops strictly below the slicing level sl , i.e. the loops with depths $d > sl$ are assigned to the lower level of the hierarchical model denoted here as $\mathbf{L}_l$. Given a polyhedral program model, the slicing of the model into two layers is performed by comparing the level of each model component with the selected slicing level. In the discussion that follows, we use the following terminology for the comparison of levels:

- **Case A:** component level l_m is above or at the slicing level sl , i.e. $0 \leq l_m \leq sl$
- **Case B:** component level l_m is strictly below the slicing level sl , i.e. $sl < l_m \leq l_n$

where l_n is the total number of levels. Let us illustrate the slicing transformation on a P/C pair of statements shown in Figure 4.13(b). This model contains two PRDG nodes S and T . The two nodes are connected via dependence edge e that represents a dataflow dependence induced by the read-write accesses from the statements S and T to the program variable `block[j][i]`. The iteration vector of node S is $\vec{x}_S = [f, is, js]^T$, and the iteration vector of node T is $\vec{x}_T = [f, it, jt]^T$. The common nesting level (See Definition 11) of statements S and T is $n_{S,T} = 1$. The iteration vectors representing common dimensions are $\vec{x}_S^c = [f]$ and $\vec{x}_T^c = [f]$, which leads to the distance vector $\vec{d} = \vec{x}_T^c - \vec{x}_S^c = \vec{0}$. Zero-valued distance vector means that the dependence is induced by the textual order of statements at the common loop nesting level

$n_{S,T} = 1$. According to the textual order encoding scheme, the dependence level (See Subsection sec:hiprdg:levels) for this dependence is $dl(e_{S,T}) = 1+$. The difference between the values 1 and 1+ is relevant for the proposed slicing and encapsulation, as it affects how much code is going to be encapsulated.

4.5.1 Node Splitting

The first step of the slicing transformation consist of splitting the PRDG nodes at the slicing level sl . Splitting into two levels is realized with the construction of two nodes out of a single PRDG node, and adjusting the iteration domains, iteration vectors, and statements in the two nodes. After splitting one node contains only the "lower" iteration space dimensions of node S , while the other node contains only the "higher" iteration space dimensions of node S . This is illustrated in detail in Figure 4.16 for node S . The PRDG node S is split into its higher part S^H and its lower part S^L as

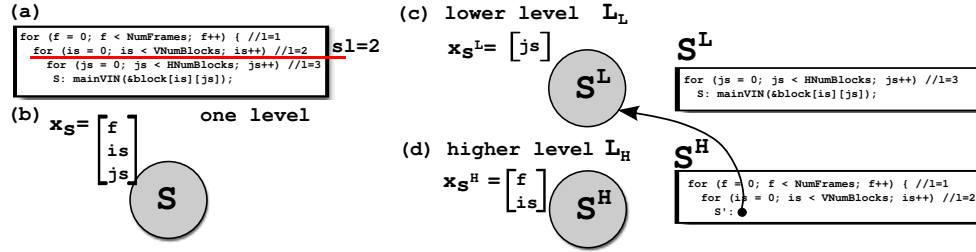


Figure 4.16: Splitting of PRDG node S at slicing level $sl = 2$.

follows:

- Node S^H with the iteration vector $\vec{x}_{S^H} = [f, is]^T$
- Node S^L with the iteration vector $\vec{x}_{S^L} = [js]^T$

where $\vec{x}_S = [f, is, js]^T$ is the iteration vector of the statement S in the SANLP. The iterators denoting node dimensions above the slicing level $sl = 2$, i.e. iteration vector components from the 1-st to the 2-th dimension (loops f and is) are assigned to the iteration domain of the node S^H in the higher level of hierarchy L_H as shown in Figure 4.16(d), while the node dimensions below the slicing level, i.e. the 3-rd iteration vector component is assigned to the iteration domain of the node's lower slice S^L in the lower level of hierarchy as shown in Figure 4.16(c). The same process is repeated for the node T . As a result we obtain nodes T^H with the iteration vector (f, it) and node T^L with the iteration vector (js) .

The lower level node S^L invokes the unmodified statement S of the source SANLP. It processes only the iterations of the js for loop. The higher level node S^H now

4.6. CONSTRUCTION OF A MULTI-LEVEL PROGRAM (MLP)

invokes a derived statement S' . The invocation of statement S' on (f, is) domain of node S^H leads to the execution of the node S^L as illustrated by the arrow in Figure 4.16, which in turn executes statement S for all iterations in the lower level js domain. This means that S^L and S^H together run through the exact same iterations as S , but split in two different modules.

4.5.2 Dependence Placement

Second, the dependence edges must be analyzed and assigned to the appropriate HiPRDG layer. The decision in which PRDG to place an edge is based on the level comparison. We differentiate two main cases for the *slicing rule*:

- **Case A:** ABOVE/AT ($dl \leq sl$) - Dependence level is above or the same as the slicing level. In this case, the dependence edge stays in the top-level graph at the higher layer L_H .
- **Case B:** BELOW ($dl > sl$) - Dependence level is strictly below the slicing level. In this case, the dependence edge is located in the lower layer L_L . It is assigned to the PRDG that contains the source and the destination nodes of the dependence's P/C pair.

Dependence edges of the original program model are assigned to different components according to the slicing rule. We already computed the dependence level of the dependence e in the PRDG shown in Figure 4.13(b). The dependence e has dependence level $dl(e) = 1+$, i.e. it is induced by the textual order within the body of the f loop. If we use slicing level $sl = 2$, the dependence edge will remain above the slicing level (Case A). After slicing the dependence edge connects the higher-level nodes obtained from the PRDG, namely the nodes S' (corresponding to S^H) and T' (corresponding to T^H) shown in Figure 4.14(b). We complete the transformation by adjusting the linear (in)equalities in the specification of dependences to contain only the iteration space dimensions used at the given program level.

The result of the slicing transformation is a set of stand-alone PRDGs. Each PRDG is associated with some program level and assigned to some HiPRDG node at that level. Moreover, a PRDG at one level is a definition in form of the fully-fledged polyhedral model of some derived statement at one level up. Which derived statement it exactly defines is determined by the annotation of its HiPRDG edge.

4.6 Construction of a Multi-Level Program (MLP)

In this section, we address the last phase of the workflow depicted in Figure 4.3, i.e. the construction of a multi-level program. The input to the *program construction*

4.6. CONSTRUCTION OF A MULTI-LEVEL PROGRAM (MLP)

phase is the HiPRDG model and the specification of derived token granularities, i.e. data types at each level of the HiPRDG. To generate the MLP program code, we perform a bottom-up traversal of the HiPRDG. Each HiPRDG node is separately transformed into an independent program module. The designated root node of the HiPRDG is used to generate the program *main*. Derived statements are substituted by invocations of the corresponding program modules generated from the lower-level HiPRDG nodes. Communication between program modules is realized using tokens.

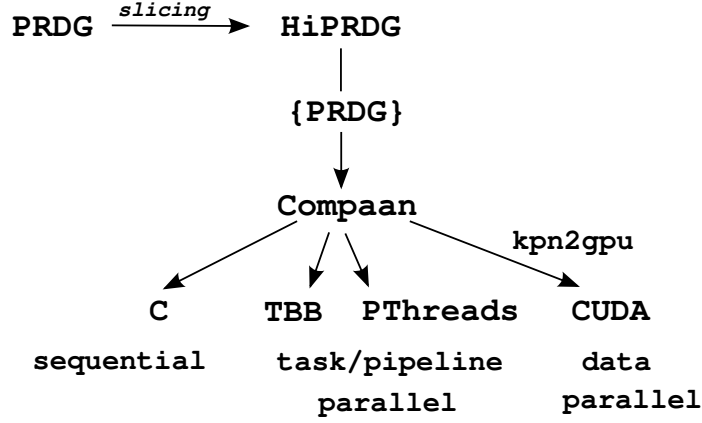


Figure 4.17: A HiPRDG contains one or more PRDGs at each level. For each PRDG we can generate different target code (e.g. PThreads, CUDA).

Each leaf node can be processed differently by Compaan or other polyhedral compilers as illustrated in Figure 4.17. For each leaf node, we can generate sequential C code, multi-threaded code taking advantage of task and pipeline parallelism, or using the KPN2GPU extension presented in Section 3 generate data parallel CUDA code.

We illustrate each of the steps in the derivation of a MLP on the running example of M-JPEG P/C pair given in Figure 4.18. It shows the HiPRDG of the M-JPEG example extended with tokens passed between the components. To more easily follow the generated code, we renamed some of the components in the HiPRDG, e.g. derived statement S' from level $L0$ in Figure 4.13 is now denoted as DS .

First, we generate the body of each program module from the polyhedral specification of the HiPRDG node, i.e. the PRDG with which we annotated the HiPRDG node. Details of program module body generation are given in Section 4.6.2. Second, we create an interface to the generated code by encapsulating it into a function definition. The formal arguments to the program module are obtained from the specification of the tokens for the given level and the prototype of the statements processed within a given HiPRDG node. The token specifications for each level are produced in

4.6. CONSTRUCTION OF A MULTI-LEVEL PROGRAM (MLP)

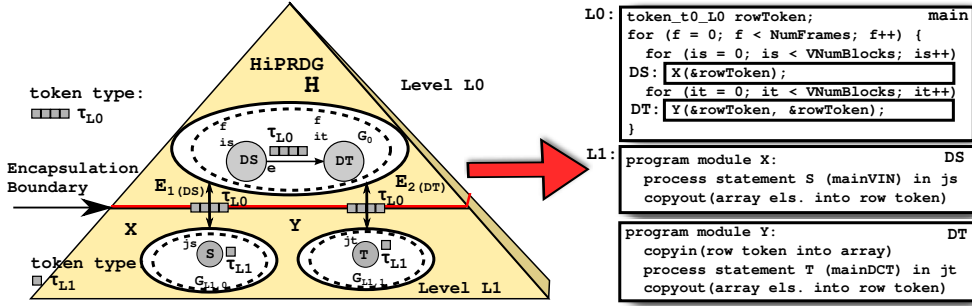


Figure 4.18: An example: (a) A two-level HiPRDG of M-JPEG P/C pair. (b) A simple MLP pseudocode with two program modules derived from HiPRDG nodes X and Y , and *main* derived from PRDG of the root node R .

the token granularity selection phase. Encapsulation into program module is covered in Section 4.6.3. The body of the program module is wired to the program module interface by inserting the methods for bringing the data from the tokens passed as arguments to the function into data space of the function. Generation of the wiring is covered in Section 4.6.4. The special case is the root node R of the HiPRDG, which requires only program body generation and encapsulation into the *main*.

Algorithm 1 MLP Construction

- 1: Inputs: (1) A HiPRDG H , (2) Per-level specifications of token data types.
 - 2: Output: A MLP
 - 3: for level $L_k = L_{max}$ to L_0 of HiPRDG H
 - 4: foreach HiPRDG node X_i in L_k
 - 5: $G \leftarrow$ PRDG annotating the node X_i
 - 6: Generation of program module body from G :
 - 7: (1) PRDG scheduling and transformations (parallelization),
 - 8: optional construction of intermediate models, such as PPN.
 - 9: (2) Code generation
 - 10: (3) Substitution of statements with function calls
 - 11: Encapsulation - Generation of program module interface for node X_i
 - 12: Wiring of the program module body to the interface
-

The steps of MLP generation are summarized in Algorithm 1. The input is an HiPRDG, and the output is a multi-level program (MLP). The resulting multi-level program can contain sequential and parallel program modules featuring arbitrary forms of parallelism.

4.6.1 Preparatory Step

Figure 4.18(a) shows that after slicing, processing at the top-level $L0$ proceeds on tokens of type τ_{L0} (i.e. composite tokens corresponding to a collection of blocks, such as 4-block tokens in Figure 4.18), while processing at level $L1$ still proceeds on tokens of type τ_{L1} that corresponds to the original block token-type in the source code. To generate a MLP, we first need to introduce data structures representing the novel composite token types, which is here τ_{L0} . The composite data structures for composite tokens are generated as explained in Section 4.3.2. This leads to $L1$ tokens being defined as blocks and assigned alias type `token_t0_L1`, and composite $L0$ tokens being defined as arrays of blocks and encapsulated into structs of type `token_t0_L0`. The data type definitions are given in Listing 4.3.

```

1 // L1 Token Type ~ original data type (TBlock)
2 typedef TBlock token_t0_L1;
3
4 // L0 Token Type ~ 1D array of TBlock elements
5 struct token_t0_L0
6 {
7     token_t0_L1 elements[HNumBlocks];
8 };

```

Listing 4.3: Multi-Level Program Data Types

Passing the data from level $L0$ to the lower level $L1$ in Figure 4.18 for processing, requires sending of a collection of blocks (a row) represented as τ_{L0} token to the program modules at level $L1$. To correctly process this collection of blocks at level $L0$, we first must perform conversion from τ_{L0} type into a dataspace of single blocks. This is realized by means of COPYIN/COPYOUT operations. The COPYIN operation takes as input a coarser-grain τ_{L0} token and copies it into a data space of single blocks. The COPYOUT operation combines single block tokens into a collection of blocks. A prototype implementation of the two methods is given below in Listing 4.4.

```

9 //////////////////////////////////////////////////
10 // COPYIN Procedure
11 // from a row token (data type token_t0_L0)
12 // with token cardinality TC = HNumBlocks
13 // into block array
14 // starting at offset position: start
15 //////////////////////////////////////////////////
16 void COPYIN(token_t0_L0* rowToken, unsigned int TC,
17             token_t0_L1* blocks, unsigned int start)
18 {
19     for (unsigned int count = 0; count < TC; count++)

```

4.6. CONSTRUCTION OF A MULTI-LEVEL PROGRAM (MLP)

```
20         blocks[start + count] = rowToken->elements[count];
21     }
22     //////////////////////////////////////
23     // COPYOUT Procedure
24     // from data space A with elements of type token_t
25     // starting at offset position: start
26     // into composite token tA of type ctoken_t
27     // with token cardinality TC
28     //////////////////////////////////////
29     void COPYOUT(token_t0_L0* rowToken, unsigned int TC,
30                 token_t0_L1* blocks, unsigned int start)
31     {
32         for (unsigned int count = 0; count < TC; count++)
33             rowToken->elements[count] = blocks[start + count];
34     }
```

Listing 4.4: MLP Access Helpers

4.6.2 Program Module Body Generation

The generation of the body of a program module from its PRDG is composed of the following steps:

- PRDG scheduling and parallelizing transforms
- Code generation
- Statement substitution

As indicated in Section 4.6, the PRDG can be transformed and optimized (or parallelized) for the given target architecture using different compiler tools that accept a PRDG as input. The transformed model is then passed to the code generator to obtain the program code. At this stage, it is possible to construct an intermediate model, such as a PPN, to facilitate code generation for a desired target architecture. For example, generation of a PPN for the root node R facilitates task-parallel execution and streaming on the platform-level. After the model is transformed into the desired form, it is sent to a code generator tool to obtain a valid program code in desired target language.

Let us illustrate the generation of the body of a program module on the example of node X , which appears as a leaf of the HiPRDG in Figure 4.18. The node X is annotated with a simple PRDG named $G_{L1,0}$. The graph $G_{L1,0}$ contains only a single node that executes unmodified statement S . The one-dimensional node domain captures all executions of S in for loop js . Generation of sequential code from graph $G_{L1,0}$ would lead to pseudocode in Figure 4.19(a1) and graph $G_{L1,1}$ would lead to pseudocode in Figure 4.19(b1).

4.6. CONSTRUCTION OF A MULTI-LEVEL PROGRAM (MLP)

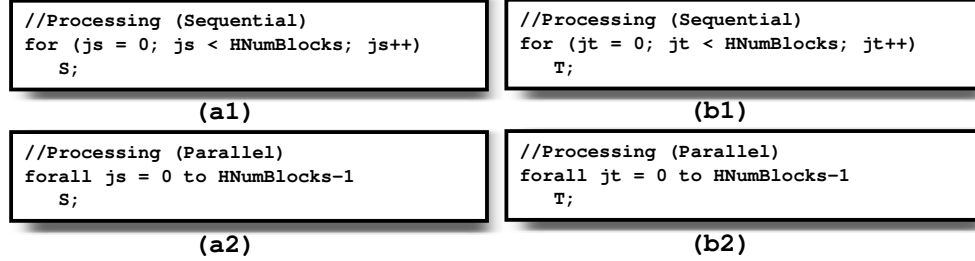


Figure 4.19: Processing blocks of modules X and Y .

The graphs $G_{L1,0}$ and $G_{L1,1}$ in Figure 4.18 contain no dependence edges. The absence of dependences leads to parallel code, which we illustrate by **forall**² loops in pseudocode in Figure 4.19(a2) for node X and Figure 4.19(b2) for node Y . In general, at this stage the designer is capable of generating a sequential or a parallel code - whatever best fits the given target architecture.

In both cases, the statements need to be substituted by actual function calls. In case of leaf nodes this is rather simple. Statement S invokes function `mainVIN` from program source, and statement T invokes function `mainDCT`. Both functions are unmodified and process blocks, which are now denoted as τ_{L1} tokens. After the introduction of a local variable representing the node's footprint on block dataspace, i.e. a row represented by a one-dimensional array of blocks, we can substitute the statement S with a function call and pass an element of a data space as the actual argument of the function call, as illustrated in Figure 4.20

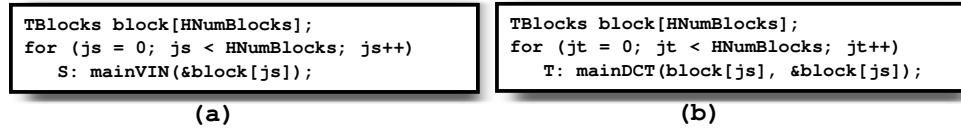


Figure 4.20: Processing blocks of modules X and Y (Statements substituted with the actual function calls).

The generation of program module bodies for non-leaf nodes involves substitution of *derived statements* with calls to program modules. As an example let us have a look at HiPRDG root node R . The sequential code implementing PRDG G_0 is shown in Listing 4.5.

²All iterations of a forall loop are executed in parallel.

4.6. CONSTRUCTION OF A MULTI-LEVEL PROGRAM (MLP)

```

1  for (f = 0; f < NumFrames; f++) {
2      for (is = 0; is < VNumBlocks; is++)
3          DS:...;
4      for (it = 0; it < VNumBlocks; it++)
5          DT:...;
6  }

```

Listing 4.5: Sequential code for G_{L0} (HiPRDG root R).

For loops encapsulate calls to derived statements DS and DT . HiPRDG edge $E_1(DS)$ reveals that derived statement DS calls a program module generated from HiPRDG node X . Similarly, HiPRDG edge $E_2(DT)$ reveals that derived statement DT calls a program module generated from HiPRDG node X . Thus, in order to substitute DS and DT with actual function calls, we need to generate the modules X and Y first. This is addressed in Section 4.6.3, resulting in declarations of program modules X and Y in Figure 4.21(b). Now we can substitute statement DS with a function call X and statement DT with a function call Y . The result is the top-level program in Figure 4.18(b).

4.6.3 Encapsulation/Interface Generation

The program module body illustrated by the code snippet in Figure 4.20 needs to be encapsulated in order to invoke it as a program module. We approach this by generating a program module interface out of each HiPRDG node. To create the interface we need a name of the function and formal arguments of the function. Let us illustrate the generation of a program module interface on the example of Figure 4.21. For each of the non-root nodes of the HiPRDG node, we generate a function name

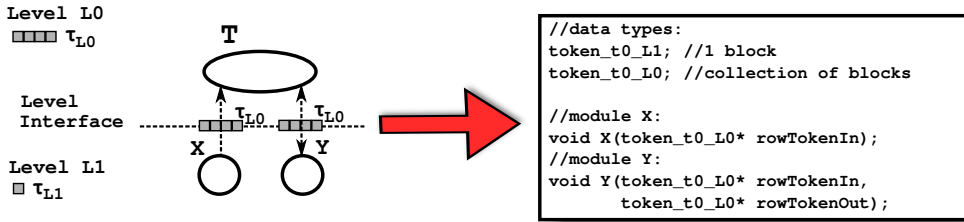


Figure 4.21: One HiPRDG node - one program module. Nodes communicate composite tokens of type τ_{L0} which corresponds to modules' arguments.

corresponding to the unique identifier of the HiPRDG node. Thus, HiPRDG node X becomes function X , and HiPRDG node Y becomes function Y , as illustrated in Figure 4.21. The root node R is encapsulated into main. Next, we need to obtain the formal arguments of the program modules. The formal arguments of the program

4.6. CONSTRUCTION OF A MULTI-LEVEL PROGRAM (MLP)

modules are obtained by analyzing the tokens passed to and from the program module. Let us have a look at the interface between two levels illustrated in Figure 4.21. The root node R operating at level $L0$ gets row tokens of type τ_{L0} from node X at level $L1$ and passes them to node Y at level $L1$. The token type τ_{L0} is implemented as data type `token_t0_L0` (see Listing 4.3) and corresponds to one-dimensional collection of blocks. As a consequence the program modules implementing HiPRDG nodes X and Y have to accept tokens of type τ_{L0} , resulting in the interfaces in Figure 4.21(b). The communication between the nodes R and Y is in both directions. Thus, the function Y has both input and output arguments of type τ_{L0} .

4.6.4 Automatic Type Conversion

Once we have generated the program module interface and its body, it is necessary to connect these parts together. In Section 4.6.1, we explained how to bring the data from one token type into the other and vice versa by introducing the methods `COPYIN` and `COPYOUT`. A prototype implementation of the two methods is given in Listing 4.4. Now it is the time to put these two methods into practice. In module

<pre>//module X: void X(token_t0_L0* rowTokenIn) { //Local data space token_t0_L1 block[HNumBlocks]; //Processing (SEQ/PAR) for (js = 0; js < HNumBlocks; js++) S: mainVIN(&block[js]); COPYOUT(rowTokenOut<-block); }</pre>	<pre>//module Y: void Y(token_t0_L0* rowTokenIn, token_t0_L0* rowTokenOut) { //Local data space token_t0_L1 block[HNumBlocks]; COPYIN(block<-rowTokenIn); //Processing (SEQ/PAR) forall (jt = 0 to HNumBlocks-1) T: mainDCT(block[jt], &block[jt]); COPYOUT(rowTokenOut<-block); }</pre>
(a)	(b)

Figure 4.22: Pseudocode of the program modules after wiring the program module body to the program module interface.

X we need to pack the blocks produced by the function `mainVIN` into a composite token type τ_{L0} and send it to level $L0$. Similarly, in order to perform the unmodified `mainDCT` in module Y it is necessary to first unpack blocks from the input composite token into a local data array `block`, and then we can process it. Once the DCT has completed, we pack the resulting blocks into output token of type τ_{L0} and send it back to level $L0$. This is realized by inserting a call to the `COPYIN` method at the beginning of the program module implementation. All input arguments are brought with this method into local data space of the program module. By analogy, we insert a call to the `COPYOUT` method at the end of the program module implementation. After wiring, we obtain the definitions of program modules as illustrated by pseudocode in

Figure 4.22.

4.7 Results of MLP Construction

The result of the code generation from the hierarchical program model (HiPRDG) is a structured, multi-level program. As an illustration, in Figure 4.23 we show the three program modules, i.e. the functions *X* and *Y*, and the new *main* operating on new data structures, generated from the three-node HiPRDG depicted in Figure 4.18.

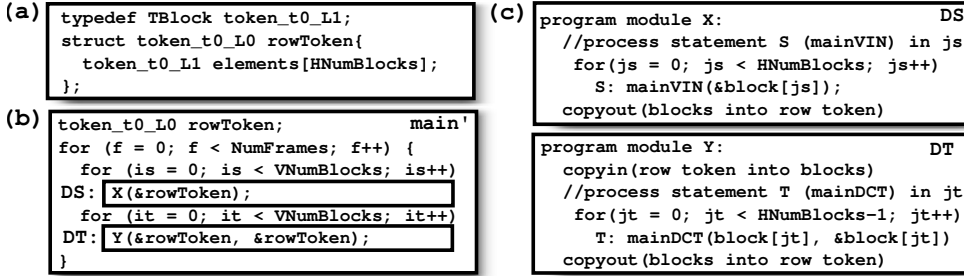


Figure 4.23: Resulting multi-level program data types and modules.

Figure 4.23(a) shows the data representation after splitting the program into two levels. The specification of composite data structures is automatically obtained after the analysis and selection of token granularity. As an illustration, in the running example, we have chosen to process row-sized tokens on the top-level (*L0*), and blocks at the lower level (*L1*) of the MLP. This choice lead us to slicing the program at loop nesting level $l = 2$. This results in the new *main* (*main'*) shown in Figure 4.23(b). The *main'* contains derived statements and operates on row-sized tokens. The hierarchical restructuring that we performed on the program model resulted in the replacement of the loop subnests containing loops *js* and *jt* in *main'* with derived statements *DS* and *DT*. The derived statements *DS* and *DT* invoke functions *X* and *Y* that are obtained via encapsulation of the program code generated from PRDG of these two loop subnests into independent program modules. The computation in program modules of the multi-level program can be sequential or parallel. As a first step, let us illustrate a completely sequential two-level program. In Figure 4.23(c), we give sequential pseudocode for the functions *X* and *Y*. Together, *main'*, *X*, and *Y* execute in a structured way all iterations of the original program.

In line with the discussion on program module generation, it is also possible to generate a task-parallel PPN from the top-level graph containing derived statements

DS and DT , instead of directly from the program code containing statements S and T . For comparison, we show two pseudocodes side by side in Figure 4.24.

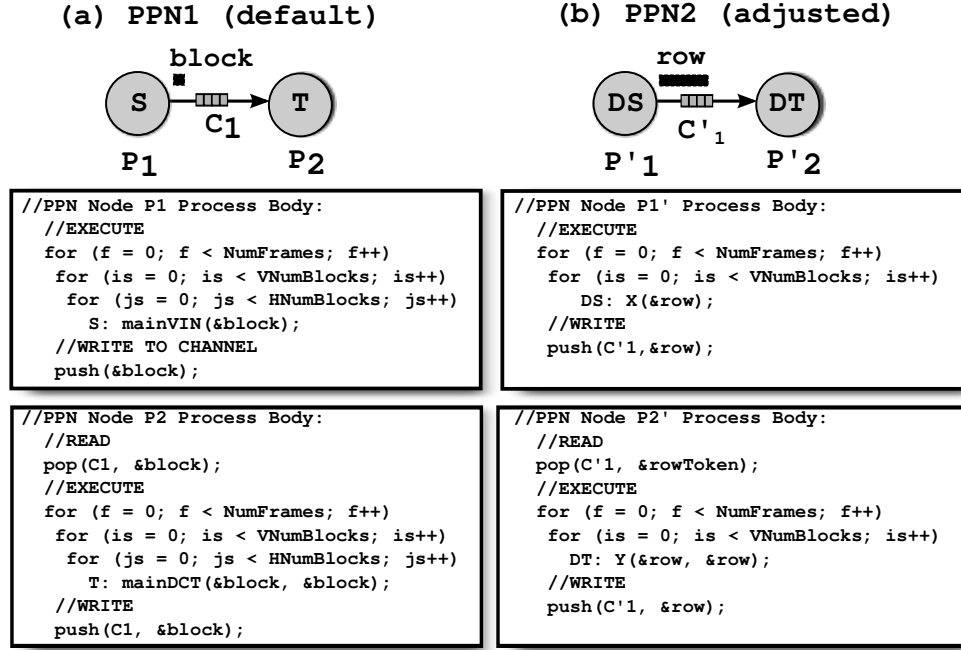


Figure 4.24: Default PPN working on blocks, and a PPN with adjusted granularity.

Both PPNs have two nodes and execute the same functionality. However, while the default PPN illustrated in Figure 4.24(a) works on blocks, as specified in the SANLP, the PPN obtained from our top-level graph works on rows, as illustrated in Figure 4.24(b). This shows that using our approach, it is possible to adjust the token granularity in a PPN without having to manually rewrite the program code. This leads to significant performance improvements as will be shown in the M-JPEG case study in Section 6.5.

The multi-level structuring of the program model and encapsulation enable us to perform different types of parallelization at different levels. Each program module can be further parallelized and optimized independently, or it could be simply sequentially processed. Figure 4.25(a) shows the sequential code for the DCT computation in module Y , while in Figure 4.25(b) we see pseudocode for parallel processing of DCT instances. All iterations of the *forall* loop in Figure 4.25(b) can be executed in parallel. The abstract *forall* loop can be replaced by a compiler with an OpenMP parallelization pragma for generation of data parallel CPU code, a TBB parallel processing construct, or for example, KPN2GPU can be applied to generate CUDA code, as illustrated in Figure 4.25(c). As a result of our approach, we are now capable of

4.8. CONCLUSIONS AND FUTURE WORK

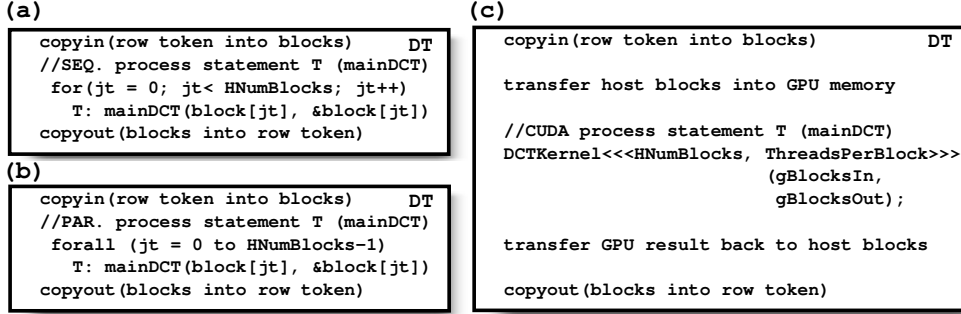


Figure 4.25: Independent parallelization of DCT program module (HiPRDG node Y).

obtaining a two-level parallel M-JPEG realization which features task and pipeline parallelism by taking advantage of the PPN model at the top-level, and whose program modules internally have data parallelism that can be used for GPU acceleration. We present the overall performance improvements achieved by the multi-level parallelization in Section 6.7.

4.8 Conclusions and Future Work

In this chapter, we introduced the hierarchical intermediate program representation in the polyhedral model called HiPRDG, and presented a structured method for derivation of a HiPRDG from the standard polyhedral model of the application. We also showed how to derive a structured multi-level program (MLP) from a HiPRDG. The hierarchical representation leads to multi-level parallel programs that are well suited for mapping onto heterogeneous platforms, allowing us to target different architectural components with different types of parallelism. As a result of the techniques presented in this chapter, we are now capable of obtaining multi-level parallel programs featuring task, data, and pipeline parallelism.

PPN Execution on Heterogeneous Platforms with GPUs

5.1 Introduction

In Chapter 4, we have shown how to construct a two-level program featuring task, pipeline and data parallelism. The top-level of the resulting program contains coarse-grain autonomous tasks communicating via channels that are generated as FIFO buffers from the PPN specification. Statements executed by computationally intensive tasks are further transformed for data parallelism in order to be offloaded for execution on an accelerator. In this chapter, we introduce novel techniques for execution of PPN process' statements on an accelerator while improving the efficiency of host-accelerator communication.

Let us again consider the two-level M-JPEG PPN illustrated in Figure 4.2. The nodes of the top-level PPN, which are denoted in this figure as S' , T' , Q and VOU , are implemented as parallel tasks and mapped for execution on different threads running on a multicore CPU of the host platform. Communication between the four tasks is organized exclusively using FIFO buffers. At each iteration, the S' process writes a token into its output buffer. The T' process reads the token from the buffer, and executes the next process iteration which executes the statement, that contains a call to the `mainDCT` function. Our goal at this stage is simple - and that is to offload the execution of `mainDCT` onto an accelerator.

After generating data parallel accelerator code for the `mainDCT` (either manually or e.g., using the approach presented in Chapter 3), it is necessary to provide the host-side accelerator management code for *kernel offloading* onto a GPU. To execute the node T (DCT) on the GPU, two issues need be resolved:

5.1. INTRODUCTION

- (1) How to manage the offloading of PPN process iterations onto an accelerator (such as GPU)
- (2) How to reduce host-accelerator communication overheads, i.e. how to improve the communication efficiency

The GPU accelerator is typically seen as a co-processor managed by the host. The traditional model for kernel offloading consists of three phases:

- **copy-in** - data transfer from host memory (main memory) to device memory (GPU global memory),
- **kernel** - execution of process iteration (kernel) on the GPU
- **copy-out** - data transfer from device memory to host memory

In compiler-assisted parallelization frameworks, the kernel offloading is traditionally realized with a *drop-in code replacement*. The drop-in code replacement refers to the substitution of the sequential code on the host-side with the kernel offloading mechanism. We refer to this mechanism as *Synchronous Offloading of a Kernel* (SOK). The SOK mechanism is illustrated for function call Y in Figure 5.1.

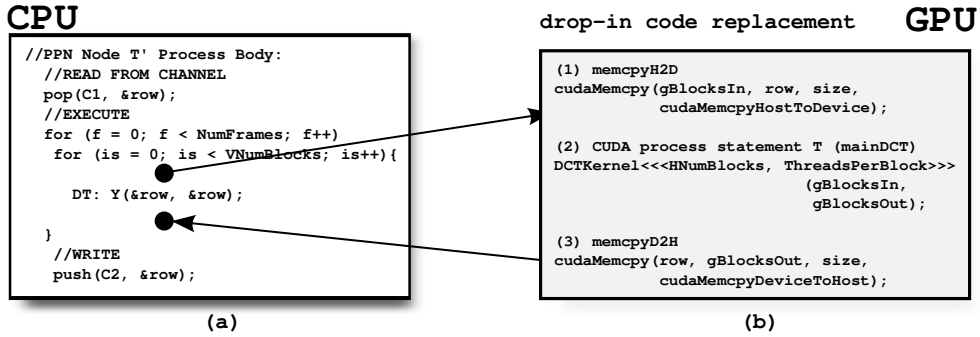


Figure 5.1: Blocking kernel offloading using the drop-in code replacement.

The body of the PPN process T' is shown in Figure 5.1(a). At each iteration (f, is), T' executes statement DS . The statement DS invokes the function Y that implements the DCT processing. It takes as input argument a variable of type `row` that is an array of `HNumBlocks` block elements, transforms it, and returns the resulting `row`. To execute the DCT on the GPU, the call to function Y is simply replaced by the drop in code replacement shown in Figure 5.1(b). Figure 5.1(b) shows the three steps of synchronous kernel offloading. First, we transfer the input data from the variable `row` in the host memory (main memory) to the variable `gBlocksIn` in the GPU memory using a CUDA API call `cudaMemcpy` indicating the direction as

`cudaMemcpyHostToDevice`. For brevity, we denote `cudaMemcpy` calls from host to device as *memcpyH2D*, and `cudaMemcpy` calls from device to host as *memcpyD2H*. Second, we launch the kernel that implements the DCT processing. The kernel reads the data from the `gBlocksIn` array and writes it to the `gBlocksOut` array, which is also in the GPU global memory. Once the kernel runs to completion, we transfer the results from the GPU memory `gBlocksOut` to the host memory `rowToken`. Once all three actions are completed, the control is returned to the process T' , which only then proceeds with the execution of the next process iteration. We illustrate the GPU execution timeline using the SOK mechanism in Figure 5.2(a).

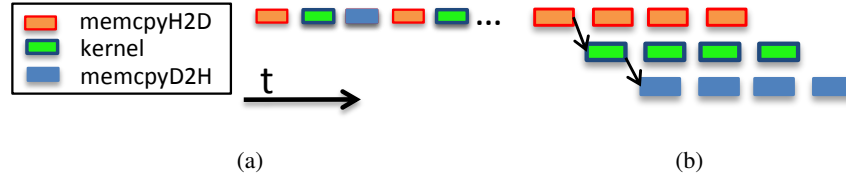


Figure 5.2: (a) Have: Synchronous execution of CUDA operations (no overlaps), (b) Want: Asynchronous execution (overlapped).

The main limitation of the SOK mechanism is that this type of kernel offloading is blocking. The PPN process can not start the data transfer from host to device and the computation of the next iteration, until all three phases of the current iteration have completed. In case of streaming applications, this inhibits the pipelining of data transfers and processing on the GPU. The impact of data transfers can be mitigated by overlapping data transfers and computation (see Figure 5.2(b)), as demonstrated in several application case studies [14, 143]. This can be achieved following NVIDIA’s technical note [100] that contains an example code pattern that enables overlapping of data transfers and kernel execution. State of the art frameworks for run-time task scheduling, such as StarPU [10], use asynchronous data transfers for communication with the GPU. Farago and Nikolov [52] experimented with the SOK mechanism in the context of PPNs, which results in the sequential timeline given in Figure 5.2(a).

The question that we address in this chapter is how to efficiently offload kernel execution from a PPN node to an accelerator and achieve overlapping of computation and communication phases as illustrated in Figure 5.2(b).

5.2 Approach

To enable overlapping of computation and communication phases of consecutive PPN process iterations illustrated in Figure 5.2(b), we propose a model-based *Asynchronous Offloading of a Kernel* (AOK) mechanism for kernel offloading to acceler-

5.2. APPROACH

ators, which leverages the asynchronous, data-driven nature of the PPN model combined with asynchronous data transfers to the accelerator.

As a step to asynchronous communication with the accelerator, we differentiate between two types of channels: the channel between processes executing their process iterations on the same device (e.g. a host CPU), and the channels between two processes executing process iterations on different devices (e.g. a host CPU and a GPU accelerator). For simplicity, we refer to the first type of channels as a Shared Memory Channel (SMC) (or channel type A) and the second type as Distributed Memory Channel (DMC), or simply channel type B. When the SOK mechanism is used, the PPN process makes use only of SMC channels to connect to other processes, as illustrated in Figure 5.3(a). In this figure, the data transfers between the host and GPU are denoted as *dataxfer*. For processing on the GPU, the process P first puts data into the SMC channel Ch_1 , the process T' reads the token from the SMC channel Ch_1 , and then invokes the kernel offloading mechanism shown in Figure 5.1(b). The SOK mechanism invokes three CUDA operations: (1) synchronous data transfer from host to GPU, (2) kernel execution, and (3) synchronous data transfer from GPU to host. The timeline of the CUDA operation execution for the three consecutive process iterations is shown in Figure 5.3(c). Only after the all three CUDA operations have completed the control is returned into process T' , which then puts the results of GPU execution into the SMC channel Ch_2 .

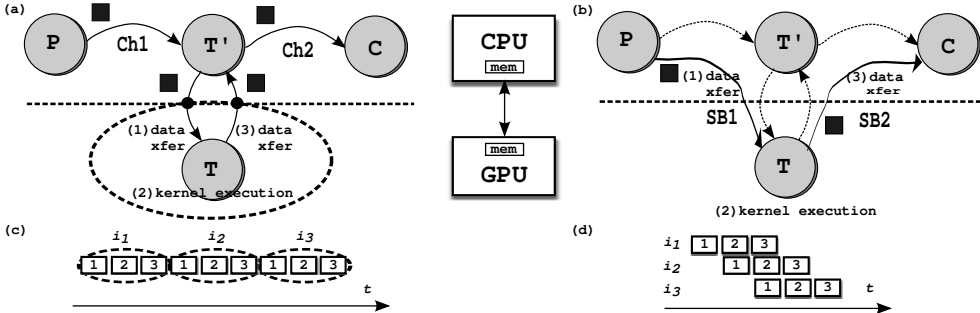


Figure 5.3: Solution Approach: (a) SOK vs (b) AOK

The core idea of our approach for asynchronous kernel offloading is to directly send the data produced by P to the GPU without transferring it first to the host memory of process T' , by taking advantage of different channel types. Following the AOK approach, process P does not block waiting for the data transfer to complete. Instead, process P is free to proceed with the next process iteration and start the next data transfers to the GPU. As soon as the GPU receives the input data from process P' , it launches the DCT kernel, and once the kernel is completed, we directly transfer the results from GPU memory to the process C . The arrows between processes

P and T' , and T' and C in Figure 5.3 indicate synchronization that takes place between processes, and arrows between processes P and T and T and C indicate the host-accelerator data transfers. So, instead of having P transfer data from the host memory buffer to the host memory buffer of T' , and then having T' transfer the data from the host memory buffer into GPU memory using a synchronous data transfer call, P transfers data directly from its host memory buffer into the GPU memory buffer access by T , as indicated by `dataxfer` marks in Figure 5.3. To realize this pattern we take advantage of asynchronous DMA transfers between the host and GPU accelerator. This sort of asynchronous execution enables pipelining of GPU operations (e.g. kernel execution from the first iteration (denoted as i_1) can be overlapped with the host-accelerator data transfer from the second iteration (denoted as i_2)), thus facilitating the desired overlapping of communication and computation. The timeline of the CUDA operation execution for three consecutive process iterations is shown in Figure 5.3(d).

Our approach is inspired by the technical note from NVIDIA [100] that shows how to use the concept of a CUDA *stream* to realize overlapping of communication and computation. A CUDA stream describes a sequences of GPU operations (kernel executions, data transfers) that execute in-order. The operations from different streams execute in parallel. This enables overlapping of a kernel execution in stream s_1 with a data transfer in stream s_2 .

In addition, we leverage double-DMA capabilities of Fermi-architecture GPUs to achieve the overlapping of not only data transfers to the GPU and kernel execution, but also to additionally overlap GPU data transfers in different streams. This allows us to upload an input token from iteration i_3 to the GPU (data xfer in phase 1), while we are at the same time executing the kernel for iteration i_2 , and downloading the results of iteration i_1 to the host memory (data xfer in phase 3), as shown in Figure 5.3(d). Coupled with the PPN properties of data-driven asynchronous execution, the technical capabilities of modern GPUs enable us to design a model-based approach for overlapping communication and computation on GPU based systems.

The chapter is structured as follows. In Section 5.3.1, we introduce a classification of PPN channels that enables model-driven channel design and mapping. In Section 5.3.2, we give an overview of traditional channel design, and explain extensions that improve its efficiency by reducing the amount of data transferred at each channel access. In Section 5.4, we cover the AOK mechanism. First, we present our *stream buffer* design for type B channels that enables model-driven overlapping of computation and communication in Section 5.4.1 Second, we show how to apply the stream buffer design to enable AOK in a PPN in Section 5.4.2. Finally, we show experimental results in Section 5.5, and discuss our findings.

5.3 Model-Driven Communication Design

5.3.1 Classification of PPN Channels

Let's consider a heterogeneous platform containing several discrete devices (architectural components) each with its own private memory. For efficient design and mapping of communication between PPN processes, it is necessary to consider whether the processes communicate through the physically same piece of memory, or not. For example, two processes entirely executing on the host CPU both access the main memory. However, if one process executes on the CPU and the other offloads its iterations to the GPU, these two processes access two physically different memories: the main memory, and the GPU device memory. According to the memory accessed by the two processes, we propose the following classification of PPN channels into two main categories:

- **A. Shared Memory Channel (SMC)**
- **B. Distributed Memory Channel (DMC)**

A Shared Memory Channel (SMC) connects two PPN processes that execute on the same device, and thus access the same memory. If both processes connected by the channel execute on the CPU, such channel is further classified as host-to-host (H2H) SMC. If both processes execute on an accelerator device, such as GPU, such channel is further classified as device-to-device (D2D) SMC.

A Distributed Memory Channel (DMC) connects two PPN processes that execute their process iterations on two devices, each with their own memory. For example, the VIN process executes exclusively on the host CPU, while the DCT process executes kernels on the GPU. In this case, we have a distributed memory system, i.e. both host and accelerator have their own memory space. If the channel's producer process executes on the host and its consumer process executes on the GPU, we further classify this channel as host-to-device (H2D) DMC. Analogously, if the channel's producer process executes on the GPU and its consumer process executes on the host, we further classify this channel as device-to-host (D2H) DMC. While SMC channels can be simply realized using some of the available FIFO libraries, host-GPU DMC channels require closer consideration as the communication and synchronization must be carefully handled.

5.3.2 SMC Design

The PPN channels are typically designed as first-in first-out (FIFO) buffers. There is large number of FIFO buffer implementations available. For illustration, we will make a brief overview of a general FIFO buffer design based on a well-known concept

of a circular buffer (ring buffer) and explain a simple extension to this design, that results in minimized data movement over the bus.

A circular buffer is a fixed-size bounded buffer allocated in a single piece of memory. The circular buffer of size m has m slots, which are written and read in a circular fashion. The producer process writes tokens one by one into the buffer. After all slots have been filled up, the producer starts writing from the beginning of the buffer. If there are no empty slots left, we block the producer process until the consumer process reads the next token and thus another slot becomes available. Thus we realize SMC channels by using circular buffers and blocking read/write accesses.

The point-to-point communication in a PPN results in PPN channels having the *single producer - single consumer* (SPSC) property. Due to this SPSC property, the channel's producer and consumer PPN nodes can read and write data concurrently from the channel as long as there is a sufficient number of full/empty buffer slots available.

The current realisation of PPN processes requires physical transfer of data tokens from channels into a local memory upon read, and from the local memory into the channel upon write. However, when two processes are connected with SMC, these data transfers are unnecessary. Instead of physically copying a data token into a local variable of the process, it is sufficient to acquire a pointer to the buffer slot holding the data token. For example, instead of copying a complete token (e.g. 1KB block in M-JPEG or larger) into a local variable and then processing the local variable as input argument to the process, it is sufficient to take the pointer to the token and pass it as the input argument to the process. We call this type of access *direct access to memory*. When direct access to memory is used, additional modifications to the PPN R/E/W execution protocol are required. For this we propose use of an *Acquire Direct Access to Memory* (ADAM) protocol. In the ADAM protocol, a process first acquires pointers to the tokens that are the actual input and output arguments. Then, it evaluates the process function, and finally, it releases both pointers. The use of the ADAM protocol considerably reduces the amount of data traffic and has significant performance impacts on memory bounded applications.

5.3.3 Synchronous DMC Design

Each PPN process connected to a DMC channel accesses a physically separate portion of memory on the system. To realize the communication between the producer PPN process and the consumer PPN process, we need to transfer the data between the memories of the producer and the consumer.

To realize DMC channels we use a distributed double buffering scheme. Let us illustrate this on the host-to-device DMC in Figure 5.4. In this figure, we see a P/C pair of processes. The producer process of the P/C pair, denoted as P , executes

5.3. MODEL-DRIVEN COMMUNICATION DESIGN

iterations on the host CPU, a the consumer process of the P/C pair, denoted as T' , executes iterations on the GPU. The data produced by P can be used as the input argument at T' only after this data is transferred from the host memory to the GPU memory.

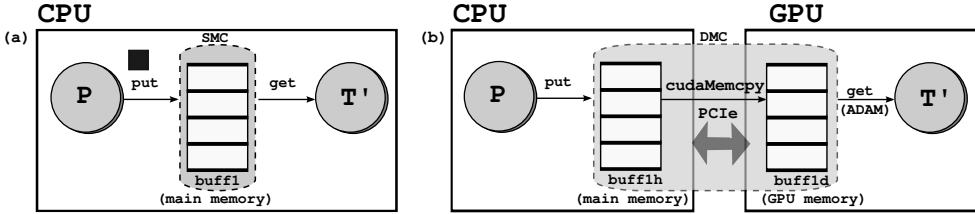


Figure 5.4: SMC vs DMC

To realize this, we first introduce a distributed double buffering scheme. We create one circular buffer in the host memory labeled as $buff1h$ and one circular buffer of the same size in the GPU memory labeled as $buff1d$. For each circular buffer, we use the SMC design described in Section 5.3.2. The producer writes the data into the $buff1h$, and the transformer T reads the data from the $buff1d$.

The data that P writes into the host-side buffer has to be transferred into the GPU-side buffer. There needs to be a mechanism for buffer-to-buffer communication in place. We realize buffer-to-buffer communication by initiating a data transfer ($cudaMemcpy$) from host to device. The producer initiates the transfer to the GPU, since it is capable of starting the transfer as soon as it has produced the data.

Writing to the DMC from the producer is composed of several operations. First, the producer tries to acquire a slot for writing in host-side memory buffer ($buff1$). Second, the producer puts the data into host-side memory buffer ($buff1$). Third, the producer invokes the $cudaMemcpy$ call to transfer data from the host-side buffer into device-side buffer $buff2$. The $cudaMemcpy$ operation is synchronous, which means that the control returns to the producer only after the GPU transfer has completed. Once the $cudaMemcpy$ returns, the producer signals to the process T' controlling the GPU that the input data is available. The GPU then issues the kernel. In this (synchronous) mode of operation, all GPU operations are issued in the same CUDA stream by default.

The basic DMC model uses synchronous data transfers between host and GPU buffer slots, which results in blocking of the producer process until the transfer has completed. As a consequence, the host-accelerator data transfers cannot be overlapped with the computation.

5.4 Asynchronous Offloading of Kernels (AOK)

The AOK mechanism requires asynchronous host-accelerator communication. We first present an efficient design for asynchronous host-accelerator (i.e. CPU-GPU) communication in Section 5.4.1 and then we show how to leverage this design to realize AOK in PPN in Section 5.4.2.

5.4.1 Asynchronous Stream Buffer Design

To improve the communication efficiency between host and accelerator, we want to overlap computation and data transfers. In this section we extend the DMC design to take advantage of asynchronous data transfers and host-accelerator DMA. The result of this extension is what we call the *stream buffer* design.

Our goal is to overlap different *GPU operations*. We distinguish three main categories: data transfers from host to accelerator (type 1 in Figure 5.5), kernel execution (type 2 in Figure 5.5), and data transfers from accelerator to host (type 3 in Figure 5.5). In CUDA, different GPU operations can execute concurrently provided that they are issued in different CUDA streams. In addition, the following requirements must be satisfied: data transfers are used in combination with *pinned memory*¹ on the host and only asynchronous data transfers are used.

Let us first introduce CUDA streams for the execution of operations in the three categories. For each type of data transfer we introduce a separate stream, namely an *upload stream* denoted as s_{H2D} for data transfers from host to accelerator. and a *download stream* denoted as s_{D2H} for data transfers from accelerator to host. Execution of GPU operations in a dedicated stream increases the concurrency from the sequential execution in the default stream s_0 illustrated in Figure 5.5(a) to the concurrent execution of operations in the three streams as depicted in Figure 5.5(b).

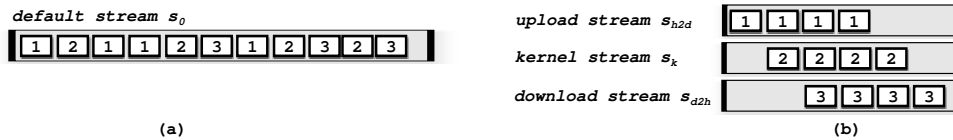


Figure 5.5: Single stream vs Dedicated streams.

The stream buffer design in Figure 5.4(b) issues host-to-accelerator data transfers into the upload stream s_{H2D} . To execute independent kernels concurrently, we also introduce a separate kernel execution streams for different kernels. In the M-JPEG example, we offload only the DCT kernel, hence it is sufficient to have a single kernel

¹Pinned memory is non-pageable portion of system memory, i.e. it can not be swapped out to disk by the operating system.

5.4. ASYNCHRONOUS OFFLOADING OF KERNELS (AOK)

execution stream, which we denoted as s_K . The accelerator-to-host data transfers are issued into the download stream s_{D2H} .

Let us now explain how we combine the concept of stream-based execution with asynchronous transfers to achieve the desired overlapping. Use of CUDA streams is only possible with asynchronous data transfers. CUDA provides an asynchronous API for its device-to-host and host-to-device `cudaMemcpy` operations. The asynchronous data transfers are realized using `cudaMemcpyAsync`. The invocation of an asynchronous CUDA call immediately returns control to the caller (e.g. producer process), without waiting for the GPU operation to complete. This means that the producer process can immediately proceed with processing the next iteration, in which it produces the next result and issues the next host-to-device data transfer. The host-to-device transfers are executed in order, resulting in a sequence of data host-to-device transfers (type 1) in stream s_{H2D} shown in Figure 5.5(b).

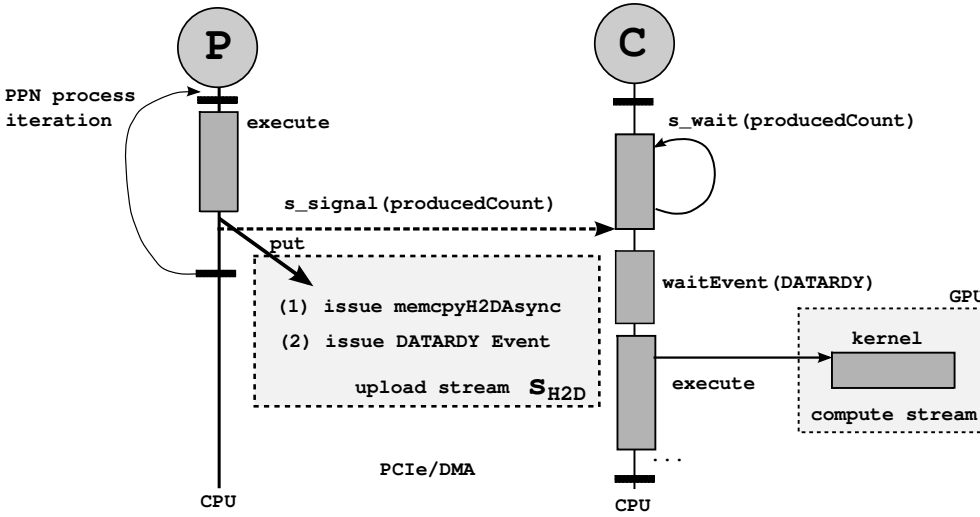


Figure 5.6: Stream Buffer Design with Asynchronous Data Transfers: Interaction Diagram.

When using synchronous data transfers, the producer has to wait until a data transfer to the GPU completes. After the data transfer has finished, the producer process signals to the consumer process that input data is available, and that it can launch its GPU kernel. As the end time of asynchronous host-device data transfers is not known, it is not possible to use the same signaling scheme in the asynchronous execution mode. Thus, the use of asynchronous data transfers imposes additional synchronization requirements.

These additional synchronization requirements are captured in the *producer-initiates consumer-completes* (PICC) protocol for asynchronous communication over DMC-

5.4. ASYNCHRONOUS OFFLOADING OF KERNELS (AOK)

type channels between PPN processes that run on host and accelerator. The communication pattern using the PICC protocol is illustrated by the interaction diagram in Figure 5.6. The PICC protocol starts when the producer process completes its *execute* phase and needs to write the results into the channel, i.e. when it invokes the *put* operation on the buffer. As the first step of the *put* operation, the producer process *initiates* an asynchronous data transfer (`memcpyH2DAsync`) to the GPU. To support the use of asynchronous data transfers, we introduce an additional GPU-side signaling mechanism. The role of the GPU-side signaling mechanism is to record a *DATARDY* event on the GPU after the asynchronous data transfer `memcpyH2DAsync` initiated by the producer *P* has completed. We realize this by issuing a GPU-side *DATARDY* event in the upload stream s_{H2D} *immediately* after the asynchronous data transfer call, as the second step of the *put* operation. Since the operations in the same CUDA stream must (1) run to completion and (2) execute in order, the GPU captures the *DATARDY* event only after the data transfer completes, even though the data transfer call is asynchronous. The consumer needs to wait on the *DATARDY* event on the GPU to occur, which is illustrated by `waitEvent(DATARDY)` phase in the consumer side of the interaction diagram. Observing a *DATARDY* event, indicates to the consumer that the data transfer is complete. Hence, the consumer completes the PICC protocol. As soon as *DATARDY* event is captured, the consumer process continues its execution. If the kernel requires data from multiple producers, the consumer process waits on multiple *DATARDY* events. If *DATARDY* events for all input arguments have been captured, the consumer process *C* exits the blocking read state, and enters its execution phase, in which it issues its GPU kernel in the compute stream.

In the CUDA programming model, waiting on an GPU event that has not yet been issued is not defined. To insure the correct execution of the PICC protocol with the current CUDA API, it is necessary to include additional host-side synchronization that indicates to the consumer when it is safe to start waiting on the GPU-side *DATARDY* event. We realized this in practice with a CPU-side semaphore `producedCount` in Figure 5.6. The producer increases its `produceCount` semaphore immediately after it issues the asynchronous data transfer. The consumer waits on the `produceCount` semaphore until the producer sets the new value. This indicates to the consumer process *C* that the data transfer has started and that it should start waiting on the *DATARDY* event indicating that the transfer has finished.

5.4.2 Application in PPN Execution

In this section, we instantiate PPN channels between the CPU and the GPU (DMC type) using a stream buffer design presented in Section 5.4.1. The use of the stream buffer design enables asynchronous host-accelerator communication in a PPN, and allows us to use AOK in a PPN instead of the SOK mechanism. Let us now illustrate

5.4. ASYNCHRONOUS OFFLOADING OF KERNELS (AOK)

AOK in a PPN on a three-node pipeline illustrated in Figure 5.7. We will discuss the flow of data from the CPU producer process P via the GPU transformer process T' to the CPU consumer process C .

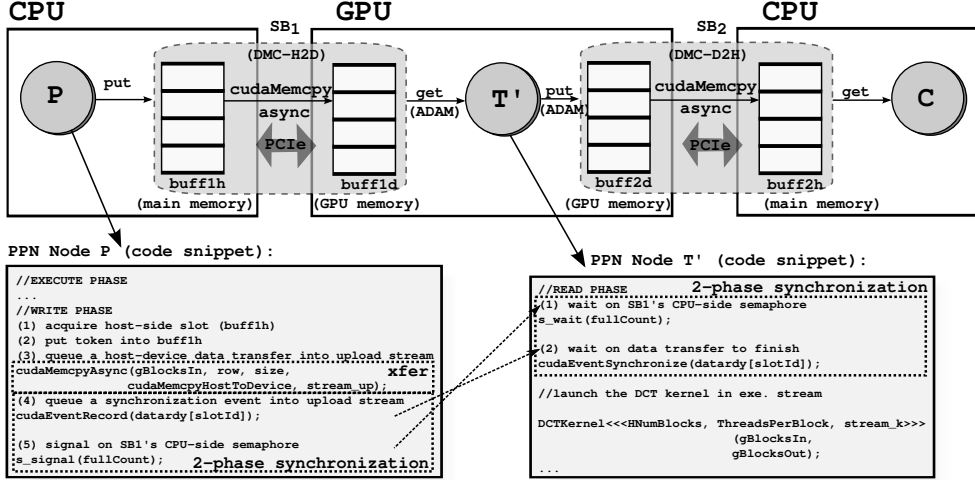


Figure 5.7: AOK using stream buffers for communication.

Each PPN process is executed by an independent thread of execution, and proceeds asynchronously in a data-driven fashion. The processes P and C execute on the CPU, while process T executes the DCT kernel on the GPU. Let us explain how the nodes execute using the asynchronous mechanisms for data transfers and kernel offloading. After generating data, process P puts data into the stream buffer SB_1 that implements the host-to-device DMC channel. Once a token is in the host-side buffer $buff1h$, P first queues up the asynchronous host-device data transfer into the upload stream s_{H2D} denoted in code as `stream_up`. Second, P starts the two phase CPU-GPU synchronization according to the PICC protocol presented in Section 5.4.1. A `DATARDY` event is queued up by P after the data transfer in the upload stream `stream_up` to signal the end of data transfer. Second, the P signals via host-side semaphore to the process T' that the data transfer has been initiated, and proceeds to execute the next process iteration.

Process T' waits on the host-side semaphore before it starts the GPU-side synchronization. As an important difference to the SOK mechanism, the T' must also acquire an empty buffer slot for storing its results before it can proceed with the execution. We realize this by blocking the process T' that makes use of AOK until all input data and an empty slot in the output channel are available in the GPU memory. Thus, the blocking parts of the blocking on read and the blocking on write phases of the process execution are now combined into a single phase at the start of each process iteration.

As a consequence, only this step of the process execution with AOK is blocking. All subsequent steps, i.e. reading data, processing data, writing data, and starting data transfer to the consumer, are non-blocking.

After process T' observes the GPU event, a new data transfer can take place on the PCIe bus while the GPU executes the DCT kernel of process T' . When T' completes the kernel execution on the GPU, it starts a `put` operation into the stream buffer SB_2 . As part of the `put` operations, T' issues an asynchronous data transfer in download stream s_{D2H} , and enqueues its `DATARDY` event. Once the consumer C captures the `DATARDY` event, it proceeds with consuming the data produced by the GPU. In the meanwhile, the GPU is already executing the DCT kernel on the next token.

This mode of operation is significantly different from synchronous kernel offloading, as (1) we avoid unnecessary data transfers from the host memory buffer of P into host memory buffer of T' , (2) the data transfers and the kernel execution are asynchronous. By combining the stream buffer mechanism with asynchronous execution in a PPN, the computations on the CPU, on the GPU, and a data transfers from CPU to GPU are overlapped. On a high-end GPU, such as a Tesla C2050 with two DMA engines it is also possible to overlap data transfers in different directions. Given a streaming video application with tokens representing frames, the following activities occur simultaneously:

- Transformer process T executes GPU kernel on frame k
- The first DMA engine uploads frame $k + 1$ from host buffer to GPU
- Producer process P executes `produce` function to obtain frame $k + 2$
- The second DMA engine downloads frame $k - 1$ from GPU to host
- Consumer process C executes `consume` function on frame $k - 2$

Thus, the data-driven asynchronous execution model with stream buffer design enables overlapping of PPN computation on the host (CPU), data transfers to the GPU, as well as kernel execution on the GPU, resulting in simultaneous utilization of all platform devices.

5.5 Experimental Results

To determine the efficiency of the stream buffer design and its application for AOK in PPNs, we perform a number of experiments. Let us first describe the experimental setup.

5.5. EXPERIMENTAL RESULTS

5.5.1 Platform Specification

The test platform contains an Intel Core i7-920 Nehalem architecture 2.66GHz processor, Intel Motherboard, and a high-end NVIDIA Tesla C2050 GPU. The Intel Core i7 (Nehalem) is a multi core, Hyper-threading technology (HT) enabled design [44]. Each socket supports one to eight cores, which share the level 3, a local integrated memory controller (IMC) and an Intel QuickPath Interconnect (QPI). The interface to the GPU is via PCIe 2.0 x16 bus. PCIe 1.x is often quoted to support a data rate of 250 MB/s in each direction, per lane. This figure is a calculation from the physical signaling rate (2.5 Gbaud) divided by the encoding overhead (10 bits per byte). This means a sixteen lane (x16) PCIe card would then be theoretically capable of $16 \times 250 \text{ MB/s} = 4 \text{ GB/s}$ in each direction. For PCI 2.x with data rate of 500MB/s in each direction, the theoretical bandwidth is up to 8GB/s in each direction, which gives us a theoretic maximum of 16GB/s for bi-directional communication. In practice, these numbers are smaller, since they depend also on manufacturer's design and the profile of the traffic, which is a function of the high-level (software) application and intermediate protocol levels.

5.5.2 Platform Micro-Benchmarks

To obtain estimates of the actual platform performance we conducted a set of micro-benchmarks. First, we measured the host memory performance. Figure 5.8 shows the measured bandwidth of main memory accesses achieved on the host side. We did four experiments. We measured the performance of a data copy from a for loop (`memcpy(for - loop)`) which is typically used in PPN implementations, and the performance of the GNU C library (`memcpy(glibc)`). In addition, we include the performance achieved separately by 32-bit read accesses `rd32 - bit` and 32-bit write accesses `wr32 - bit`. The experiment was conducted using a single thread of execution and the results were averaged over 100 iterations. Special care has been taken to avoid measuring accesses to CPU caches instead to the memory subsystem. The cache contents were manually "trashed" before conducting each measured memory access. The trashing ensures that the cache does not hold values copied in the previous benchmark iteration, and that we obtain out-of-cache memory access numbers. The results show that the GNU C `memcpy(glibc)` performs significantly better than when we perform add read and write accesses directly in the for loop. Since our PPN implementation uses manual read/write accesses, it is relevant to be aware of this performance gap. In addition, we found out that all four methods achieve less than 4 GB/s for tokens smaller than 16 KB. The token size in the default M-JPEG PPN is only 1 KB, which results in data access performance below 1 GB/sec. This points to the importance of being able to adjust and tune the token granularity as proposed in Chapter 4.

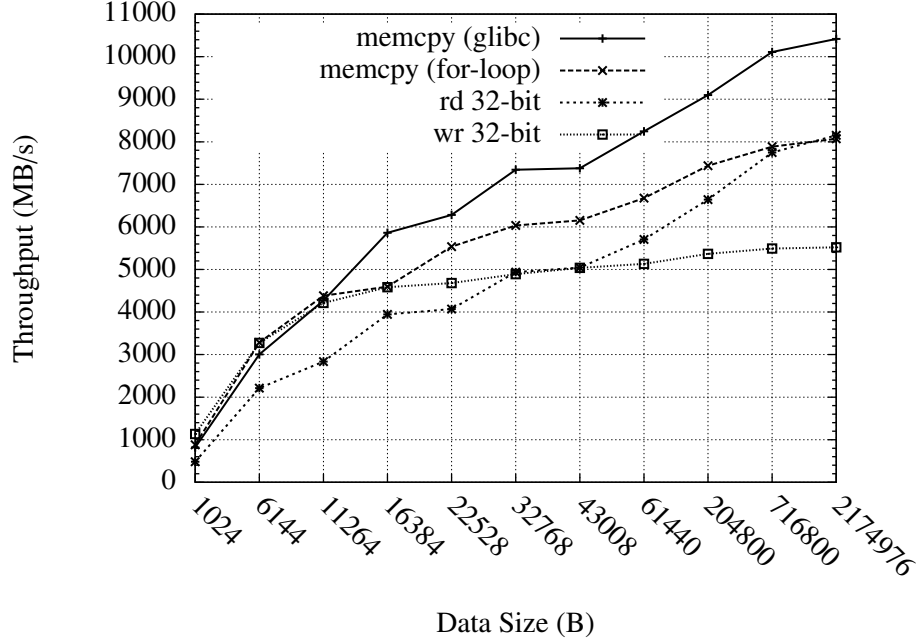


Figure 5.8: Throughput of Host Memory Accesses. Experiment details: Single Thread, Average over 100 iterations.

Second, we measured the scalability of parallel accesses to memory. Contention for shared resources, such as host memory bandwidth, increases with the number of threads accessing the memory. This contention can reach a point that adding more threads does not increase performance. In each experiment, we measured the time required to transfer a fixed total amount of data (64MB) from one location in host memory to another using different number of threads. Each thread gets one chunk of 64MB for transfer. As illustrated in Figure 5.9, the memory bandwidth is already saturated with 2 only threads. Further increasing the number of threads actually reduces performance. This shows that the performance of host memory subsystem is low and sustains only two memory-intensive parallel tasks.

Third, we measured performance of host-device data transfers over PCIe bus. The performance numbers in Figure 5.10 show the throughput achieved over PCIe bus as a function of data transfer size. We measured the performance of both synchronous and asynchronous host-device data transfers. The synchronous transfers via `cudaMemcpy` API call access pageable memory on the host, while asynchronous transfer via `cudaMemcpyAsync` API call directly access non-pageable (pinned) memory buffer for direct memory access (DMA) transfer. As Figure 5.10 shows, asyn-

5.5. EXPERIMENTAL RESULTS

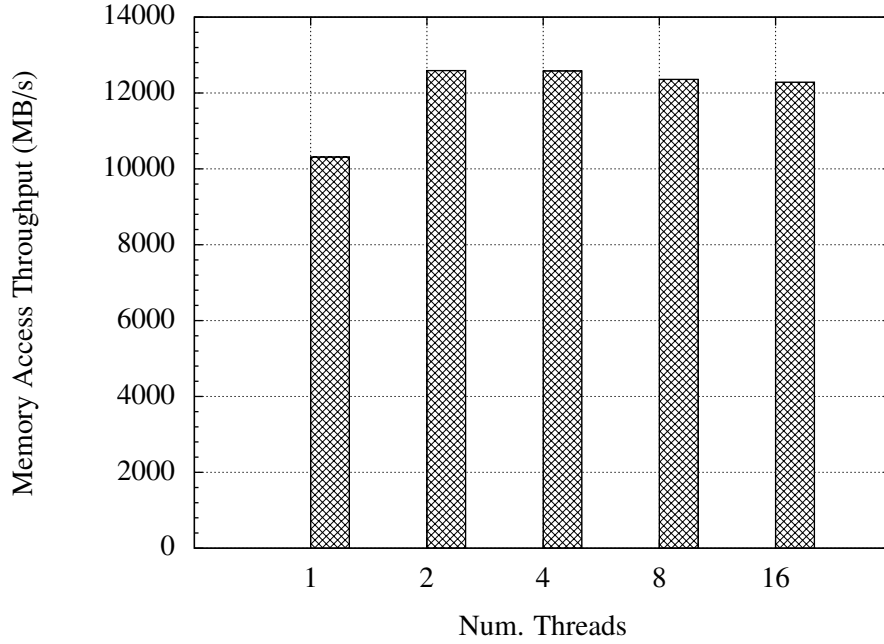


Figure 5.9: Scalability of Host Memory Accesses. Experiment details: Total data size 64MB (fixed), each thread copies an equal non-overlapping portion of data. Memory subsystem saturated with only two threads accessing memory concurrently.

chronous transfers from pinned memory (`htod-async-pinned` and `dtoh-async-pinned`) achieve a steady throughput between 5000 – 6000 MB/s for data transfers larger than 1MB. The performance of synchronous data transfers from pageable memory (`htod-sync-pageable` and `dtoh-sync-pageable`) turned out to be lower in practice and also less predictable. For example, `dtoh-sync-pageable` fluctuates around 1500 MB/s, while the performance of `htod-sync-pageable` suddenly jumps for transfers larger than 60 KB. The significantly better performance of asynchronous transfers can be explained by the use of pinned memory and use of an DMA engine. The pages in pinned memory can not be swapped out of RAM by the operating system, which means that there is no virtual memory management overhead involved in waiting for the operating system to load the pages from the disk. In addition, the use of a DMA engine eliminates the need to involve the CPU in the data transfer. The DMA engine puts directly the data from the pinned memory on the PCIe bus.

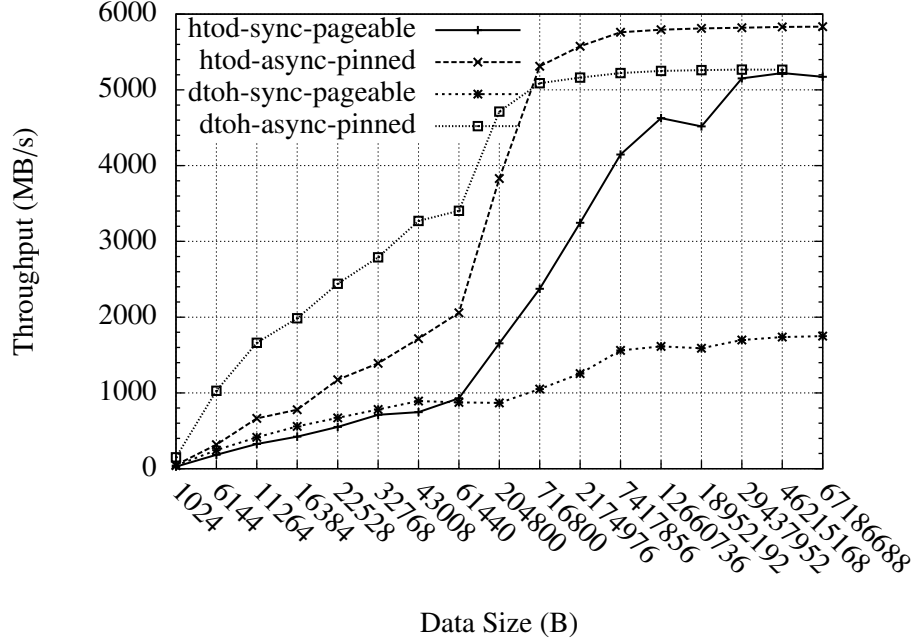


Figure 5.10: Throughput of Host-Accelerator Data Transfers. Comparison of performance for synchronous transfers from pageable memory and asynchronous data transfers requiring pinned memory.

5.5.3 Stream Buffer

We show the maximal communication performance improvement achieved using stream buffer design over synchronous data transfers in Figure 5.11. In this experiment, we transferred data in fixed-size packets over PCIe bus using synchronous host-device transfers from pageable memory, denoted as `default (sync xfers)` trace, and asynchronous host-device transfers from pinned memory, denoted as `streaming (sb - asyncq)` trace. The experiment has been repeated for a large number of iterations. We measured the time to complete a round-trip: we transfer the tokens from the host memory to the GPU memory and back. We show the round-trip throughput (MB/s) as a function of data size. The maximal theoretical round-trip throughput with complete overlapping of data transfers in both directions to $2 \times 8 = 16$ GB/s for a PCIe 2.0 bus. In practice, the host-device data transfers achieve 6 GB/s for large data sizes as illustrated in Figure 5.10. This brings the maximum achievable round-trip throughput with overlapping of data transfers to $2 \times 6 = 12$ GB/s. The largest round-trip that we measured was around 8 GB/s, as shown in the last bar of Figure 5.11.

5.5. EXPERIMENTAL RESULTS

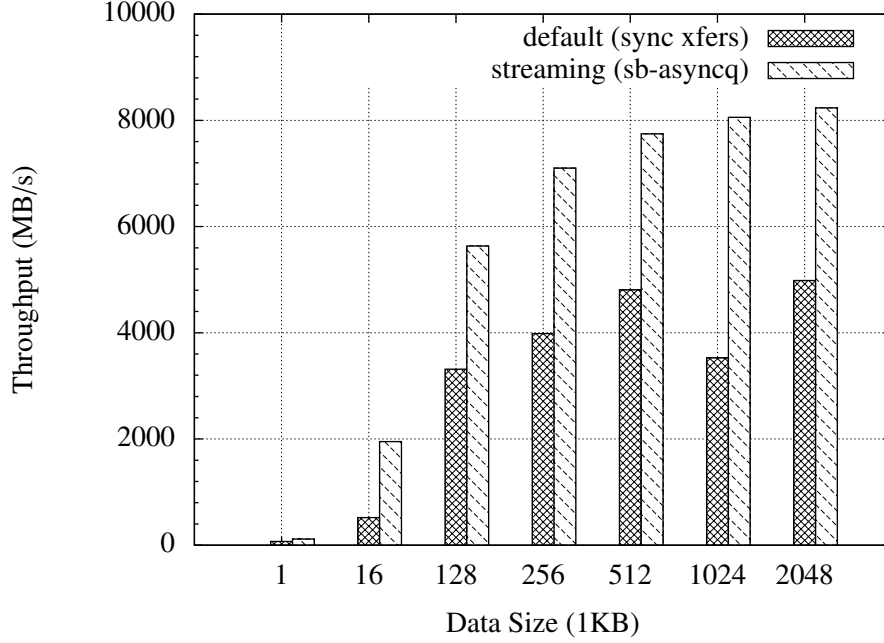


Figure 5.11: Increase of Round-Trip Throughput Using The Asynchronous Stream Buffer Design.

Table 5.5.3 shows the average achieved bandwidth on PCIe bus in each direction in case of default (synchronous) data transfers and streaming (asynchronous) data transfers using stream buffer design. The last column gives percentage of time that the transfers were overlapped in case of streaming asynchronous data transfers for different data sizes. As it can be seen for very small data transfers (1024B) there is no overlap due to large runtime overheads. However, for larger, and thus longer, data transfers, we observed more than 90% of time overlap between host-to-device and device-to-host data transfers.

The percentage of the overlapped time is shown in the last column of Table 5.5.3. We also measured the average PCIe transfer bandwidth for both synchronous and asynchronous designs. The results are given in Table 5.5.3. In line with micro benchmarks, we observed higher bandwidth for host-to-device transfers in both settings. We also observed an interesting anomaly. The experiments revealed that the time to complete individual asynchronous host-to-device and device-to-host transfer increases for overlapped transfers. This indicates that the implementation of the PCIe bus on the system does not support simultaneous transfers in both directions at the full speed of PCI. As a consequence, the average bandwidth of individual host-to-

5.6. CONCLUSIONS

Transfer Type	Data Size [B]	Average H2D PCIe Transfer BW [MB/s]	Average D2H PCIe Transfer BW [MB/s]	Overlap [%]
Default	1024	413.644	605.394	0
Streaming	1024	315.626	313.128	0
Default	131072	5534.44	5161.91	0
Streaming	131072	4408.32	4492.17	34
Default	1048576	6041.44	5531.42	0
Streaming	1048576	4452.45	4550.42	86
Default	2097152	6082.41	5527.5	0
Streaming	2097152	4439.63	4531.64	92

Table 5.1: Percent overlap with streaming

device and device-to-host transfers for the streaming case is somewhat smaller, but due to the overlapping we still achieve significant performance gains.

5.5.4 PPN Execution with Asynchronous Kernel Offloading

Let us now give the performance comparison of SOK and AOK in the context of PPN execution. For the experiment, we used a PPN composed of three nodes connected in a simple P/T/C pipeline. Producer P and consumer process C execute on the host, while transformer T offloads its process iteration onto the GPU.

In Figure 5.12, we show the throughput increase when asynchronous methods are used instead of simple synchronous kernel offloading. The PPN using the synchronous kernel offloading (SOK) is represented by the `ppn – gpu – offload` trace. The PPN using the asynchronous kernel offloading (AOK) is represented by the `ppn – gpu – sb – adam – asynq` trace. The two traces in between are hybrid variants: the `ppn – gpu – sb` trace represents the PPN using the stream buffer design with distributed double-buffering and synchronous communication, and the `ppn – gpu – sb – adam` trace is its optimization that in addition uses direct memory access to buffer slots. In this experiment, we observed significant throughput increase when using the methods developed in this chapter. The AOK approach lead to up to a 4.3× improvement over the traditional SOK approach for kernel offloading.

5.6 Conclusions

To minimize the impact of host-accelerator data transfers, we created a novel method for kernel offloading in PPNs. First, we realized efficient host-accelerator commu-

5.6. CONCLUSIONS

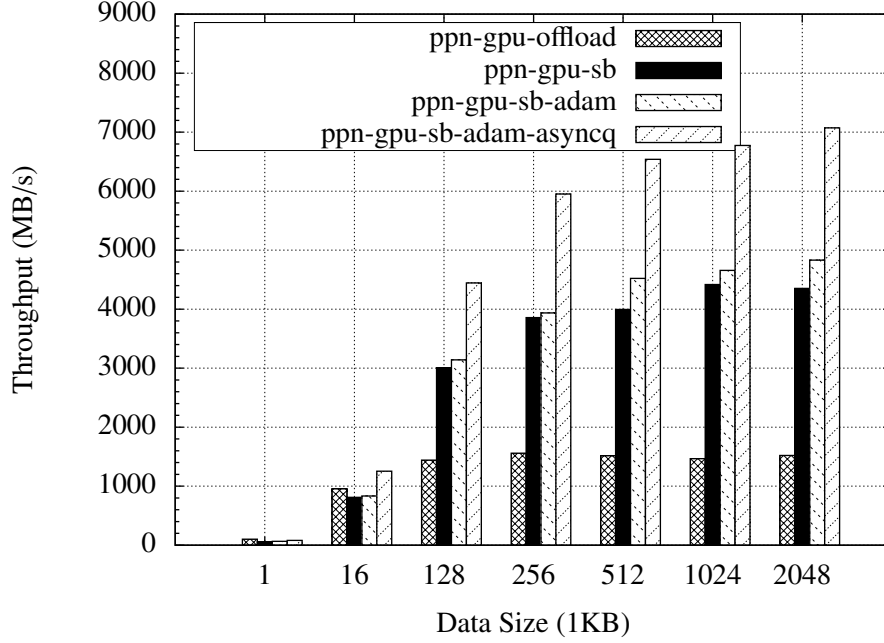


Figure 5.12: Throughput Increase by Using AOK vs SOK in a PPN

nication which supports overlapping of data transfers with computation by taking advantage of GPU streaming concepts and asynchronous data transfers. This lead to the introduction of an efficient stream buffer design for host-accelerator channels. Second, we leveraged the stream buffer design to introduce support for asynchronous kernel offloading in the PPN. The extension of the PPN execution model with support for streaming and asynchronous kernel offloading, results in a model-driven approach for overlapping data transfers and computation on heterogeneous platforms with accelerators.

Chapter 6

M-JPEG Case Study

6.1 Introduction

To evaluate the benefits of the techniques presented in previous chapters, in this chapter we conduct a series of experiments on the M-JPEG encoder. The M-JPEG encoder is a streaming multimedia application from the realm of video compression that performs lossy still-image compression on the stream of input frames, and as a result generates an output data stream of reduced data size. Although the M-JPEG standard defines a relatively simple encoding workflow (in terms of video compression standards), it is still a very interesting application for a parallelization case study since it contains inherent task and data parallelism. On the one hand, the M-JPEG encoder is a typical streaming application, and as such it is easily modelled as a pipeline of tasks. On the other hand, it contains computationally intensive tasks, such as discrete cosine transform (DCT), that feature inherent data parallelism. As such, M-JPEG encoder provides rich experimentation opportunities with different types of parallelism. As the basis for our experiments, we adapted the M-JPEG encoder that was originally developed at LERC to demonstrate the COMPAAN/LAURA approach [121].

6.2 M-JPEG Encoder and its PPN

An overview of the M-JPEG encoder workflow is given in Appendix C.1. The SANLP of the M-JPEG encoder used for experiments is given in Listing C.2. Running the code in Listing C.2. through the COMPAAN compiler results in the PPN shown in Figure C.2. The M-JPEG PPN automatically generated by the COMPAAN compiler is a four-node pipeline. The processes exchange data via tokens, which by default correspond to 8×8 pixel blocks. The four processes are connected by channels im-

plemented as FIFO buffers. For each of the four processes, a task is generated in C. Following the typical PPN implementation and mapping approach, each PPN process is assigned for execution to a single asynchronous processing entity, implemented as a POSIX thread. Each POSIX thread is typically assigned for execution to a different core of the platform's multicore CPU. The obtained parallel program exhibits task and pipeline parallelism. The code within the tasks is sequential. The whole PPN executes in asynchronous data-driven fashion.

6.3 Experimental Setup

6.3.1 Application Configuration

We performed experiments by running the M-JPEG PPN on a stream of 100 frames. Each frame is a color image of 128×128 pixels. In the given M-JPEG encoder implementation, the source image is given in YUV color space. Frames are split in 8×8 -pixel blocks (1 KB each), which are processed independently. The baseline performance results are obtained by running the default PPN automatically generated by running the `COMPAA` compiler on the M-JPEG SANLP. The experiments in the subsequent sections are designed to evaluate performance improvement over default PPN that are achieved by applying techniques from the previous three chapters.

6.3.2 Platform

The test platform used for all experiments in this chapter features an Intel Core i7-920 Nehalem architecture 2.66GHz processor, Intel Motherboard, and an NVIDIA Tesla C2050 GPU. The Tesla C2050 GPU has 448 streaming processors (SPs) organized in 14 streaming multiprocessors (SMPs) with 32 SP cores each. The Intel Core i7 (Nehalem) is a multi core, Hyper-threading technology (HT) enabled design [44]. Each socket supports one to eight cores, which share a last level cache (L3), a local integrated memory controller (IMC) and an Intel QuickPath Interconnect (QPI). The interface to the GPU is via PCIe 2.0 x16 bus. The microbenchmarks for the performance of the host memory subsystem and the host-GPU link are given in Section 5.5.2. The NVIDIA Tesla C2050 GPU is a second-generation GPU for general purpose computing featuring the Fermi architecture [105]. The Fermi architecture is the first CUDA-capable architecture with support for the task parallelism. In addition, the Tesla-line of GPUs contains two DMA engines, which are typically found only in high-end GPUs for high performance computing. As a result, Tesla C2050 GPU supports concurrent DMA transfers from and to the GPU over PCIe bus.

6.3.3 Experiments

To obtain the baseline, we measured the **initial performance** of the default PPN obtained by the COMPAAN compiler and analyzed its results. Our observations on the performance of the task-parallel M-JPEG are given in Section 6.4.

First, we applied techniques for multi-level parallelization presented in Chapter 4 to obtain a multi-level M-JPEG, illustrated in Figure 4.2. At the top-level (Level1) of the program we again constructed a task-parallel program from a PPN, but with one significant difference. The tokens processed and exchanged between the processes in the new M-JPEG PPN have different granularity than the tokens in the default PPN. The tokens in the new M-JPEG PPN are not limited to 8×8 pixel blocks, but they can also represent entire frames. We show the impact of the **token granularity** adjustment in the PPN in Section 6.5.

Second, at the bottom-level (Level2) of the multi-level program model depicted in Figure 4.2, there is a node T which executes the DCT computation. We further parallelized this node to reveal its **data parallelism**. Using the techniques described in Chapter 3, we obtained a data parallel CUDA kernel for DCT. We first measured in isolation the computational performance of the default kernel resulting from the KPN2GPU, and then showed the improvements after including the optimizations proposed in Chapter 3. We then show the total speedup of the DCT processing on the GPU which is affected not only by the computation but also by the time to transfers the input data for the DCT to GPU and to transfer the results back. The overall results of DCT processing on Tesla C2050 GPU are given in Section 6.6.

Third, we measured the performance of the **overall solution**. The result of Chapter 4 is a multi-level parallel program that features task, data, and pipeline parallelism. The coarse-grain tasks are mapped onto platforms devices (CPUs, GPU). At the platform level (Level1), we exploit task and pipeline parallelism. At the component level (Level2), we explore data parallelism in the DCT node by executing its process iterations on Tesla C2050 GPU. To obtain the overall results, we measured the performance of the multi-level parallel M-JPEG using the stream buffer design with **kernel offloading** methods presented in Chapter 5. The results for the overall solution are given in Section 6.7.

6.4 The Performance of Default M-JPEG PPN

To evaluate the performance of the initial task-parallel program obtained using PPN model, we compared the throughput of the sequential M-JPEG encoder with the throughput of the default parallel M-JPEG encoder derived from the PPN model. The performance of the default PPN model can already be improved using a waste range of techniques, such as node splitting and merging [93], which were developed

6.5. ADJUSTING TOKEN GRANULARITY BY ENCAPSULATION

in the DAEDALUS context at LERC [2]. In this chapter, we are interested to measure only the incremental improvement stemming from the techniques proposed in Chapters 3,4, and 5, and for this purpose we use the default PPN derived by the COMPAAN compiler as the baseline.

We report the results as the average throughput in KB/s for the entire stream. The throughput measured for the sequential M-JPEG encoder amounts to 385 KB/s. The throughput measured for the automatically generated parallel program obtained from the M-JPEG PPN is 470 KB/s, which corresponds to an improvement of approximately 22% over the sequential version. To estimate the computational demands of M-JPEG encoder tasks, we profiled the sequential application and compared the execution time spent in each stage. The results are shown in Table 6.4.

M-JPEG Task	vin	dct	q	vle
Execution Time [%]	2.50	48.50	25	24

Table 6.1: Percentage of M-JPEG Execution Time spent in each task.

The analysis of M-JPEG performance lead to the following observations.

First, Table 6.4 shows that the DCT node is the bottleneck node of the M-JPEG pipeline. As the execution time of a bottleneck node determines the overall execution time of a pipeline application, the overall performance could be potentially improved by acceleration of the DCT node.

Second, the overall application performance in the parallel case is also affected by large amount of (physical) data movement operations involved in realizing the PPN channel-based communication.

Third, the default M-JPEG PPN uses small fixed-size tokens corresponding to 1 KB image blocks. The use of small tokens leads to inefficient data transfers as indicated by experiments conducted in Section 5.5.2 (Figure 5.8). In addition, small tokens require large amount of synchronization between nodes.

In the subsequent experiments, we show how the techniques proposed in the previous chapters alleviate these issues.

6.5 Adjusting Token Granularity by Encapsulation

A well known problem in parallel applications is the cost of communication and synchronization, as it can easily outweigh the benefits of parallelization [118]. The overall percentage of time spent in synchronization can be reduced if we could reduce the number of synchronization points. In the PPN model, the synchronization points are induced by blocking read and blocking write operations. The number of blocking operations can be reduced by reducing the number of process iterations that are exe-

6.5. ADJUSTING TOKEN GRANULARITY BY ENCAPSULATION

cuted in order to process the same amount of input data. We achieve this by packing blocks into larger composite tokens. This is realized by using the token composition concepts and techniques presented in Chapter 4.

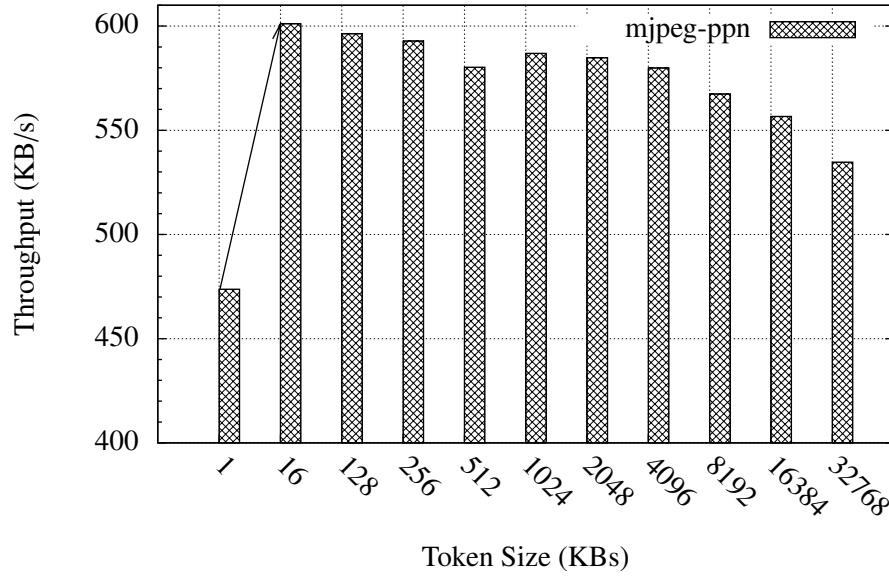


Figure 6.1: Effect of Token Granularity on Parallel Execution Time

The default M-JPEG PPN generated by the *COMPAAN* compiler communicates tokens of the size of a single image block between the processes. For example, in order to process a sequence of 10 frames, with each frame containing 8×16 blocks, each PPN process executes 1280 iterations. There are two synchronization points in each process iteration (one for blocking on read and one for blocking on write). This results in 2560 synchronizations during process execution. By coarsening the granularity of the composite token to token cardinality $TC = 128$ blocks, the number of synchronizations is cut by the factor of MR , i.e. only 200 synchronization points are required for the complete frame sequence.

The result of token granularity coarsening is shown in Figure 6.1. The horizontal axis represents token size as a multiple of 1 KB units, i.e. blocks. The *mjpeg-ppn* bars show the throughput of the task-parallel M-JPEG obtained by using the PPN model of computation in KB/s. The first bar corresponds to the reference performance of the default PPN generated by the *COMPAAN* compiler. The subsequent bars show how the PPN throughput changes with the change in token granularity. We

observed an improvement of the `mjpeg-ppn` execution time of 27% after we tuned the token granularity size. The optimal token size for M-JPEG execution following PPN model of computation on the given test platform is found to be 16 KB. As a result of token granularity coarsening, the speedup over sequential version increased from 22% for the initial PPN to 55% for the PPN with the adjusted token granularity.

6.6 Leveraging Data Parallelism for GPU Acceleration

6.6.1 DCT Kernel Execution

Combining task and data parallelism has been considered in many areas of parallel and distributed computing - from programming large scale distributed systems to programming second-generation GPUs with support for concurrent kernel execution [18] and Cell B.E. [133]. We pursue this idea by exploiting inherent data parallelism in DCT node of the M-JPEG encoder. We first performed the hierarchical splitting of M-JPEG. At *Level1*, we generate a PPN featuring task parallelism at selected token granularity. The blocks in each composite token that comes into the DCT node are offloaded for processing on GPU SMPs. To generate a CUDA kernel for the DCT computation, we created a *Level2* PPN from the code in `mainDCT` function (See Appendix C, Listing C.2) Following the approach outlined in Chapter 3, we obtain a pipeline of data parallel CUDA kernels. Each 8×8 block is processed by a thread block consisting of 64 CUDA threads.

Let's now evaluate the performance of the CUDA code obtained using the techniques presented in Chapter 3. The support for composite tokens introduced in Section 3.8.3 allows us to obtain a pipeline of 14 data parallel kernels (one kernel for each loop nest in `mainDCT` function). Figure 6.2 shows the summary of the computational speedups achieved by executing DCT computations on the GPU. The horizontal axis shows the number of composite tokens processed on the GPU as a multiple of 1 KB blocks. The vertical axis shows computational speedup as a function of GPU execution time divided by CPU execution time for the same M-JPEG code.

The baseline performance of the data parallel DCT CUDA code produced in line with Chapter 3 is shown as trace `kpn2gpu-dp-def` in Figure 6.2. It achieves a speedup of only $3.5\times$ compared to the sequential DCT code running on the CPU. This is primarily due to the excessive global-memory communication and kernel launch overheads caused by mapping each PPN node of the DCT computation pipeline onto a separate kernel. To improve the performance we manually modified the code to include the optimizations presented in Section 3.8:

- First, to avoid kernel launch overheads, we merged 14 kernels of the DCT computation pipeline into a single CUDA DCT kernel.

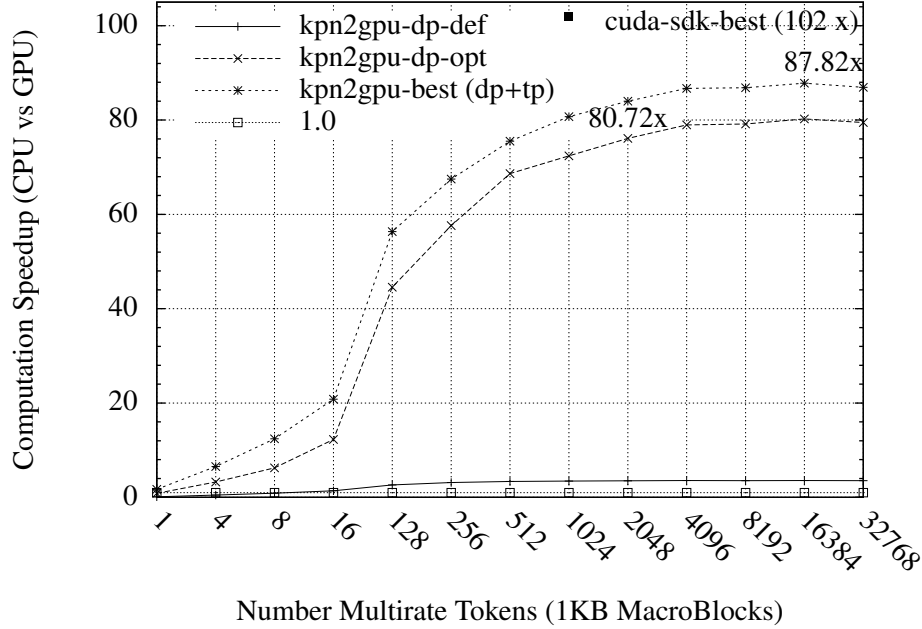


Figure 6.2: Computational Speedup is given as a ratio of the time required for sequential processing of reference C code on a single CPU core vs the GPU processing time

- Second, we remapped the PPN channels connecting 14 nodes from GPU's global to GPU shared memory.
- Third, we increased amount of data reuse by exploiting the multiplicity property discussed in Section 3.8.3 to load the input blocks only once into a shared memory buffer instead of reading it again from the global memory for each thread separately.

The resulting data parallel DCT implementation is shown in Figure 6.2 as the **kpn2gpu-dp-opt** trace. This optimized version is approximately 80× faster than its sequential source code executing on the CPU.

As the last optimization, we added task parallelism stemming from independent processing of different color channels (YUV). The results are presented as **kpn2gpu-best-(dp + tp)** trace, which reaches a speedup of 87× for 16234 KB input data size. The hand-optimized DCT [101] in the CUDA SDK achieves a 102× speedup over its sequential counterpart executed on the same CPU.

The CUDA DCT performance increase from only 3.5× to 87× speedup has been achieved as a combination of reducing synchronization costs by merging PPN nodes

6.6. LEVERAGING DATA PARALLELISM FOR GPU ACCELERATION

in the pipeline, reducing communication costs by smarter mapping of PPN channels to the rich memory hierarchy of the GPU, and taking advantage of data reuse in the kernel by exploiting the multiplicity property for improving data locality. The tremendous performance jump achieved by leveraging the concept of multiplicity in the PPN specification points to the relevance of further work on directions outlined in Section 3.8.3.

6.6.2 Offloading Computation to the GPU

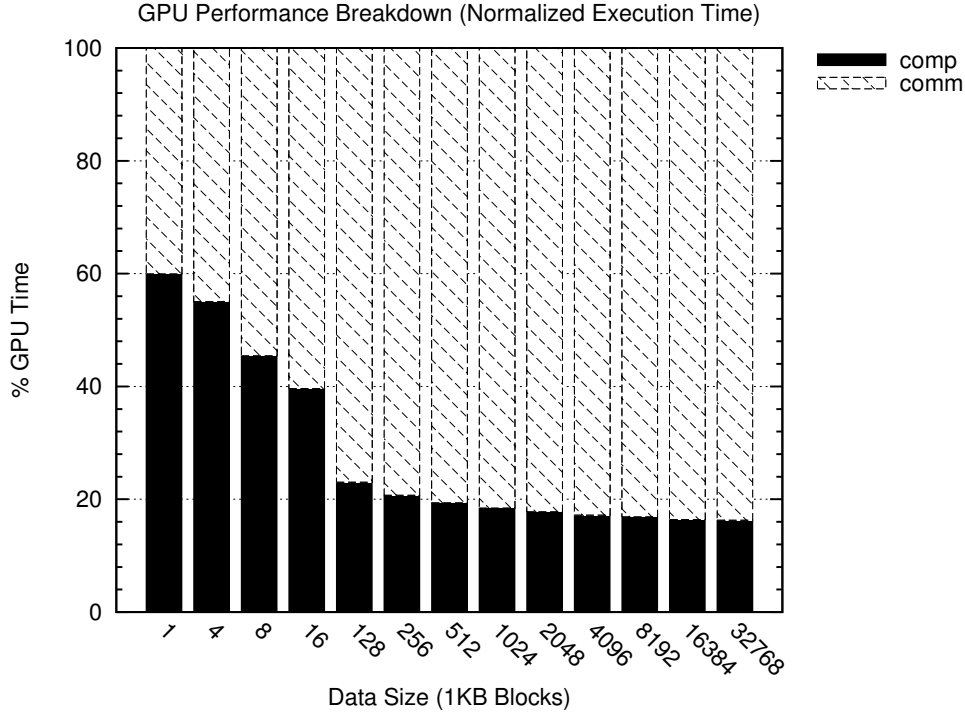


Figure 6.3: Breakdown of GPU time: Time to compute the DCT kernel `comp` vs. the time spent in transferring the input and output data to the host (`comm`).

The computational speedup is only a single side of the performance coin. To describe the effect of GPU acceleration on the overall M-JPEG application performance, it is also necessary to consider communication and runtime overheads. Figure 6.3 depicts normalized GPU time versus data size. GPU time in this section denotes the total time required to send the data to the GPU, compute it on the GPU, and send it back to host. The GPU time is measured on the GPU using the NVIDIA's CUDA profiler tool. Each of the bar shows the ratio of time spent in communication

6.6. LEVERAGING DATA PARALLELISM FOR GPU ACCELERATION

(data transfers from host to GPU, and from GPU to host) and time spent in computation on the GPU, i.e. kernel execution time. For small number of elementary tokens (1KB blocks), the time to transfer the data is still comparable to the computation time. However, as the utilization of the GPU increases, the computation throughput grows. The time to transfer the data becomes significantly larger than the actual GPU computation time. For larger data sizes (higher token cardinality (TC) factor), communication takes up more than 80% of the GPU time. This points to the significance of (a) being able to adjust the token granularity, (b) overlap the transfers. The signif-

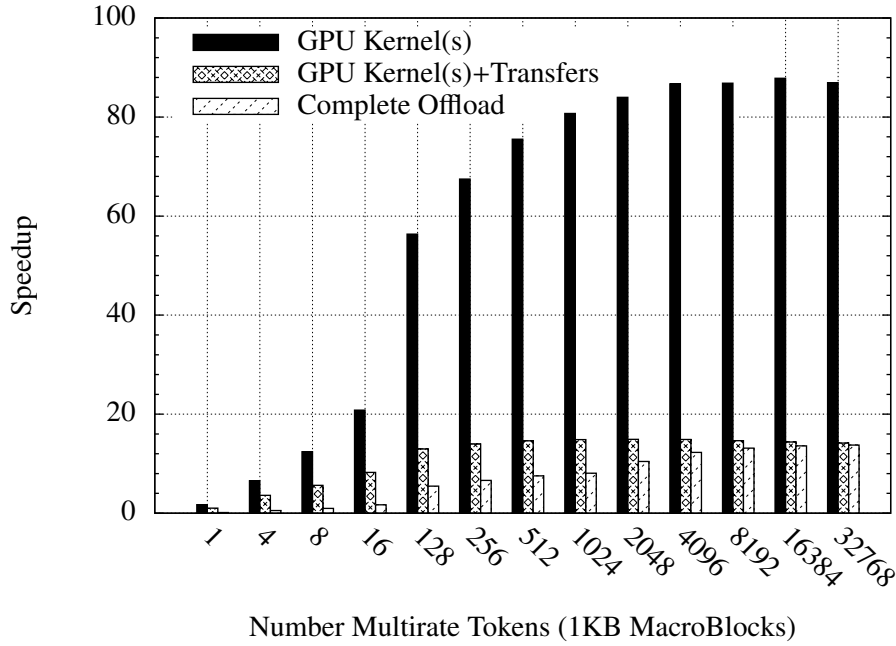


Figure 6.4: Comparison of speedup achieved with offloading DCT transform onto GPU: (a) computational speedup, (b) gpu (comp+comm) speedup, (c) complete offloading speedup including runtime overheads.

ificance of adjusting token granularity was already shown in Section 6.5. Before we discuss the transfer overlapping for which we introduced the concepts in Chapter 5, let us first see what is the overall performance gain achieved by DCT acceleration on the GPU when data transfers are taken into account. Figure 6.4 shows a comparison of the speedups achieved by measuring the following times: (a) kernel computation time on the GPU (denoted as GPUKernel(s)), (b) kernel computation time and (synchronous) data transfers to the GPU (host-to-device and device-to host)(denoted as GPUKernel(s) + Transfers), (c) kernel computation time, data transfers (denoted

6.7. OVERALL PERFORMANCE RESULTS WITH GPU ACCELERATION

as `CompleteOffload`), which additionally accounts for CUDA runtime overheads. The results for variants (a) and (b) are obtained using CUDA profiler to record timestamps on the GPU, while the results for variant (c) are measured using a CPU-based timer which was started before the start of synchronous offload and stopped after the synchronous offload has returned control to the calling thread. The vertical axis shows speedup, as the remainder of the CPU execution time for sequentially executed DCT source code with the GPU time for three different measurement scenarios described above. The horizontal axis shows data size as the number of 1KB blocks sent to a GPU for processing. So far, we found out that the best KPN2GPU variant of `mainDCT` achieves more than 12 GB/s computation throughput on the GPU. Compared to the throughput of the original DCT code which is 180 MB/s, this results in approximately $87\times$ computational speedup. However, when data transfers and runtime overheads are taken into account, the overall performance numbers drop since extensive amount of time goes on transferring the data to and from the GPU. When communication to/from the GPU is included, the best performing code achieved slightly more than 2 GB/s throughput. This corresponds to approximately $14\times$ speedup with data transfers. Finally, when all run time overheads for synchronous offloading are taken into account, We found out that for < 128 blocks, overheads of offloading to the GPU cause a slow-down instead of speed-up. The maximal achieved speedup with synchronous kernel offloading is reduced to $13.76\times$ for large data sizes. There is also a further performance drop for complete offloading of small data sets. However, we believe that Figure 6.4 may be giving an overly pessimistic result for the complete offload performance, since the execution time for smaller data sizes lasts shorter and the measurement may be affected by timer precision. For the same reason, the recorded performance may also suffer from synchronization required on the CPU side to complete the measurement. Detailed time breakdown and investigation of overheads affecting offload performance within a complete application would definitely require more sophisticated performance analysis tools for hybrid application than the tools publicly available today. In any case, the results presented above point out the importance of efficient offloading and efficient communication with the GPU, and the benefit of leveraging asynchronous transfers for overlapping communication and computation whenever possible.

6.7 Overall Performance Results with GPU Acceleration

In this section, we analyze the overall performance obtained by generating and mapping a two-level parallel program using the methods described in previous sections. Partial results for token granularity and GPU offloading were given in the previous sections. In this section, we show these results in the context of complete M-JPEG

6.7. OVERALL PERFORMANCE RESULTS WITH GPU ACCELERATION

PPN execution with task, data, and pipeline parallelism on the streaming platform.

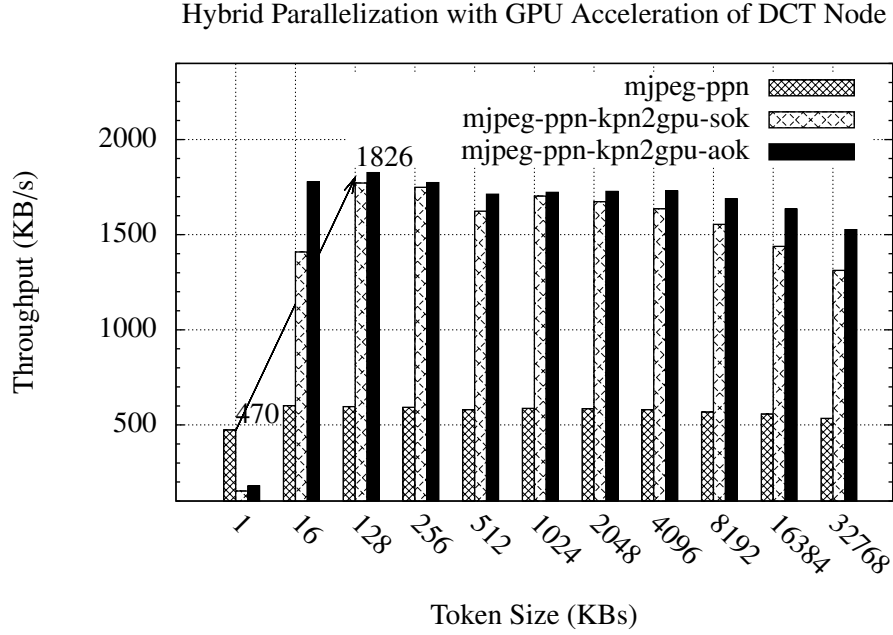


Figure 6.5: Hybrid PPN Parallelization with Data Parallel DCT (Converted with KPN2GPU) and Host-Accelerator Stream Buffers as a Function of Token Size.

Figure 6.5 depicts PPN application throughput as a function of token size in the top-layer PPN. The bar chart gives performance comparison of the following PPN variants:

- mjpeg-ppn
- mjpeg-ppn-kpn2gpu-sok
- mjpeg-ppn-kpn2gpu-aok

The first variant mjpeg-ppn represents the initial PPN obtained by the COMPAAN compiler, converted into a multi-threaded program, and executed on the multicore CPU. The second variant mjpeg-ppn-kpn2gpu-sok is an extension of the initial PPN with synchronous offloading of parallelized DCT to the GPU. The third variant mjpeg-ppn-kpn2gpu-aok is a variation of mjpeg-ppn-kpn2gpu-sok, which uses the principles presented in Chapter 5 for asynchronous communication and asynchronous DCT kernel offloading.

6.7. OVERALL PERFORMANCE RESULTS WITH GPU ACCELERATION

For all variants, the low point occurs for the default PPN token size which corresponds to a single block. For 1 KB tokens, the throughput of `mjpeg-ppn` is 470 KB/s, while the throughputs of `mjpeg-ppn-kpn2gpu-sok` and `mjpeg-ppn-kpn2gpu-aok` are even below 100 KB/s. This dramatic performance drop is due to the inefficient processing on the GPU when the whole workload contains only one single block. In this case, there is simply not enough work to utilize multiple SMPs on the GPU. All 14 SMPs on the Tesla C2050 are idle apart from a single SMP. In addition, the kernel offloading overheads for small token data sizes are large in comparison to the duration of the token processing as shown in Section 6.6.2.

The two observed problems can be alleviated by processing composite tokens, which have more than 1 block. In Section 6.5 we demonstrated the performance improvements achieved on the multi-core CPU by tuning the PPN token size. The use of TC tokens is however not only beneficial for reducing the amount of synchronization in PPN, but it also allows us to achieve better utilization of GPU SMPs. Figure 6.5 clearly shows a considerable performance leap that occurs with increase of the token granularity for all three variants. The performance increase is much larger for PPN variants with GPU acceleration of DCT, as the efficiency of GPU processing increases with the number of independent work items. With composite token tokens it is possible to occupy multiple SMPs, and thus achieve much higher GPU accelerator utilization. For example, with token cardinality factor $TC = 128$, 128 blocks are processed on the GPU in a single offload. As a result of better GPU utilization and smaller runtime overheads, the overall performance of the parallel M-JPEG with GPU acceleration increases from 480 KB/s to 1826 KB/s.

Use of the AOK mechanism brings an improvement of 5-28 % to the overall performance depending on the data size. Contrary to our expectations, the performance improvement in this case is not primarily due to asynchronous data transfers and thus streaming to/from the GPU. In depth investigation of GPU figures within CUDA GPU profiler tool revealed that there are only a few overlapping data transfers between host and GPU accelerator, since the best M-JPEG PPN data rate is only around 460 KB/s. This is much smaller than the transfer rate of the PCIe link connecting the GPU to the host, which is around 4 GB/s. For any overlaps to be possible, the data rate of the application must be much higher, preferably closely matching or larger of the link data rate. Otherwise, there are large gaps between the GPU transfers, and there is nothing to be overlapped. The primary performance impact of AOK in this case comes from the reduction of communicated data volume achieved by using the more efficient PPN channel design which reduces the pressure on the memory subsystem.

The most significant part of performance improvement comes from the speed-up gained by data-parallel processing of `mainDCT` node on the GPU. Detailed performance/time breakdown is illustrated for three different token sizes for the first variant `mjpeg-ppn` in Figure 6.6 and for the second variant `mjpeg-ppn-kpn2gpu-sok` in

6.7. OVERALL PERFORMANCE RESULTS WITH GPU ACCELERATION

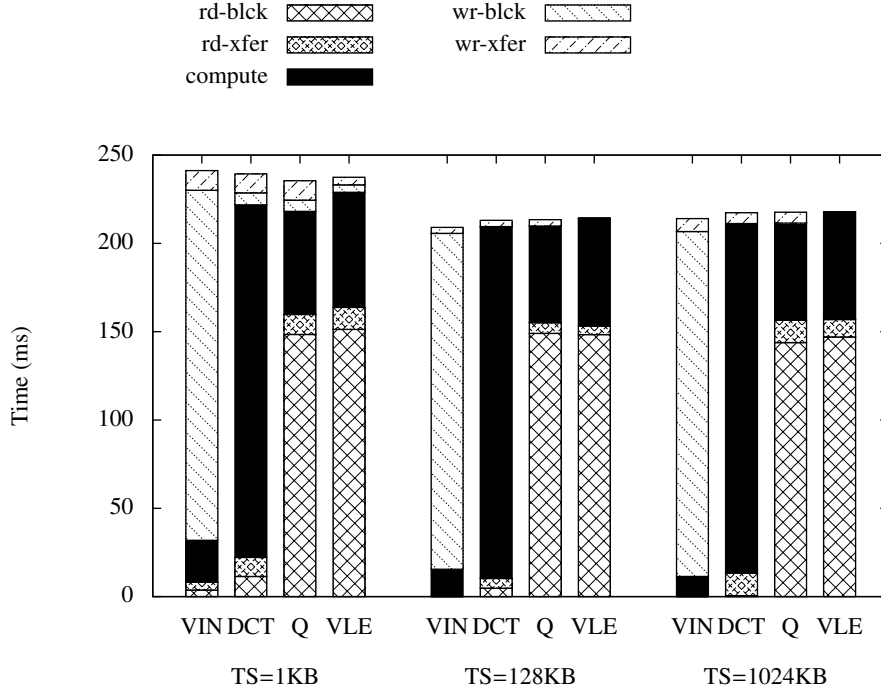


Figure 6.6: Execution Time Breakdown for Parallel PPN Execution of M-JPEG Tasks (`mjpeg-ppn`). Each task executes a five-phase sequence: block on read, read data transfer, computation, block on write, write data transfer.

Figure 6.7. The detailed performance breakdown for the third variant `mjpeg-ppn-kpn2gpu-aok` is left out due to technical hurdles involved in detailed performance benchmarking when using asynchronous data transfers and execution on the GPU.

For token size 128 KB (stemming from the token cardinality factor $TC = 128$), the DCT task amounts to 60.3% of total computation time, as illustrated by the second stacked histogram in Figure 6.6. By offloading the DCT task to the GPU and processing it in data parallel manner, the compute phase of the DCT task with GPU offloading is reduced to 22% of the total execution time. The performance breakdown with DCT execution on the GPU is illustrated by the second stacked histogram in Figure 6.7.

After GPU acceleration of DCT, the new bottleneck task is VLE with 37.6% computation time. Since the new bottleneck is the VLE task, further work on the optimization of the DCT computation on the GPU would have a small impact on the overall execution time. The new bottleneck task VLE is difficult to parallelize for GPU, due to its computational structure. While in [13], we presented a novel approach for parallel variable-length encoding (PAVLE) on the GPU, this parallelization approach requires use of a completely different algorithm and is a non-trivial task for a compiler. Some

6.8. CONCLUSIONS

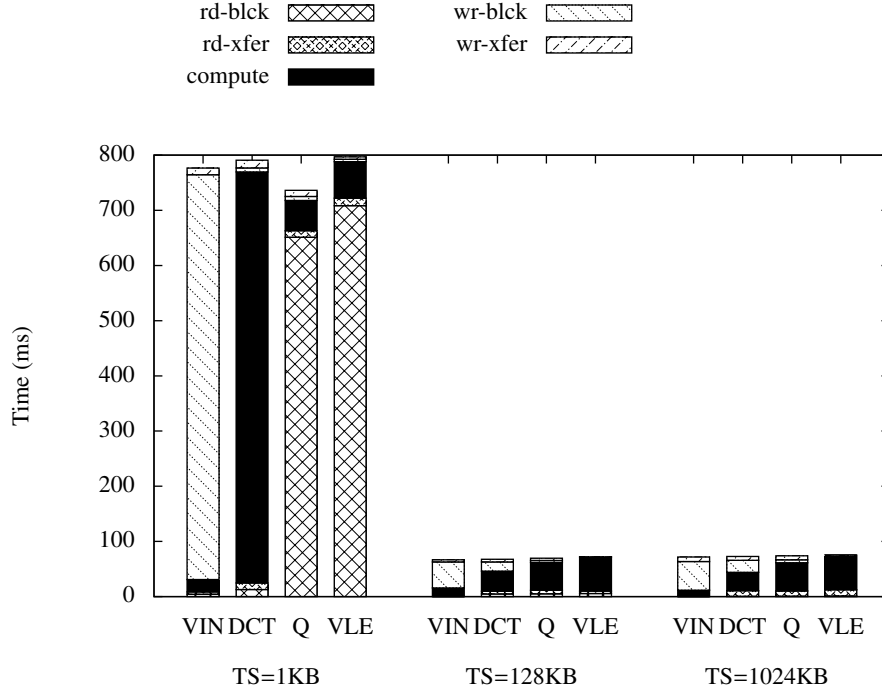


Figure 6.7: Execution Time Breakdown for Parallel PPN Execution of M-JPEG Tasks with DCT Task Offloading to a GPU Accelerator (`mjpeg-ppn-kpn2gpu-sok`). Compute bar for DCT task represents complete GPU offloading time. The offloading time is composed of data transfers to/from GPU, sub-PPN computation on a GPU, and CUDA runtime API overheads.

further gains are achieved with the use of stream buffer design since it efficiently reduces pressure on the memory subsystem due to its improved data access protocol and thus improves the overall application performance.

6.8 Conclusions

In this chapter, we performed a case study on multi-level parallelization on the example of M-JPEG encoder. Using concepts and techniques presented in previous chapters, we generated a top-level PPN featuring task and pipeline parallelism with adjusted token granularity. Parallelization of the `mainDCT` functionality using the `KPN2GPU` tool revealed inherent data parallelism within the DCT node. Finally, we mapped the two-level PPN onto a heterogeneous platform with a modern Tesla C2050 GPU accelerator. We exploited both data and task parallelism within the DCT node for efficient mapping onto second-generation Fermi-architecture architecture GPU, and

used the stream buffer design to improve efficiency of PPN communication channels. After transformations with the techniques presented in the previous chapters, the resulting multi-level program performs $4\times$ faster compared to the default task-parallel PPN obtained by the COMPAAAN compiler.

6.8. *CONCLUSIONS*

Conclusion

Modern heterogeneous platforms support not only the traditional types of parallelism, such as ILP, vector, data, task, and pipeline parallelism, but also provide platform-level and component-level parallelism, i.e. it is possible to exploit not only a number of different architectural components on the platform, but also different types of parallelism within each component. To exploit the rich parallelism opportunities offered by heterogeneous platforms, we believe that having a multi-level program model is a prerequisite. This thesis makes a first but important step in this direction by introducing a hierarchical internal representation into the polyhedral framework, and a novel method to derive this representation from the standard polyhedral program model and then transform it into a multi-level program (MLP) featuring different forms of parallelism. As such, this thesis opens doors for future research on highly efficient tailor-made parallel program generation and auto-tuning for the next generations of multi-level heterogeneous platforms with diverse accelerators.

7.1 Summary of Work and Contributions

In this thesis, we presented a novel methodology for transformation and more efficient mapping of streaming applications onto heterogeneous platforms with GPU accelerators. Compared to homogeneous multicore platforms, heterogeneous platforms pose not only additional, but also more complex programming challenges to application designers. On the one hand, heterogeneous platforms offer multiple levels and multiple types of parallelism. On the other hand, the sheer range of parallelization opportunities makes it harder to implement and map parallel applications onto such platforms. The concepts and techniques presented in this thesis are designed to exploit task, data, and pipeline parallelism on heterogeneous platforms with massively

7.1. SUMMARY OF WORK AND CONTRIBUTIONS

data-parallel accelerators (e.g., GPUs).

On the conceptual level, our work revolves around HiPRDG, the hierarchical intermediate program representation that we have introduced into the polyhedral model (Chapter 4). The intermediate representation of an application in the form of the HiPRDG facilitates automatic derivation of structured multi-level programs featuring different types of parallelism at each level. For example, by first transforming the application into HiPRDG and then generating the parallel code, it is possible to obtain a top-level PPN with coarse-grain tasks which can be executed in a pipeline fashion. Further, it is possible to discover fine-grain data parallelism at the next level of the HiPRDG, and automatically generate task and data parallel kernels using the KPN2GPU tool which implements the parallelization approach described in Chapter 3. Finally, using the offloading principles described in Chapter 5, task and data parallel kernels obtained by the KPN2GPU can be offloaded for computation on a GPU. Moreover, the stream buffer design enables streaming of data to the GPU, and overlapping of computation and communication. The multi-level parallel program obtained in this manner can execute in asynchronous data-driven manner on a heterogeneous computing platform.

Contributions The main contributions of this thesis are the following:

- (I) A method for discovery and exploitation of data and task parallelism in PPN representation for mapping onto massively parallel accelerators, such as GPUs.
- (II) A novel hierarchical program representation in the polyhedral model and a method for hybrid generation of multi-level (parallel) programs with token granularities adjustable at each level.
- (III) A novel solution for efficient stream buffer design for model-based overlapping of communication and computation on heterogeneous platforms with discrete accelerators.

Contribution I We presented a systematic approach for identification, extraction and modeling of fine grain data parallelism in the polyhedral process network (PPN) specification in Chapter 3. We leverage a state of the art method to identify data parallelism and introduce a data parallel view (DPV) onto PPNs. The DPV is an extension of the PPN model with concepts required for capturing data parallelism within PPN nodes and channels. As an extension to the COMPAAAN compiler framework, we implemented the KPN2GPU tool. The KPN2GPU tool detects data parallelism in the program specification, builds a DPV model which captures data and task parallelism, and provides a CUDA compiler back end for automatic kernel and host code generation. In addition, we provide a model-based method for exploiting the

recently introduced hardware/software support for concurrent kernel execution [105] on modern compute-capable GPUs. This makes it possible not only to exploit data parallelism, but also to exploit task-parallel execution for better GPU utilization.

Contribution II In Chapter 4, we introduced a novel hierarchical program representation in the polyhedral model called *Hierarchical Polyhedral Reduced Graph* (HiPRDG). HiPRDG is an intermediate program model based on polyhedral representation that spans multiple levels. At the top-level, there is a standard polyhedral reduced dependence graph (PRDG) which can be used as a starting point for model traversal. Zooming in into each node of the top-level PRDG reveals the definition of the node’s statement. The definition is captured in form of a PRDG and stored within a node at the lower level of hierarchy. By providing a modular representation that is structured into multiple levels of hierarchy, the HiPRDG model is well suited to be used as a basis for hybrid generation of multi-level parallel programs. To split a standard program model into a multi-level program model we introduced a technique called *slicing*. We first decide on the number of levels and desired token granularity for each level, and then slice the program model at the loop levels (depths) corresponding to the desired token granularity at each level. Finally, we showed how to generate a coarse-grain PPN of a streaming application for exploiting task and pipeline parallelism at the platform-level. Further, we used the notion of hierarchy in the model to generate data-parallel bodies for selected tasks for leveraging massive parallelism within the accelerator. The result of the derivation is a hybrid (task, data, and pipeline parallel) multi-level parallel program which can be mapped to a heterogeneous platform with a data parallel accelerator in a more efficient manner than the initial PPN. Case studies in Chapter 6 showed that by leveraging hybrid multi-level parallelization and accelerator offloading a 4× performance improvement can be achieved compared to the performance of the initial PPN. In addition, the techniques presented in Chapter 4 implicitly enable adjustment of token granularity, thus also addressing one of the key challenges for mapping streaming applications onto heterogeneous platform. Adjustment of the token granularity alone has been demonstrated to improve performance of the task-parallel execution by 27% on a multi-core CPU, with even more profound impact on the GPU execution.

Contribution III In Chapter 5, we extended the baseline PPN mapping model with concepts required for offloading computation of PPN nodes on a GPU. We also introduced the communication protocol which significantly reduces the amount of data movement involved with PPN implementation on shared-memory architectures (such as multicore CPUs), thus reducing the pressure on the memory subsystem. As the main part of Chapter 5, we introduced an efficient *stream buffer* design [17] for communication between a pair of producer-consumer processes executing on heteroge-

neous devices. Using the stream buffer design for host-accelerator communication, data transfers to/from GPU and computation can be pipelined on high-end GPUs equipped with two DMA engines. For data-intensive streaming applications, the time spent in copying the data from host memory to GPU memory, and back, can outweigh the benefits of GPU parallelization. However, the experiments in Chapter 5 indicate that a significant improvement of the overall GPU execution time is possible using the stream buffer design. The mechanisms described in this chapter provide a model-based method to achieve data-driven asynchronous execution with overlapping of communication and computations.

7.2 Prerequisites for Further Progress

The research presented in this thesis creates a bridge between distributed memory parallelization with PPNs and shared memory parallelization techniques for exploiting data parallelism. As such, it opens numerous possibilities for further research, but it also poses a set of novel requirements that an experimentation framework would need to satisfy. Combining transformations implemented in different, even closely related polyhedral frameworks, such as [1,2,81] to name a few, is nowadays a technically challenging and time consuming task standing in the way of research progress. First and foremost, further research on hybrid, multi-level parallelization for heterogeneous platforms requires a standardized polyhedral intermediate representation for data exchange between polyhedral compiler frameworks. The polyhedral compiler research community would also greatly benefit from a unified, modular and extensible open source framework based on standardized interfaces, which could be used as a basis for further research on transformations, experimentation and reporting scientific progress.

7.3 Directions for Future Work

The research work presented in previous chapters creates a basis for multi-level hierarchical representation in the polyhedral model and opens the doors for further research and experimentation in numerous directions. Some of the ideas for further work are the following:

Parallelism affinity An attractive topic for further research is guided derivation of parallel, multi-level programs. For this purpose, we propose to use the notion of *parallelism affinity* to describe different platform components. With the concept of parallelism affinity, we can specify what is the preferred type of the parallelism for the given architectural component, e.g., task parallelism for a multicore CPU and

data parallelism for a GPU. Moreover, it would be easy to extend the parallelism affinity concept to include information on the number, types and characteristics of parallelism levels for each component. Similarly, we can use the notion of *communication affinity* to describe the preferred type and characteristics of the communication on the platform, e.g. whether it exposes shared memory or distributed memory architecture at some level. The combination of multi-level parallelism affinity and communication affinity specifications of heterogeneous platforms can be used to guide automatic derivation, mapping, and performance tuning of architecture-specific program modules from our hierarchical intermediate representation leading to efficient, custom-tailored generation of parallel code.

Balancing pipeline and data parallelism Data parallelism and pipeline parallelism are opposing concepts in the sense that opting for maximal data parallelism in a streaming application leaves no pipeline parallelism and vice versa. We can generate applications that exploit both data and pipeline parallelism by slicing the program model into multiple levels, and then transforming the components in one level for data or for pipeline parallelism. The size of the tokens passed between the levels determines the granularity of tasks at each level, and implicitly the balance between top-level pipeline parallelism and data parallelism. It is possible to perform empirical search on program variants to select a good combination of parameters for the program execution. Another interesting topic would be the design of an analytical model for determining the optimal balance of data and pipeline parallelism for the given platform and application combination.

Data space transformations Another promising topic for further research are data space transformations in the polyhedral model for enabling the construction and exchange of arbitrary composite tokens between program modules. The extension and further refinement of support for composite tokens could possibly increase the general applicability of PPNs for processing not only streaming applications, but also for acceleration of iterative numerical applications that process large data sets (*Big Data*), e.g., by partitioning the data set processed at each simulation step into tokens and streaming these tokens to and from the GPU.

Overall, our approach is currently targeted towards parallelization of computationally intensive streaming applications and their mapping onto heterogeneous platforms. However, the proposed future work directions will improve both the application coverage and the performance of automatic parallelization and mapping, making our model-based approach a promising alternative to manual parallelization.

7.3. DIRECTIONS FOR FUTURE WORK

Appendix A

Application Source Codes

A.1 Predictor

```
1 int produce();
2 int predict(int um, int ml);
3 void consume(int a);
4
5 #pragma compaan_procedure predictor
6 void predictor(void)
7 {
8     int a[5][5];
9     int i,j;
10
11     for (i = 0; i < 4; i = i + 1)
12         for (j = 0; j < 4; j = j + 1)
13             a[i][j] = produce();           //Statement P
14
15     for (i = 1; i <= 4; i = i + 1)
16         for (j = 1; j <= 4; j = j + 1)
17             a[i][j] = predict(a[i-1][j], a[i][j-1]); //Statement T
18
19     for (i = 1; i <= 4; i = i + 1)
20         for (j = 1; j <= 4; j = j + 1)
21             consume(a[i][j]);             //Statement C
22 }
```

Listing A.1: Predictor SANLP

A.2 Grid

A.3. SOBEL

```
1 int produce();
2 int grid(int a);
3 void consume(int a);
4
5 #pragma compaan_procedure grid
6 void grid(void)
7 {
8     int a[10];
9     int i,j,x,y;
10
11     for (x = 0; x < 10; x = x + 1)
12         a[x] = produce();           //Statement P
13
14     for (i = 1; i <= 4; i = i + 1)
15         for (j = 1; j <= 4; j = j + 1)
16             a[i+j] = grid(a[i+j]);  //Statement T
17
18     for (y = 2; y < 8; y = y + 1)
19         consume(a[y]);              //Statement C
20 }
```

Listing A.2: Grid SANLP

A.3 Sobel

```
1
2 #define MAX_WIDTH 250
3 #define MAX_HEIGHT 250
4
5 #define abs(x) ( (x < 0) ? -(x) : (x) )
6
7 void readPixel(int * p);
8 void writePixel(int value);
9
10 void gradient(const int a1, const int a2, const int a3, const int a4,
11     const int a5, const int a6, int* out);
12
13 void absVal(const int x, const int y, int* out);
14
15 #pragma compaan_procedure sobel
16 void sobel(
17     #pragma compaan_parameter 10 25
18     int M,
19     #pragma compaan_parameter 10 25
20     int N, int image_in[MAX_WIDTH][MAX_HEIGHT],
21     int image_out[MAX_WIDTH][MAX_HEIGHT])
```

```
22 {
23
24   int i, j;
25   int image[M][N];
26   int Jx[M][N];
27   int Jy[M][N];
28   int av[M][N];
29
30   for (j = 0; j < M; j++)
31     for (i = 0; i < N; i++)
32       image[j][i] = image_in[j][i];
33
34   for (j = 1; j < M - 1; j++) {
35     for (i = 1; i < N - 1; i++) {
36       gradient(image[j - 1][i - 1], image[j][i - 1], image[j + 1][i - 1],
37               image[j - 1][i + 1], image[j][i + 1], image[j + 1][i + 1],
38               &(Jx[j][i]));
39       gradient(image[j - 1][i - 1], image[j - 1][i], image[j - 1][i + 1],
40               image[j + 1][i - 1], image[j + 1][i], image[j + 1][i + 1],
41               &(Jy[j][i]));
42       absVal(Jx[j][i], Jy[j][i], &(av[j][i]));
43     }
44   }
45
46   for (j = 1; j < M - 1; j++)
47     for (i = 1; i < N - 1; i++)
48       image_out[j][i] = av[j][i];
49
50 }
```

Listing A.3: Sobel SANLP

A.3. *SOBEL*

Appendix B

KPN2GPU

B.1 Predictor

B.1.1 PPN Model

Default PPN (Compaan)

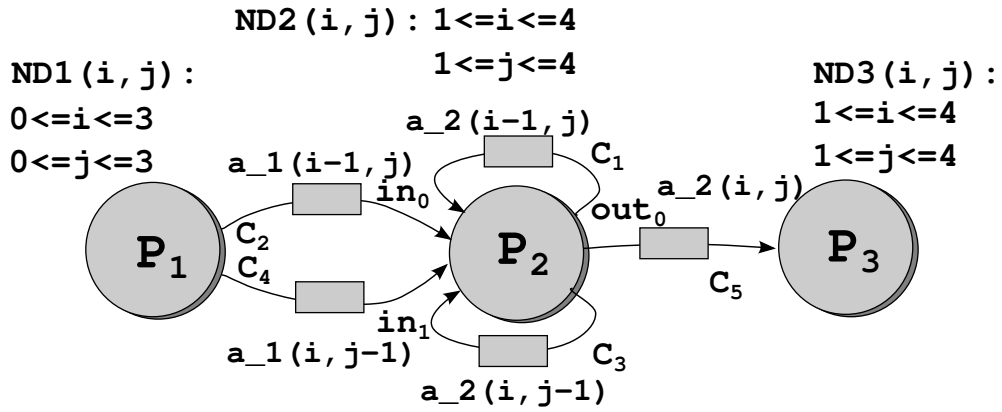
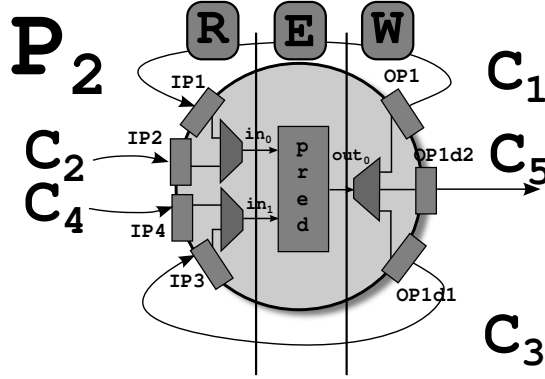


Figure B.1: The PPN obtained from the source code in Listing A.1.

Specification of Process P_2

B.1.2 Node P_2 : Space-Time Mapping

- schedule: $t(P_2, (i, j)) = i + j - 2$



$$\begin{aligned}
 D_{P_2} &= \{(i, j) \mid 1 \leq i \leq 4 \wedge 1 \leq j \leq 4\} \\
 \text{IPD1} &= D_T \cap \{(i, j) \mid i \geq 2\} \\
 \text{IPD2} &= D_T \cap \{(i, j) \mid i = 1\} \\
 \text{IPD3} &= D_T \cap \{(i, j) \mid j \geq 2\} \\
 \text{IPD4} &= D_T \cap \{(i, j) \mid j = 1\} \\
 \text{OPD1} &= D_T \cap \{(i, j) \mid i \leq 3\} \\
 \text{OPD1d1} &= D_T \cap \{(i, j) \mid j \leq 3\} \\
 \text{OPD1d2} &= D_T \\
 M_{C1} = M_{C2} &= \begin{bmatrix} 1 & 0 & -1 \\ 0 & 1 & 0 \end{bmatrix} \\
 M_{C3} = M_{C4} &= \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & -1 \end{bmatrix}
 \end{aligned}$$

Figure B.2: (a) Process P_2 , (b) Specification of P_2 's domain, port domains, and mapping matrices of incoming channels C_1 - C_4 .

- allocation: $p(P_2, (i, j)) = j$

Resulting space-time mapping (schedule + allocation): $T_{P_2} = \begin{bmatrix} 1 & 1 & -2 \\ 0 & 1 & 0 \end{bmatrix}$

Inverse space-time mapping: $T_{P_2}^{-1} = \begin{bmatrix} 1 & -1 & 2 \\ 0 & 1 & 0 \end{bmatrix}$

Coordinate system transformation: $\begin{bmatrix} i \\ j \\ 1 \end{bmatrix} = \begin{bmatrix} 1 & -1 & 2 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} t0 \\ p0 \\ 1 \end{bmatrix}$

Iterators expressed in space-time coordinates: $i = t0 - p0 + 2, j = p0$.

- Width $W = p0_{max} = 4$: Default num threads for the allocation
- Depth $D = t0_{max} = 7$: Default num time steps for the schedule

B.1.3 Node P_2 : DPV Components

Parallel Node Domain

$$D_{P_2}^{\parallel} = \{(p0, t0) \mid (t0 - p0 + 1 \geq 0) \wedge (-t0 + p0 + 2 \geq 0) \wedge (p0 - 1 \geq 0) \wedge (-p0 + 4 \geq 0)\}$$

$$\begin{aligned}
 \text{IPD1} &= D_T^{\parallel} \cap \{(p0, t0) \mid t0 - p0 \geq 0\} \\
 \text{IPD2} &= D_T^{\parallel} \cap \{(p0, t0) \mid t0 - p0 - 1 = 0\} \\
 \text{IPD3} &= D_T^{\parallel} \cap \{(p0, t0) \mid p0 - 2 \geq 0\} \\
 \text{Port Domains} \quad \text{IPD4} &= D_T^{\parallel} \cap \{(p0, t0) \mid p0 - 1 = 0\} \\
 \text{OPD1} &= D_T^{\parallel} \cap \{(p0, t0) \mid -t0 + p0 + 1 \geq 0\} \\
 \text{OPD1d1} &= D_T^{\parallel} \cap \{(p0, t0) \mid -p0 + 3 \geq 0\} \\
 \text{OPD1d2} &= D_T^{\parallel}
 \end{aligned}$$

Data Parallel Channels

Input channels for argument in_0

DPC1 Input channel for argument in_0 , when read from P'_2 . Derived from PPN self-link C_1 with mapping M_{C_1} .

- Recall : $M_{C_1} = \begin{bmatrix} 1 & 0 & -1 \\ 0 & 1 & 0 \end{bmatrix}$
- DPC1 Mapping: $M'_{C_1} = T_{P_2} M_{C_1} T_{P_2}^{-1} = \begin{bmatrix} 1 & 0 & -1 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$
- DPC1 Array: $a_2[7][4]$
- Read access: $in0 \leftarrow a_2(t0 - 1, p0)$

DPC2 Input channel for argument in_0 , when read from P'_1 . Derived from external PPN channel C_2 with mapping M_{C_2} .

- Recall : $M_{C_2} = \begin{bmatrix} 1 & 0 & -1 \\ 0 & 1 & 0 \end{bmatrix}$
- DPC2 Mapping: $M'_{C_2} = T_{P_1} M_{C_2} T_{P_1}^{-1}$, where $T_{P_1} = I$.
- DPC2 Mapping: $M'_{C_2} = \begin{bmatrix} 1 & -1 & 1 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$
- DPC2 Array: $a_1[5][5]$
- Read access: $in0 \leftarrow a_1(t0 - p0 + 1, p0)$

Input channels for argument in_1

DPC3 Input channel for argument in_1 , when read from P'_2 . Derived from PPN self-link C_3 with mapping M_{C_3} . Implemented as array a_2 .

- Recall : $M_{C_3} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & -1 \end{bmatrix}$
- DPC3 Mapping: $M'_{C_3} = T_{P_2} M_{C_3} T_{P_2}^{-1} = \begin{bmatrix} 1 & 0 & -1 \\ 0 & 1 & -1 \\ 0 & 0 & 1 \end{bmatrix}$
- DPC3 Array: $a_2[5][5]$
- Read access: $in1 \leftarrow a_2(t0 - 1, p0 - 1)$

DPC4 Input channel for argument in_1 , when read from P'_1 . Derived from external PPN channel C_4 with mapping M_{C_4} .

- Recall : $M_{C_4} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & -1 \end{bmatrix}$
- DPC2 Mapping: $M'_{C_4} = T_{P_1} M_{C_4} T_{P_1}^{-1}$, where $T_{P_1} = I$.
- DPC2 Mapping: $M'_{C_4} = \begin{bmatrix} 1 & -1 & 1 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$
- DPC4 Array: $a_1[5][5]$
- Read access: $in0 \leftarrow a_1(t0 - p0 + 1, p0)$

Output channels for argument out_0

DPC5 External output channel for argument out_0 produced by DPP P'_2 , derived from PPN channel C_5 .

- DPC5 Array: a_2
- Write access: $out0 \leftarrow a_2(t0, p0)$

DPC1, DPC3 Self-links feeding argument out_0 to the next iteration of DPP P'_2 . See the specification above.

B.1.4 Predictor: Host Code

```
1 void runTest( int argc, char** argv){
2     //Initialization
3     printf("Allocating_memory_on_CPU...\n");
4     int* a_1 = (int*) malloc(ga_1_MEM_SIZE);
5     int* a_2 = (int*) malloc(ga_2_MEM_SIZE);
6
7     printf("Allocating_memory_on_GPU...\n");
8     int* ga_1; cudaMalloc((void**) &ga_1, ga_1_MEM_SIZE);
9     int* ga_2; cudaMalloc((void**) &ga_2, ga_2_MEM_SIZE);
10
11     // Kernel Configuration
12     int ND_1_GRIDDim = 1; int ND_1_TBDim = ND_1_W;
13     int ND_2_GRIDDim = 1; int ND_2_TBDim = ND_2_W;
14     int ND_3_GRIDDim = 1; int ND_r_TBDim = ND_3_W;
15
16     // Kernel Calls
17     ND_1_Kernel<<< ND_1_GRIDDim, ND_1_TBDim >>>(ga_1);
18
19     ND_2_Kernel<<< ND_2_GRIDDim, ND_2_TBDim >>>(ga_1, ga_2);
20
21     ND_3_Kernel<<< ND_3_GRIDDim, ND_3_TBDim >>>(ga_2);
22
23     cudaMemcpy(a_1, ga_1, ga_1_MEM_SIZE, cudaMemcpyDeviceToHost);
24     cudaMemcpy(a_2, ga_2, ga_2_MEM_SIZE, cudaMemcpyDeviceToHost);
25
26     //Clean up
27     free(a_1);
28     free(a_2);
29
30     cudaFree(ga_1);
31     cudaFree(ga_2);
32
33     cudaThreadExit();
34 }
```

Listing B.1: Generated CUDA host code.

B.1.5 Node P'_2 : CUDA Kernel (Default)

```
1 __host__ __device__ int predict(int um, int ml);
2 #define ND_2_W (4)
3 #define ND_2_D (7)
4
5 #define ACTIVE_ND_2 ((t0-p0+1 >= 0) && (-t0+p0+2 >= 0) && (p0-1 >= 0) && (-p0
    +4 >= 0))
```

B.1. PREDICTOR

```
6
7 #define ND_2IP_1 ((ACTIVE_ND_2) && (((t0-p0 >= 0))))
8 #define ND_2IP_2 ((ACTIVE_ND_2) && (((t0-p0+1 == 0))))
9 #define ND_2IP_3 ((ACTIVE_ND_2) && (((p0-2 >= 0))))
10 #define ND_2IP_4 ((ACTIVE_ND_2) && (((p0-1 == 0))))
11
12 #define ND_2OP_1 ((ACTIVE_ND_2) && (((-t0+p0+1 >= 0))))
13 #define ND_2OP_1_d1 ((ACTIVE_ND_2) && (((-p0+3 >= 0))))
14 #define ND_2OP_1_d2 (ACTIVE_ND_2)
15
16 #define a_2_stride ND_2_W
17 #define a_1_stride ND_1_W
18 #define DPC1(t,p) ga_2[ a_2_stride * (t-1) + (p) ]
19 #define DPC2(t,p) ga_1[ a_1_stride * (t-p+1) + (p) ]
20 #define DPC3(t,p) ga_2[ a_2_stride * (t-1) + (p-1) ]
21 #define DPC4(t,p) ga_1[ a_1_stride * (t-p+2) + (p-1) ]
22 #define DPC5(t,p) ga_2[ a_2_stride * (t) + (p) ]
23
24 __global__ void ND_2_Kernel( int *ga_1, //input channel
25                             int *ga_2 //input-output channel
26                             )
27 {
28     int in_0;
29     int in_1;
30     int out_0;
31
32     // Mapping: CUDA Threads to Processing Entities
33     // threadIdx.x - unique thread identifier
34     int p0 = ((threadIdx.x) + ((1)));
35
36     // Number of synchronous time steps in PND
37     for(int t0 = 0; t0 < ND_2_D; t0++)
38     {
39         ///////////////////////////////////
40         // Process Iteration t0
41         ///////////////////////////////////
42
43         ///////////////////////////////////
44         // Phase I: READ
45         ///////////////////////////////////
46         if (ND_2IP_1) {
47             in_0 = DPC1(t0, p0);
48         }
49         if (ND_2IP_2) {
50             in_0 = DPC2(t0, p0);
51         }
52         if (ND_2IP_3) {
53             in_1 = DPC3(t0, p0);
54         }
55     }
```

```

55     if (ND_2IP_4) {
56         in_1 = DPC4(t0, p0);
57     }
58
59     //////////////////////////////////////
60     // Phase II: EXECUTE
61     //////////////////////////////////////
62     if (ACTIVE_ND_2) {
63         out_0 = predictor(in_0, in_1);
64     }
65
66     //////////////////////////////////////
67     // Phase III: WRITE
68     // (for each output arg - 1 write per memory array!)
69     //////////////////////////////////////
70     if ((ND_2OP_1) || (ND_2OP_1_d1) || (ND_2OP_1_d2)) {
71         DPC5(t0, p0) = out_0;
72     }
73     __syncthreads();
74
75 } //end for
76
77 } //end ND_2_Kernel

```

Listing B.2: CUDA kernel: default.

B.1.6 Node P'_2 : CUDA Kernel (Optimized)

```

1 //Channel width
2 #define sa_2_stride (4)
3
4 //(a) Default buffer size
5 #define DPC13_SIZE (7*4)
6 #define DPC1(t,p) sa_2[ sa_2_stride * (t-1) + (p) ]
7 #define DPC3(t,p) sa_2[ sa_2_stride * (t-1) + (p-1) ]
8 #define DPC13(t,p) sa_2[ sa_2_stride * (t) + (p) ]
9
10 //(b) Optimized buffer size
11 #define DPC13_SIZE (1*4)
12 #define DPC1(t,p) sa_2[ p ]
13 #define DPC3(t,p) sa_2[ p-1 ]
14 #define DPC13(t,p) sa_2[ p ]
15
16 __global__ void ND_2_Kernel( int *ga_1, //input channel
17                             int *ga_2 //input-output channel
18                             )
19 {
20     __shared__ int sa_2[DPC13_SIZE];    //self-links

```

B.1. PREDICTOR

```
21
22     int in_0;
23     int in_1;
24     int out_0;
25
26     // Mapping of virtual processors to CUDA threads
27     int p0 = ((threadIdx.x) + ((1)));
28
29     for(int t0 = 0; t0 < ND_2_D; t0++)
30     {
31         //////////////////////////////////////
32         // Process Iteration t0
33         //////////////////////////////////////
34
35         //////////////////////////////////////
36         // Phase I: READ
37         //////////////////////////////////////
38         if (ND_2IP_1) {
39             in_0 = DPC1(t0, p0);
40         }
41         if (ND_2IP_2) {
42             in_0 = DPC2(t0, p0);
43         }
44         if (ND_2IP_3) {
45             in_1 = DPC3(t0, p0);
46         }
47         if (ND_2IP_4) {
48             in_1 = DPC4(t0, p0);
49         }
50         __syncthreads();
51
52         //////////////////////////////////////
53         // Phase II: EXECUTE
54         //////////////////////////////////////
55         if (ACTIVE_ND_2) {
56             out_0 = predictor(in_0, in_1);
57         }
58
59         //////////////////////////////////////
60         // Phase III: WRITE
61         // (for each output arg - 1 write per memory array!)
62         //////////////////////////////////////
63         // Only writes to self-links must complete before
64         // the first threads start read phase of the next process iteration
65         if ((ND_2OP_1) || (ND_2OP_1_d1)) {
66             DPC13(t0, p0) = out_0;
67         }
68         __syncthreads();
69     }
```

```
70     if (ND_2OP_1_d2) {
71         DPC5(t0, p0) = out_0;
72     }
73
74 } //end for
75
76 } //end ND_2_Kernel
```

Listing B.3: CUDA kernel: optimized.

B.1.7 Scaling

Adjusted Host Code

Given the data parallel tile domain in space-time coordinates $(p1, t1)$ with maximal parallel width W_2 and the maximal depth D_2 , we generate the scalable CUDA host code as follows:

```
1  // tile time steps
2  for(int t1 = 0; t1 < D2; t1++)
3  {
4      // Kernel Configuration
5
6      // Number of thread blocks in the kernel launch
7      int ND_2_GRIDDim = W2;
8
9      // Number of threads in each thread block corresponds to tile width
10     int ND_2_TBDim = TY;
11
12     //Kernel launch
13     ND_2_Kernel<<< ND_2_GRIDDim, ND_2_TBDim >>>(t1, ga_1, ga_2);
14
15 }
```

Listing B.4: Scaled-up CUDA Host Code.

Adjustments to the Kernel Code

```
1
2 // max width of this DPP (P_2)
3 #define ND_2_W 8
4 // max tile width of this DPP (P_2)
5 #define ND_2_W2 2
6
7 // width of its producer DPP (P_1)
8 #define ND_1_W 9
```

B.1. PREDICTOR

```
9
10 // Tile width across p-dimension
11 #define TX 4
12 #define TY 4
13 #define ND_2_WTile (TY)
14 #define DPC13_SIZE (1 * ND_2_WTile)
15 //tile offset in x-axis: TX*t1
16 #define tOffsetX (TX * (t1-blockIdx.x))
17 //tile offset in y-axis: TY*p1
18 #define tOffsetY (TY * blockIdx.x)
19
20 // Channel accesses
21 #define ga_2_stride ND_2_W
22 #define sa_2_stride ND_2_WTile
23 #define ga_1_stride ND_1_W
24
25 #define DPC1(t,p) sa_2[ sa_2_stride * ((t-1) - tOffsetX) + ((p) - tOffsetY) ]
26 #define DPC2(t,p) ga_1[ ga_1_stride * (t-p+1) + (p) ]
27 #define DPC3(t,p) sa_2[ sa_2_stride * ((t-1) - tOffsetX) + ((p-1) - tOffsetY)
    ]
28 #define DPC4(t,p) ga_1[ ga_1_stride * (t-p+2) + (p-1) ]
29 #define DPC5(t,p) ga_2[ ga_2_stride * (t) + (p) ]
30
31 __global__ void ND_2_Kernel( int t1, //current tile time step
32                             int *ga_1, //input channel
33                             int *ga_2 //output channel
34                             )
35 {
36     __shared__ int sa_2[DPC13_SIZE];    //self-links
37
38     int in_0;
39     int in_1;
40     int out_0;
41
42     // Mapping: 2-Level CUDA Thread Hierarchy to Processing Entities
43     // blockIdx.x - unique thread block identifier
44     // threadIdx.x - unique thread identified within a thread block
45     // blockDim.x - number of threads in a thread block (block width)
46     int p0 = TY * blockIdx.x + threadIdx.x + 1;
47
48     // Process Control: Execute a number of synchronous time steps in PND tile
49     // Lower and upped bounds ~ the first and the last iteration of the tile
50     for(int t0 = TX * (t1 - blockIdx.x); t0 < TX * (t1 - blockIdx.x) + TX - 1;
51         t0++)
52     {
53         //////////////////////////////////////
54         // Process Iteration t0
55         //////////////////////////////////////
```

```

56  //////////////////////////////////////
57  // Phase I: READ
58  //////////////////////////////////////
59  if (ND_2IP_1) {
60      in_0 = DPC1(t0, p0);
61  }
62  if (ND_2IP_2) {
63      in_0 = DPC2(t0, p0);
64  }
65  if (ND_2IP_3) {
66      in_1 = DPC3(t0, p0);
67  }
68  if (ND_2IP_4) {
69      in_1 = DPC4(t0, p0);
70  }
71  __syncthreads();
72
73  //////////////////////////////////////
74  // Phase II: EXECUTE
75  //////////////////////////////////////
76  if (ACTIVE_ND_2) {
77      out_0 = predictor(in_0, in_1);
78  }
79
80  //////////////////////////////////////
81  // Phase III: WRITE
82  //////////////////////////////////////
83  if (ND_2OP_1 || (ND_2OP_1_d1)) {
84      DPC13(t0, p0) = out_0;
85  }
86  __syncthreads();
87  if (ND_2OP_1_d2) {
88      DPC5(t0, p0) = out_0;
89  }
90
91  } //end for
92
93
94 } //end ND_2_Kernel

```

Listing B.5: Scaled-up CUDA Kernel Code. Additionally parametrized in grid time step and thread block index.

B.1. PREDICTOR

Appendix C

M-JPEG Encoder

C.1 Overview

Various standards have been developed for compression of digital video signals. The video compression standards can be broadly classified into still image-based compression approaches, and motion estimation-based approaches. Motion JPEG (M-JPEG) belongs to the class of still image-based compression approaches. M-JPEG standard specifies a video codec in which each frame of the video stream is encoded independently as a still image using JPEG standard for image compression. JPEG is a well-known image compression standard, which is named after Joint Photographic Experts Group - the committee that developed and released the standard in 1991.

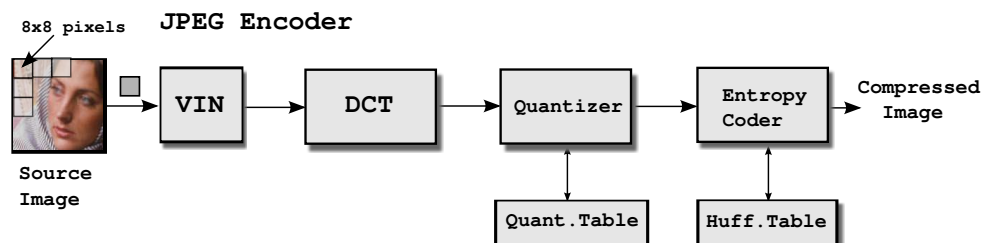


Figure C.1: Block Diagram of JPEG Encoder. In M-JPEG, the still image compression with JPEG encoder is applied to each frame in video sequence individually.

Figure C.1 shows JPEG encoder block diagram. The JPEG encoder partitions the input image into 8×8 blocks of pixels (blocks). Each block is processed indepen-

C.2. CODE LISTINGS

dently. Each block of the image passes through the DCT module. The DCT module performs a highly efficient 2-dimensional DCT transform to de-correlate the image signal and extract its frequency coefficients. The DCT coefficients are passed to the quantizer, which normalizes the DCT coefficients by a 8×8 quantization matrix and then rounds them off to the nearest integer. The compression ratio, and thus the quality of the encoded image, are determined by the quantization step. The output of the quantization stage is passed to the entropy coder which performs several encoding steps, including run-length encoding and variable-length encoding using the Huffman compression algorithm, on the quantized coefficients. The output of the entropy coder is packed into compressed bitstream to generate the JPEG-encoded image, and stored into a file. The M-JPEG encoder simply applies the JPEG encoding on each frame in the sequence.

As a still image-based compression approach, the M-JPEG does not perform video sequencing (motion) compression, which would allow the encoder to only encode the changes in the video sequence between the frames. However, it has the advantage that the resulting quality of video compression is independent from the motion in the image. As each individual frame is a complete JPEG compressed image, all frames will have the same guaranteed quality, which is not the case with MPEG-based standards. In addition, M-JPEG standard has the smallest latency in image processing, and it is quite easy to understand.

C.2 Code Listings

```
1  TBlock block[VNumBlocks][HNumBlocks];
2
3  for (int t = 0; t < NumFrames; t++) {
4
5      for (int i = 0; i < VNumBlocks; i++)
6          for (int j = 0; j < HNumBlocks; j++)
7              S: mainVIN(&block[i][j]);
8
9      for (int i = 0; i < VNumBlocks; i++)
10         for (int j = 0; j < HNumBlocks; j++)
11             T: mainDCT(block[i][j], &block[i][j]);
12
13     for (int i = 0; i < VNumBlocks; i++)
14         for (int j = 0; j < HNumBlocks; j++)
15             Q: mainQ(block[i][j], &block[i][j]);
16
17     for (int i = 0; i < VNumBlocks; i++)
18         for (int j = 0; j < HNumBlocks; j++)
19             V: mainVLE(block[i][j], &block[i][j]);
20 }
```

Listing C.1: Pseudocode of a M-JPEG Encoder (*main* code snippet)

```

1      int blockIn[8][8];
2      int coeff[8];
3      int tmp[8][8];
4      int blockOut[8][8];
5
6      // Step 1: Pre-shift the pixel values
7      for (i = 0; i < 8; i++)
8          for (j = 0; j < 8; j++)
9              S0: shift(&blockIn[i][j]);
10
11     // Step 2: Perform the first pass of 2D separable integer DCT
12     // Inputs: a whole 8x8 block of pixel values and the coefficients array
13     // Output: a whole 8x8 block of pixel values
14     for (i = 0; i < 8; i++)
15         for (j = 0; j < 8; j++)
16             S1: tmp[i][j] = dotProduct1(blockIn, coeff, i, j);
17
18     // Step 3: Perform the second pass of 2D separable integer DCT
19     // Inputs: a whole 8x8 block of pixel values and the coefficients array
20     // Output: a whole 8x8 block of pixel values
21     for (i = 0; i < 8; i++)
22         for (j = 0; j < 8; j++)
23             S2: blockOut[i][j] = dotProduct2(tmp, coeff, i, j);
24
25     // Step 4: Bound the pixel values to the threshold
26     for (i = 0; i < 8; i++)
27         for (j = 0; j < 8; j++)
28             S3: bound(&blockOut[i][j]);

```

Listing C.2: Code snippet (pseudocode) from the definition of *mainDCT*. The pseudocode illustrates the processing of 8×8 blocks for a single color component.

```

1 void dotProduct1(int blockIn[][8], int coeff[][8], int tmp[][8], int i, int j)
2 {
3     int sum = 0;
4     for (int k = 0; k < 8; k++)
5     {
6         sum += blockIn[i][k] * (coeff[j][k]>>16);
7         tmp[i][j] = sum >> 8;
8     }
9 }
10
11 void dotProduct2(int tmp[][8], int coeff[][8], int blockOut[][8], int i, int j)
    )

```

C.3. M-JPEG PPN

```

12 {
13   int sum = 0;
14   for (k = 0; k < 8; k++)
15   {
16     sum += (coeff[i][k]>>16) * tmp[k][j];
17     blockOut[i][j] = sum >> 8;
18   }
19 }

```

Listing C.3: Code snippet (pseudocode) from the definition of DCT passes.

C.3 M-JPEG PPN

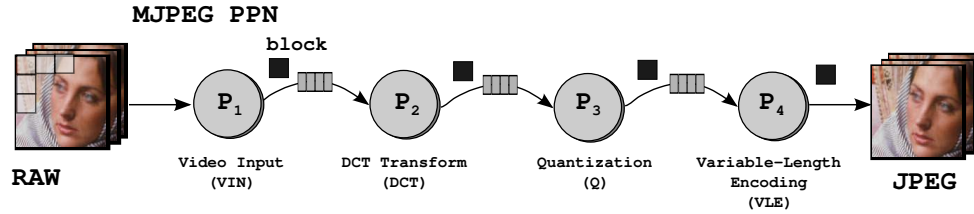


Figure C.2: Automatically generated M-JPEG PPN is a pipeline composed of four major tasks. The input to the pipeline is a stream of frames in raw format, the M-JPEG PPN performs JPEG image compression on each frame individually.

The M-JPEG SANLP in Listing C.1 is transformed by the COMPAAAN compiler into four tasks represented by PPN processes, as illustrated in Figure C.2. Each PPN process executes one statement of the SANLP and its enclosing loop nest. For example, the DCT node (P_2) executes the statement $T : \text{mainDCT}(\text{block}[i][j], \&\text{block}[i][j])$ on the iteration domain resulting from for loops f , i , and j that enclose the statement T . The four processes form a straight-forward processing pipeline. The processes are connected via channels implemented as FIFO buffers and exchange data via tokens. The token data type in the PPN generated by the COMPAAAN compiler always equals the data type of the variables in the source code, which is here a *block* of the picture that contains 8×8 pixels. Following the default PPN implementation and mapping approach in Compaan, PPN processes are assigned as tasks for execution to different threads. Each PPN process task *sequentially* executes iterations of the for loops encapsulating its program statement (e.g. the DCT node executes its copy of the code on lines 3, 9, 10, 11 in Listing C.1). This parallelization approach is used for task and pipeline parallel execution on multicore platforms.

Bibliography

- [1] The Polyhedral Compiler Collection, 2011.
- [2] Daedalus System-level Design Framework, 2012.
- [3] ACE Associated Compiler Experts bv. Parallelization using polyhedral analysis. Technical report, 2008.
- [4] A. Aho, M. Lam, R. Sethi, and J. Ullman. *Compilers: Principles, Techniques, and Tools*, volume 1009. Pearson/Addison Wesley, 2007.
- [5] J. Allen, D. Callahan, and K. Kennedy. Automatic Decomposition of Programs for Parallel Execution. In *14th Annual Symposium on Principles of Programming Languages*, pages 63–76, 1987.
- [6] J. Allen and K. Kennedy. PFC: A Program to convert Fortran to parallel form. In *IBM Conference on Parallel Computing and Scientific Computations*, 1982.
- [7] S. Amarasinghe and M. Lam. Communication optimization and code generation for distributed memory machines. In *ACM SIGPLAN Notices*, volume 28, pages 126–138. ACM, 1993.
- [8] J. Anderson, S. Amarasinghe, and M. Lam. Data and Computation Transformations for Multiprocessors. *ACM SIGPLAN Notices*, 30(8):166–178, 1995.
- [9] C. Augonnet, J. Clet-Ortega, S. Thibault, and R. Namyst. Data-aware task scheduling on multi-accelerator based platforms. In *Parallel and Distributed Systems (ICPADS), 2010 IEEE 16th International Conference on*, pages 291–298. IEEE, 2010.
- [10] C. Augonnet, S. Thibault, R. Namyst, and P. Wacrenier. StarPU: a unified platform for task scheduling on heterogeneous multicore architectures. *Euro-Par 2009 Parallel Processing*, pages 863–874, 2009.

Bibliography

- [11] E. Ayguadé et al. Hybrid/heterogeneous programming with OmpSs and its software/hardware implications. In *Programming Multi-Core and Many-Core Computing Systems*. Wiley-Blackwell, 2012.
- [12] H. Bae, L. Bachega, C. Dave, S. I. Lee, S. Lee, S. J. Min, R. Eigenmann, and S. Midkiff. Cetus: A Source-to-Source compiler infrastructure for multicores.
- [13] A. Balevic. Parallel Variable-Length Encoding on GPGPUs. In *Proceedings of the 3rd Workshop on Highly Parallel Processing on a Chip (HPPC), Euro-Par 2009*.
- [14] A. Balevic, N. Fuerst, M. Heide, S. Papandreou, and A. Weiss. CUJ2K: a JPEG2000 encoder in CUDA. Technical report, Institute for Parallel and Distributed Systems, University of Stuttgart, 2009.
- [15] A. Balevic and B. Kienhuis. A Data Parallel View on Polyhedral Process Networks. *SCOPES'11*.
- [16] A. Balevic and B. Kienhuis. Deriving a Multi-Level Program Model for Efficient Parallelization on Heterogeneous Platforms. *Proceedings of International Conference on Parallel and Distributed Computing and Networks (PDCN 2013)*.
- [17] A. Balevic and B. Kienhuis. An Efficient Stream Buffer Mechanism for Dataflow Execution on Heterogeneous Platforms with GPUs. *Data-Flow Execution Models for Extreme Scale Computing (DFM 2011) in conjunction with PACT 2011*.
- [18] A. Balevic and B. Kienhuis. Exploiting task parallel execution on CUDA: A case study. *Proceedings of the 2nd Workshop on Applications for Multi and Many Core Processors*, June 2011.
- [19] A. Balevic and B. Kienhuis. KPN2GPU: An approach for discovery and exploitation of fine-grain data parallelism in process networks. *SIGARCH Computer Architecture News* 39(4), pages 66–71, 2011.
- [20] A. Balevic and B. Kienhuis. Scaling data-intensive applications on heterogeneous platforms with accelerators. *Proceedings of 2nd International Workshop on Accelerators and Hybrid Exascale Systems, IPDPS 2012*, May 2012.
- [21] A. Balevic, B. Kienhuis, and E. Deprettere. Multi-Level Parallelization in Polyhedral Model for Heterogeneous Platforms with Accelerators. *Poster Presentation at GPU Technology Conference 2013 (GTC2013)*.

-
- [22] B. Barney et al. Introduction to parallel computing. *Lawrence Livermore National Laboratory*, 6(13):10, 2010.
 - [23] M. Baskaran, A. Hartono, S. Tavarageri, T. Henretty, J. Ramanujam, and P. Sadayappan. Parameterized tiling revisited. In *Proceedings of the 8th annual IEEE/ACM international symposium on Code generation and optimization*, pages 200–209. ACM, 2010.
 - [24] M. Baskaran, J. Ramanujam, and P. Sadayappan. Automatic C-to-CUDA code generation for affine programs. In *Proc. of Compiler Construction (CC 2010)*. Springer, 2010.
 - [25] M. M. Baskaran. *Compile-time and Run-time Optimizations for Enhancing Locality and Parallelism on Multi-core and Many-core Systems*. PhD thesis, The Ohio State University, 2009.
 - [26] M. M. Baskaran, U. Bondhugula, S. Krishnamoorthy, J. Ramanujam, A. Rountev, and P. Sadayappan. A compiler framework for optimization of affine loop nests for GPGPUs. In *Proceedings of the 22nd annual International Conference on Supercomputing*, pages 225–234. ACM, 2008.
 - [27] C. Bastoul. Generating loops for scanning polyhedra. Technical Report 2002/23, PRiSM, Versailles University, 2002. Related to the CLooG tool.
 - [28] C. Bastoul. Code generation in the polyhedral model is easier than you think. In *PACT’13 IEEE International Conference on Parallel Architecture and Compilation Techniques*, pages 7–16, Juan-les-Pins, France, September 2004.
 - [29] C. Bastoul. *Improving Data Locality in Static Control Programs*. PhD thesis, University Paris 6, Pierre et Marie Curie, France, Dec. 2004.
 - [30] A. J. Bernstein. Analysis of Programs for Parallel Processing. *IEEE Transactions on Electronic Computers*, pages 15:757–763, October 1966.
 - [31] A. J. Bik. *The Software Vectorization Handbook*. Intel Press, 2004.
 - [32] U. K. R. Bondhugula. *Effective Automatic Parallelization and Locality Optimization Using The Polyhedral Model*. PhD thesis, The Ohio State University, 2008.
 - [33] U. Bondhugula et al. PLuTo: a practical and fully automatic polyhedral program optimization system. In *Proc. of PLDI’08, Tucson, AZ, 2008*.
 - [34] J. T. Buck, S. Ha, E. A. Lee, and D. G. Messerschmitt. Ptolemy: A framework for simulating and prototyping heterogeneous systems. 1994.

Bibliography

- [35] R. D. Chamberlain, M. A. Franklin, E. J. Tyson, J. H. Buckley, J. Buhler, G. Galloway, S. Gayen, M. Hall, E. Shands, and N. Singla. Auto-pipe: Streaming applications on architecturally diverse systems. *Computer*, 43(3):42–49, 2010.
- [36] C. Chen, J. Chame, and M. Hall. CHiLL: A framework for composing high-level loop transformations. *U. of Southern California, Tech. Rep*, 2008.
- [37] N. Coop. CUDA C Programming Guide V3.2. Technical report, Sept. 2010.
- [38] N. Coop. CUDA Technical Documentation: Programming and Best Practices Guide V3.2. Technical report, Sept. 2010.
- [39] R. Cytron, J. Ferrante, B. Rosen, M. Wegman, and F. Zadeck. An efficient method of computing static single assignment form. In *Proceedings of the 16th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 25–35. ACM, 1989.
- [40] L. Dagum and R. Menon. Openmp: an industry standard api for shared-memory programming. *Computational Science Engineering, IEEE*, 5(1):46–55, 1998.
- [41] J. Darlington, A. Field, P. Harrison, P. Kelly, D. Sharp, Q. Wu, and R. While. Parallel programming using skeleton functions. In *PARLE’93 Parallel Architectures and Languages Europe*, pages 146–160. Springer, 1993.
- [42] A. Darte, Y. Robert, and F. Vivien. *Scheduling and Automatic Parallelization*. Springer, 2000.
- [43] A. Darte, F. Vivien, et al. A comparison of nested loops parallelization algorithms. Technical report, 1995.
- [44] David Levinthal. Performance Analysis Guide for Intel Core i7 Processor. Technical report.
- [45] J. Davis, M. Goel, C. Hylands, B. Kienhuis, E. Lee, J. Liu, X. Liu, L. Muliadi, S. Neuendorffer, J. Reekie, et al. Ptolemy ii: Heterogeneous concurrent modeling and design in java. *University of California, Berkeley, Tech. Rep. UCB/ERL M*, 99, 1999.
- [46] E. F. Deprettere, E. Rijpkema, and B. Kienhuis. Translating imperative affine nested loop programs into process networks. In *Embedded processor design challenges*, pages 89–111. Springer, 2002.

-
- [47] R. Dolbeau, S. Bihan, and F. Bodin. HMPP: A hybrid multi-core parallel programming environment. In *First Workshop on General Purpose Processing on Graphics Processing Units*, 2007.
 - [48] A. Duran, E. Ayguade, R. M. Badia, J. Labarta, L. Martinell, X. Martorell, and J. Planas. OmpSs: a proposal for programming heterogeneous multi-core architectures. *Parallel Processing Letters*, 21(02):173–193, 2011.
 - [49] R. Eigenmann and J. Hoeflinger. Parallelizing and vectorizing compilers, 2000.
 - [50] J. Eker, J. W. Janneck, E. A. Lee, J. Liu, X. Liu, J. Ludvig, S. Neuendorffer, S. Sachs, and Y. Xiong. Taming heterogeneity-the ptolemy approach. *Proceedings of the IEEE*, 91(1):127–144, 2003.
 - [51] J. Enmyren and C. W. Kessler. Skepu: a multi-backend skeleton programming library for multi-gpu systems. In *Proceedings of the fourth international workshop on High-level parallel programming and applications*, pages 5–14. ACM, 2010.
 - [52] T. Farago, H. Nikolov, and E. Deprettere. A framework for heterogeneous desktop parallel computing. Technical report, LIACS - Leiden University, The Netherlands, December 2008.
 - [53] P. Feautrier. Polyhedral Model: Past, Present, Future.
 - [54] P. Feautrier. Parametric integer programming. *RAIRO Recherche operationnelle*, 22(3):243–268, 1988.
 - [55] P. Feautrier. Dataflow analysis of array and scalar references. *International Journal of Parallel Programming*, 20(1):23–53, 1991.
 - [56] P. Feautrier. Some efficient solutions to the affine scheduling problem. Part I. One-dimensional time. *IJPP’92*, 21(5):313–347, 1992.
 - [57] P. Feautrier. Some efficient solutions to the affine scheduling problem. Part II. Multi-dimensional time. *IJPP’92*, 21(5):313–347, 1992.
 - [58] P. Feautrier. Automatic parallelization in the polytope model. *The Data Parallel Programming Model*, pages 79–103, 1996.
 - [59] P. Feautrier. PIP/Piplib, a parametric integer linear programming solver, 2003.
 - [60] P. Feautrier. Scalable and structured scheduling. *IJPP’06*, 34(5):459–487, 2006.

- [61] R. Ferrer, J. Planas, P. Bellens, A. Duran, M. Gonzalez, X. Martorell, R. Badia, E. Ayguade, and J. Labarta. Optimizing the exploitation of multicore processors and gpus with openmp and opencl. *Languages and Compilers for Parallel Computing*, pages 215–229, 2011.
- [62] M. J. Flynn and K. W. Rudd. Parallel architectures. *ACM Computing Surveys (CSUR)*, 28(1):67–70, 1996.
- [63] I. Foster. *Designing and Building Parallel Programs*. Addison-Wesley, 1996.
- [64] A. Goderis, C. Brooks, I. Altintas, E. A. Lee, and C. Goble. Heterogeneous composition of models of computation. *Future Generation Computer Systems*, 25(5):552–560, 2009.
- [65] M. Gordon, W. Thies, and S. Amarasinghe. Exploiting coarse-grained task, data, and pipeline parallelism in stream programs. In *Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 151–162. ACM, 2006.
- [66] C. Gregg and K. Hazelwood. Where is the data? Why you cannot debate CPU vs. GPU performance without the answer. In *Performance Analysis of Systems and Software (ISPASS), 2011 IEEE International Symposium on*, pages 134–144. IEEE.
- [67] M. Griebl. Automatic parallelization of loop programs for distributed memory architectures. *University of Passau*, 2004.
- [68] M. Griebl and C. Lengauer. The loop parallelizer LooPo. In *Proc. Sixth Workshop on Compilers for Parallel Computers*, volume 21, pages 311–320, 1996.
- [69] S. v. Haastregt and B. Kienhuis. Enabling automatic pipeline utilization improvement in polyhedral process network implementations. In *Application-Specific Systems, Architectures and Processors (ASAP), 2012 IEEE 23rd International Conference on*, pages 173–176. IEEE, 2012.
- [70] M. Hall, J. Chame, C. Chen, J. Shin, G. Rudy, and M. Khan. Loop transformation recipes for code generation and auto-tuning. *Languages and Compilers for Parallel Computing*, pages 50–64, 2010.
- [71] M. Harvey and G. De Fabritiis. Swan: A tool for porting CUDA programs to OpenCL. *Computer Physics Communications*, 2011.
- [72] J. L. Hennessy and D. A. Patterson. *Computer architecture: a quantitative approach*. Morgan Kaufmann, 2002.

-
- [73] A. Ilić and L. Sousa. Chps: An environment for collaborative execution on heterogeneous desktop systems. *International Journal of Networking and Computing*, 1(1):pp–96, 2011.
 - [74] F. Irigoin and R. Triolet. Dependence approximation and global parallel code generation for nested loops. In *Proceedings of the International Workshop on Parallel and Distributed Algorithms*, October 1988.
 - [75] F. Irigoin and R. Triolet. Supernode partitioning. In *Proceedings of the 15th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 319–329. ACM, October 1988.
 - [76] G. Kahn and D. MacQueen. Coroutines and Networks of Parallel Processes. In *Proceedings of IFIP Congress 77*, pages 993–998, 1977.
 - [77] G. Kahn and D. MacQueen. Coroutines and Networks of Parallel Processes. In B. Gilchrist, editor, *Proceedings of IFIP Congress 77*, pages 993–998. North Holland Publishing Company, 1977.
 - [78] B. Kienhuis. Matparser: An array dataflow analysis compiler. *University of California at Berkeley, Tech. Rep*, 2000.
 - [79] B. Kienhuis and E. Deprettere. Modeling stream-based applications using the SBF model of computation. *The Journal of VLSI Signal Processing*, 34(3):291–300, 2003.
 - [80] B. Kienhuis, E. Deprettere, P. Van Der Wolf, and K. Vissers. A Methodology to Design Programmable Embedded Systems - The Y-chart Approach. In *Embedded Processor Design Challenges*, pages 321–324. Springer, 2002.
 - [81] B. Kienhuis, E. Rijpkema, and E. Deprettere. Compaan: Deriving process networks from matlab for embedded signal processing architectures. In *Proc. of CODES’00*, pages 13–17. ACM, 2000.
 - [82] S. Kung. Vlsi array processors. *Englewood Cliffs, NJ, Prentice Hall*, 1988, 685 p. *Research supported by the Semiconductor Research Corp., SDIO, NSF, and US Navy.*, 1, 1988.
 - [83] E. A. Lee and T. M. Parks. Dataflow Process Networks. *Proc. of the IEEE*, 83(5):773–801, 2002.
 - [84] S. Lee, S. J. Min, and R. Eigenmann. OpenMP to GPGPU: a compiler framework for automatic translation and optimization. In *Proceedings of the 14th ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 101–110. ACM, 2009.

Bibliography

- [85] S. I. Lee, T. A. Johnson, and R. Eigenmann. Cetus - an extensible compiler infrastructure for source to source transformation. In *Proc. of LCPC'04*.
- [86] C. Lengauer. Loop parallelization in the polytope model. *LECTURE NOTES IN COMPUTER SCIENCE*, pages 398–398, 1993.
- [87] M. Leyton and J. M. Piquer. Skandium: Multi-core programming with algorithmic skeletons. In *Parallel, Distributed and Network-Based Processing (PDP), 2010 18th Euromicro International Conference on*, pages 289–296. IEEE, 2010.
- [88] E. Lindholm, J. Nickolls, S. Oberman, and J. Montrym. NVIDIA Tesla: A unified graphics and computing architecture. *Micro, IEEE*, 28(2):39–55, 2008.
- [89] E. Lindholm, J. Nickolls, S. Oberman, and J. Montrym. NVIDIA Tesla: A unified graphics and computing architecture. *Micro, IEEE*, 28(2):39–55, 2008.
- [90] J. Liu, X. Liu, and E. Lee. Modeling distributed hybrid systems in ptolemy ii. In *American Control Conference, 2001. Proceedings of the 2001*, volume 6, pages 4984–4985 vol.6, 2001.
- [91] G. Martinez, M. Gardner, and W. Feng. CU2CL: A CUDA-to-OpenCL translator for multi-and many-core architectures. In *Parallel and Distributed Systems (ICPADS), 2011 IEEE 17th International Conference on*, pages 300–307. IEEE, 2011.
- [92] C. Meenderinck and B. Juurlink. A case for hardware task management support for the starss programming model. In *Proc. 13th Euromicro Conf. on Digital System Design: Architectures, Methods and Tools*, 2010.
- [93] S. Meijer. *Transformations for Polyhedral Process Networks*. PhD thesis, Leiden University, The Netherlands, 2010.
- [94] S. Meijer, B. Kienhuis, A. Turjan, and E. de Kock. A process splitting transformation for Kahn process networks. In *Design, Automation & Test in Europe Conference & Exhibition, 2007. DATE'07*, pages 1–6. IEEE, 2007.
- [95] S. Neuendorffer and E. Lee. Hierarchical reconfiguration of dataflow models. In *Formal Methods and Models for Co-Design, 2004. MEMOCODE'04. Proceedings. Second ACM and IEEE International Conference on*, pages 179–188. IEEE, 2004.
- [96] J. Nickolls, I. Buck, M. Garland, and K. Skadron. Scalable parallel programming with CUDA. *Queue*, 6(2):40–53, 2008.

- [97] H. Nikolov, T. Stefanov, and E. Deprettere. Systematic and automated multi-processor system design, programming, and implementation. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, 27(3):542–555, 2008.
- [98] H. Nikolov, M. Thompson, T. Stefanov, A. Pimentel, S. Polstra, R. Bose, C. Zissulescu, and E. Deprettere. Daedalus: toward composable multimedia mp-soc design. In *Proceedings of the 45th annual Design Automation Conference, DAC '08*, pages 574–579, New York, NY, USA, 2008. ACM.
- [99] H. N. Nikolov. *System-Level Design Methodology for Streaming Multi-Processor Embedded Systems*. PhD thesis, Leiden University, The Netherlands, 2009.
- [100] NVIDIA Corp. Maximizing GPU Efficiency in Extreme Throughput Applications. Technical report, Sept. 2009.
- [101] A. Obukhov and A. Kharlamov. Discrete Cosine Transform for 8x8 Blocks with CUDA, 2008.
- [102] OpenACC Architecture Review Board. OpenACC application program interface version 3.0, May 2008.
- [103] OpenACC Consortium. OpenACC application program interface version 1.0, 2012.
- [104] T. Parks, J. Pino, and E. Lee. A comparison of synchronous and cycle-static dataflow. In *Signals, Systems and Computers, 1995. 1995 Conference Record of the Twenty-Ninth Asilomar Conference on*, volume 1, pages 204–210. IEEE, 1996.
- [105] D. Patterson. The top 10 innovations in the new NVIDIA Fermi architecture. Technical report, 2010.
- [106] D. Patterson and J. Hennessy. *Computer organization and design: the hardware/software interface*. Morgan Kaufmann, 2009.
- [107] J. Planas, R. Badia, E. Ayguadé, and J. Labarta. Hierarchical task-based programming with starss. *International Journal of High Performance Computing Applications*, 23(3):284–299, 2009.
- [108] S. Pop, A. Cohen, C. Bastoul, S. Girbal, G. Silber, and N. Vasilache. Graphite: Loop optimizations based on the polyhedral model for gcc. 2006.

- [109] S. Pop, A. Cohen, C. Bastoul, S. Girbal, G. A. Silber, and N. Vasilache. Graphite: Loop optimizations based on the polyhedral model for gcc. In *Proc. of the 4th GCC Developer's Summit*, pages 179–198, June 2006.
- [110] L. Pouchet, C. Bastoul, A. Cohen, and J. Cavazos. Iterative optimization in the polyhedral model: Part ii, multidimensional time. In *ACM SIGPLAN Notices*, volume 43, pages 90–100. ACM, 2008.
- [111] L. Pouchet, U. Bondhugula, C. Bastoul, A. Cohen, J. Ramanujam, and P. Sadayappan. Combined iterative and model-driven optimization in an automatic parallelization framework. In *Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–11. IEEE Computer Society, 2010.
- [112] J. Ramanujam and P. Sadayappan. Tiling multidimensional iteration spaces for multicomputers. *Journal of Parallel and Distributed Computing*, 16(2):108–120, 1992.
- [113] L. Renganarayanan, D. Kim, S. Rajopadhye, and M. Strout. Parameterized tiled loops for free. In *Proceedings of the 2007 ACM SIGPLAN conference on Programming language design and implementation*, pages 405–414. ACM, 2007.
- [114] E. Rijkema. *Modeling Task Level Parallelism in Piece-wise Regular Programs*. PhD thesis, Leiden University, The Netherlands, 2002.
- [115] E. Rijkema, E. F. Deprettere, and B. Kienhuis. Deriving process networks from nested loop algorithms. *Parallel Processing Letters*, 10(02n03):165–176, 2000.
- [116] E. Rypkema, E. F. Deprettere, and B. Kienhuis. Compilation from matlab to process networks. In *In Second International Workshop on Compiler and Architecture Support for Embedded Systems*, 1999.
- [117] E. Schweitz, R. Lethin, A. Leung, and B. Meister. R-stream: A parametric high level compiler. In *High Performance Embedded Computing Workshop*, 2006.
- [118] S. Setia, M. Squillante, and S. Tripathi. Analysis of processor allocation in multiprogrammed, distributed-memory parallel processing systems. *Parallel and Distributed Systems, IEEE Transactions on*, 5(4):401–420, 1994.
- [119] T. Stefanov. *Converting weakly dynamic programs to equivalent process network specifications*. 2004.

-
- [120] T. Stefanov, B. Kienhuis, and E. Deprettere. Algorithmic transformation techniques for efficient exploration of alternative application instances. In *Proceedings of the 10th International Symposium on Hardware/Software Code-sign (CODES'02)*, pages 7–12. ACM, 2002.
 - [121] T. Stefanov et al. System design using Kahn process networks: the Compaan/Laura approach. In *Proc. of DATE'04*, volume 1, 2004.
 - [122] J. E. Stone, D. Gohara, and G. Shi. Opencl: A parallel programming standard for heterogeneous computing systems. *Computing in science & engineering*, 12(3):66, 2010.
 - [123] J. Teich and L. Thiele. Partitioning of processor arrays: A piecewise regular approach. *Integration, the VLSI journal*, 14(3):297–332, 1993.
 - [124] J. Teich and L. Thiele. Exact partitioning of affine dependence algorithms. In *Embedded Processor Design Challenges*, pages 135–153. Springer, 2002.
 - [125] J. Teich, L. Thiele, and L. Zhang. Scheduling of partitioned regular algorithms on processor arrays with constrained resources. In *Application Specific Systems, Architectures and Processors, 1996. ASAP 96. Proceedings of International Conference on*, pages 131–144. IEEE, 1996.
 - [126] The Portland Group. PGI Compiler User's Guide, 2012.
 - [127] W. Thies, M. Karczmarek, and S. Amarasinghe. StreamIt: A language for streaming applications. In *Compiler Construction*, pages 49–84. Springer, 2002.
 - [128] M. Thompson, H. Nikolov, T. Stefanov, A. D. Pimentel, C. Erbas, S. Polstra, and E. F. Deprettere. A framework for rapid system-level exploration, synthesis, and programming of multimedia mp-socs. In *Proceedings of the 5th IEEE/ACM international conference on Hardware/software codesign and system synthesis*, pages 9–14. ACM, 2007.
 - [129] K. Trifunovic, D. Nuzman, A. Cohen, A. Zaks, and I. Rosen. Polyhedral-model guided loop-nest auto-vectorization. In *Parallel Architectures and Compilation Techniques, 2009. PACT'09. 18th International Conference on*, pages 327–337. IEEE, 2009.
 - [130] A. Turjan. *Compiling Nested Loop Programs to Process Networks*. PhD thesis, University of Leiden, 2007.

- [131] A. Turjan, B. Kienhuis, and E. Deprettere. Translating affine nested-loop programs to process networks. In *Proc. of CASES'04*, pages 220–229. ACM, 2004.
- [132] A. Turjan, B. Kienhuis, and E. Deprettere. Classifying interprocess communication in process network representation of nested-loop programs. *ACM Transactions on Embedded Computing Systems (TECS)*, 6(2):13, 2007.
- [133] A. Varbanescu, H. Sips, K. Ross, Q. Liu, A. Natsev, J. Smith, and L. Liu. Evaluating application mapping scenarios on the Cell/B.E. *Concurrency and Computation: Practice and Experience*, 21(1):85–100, 2009.
- [134] S. Verdoolaege. Polyhedral Process Networks. In *Handbook of Signal Processing Systems*, pages 931–965. Springer, 2010.
- [135] S. Verdoolaege, H. Nikolov, and T. Stefanov. PN: a tool for improved derivation of process networks. *EURASIP Journal on Embedded Systems*, 2007(1):19–19, 2007.
- [136] R. Wilson, R. French, C. Wilson, S. Amarasinghe, J. Anderson, S. Tjiang, S. Liao, C. Tseng, M. Hall, M. Lam, et al. Suif: An infrastructure for research on parallelizing and optimizing compilers. *ACM Sigplan Notices*, 29(12):31–37, 1994.
- [137] M. Wolf and M. Lam. A loop transformation theory and an algorithm to maximize parallelism. *IEEE Transactions on Parallel and Distributed Systems*, 2(4):452–471, October.
- [138] M. Wolf and M. Lam. A data locality optimizing algorithm. *ACM Sigplan Notices*, 26(6):30–44, 1991.
- [139] M. Wolfe. Compilers and More: Parallel Programming Made Easy?
- [140] M. Wolfe. More iteration space tiling. In *Proceedings of the 1989 ACM/IEEE conference on Supercomputing*, pages 655–664. ACM, 1989.
- [141] M. Wolfe. *Optimizing Supercompilers for Supercomputers*. PhD thesis, University of Illinois at Urbana-Champaign, 1989.
- [142] M. Wolfe. *High performance compilers for parallel computing*, volume 179. Addison-Wesley, 1996.
- [143] R. Wu, B. Zhang, and M. Hsu. Clustering billions of data points using GPUs. In *Proceedings of the combined workshops on UnConventional high performance computing workshop plus memory access workshop*, pages 1–6. ACM, 2009.

- [144] Y. Yang, P. Xiang, J. Kong, and H. Zhou. A GPGPU compiler for memory optimization and parallelism management. *SIGPLAN Not.*, 45(6):86–97, 2010.
- [145] C. Zissulescu. *Synthesis of a Parallel Data Stream Processor from Dataflow Process Networks*. PhD thesis, Leiden University, The Netherlands, 2008.

Index

- $\mathbb{Z}$ -polyhedron, 19
- (W,D) parameters, 46
- encapsulation boundary, 81
- HiPRDG, 96
- Hierarchical Polyhedral Reduced Dependence Graph, 96
- PPN process, 27, 50
- PRDG, 26
- Polyhedral Reduced Dependence Graph, 26
- absolutely parallel, 46
- allocation, 44
- anti dependence, 23
- channel, 29
- common nesting level, 24
- component-level parallelism, 7, 80
- cooperative parallelism, 53
- CUDA core, 31
- data dependence, 22
- data parallelism, 2, 42
- data space
 - index vector, 88
- dataflow dependence, 23
- dependence
 - distance vector, 22
 - level, 24
 - loop-carried, 22
 - loop-independent, 22
- dependence graph, 25
- dependence level, 24, 93
- depth, 91
- depth (refined), 91
- derived statement, 92
- distributed memory, 12
- domain, 19
- GPU, 1
- graphics processing unit, 1
- independent parallelism, 54
- input port domain, 29
- integer polyhedron, 19
- iteration domain, 19
- iteration vector, 19
- level, 91
- level of dependence, 24
- lexicographical order, 21
- loop depth, 24
- loop nest depth, 24
- loop nesting level, 24
- loop subnest, 81
- mapping, 29
- mapping (DPC), 52
- multi-level parallelism, 2, 7

- node domain, 27, 28
- operation, 2, 21
- output dependence, 23
- output port domain, 29
- P/C pair, 26
- parallel node domain, 50
- parallel process iteration, 49
- parameters vector, 19
- partitioning, 12
- pipeline parallelism, 2
- platform-level parallelism, 7, 79
- polyhedron, 19
- polytope, 19
- process, 27
- process body, 27, 28
- process domain, 27
- process function, 28
- process iteration, 28
- process iteration domain, 28
- processor array, 31
- producer-consumer pair, 26
- read dependence, 23
- read-execute-write, 27
- SAC, 23
- scattering matrix, 22
- schedule, 22, 42
- self-link, 29
- sequential order, 22
- shared memory, 12
- SIMT, 31
- single-program multiple-data (SPMD),
33
- statement, 18
 - execution instance, 21
- statement level, 91
- Static Affine Nested Loop Program, SANLP,
17
- streaming multiprocessor (SMP), 31
- streaming processor (SP), 31
- subnest, 92
- target domain, 44
- task parallelism, 2
- textual order, 22
- tiling, 12
- token, 90
 - composite, 90
 - elementary, 90
 - unit, 90
- token cardinality (TC), 90
- warp, 31

INDEX

Samenvatting

Moderne heterogene platformen bieden de mogelijkheid om de traditionele vormen van parallelisme zoals instructie-, vector-, data-, taak- en pijplijnparallelisme met elkaar te combineren. Daardoor zijn deze platformen bij uitstek geschikt voor embedded systemen omdat de platformen in staat zijn een goede performance te leveren tegen een laag energieverbruik. Echter, het is buitengewoon lastig om alle mogelijkheden voor parallelisatie volledig te benutten. Dit komt zowel door de diversiteit van de verschillende architectuurlagen en componenten als door het gebruik van verschillende programmeermodellen die meestal specifiek gericht zijn op een vorm van parallelisme. Om het hele scala aan parallelisme te kunnen uitbuiten is het daarom noodzakelijk om een programmeermodel toe te passen dat op al deze niveaus tegelijk parallelisme kan representeren. Dit proefschrift maakt een eerste stap door een stroomgebaseerd programma om te zetten naar een nieuw hiërarchische interne programmeerrepresentatie die HiPRDG genoemd wordt. Deze programmeerrepresentatie kan vervolgens omgezet worden naar een multi-niveau programma (MLP), dat zowel taak-, data- als ook pijplijnniveau parallelisme kan beschrijven op een heterogeen platform dat in het proefschrift bestaat uit multiprocessoren (CPUs) en moderne Graphics Processing Units (GPUs).

We introduceren in Hoofdstuk 3 een methode voor het automatisch vinden en exploiteren van dataparallelisme in parallelle procesnetwerken. De gevonden data parallelisme beelden we af op de zwaar parallelle versnellingselementen die aanwezig zijn in GPU's. De beschreven methode stelt ons in staat om dataparallelisme te herkennen en daaruit dan automatisch taak- en dataparallelle rekenkernen te genereren.

Om verschillende vormen van parallelisme te kunnen beschrijven, introduceren we in Hoofdstuk 4 het HiPRDG model. Dit model beschrijft een applicatie in een hiërarchische interne programmeerrepresentatie. Deze representatie maakt intensief gebruik van het polyhedrale model dat vaak gebruikt wordt voor het efficiënt transformeren en

manipuleren van computerprogramma's. Het HiPRDG model maakt het automatisch afleiden van gestructureerde multi-niveau Programma's mogelijk, waarbij elk niveau in de HiPRDG verschillende soorten parallelisme kunnen vertonen uitgedrukt in het gewenste programmeermodel. Voor een CPU is dat multithreaded code en voor een GPU is dat CUDA code. Een andere belangrijke bijdrage van het HiPRDG model is dat de granulariteit van data overdracht automatisch gewijzigd kan worden als onderdeel van multi-niveau programma's.

Om dataparallelle rekenkernen efficiënt te kunnen uitvoeren op de parallelle versnellings-elementen van een GPU, beschrijven we in Hoofdstuk 5 een nieuw stromingsbuffersysteem. Dit systeem zorgt ervoor dat stromende data transfers van en naar de GPU overlappen in tijd op een asynchrone manier. Deze asynchrone communicatie maakt het mogelijk dat CPU en GPU taken in parallel uitvoeren.

In Hoofdstuk 6 hebben wij de meerlaagse parallelisatiemethode gevalideerd met een case study gebaseerd op de M-JPEG multimedia-applicatie. Deze studie toont aan dat het tegelijk paralleliseren op meerdere niveaus en het wijzigingen van de granulariteit van data overdracht tussen taken leidt tot aanzienlijke prestatieverbeteringen vergeleken met wat mogelijk is indien slechts een enkele vorm van parallelisme gebruikt wordt.

De hiërarchische representatie HiPRDG en de multi-niveau benadering van het paralleliseren van computerprogramma's die in dit proefschrift gepresenteerd wordt, vormt de basis voor een hybride benadering waarin elke modelcomponent afzonderlijk kan worden geparalleliseerd door middel van het brede scala aan polyhedrale transformatietechnieken dat al door de compilergemeenschap is ontwikkeld. Op deze manier opent dit onderzoek de deur naar het automatisch genereren van code die is toegesneden op het optimaal uitbuiten van de specifieke eigenschappen van heterogene platformen.

Het werk beschreven in dit proefschrift levert een bijdrage aan het verbeteren van de algemeenheid, de toepasbaarheid, als ook de efficiëntie van automatische parallelisatie en werkverdeling technieken voor heterogene architecturen, waardoor onze modelgebaseerde benadering een veelbelovend alternatief is ten opzichte van handmatige parallelisatie.

Acknowledgments

A journey of a thousand miles begins with a single step. The journey towards this dissertation started in the Mathematical Gymnasium in Belgrade, a truly special place. First and foremost, I would like to thank Professor Arif Zolić for helping me to enter the fascinating world of mathematics and grow up in an intellectually thriving environment. Thanks go also to my diploma thesis advisor, Professor Srdjan Stanković, who encouraged me to pursue my interests in artificial intelligence and distributed computing, and create a work that opened me the doors of the world.

On this journey, the paths of several people crossed with mine briefly, but still made tremendous impact. In 2007, Rolf Rabenseifner at HLRS sparked my interest in parallel programming with his tutorial on OpenMP parallelization. Rolf, thanks for making parallelization so addictively interesting that I could not let it down. NVIDIA, thanks for changing the world with compute-capable GPUs and making it possible to play with parallelization at home. After spending fun two years on hands-on parallelization and GPU optimizations, I started being curious about the technology behind the scenes. A whiteboard talk in 2009 with Intel's Michael Klemm raised my interest in dependence analysis and polyhedral compilation. Michael, thank you for sharing your enthusiasm and technical expertise with me. Professor Koen De Bosschere thanks for helping me join the ACACES summer school in 2009 and enter the High Performance and Embedded Architecture and Compilation community.

I am grateful for the opportunity to further pursue my interests in parallelization at Compaan Design and LIACS, and all of that in a very special setting - working on a commercial grade compiler while still performing state of the art scientific research. This period was hard, but fruitful. While I had to tame many dragons, I came out of it stronger and more skilled. Several special people helped me along the way, and I would like to truly thank them a lot:

Joeri van Ruth for being a fantastic friend. Joeri, thanks for taking me under your wing, teaching me Linux, scripting, and a bit of cooking. Thanks to all the members

Acknowledgments

of ACE Associated Compiler Experts in Amsterdam who have accepted me as an informal member of their group and made me feel like at home. They all have a big place in my heart. On the research plane, I'd especially like to thank Marcel Beemster and Christof Douma who have always been enthusiastic to discuss my research ideas and provide their invaluable feedback. One of the last discussions that we had at ACE helped me to sharpen the concepts that lead to what later became Chapter 4. Thanks Stephen, Joeri and Doeke for jumping in with help for all kinds of technical and non-technical questions. Well deserved thanks goes also to Marius Schoorel for his carefully crafted wordings and state of the art cappuccino. And finally, many thanks to the whole 3rd floor and especially the "gezellige" corner - Doeke, Joeri, Gerrit, and Sander, who made my time at ACE truly awesome.

Many thanks go to all the guys that I have met while working on the Compaan compiler - Johan, Matthijs, Nikolay, and Giuseppe - for creating a nice and enjoyable atmosphere whenever I visited you. Special thanks go to Johan Walters for taking me through ins and outs of the Compaan compiler and teaching me plenty of his Java and design patterns wizardry on the way.

I would like to thank NVIDIA for starting the GPU computing revolution and for supporting my work. Many thanks to Chandra Cheij who made it possible to attend GTC2010, where I had an amazing opportunity to meet for the first time the people working on the core GPU technology, learn from the experts, and get inspired by breathtaking applications. An especially big thank you goes to Cliff Woolley for being an amazing technical partner. Cliff, thank you so much for always being willing to discuss all kinds of technical questions in detail, and provide very insightful answers, ideas, and leads. It has been a true pleasure working with you.

Thanks to all members of the Leiden Embedded Research Center and especially to Sven van Haastregt for helping with PN experiments and sharing his $\text{\LaTeX}$ tricks with me, and Alex Turjan for his PhD advice. In addition, I am grateful to Louis-Noël Pouchet for many fruitful discussions and support in exploring the inner workings of the Polyhedral Compiler Collection (PoCC). Many thanks to SARA's Willem Vermin for discussions on profiling and parallelization.

I also want to immensely thank Matthias Nickles who has been not only the best friend one can imagine, but also a kind of distant mentor following me and supporting me through every step of my PhD. Matthias, thank you for all your guidance during the last years, feedback on all the papers and this thesis, and your brilliant humour. A huge thank you goes to Ana Lucia Varbanescu for her great spirit, encouragement and super constructive comments that helped a lot to improve the final manuscript.

I'm truly obliged to Marko who kindly offered to host me during my EuroPar'09 visit and whose enthusiasm and positive attitude sort of swayed me to move to The Netherlands. Many thanks Sandra for your friendship and wisdom. Big thanks go to Veselin and Vladan Branković - the first for his PhD and tax advice and the second

for hosting me in Stuttgart and showing me how a great life looks like. Thanks Ute Gräter for being a great conversation partner and a friend when I needed it the most. Thanks Marina for being a true role model for enthusiasm and life energy. Many thanks to Igor and Seven Bridges Genomics for being "just around the corner" and giving me access to their high speed internet connection while I was racing against the paper deadline; your help made it possible to present the stream buffer design at PACT'11 which was one of the best conferences I ever had luck to visit. Special thanks go to Doeke for introducing me to Simba The Cat, showing me wonders of Holland, tulips, bikes, storing my tower of book boxes, and making me feel like an extended part of his family.

In addition, I'd like to thank two remarkable mathematicians whom I met at Leiden University - Renato and Samuele - for introducing me to the fun side of PhD and inviting me to their unforgettable dinner parties. Many thanks go to my dear friends Julija and Gojko who made me feel at home in Amsterdam and helped whenever needed. Thanks to Dragan and Biljana for coffees and talks on Sunday afternoons. I am indebted to all four of you for helping me with my yearly August 15 apartment moves. Without your help this essential "technology transfer" would hardly be possible. Thanks Dragoslav for being my 0-delay shipping connection to Belgrade. Thanks Anika and friends from Dutch Conversation Night for great times in Leiden. Thanks Jeff for unforgettable Hogewoerd rooftop BBQs. Thanks to my awesome friends - Filip, Nina, Olja, Guda, Rista, Laza, Zoran, Maxa, Miloš, Marija, Ivana, Andrija - and many others for all the great times back home. Also thanks to my friends all over the world - I am truly grateful for knowing all of you and happy to see you - wherever and whenever we meet!

There is also one truly special person, Ivan, whom I deeply love and care about and want to thank for being there for me, loving me and supporting me. Thank you Ivan for standing by me when needed and helping with all kinds of small things and big things. Thank you for being such an amazing partner and exploring the wonders of the world together. You are truly awesome and you mean a lot to me. Thank you so much for all your love, support, and great times together!

Last but definitely not least, I would like to express my greatest and deepest gratitude to my family (cats included). My truly greatest thanks go to my parents for their tremendous love, support and encouragement on every step of my life. For me, my mom and dad are the two absolutely most amazing, brilliant, and most wonderful people in the known Universe. They have always believed in me, pushed me to do my best and helped me in pursuing my dreams - whatever they are. They are the most responsible for me being today who I am. Mom and dad, thank you for everything, you're the best! :)

Acknowledgments

Curriculum Vitae

Ana Balevic was born on 15th December 1980 in Belgrade, Yugoslavia. As a three-year old she made this:



- an analog gramophone that she is still very proud of. Ana loved playing in her parents dark room for developing photographs, and was totally fascinated when her father brought home an Amiga 500 personal computer with graphic interface. The interest and love for engineering and science was born.

From 1995-1999, Ana attended Mathematical Gymnasium in Belgrade, where she was introduced to the wonderful world of mathematics, computing, and made friends for life. From 1999-2004, Ana studied Computer Engineering and Information Theory at University of Belgrade. In 2004, Ana graduated as Dipl.-Ing. el. (M.Sc.) at the Faculty of Electrical Engineering among the first 5 students (out of more than 1000) while at the same time running business at the family company.

After her M.Sc. studies, Ana spent a year in the DiploFoundation's Internet Governance program (in collaboration with UN, EPO and WTO) researching the interplay of science and technology and its impact on the society, and working on a project dealing with technical patents. In 2006 and 2007, Ana worked at the Center for Digital Technology and Management in Munich. She was responsible for collaboration with Siemens Communications on the CO3onSOA project which involved agile development of collaborative services using Siemens Symphonia Service-oriented architecture.

In 2007, Ana got introduced to the High Performance Computing world at HLRS, Stuttgart. At the same time, the GPU computing revolution was starting. The earliest CUDA-capable GPUs hit the market and Ana received her first GTX8800 under her desk. After playing with CUDA for a few days, Ana created her first numerical simulation, the acceleration of electromagnetic wave propagation, on the GPU. The passion for parallel computing was born. From 2007-2009, Ana worked at IPVS Stuttgart where she designed and developed parallel compression algorithms for compute-capable graphics processing units. She initiated the CUJ2K project and lead the team that developed one of the world's first JPEG2000 codec implementations entirely running on the GPU.

In 2009, Ana moved to The Netherlands where she worked with Compaan Design and LIACS on compile-time solutions for parallelizing and mapping sequential streaming applications onto heterogeneous platforms with accelerators. As a part of the project with Compaan Design and ACE Compiler Experts in Amsterdam, Ana developed the KPN2GPU tool that automatically transforms sequential program specification into GPU code as an extension to the Compaan's heterogeneous computing tool chain. As a next step, Ana created a data-driven solution for execution of streaming applications on heterogeneous platforms with GPU accelerators.

In 2012, a revelation came how to fit the pieces together - leading to the novel methodology for multi-level parallelization in the polyhedral model that enables efficient mapping and tuning of applications for heterogeneous platforms, which culminated into this dissertation.

Ana is currently working as a GPU consultant helping Leiden University Medical Center to accelerate their bioinformatics research, and is passionate about new technologies, accelerating real-world applications, and intersection of art and technology. In her free time, Ana helps Ivan on the DigiCortex project (www.digicortex.net), visits events and exhibitions, and happily bikes around.