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
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 Compaan 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 offer 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.

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 \( \text{mainVIN} \) in statement \( S \) sequentially on one microprocessor 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 \( \text{mainDCT} \) 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-
4.1. INTRODUCTION

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 \( L(jt) \).

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 \( L(js) \) and \( T’ \) corresponds to the loop subnest \( L(jt) \) running on the GPU. To achieve this execution pattern, it is 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 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

![Figure 4.1: Goal: Program structured into two-levels.](image-url)
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.](image)

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 ($f, i t$) 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 $j t$ 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.](image)

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 $L(js)$ 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 $L(js)$ corresponds to a row of the array block. This means that the data access granularity of the loop subnest $L(js)$ is a one-dimensional block array with $HNumBlocks$ elements. Thus, the access granularity of loop subnest $L(js)$ are tokens corresponding to entire rows. Analogously, the loop subnest $L(is, js)$ accesses the whole two-dimensional array. Thus, the access granularity of loop subnest $L(is, js)$ are tokens corresponding to entire frames, as illustrated in Figure 4.4.

![Figure 4.4: Access Granularities in M-JPEG code snippet.](image)

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 $L(js)$. 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 $L(js)$ into a function. The outlined function is invoked by the derived statement $S'$. The memory footprint of loop subnest $L(js)$ 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 $L(js)$. 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 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

```c
//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]);
```

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.

```c
typedef TBlock token_t0_L1;
struct token_t0_L0 rowToken{
  token_t0_L1 block[HNumBlocks];
};
```

```c
program module X:  
//process statement S (mainVIN) in js
forall (jt = 0 to HNumBlocks-1)
  S: mainVIN(block[jt]);
COPYOUT(blocks into rowToken)
```

```c
program module Y:  
//process statement T (mainDCT) in jt
forall (jt = 0 to HNumBlocks-1)
  T: mainDCT(block[jt], &block[jt]);
COPYOUT(blocks into rowToken)
```

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 $P$ is a program composed of $s$ statements $S_1, S_2, ..., S_s$. In the polyhedral model of program $P$, each statement $S$ has an associated iteration vector $\mathbf{x}_S$ 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 $\mathbf{x}_S = (i, j)^T$.

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 $(\mathcal{D}_A, \epsilon, \mathbf{s}_A)$, where

- $\mathcal{D}_A$ is a polyhedral representation corresponding to a program variable $A$, which can be for example a two-dimensional array of integers $\text{int} \{\mathbb{M}\}[\mathbb{N}]$. $\mathcal{D}_A$ is an $m$-dimensional polyhedron bounded by linear inequalities derived from the array $A$ dimensions.

- $\mathbf{s}_A$ is a $m$-dimensional vector representing the sizes of the program variable $A$ in each dimension, e.g. $\mathbf{s}_A = (M, N)$.

- $\epsilon$ is the type of a unit element of the program variable $A$, e.g. $\text{int}$ 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 $\text{int}$, 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 outermost to the innermost. With \( \vec{x}_A = (x_1, x_2, \ldots, x_m)^T \), we denote the index vector \( \vec{x}_A \) of the data space representing \( m \)-dimensional array \( A \).

The index vector of a data space \( D_A \) 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 \( F^A_{S,r} : D_S \rightarrow D_A \):

\[
\vec{x}_S \in D_S, \vec{x}_A \in D_A, \vec{x}_A = F^A_{S,r} \vec{x}_S
\]

where \( F^A_{S,r} \) 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 \( F^A_{S,r} \) corresponds to the dimensionality of array \( A \), with \( i \)th row of \( F^A_{S,r} \) being used to index the \( i \)th-dim of array \( A \), and \( n \) being the number of program parameters.

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

Figure 4.8: Example SANLP.

![Figure 4.9: Mapping Iteration Space of Statement S onto Data Space of Array A.](image)

\(^1\)In 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.
4.3 PRELIMINARIES

illustrates the projection $F_{S_{i}}^{A}$ for the code snippet. The projection $F_{S_{i}}^{A}$ for the code snippet in Figure 4.8 is shown in Figure 4.9. The iteration domain $D_{S}$ of statement $S$ in Figure (a), and the data space polyhedron $D_{A}$ 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 $D_{S}$ 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 $D_A$. 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 $D_A$. A composite token is a composite $d$-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 $\left(\epsilon, d, \vec{s}, m\right)$, where

- $\epsilon$ 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.
- $\vec{s} = (s_1, s_2, ..., s_d)^T$ 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 $L$ takes values in $[1 \ldots d_L]$, where $d_L$ is the number of nested loops in $L$ 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 $L$ 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 $l \in [1 \ldots n_S]$. 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
extending the range of the statement level to \( l \in \{1 \ldots n_S\} \cup \{n_S^+\} \). 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.

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
4.3. PRELIMINARIES

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 $\{0, 1, \ldots n_{S,T}\} \cup \{n_{S,T}^*\}$. 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.:

\[
\vec{d} = \vec{x}_T - \vec{x}_S \\
\vec{d} = \vec{x}_T - M\vec{x}_T
\]  

(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)

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

```c
for (int i=0; i<M; i++) //l=1
for (int j=1; j<N; j++) //l=2
```

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 \((i, j)\).
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 overlayed 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)**

```
SANLP:
for (f = 0; f < NumFrames; f++) {
   for (is = 0; is < VNumBlocks; is++)
      S' :
   for (it = 0; it < VNumBlocks; it++)
      T' :
}
```

**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 $E$. 

96
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.](image)

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 L0 that contains the root node R, the middle-level L1 that contains HiPRDG nodes annotated with definitions of the statements A and B in root node R’s graph $G_{L0}$, and the bottom-level L2. Let us consider node X at level L1, which is annotated with PRDG $G_{L1,X}$. 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 L2. The leftmost child of X, i.e. node Y, defines statement C in X’s graph $G_{L1,X}$. As a consequence, in the code generation phase statement C is substituted with a call to the function that encapsulates code generated from PRDG $G_{L2,Y}$ which
4.5. THE SLICING TRANSFORMATION

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 \( L_h \), 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 \( L_l \). 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 \( b\)lock[\( j \)][\( i \)]. 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 \( d_l(e_{S,T}) = 1+ \). 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 \( s_l \). 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 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 \( s_l = 2 \), 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.

![Diagram showing PRDG slicing to HiPRDG](image)

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 \( L_k = L_{\text{max}} \) 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. CONSTRUCTION OF A MULTI-LEVEL PROGRAM (MLP)

4.6.1 Preparatory Step

Figure 4.18(a) shows that after slicing, processing at the top-level \( L_0 \) proceeds on tokens of type \( \tau_{L_0} \) (i.e. composite tokens corresponding to a collection of blocks, such as 4-block tokens in Figure 4.18), while processing at level \( L_1 \) still proceeds on tokens of type \( \tau_{L_1} \) 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 \( \tau_{L_0} \). The composite data structures for composite tokens are generated as explained in Section 4.3.2. This leads to \( L_1 \) tokens being defined as blocks and assigned alias type \( \text{token}_t_{0\_L_1} \), and composite \( L_0 \) tokens being defined as arrays of blocks and encapsulated into structs of type \( \text{token}_t_{0\_L_0} \). The data type definitions are given in Listing 4.3.

```
// L1 Token Type ~ original data type (TBlock)
typedef TBlock token_t0_L1;

// L0 Token Type ~ 1D array of TBlock elements
struct token_t0_L0
{
    token_t0_L1 elements[HNumBlocks];
};
```

Listing 4.3: Multi-Level Program Data Types

Passing the data from level \( L_0 \) to the lower level \( L_1 \) in Figure 4.18 for processing, requires sending of a collection of blocks (a row) represented as \( \tau_{L_0} \) token to the program modules at level \( L_1 \). To correctly process this collection of blocks at level \( L_0 \), we first must perform conversion from \( \tau_{L_0} \) 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 \( \tau_{L_0} \) 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.

```
void COPYIN(token_t0_L0* rowToken, unsigned int TC, token_t0_L1* blocks, unsigned int start)
{
    for (unsigned int count = 0; count < TC; count++)
```

103
### 4.6. CONSTRUCTION OF A MULTI-LEVEL PROGRAM (MLP)

```c
    blocks[start + count] = rowToken->elements[count];
```

// COPYOUT Procedure
// from data space A with elements of type token_t
// starting at offset position: start
// into composite token tA of type ctoken_t
// with token cardinality TC

```c
void COPYOUT(token_t0_L0* rowToken, unsigned int TC,
             token_t0_L1* blocks, unsigned int start)
{
    for (unsigned int count = 0; count < TC; count++)
        rowToken->elements[count] = blocks[start + count];
}
```

---

**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 $j_s$. 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)

The graphs $G_{L,0}$ and $G_{L,1}$ in Figure 4.18 contain no dependence edges. The absence of dependences leads to parallel code, which we illustrate by `forall` 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 $T_{L,1}$ 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

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)

```c
for (f = 0; f < NumFrames; f++) {
    for (is = 0; is < VNumBlocks; is++)
        DS:....;
    for (it = 0; it < VNumBlocks; it++)
        DT:....;
}
```

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 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

Figure 4.21: One HiPRDG node - one program module. Nodes communicate composite tokens of type $\tau_{L0}$ which corresponds to modules’ arguments.
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 $L_0$ gets row tokens of type $\tau_{L_0}$ from node $X$ at level $L_1$ and passes them to node $Y$ at level $L_1$. The token type $\tau_{L_0}$ 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 $\tau_{L_0}$, 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 $\tau_{L_0}$.

### 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

```c
//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);
}
```

```c
//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)
  for (jt = 0 to HNumBlocks-1)
    T: mainDCT(block[jt], &block[jt]);
  COPYOUT(rowTokenOut<-block);
}
```

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 $\tau_{L_0}$ and send it to level $L_0$. 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 completes, we pack the resulting blocks into output token of type $\tau_{L_0}$ and send it back to level $L_0$. 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
4.7. RESULTS OF MLP CONSTRUCTION

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
4.7. RESULTS OF MLP CONSTRUCTION

*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.

(a) **PPN1** (default)

```cpp
//PPN Node P1 Process Body:
//EXECUTE
for (f = 0; f < NumFrames; f++)
for (is = 0; is < VNumBlocks; is++)
for (js = 0; js < HNumBlocks; js++)
S: mainVIN(&block);
//WRITE TO CHANNEL
push(C1, &block);
```

(b) **PPN2** (adjusted)

```cpp
//PPN Node P1' Process Body:
//EXECUTE
for (f = 0; f < NumFrames; f++)
for (is = 0; is < VNumBlocks; is++)
for (js = 0; js < HNumBlocks; js++)
DS: X(&row);
//WRITE
push(C'1, &row);
```

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 $Y$).](image)

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.