DERIVING A MULTI-LEVEL PROGRAM MODEL FOR EFFICIENT PARALLELIZATION ON HETEROGENEOUS PLATFORMS

Ana Balevic, Bart Kienhuis
Leiden Institute of Advanced Computer Science
Niels Bohrweg 1, 2333CA Leiden, The Netherlands
e-mail: {balevic, kienhuis}@liacs.nl

ABSTRACT
Tremendous progress in automatic parallelization has brought advanced transformations for data parallelism and locality targeting chip multicore processors, but a platform as a whole is seldom considered. Emerging heterogeneous platforms composed of loosely coupled components such as CPUs, GPUs and specialized IP cores, offer unprecedented parallelization opportunities. We propose a hierarchical, multi-level program model, called HiPRDG, which enables more efficient mapping onto heterogeneous platforms, and describe a method for its derivation from the standard program model in the polyhedral framework. In addition, we show how a HiPRDG can be used to derive a multi-level parallel program capable of exploiting task, data, and pipeline parallelism on a heterogeneous platform with a GPU, and present performance improvements on a streaming application case study.

KEY WORDS
Multi-level, parallel, polyhedral, heterogeneous platforms.

1 Introduction
Tremendous progress in automatic parallelization methods over the last decade has brought advanced transformations for data parallelism and locality targeting chip multiprocessors [1–6]. While providing powerful transformations, the existing compiler frameworks consider automatic parallelization targeting a single architectural component, such as a shared memory chip multiprocessor (CMP) or a graphics card (GPU). On the other hand, emerging heterogeneous platforms composed of multiple loosely coupled components, such as CPUs, GPUs, and specialized IP cores, offer multiple levels of parallelism and unprecedented parallelization opportunities. Leveraging both the platform-level parallelism and specific features of different compute-capable devices (CPUs, GPUs, FPGAs) still requires the manual application rewriting. To leverage multiple levels of parallelism on heterogeneous platforms, it would be advantageous to use a multi-level program model.

To bridge this gap, we propose a novel, multi-level intermediate program model called Hierarchical Polyhedral Reduced Dependence Graph (HiPRDG). A HiPRDG is a connected, acyclic graph spanning several layers. The essential feature of the HiPRDG model is the possibility to "zoom" into the specification of each node. Zooming into a HiPRDG node reveals its polyhedral specification. Is that each component of HiPRDG is also a fully-fledged polyhedral program model, which makes it possible to leverage a wide-range of advanced transformation and code generation techniques and tools developed in the polyhedral framework [1–7].

The HiPRDG model provides a basis for the automatic generation of a modular, multi-level parallel programs (MLPP) targeting multiple levels and multiple types of parallelism on a heterogeneous platform. We view a heterogeneous platform as a set of loosely coupled architectural components, each with its own private memory. Communication between components is managed explicitly. With a contemporary heterogeneous platform with a multicore CPU and a data parallel GPU as a target, we can make use of a two-level MLPP with coarse-grain task parallelism at the platform level, and data parallelism at the component level. This parallelization approach makes it possible to leverage not only the platform-level parallelism, but also to exploit the computational power of different compute-capable components on the platform.

This paper is structured as follows: In Section 2, we given an overview of the related work. In Section 3, we present a brief introduction into the polyhedral model. In Section 4, we introduce the hierarchical program model (HiPRDG), present a structured method for HiPRDG derivation, and describe steps in the generation of a multi-level parallel program. In Section 5, we show how to obtain a two-level parallel program featuring task, data, and pipeline parallelism, and demonstrate performance improvements on a streaming application case study.

2 Related Work
Several compiler frameworks and runtime environments for parallelization of sequential applications emerged over last years. HMPP [8] allows programmers to specify which parts of the sequential application are to be offloaded on a target device, such as GPU or Cell B.E. processor, and provides the wiring code and a runtime environment. The CE-TUS compiler automatically generates data parallel code for CMPs and GPUs [9]. Bondhugula et al. [5] developed an automatic parallelization framework PLuTo for enhancing parallelism and locality through tiling. Parallelization
frameworks, such as PLuTo and CHiLL [10] analyze the source code, and transform it into a tiled form which enables data reuse in multi-level caches on a single CPU, and which also supports parallel execution on chip multicore processors. Baskaran et al. [6] adapted the PLuTo approach for execution on the first generation of GPUs. Compiler frameworks in the embedded space, such as Compaan [11], also take advantage of dataflow analysis and the polyhedral model. The Compaan compiler represents streaming applications using an intermediate model that exposes task parallelism. Contemporary polyhedral frameworks use single-layered (flat) models and focus either on component-level parallelism or platform-level parallelism. The concept of hierarchy can be currently captured only by manual rewriting of the source code, and rerunning tools on each component separately to generate individual models. The approach presented in this paper aims to alleviate this issue by the introduction of a hierarchical program model that can be obtained by transformations in the polyhedral framework directly.

3 Preliminaries

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) [7], as it provides a mathematical basis for optimizing transforms, and subsequent code generation. A SANLP is a program in which each program statement is enclosed by one or more loops and if-statements, and where for-loop bounds, conditionals and array indexes are affine expressions of the enclosing loop iterators, static program parameters, and constants. An example SANLP is shown in Listing 1.

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

Listing 1. Example of a SANLP in C.

In the polyhedral model, the for-loops surrounding a statement define its iteration domain. A loop nest is represented using an n-entry column vector called its iteration vector \( \vec{x} = [x_1, x_2, \ldots, x_n]^T \) where \( x_k \) denotes the k-th loop index and \( n \) denotes the innermost loop. An instance of a statement, i.e. operation \( S(\vec{x}_S) \), is executed once for each value of the iteration vector in the domain. The iteration domain of statement \( S \) is a two-dimensional polyhedron, specified by linear inequalities \( 1 \leq i \leq M \) and \( 1 \leq j \leq N \). Statement \( S \) corresponds to the function call \( f \) with input argument \( \text{in}[i][j] \) and output argument \( \text{tmp}[i][j] \). Statement \( T \) corresponds to the function call \( g \) with input argument \( \text{tmp}[i][j] \) and output argument \( \text{out}[i][j] \). The indexes of function arguments are affine \( \text{in}[i][j] \) and \( \text{out}[i][j] \). Data flow between statements \( S \) and \( T \) is made explicit through array variable \( \text{tmp}[i][j] \).

Data Dependence According to Bernsteins’s conditions [12], two operations \( S(\vec{x}_S) \) and \( T(\vec{x}_T) \) are data dependent if they access the same variable and if at least one access is a write. In Listing 1, there is a dataflow dependence from \( T \) to \( S \) induced by a read-after-write access to a shared variable \( \text{tmp}[i][j] \). A data dependence between two operations \( (S, \vec{x}_S) \) and \( (T, \vec{x}_T) \), is denoted as \( S(\vec{x}_S) \Rightarrow T(\vec{x}_T) \). Distance vector of a dependence is defined as \( \vec{x}_T - \vec{x}_S \), 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 \( h_{S,T} \), which corresponds to the number of loops that surround both \( S \) and \( T \).

PRDG A program can be compactly represented in the polyhedral model by its Polyhedral Reduced Dependence Graph (PRDG) [2]. A statement-level polyhedral reduced dependence graph \( G = (V, E) \) is a directed multi-graph that consists of a set of vertices \( V \) (nodes) and a set of directed edges \( E \). Each program statement \( S \) in program \( P \) is represented by a node \( N_S \) in \( V \):

\[
V = \{N_S \mid S \in P\}.
\]

A PRDG node \( N_S \) is defined by its program statement \( S \), dimensionality of the statement \( d_S \) (corresponding to the number of surrounding for-loops), statement’s iteration vector \( \vec{x}_S \), and iteration domain \( D_S \subset \mathbb{Z}^{d_S} \). For each dependence in the set \( R_{S,T} \), where \( R_{S,T} \) is a non-empty set of all dependence pairs between operations of statements \( S \) and \( T \), there is a directed edge \( e_{S,T} \in E \) from node \( S \) to node \( T \). Each edge \( e_{S,T} \) in PRDG is annotated with a dependence polyhedron. The dependence polyhedron represents the homogeneous system of linear equalities and inequalities defining the relationship between the P/C pair of operations. The dependence polyhedron abstraction subsumes weaker dependence abstractions, such as level of dependence and direction vectors [2].

PPN The Polyhedral Process Network (PPN) model is a polyhedral variation of the Kahn Process Network (KPN) model [13] that defines a program as a set of processes communicating via channels. In a PPN, program statements are represented by process nodes and dependence edges by channels. For a SANLP, a PPN can be derived automatically using a tool such as Compaan [11]. The PPN model has been used as to automatically generate a parallel program featuring task and pipeline parallelism for a wide range of target architectures and program models.

4 Multi-Level Model of a Parallel Program

A standard intermediate program representation in polyhedral model is by default flat, i.e. it has only a single layer.
On the other hand, a heterogeneous platform has multiple levels (platform-level and component-level), and may support different types of parallelism (task, data, pipeline). A flat program model can not exploit these multiple layers of parallelism. In order to enable more efficient mapping of application onto heterogeneous platforms, we first introduce a novel multi-level program model, then present the slicing transformation which is used to derive it from the standard program model, and finally describe steps in the construction of a multi-level parallel program.

### 4.1 Hierarchical Polyhedral Reduced Graph

To capture the concept of a hierarchical, multi-level program representation in the polyhedral framework, we introduce an intermediate program model called Hierarchical Polyhedral Reduced Dependence Graph (HiPRDG). A HiPRDG is a connected, acyclic graph (tree) spanning several layers. It consists of a set of nodes and a set of edges. Similar as in PRDG, each HiPRDG node is characterized by a statement, iteration vector and iteration domain. However, HiPRDG provides a possibility to zoom into the statement specification. Zooming into the statement specification reveals its polyhedral model, since we annotate each statement in HiPRDG with its lower-layer polyhedral dependence graph (PRDG).

An example HiPRDG is depicted in Figure 1(b). The HiPRDG tree \( T \) represents three layers of the program: the top-layer \( L_0 \) that contains the root node \( R \), the middle-layer \( L_1 \) that contains nodes that define statements of a graph in node \( R \), and the bottom-layer \( L_2 \). Let us consider the tree node \( X \) at layer \( L_1 \). The statement of the tree node \( X \) is defined by PRDG labeled as \( G_{L1,X} \). The graph \( G_{L1,X} \) is a simple pipeline with three nodes: \( C, D, \) and \( E \). This results in tree node \( X \) having three descendants, namely nodes \( Y, Z \) and \( V \), in the next layer of the HiPRDG, i.e. layer \( L_2 \). The leftmost child of tree node \( X \), node \( Y \), is defined by PRDG \( G_{L2,Y} \). As a consequence, in the code generation phase the statement of node \( C \) is substituted with a call to a function that encapsulates code generated from PRDG \( G_{L2,Y} \).

An abstract pseudocode structure that would correspond to this HiPRDG is shown in Figure 1(a). The shadowed boxes encapsulate parts of nested loop program that correspond to the nodes of the HiPRDG. In Figure 1(a), there are three layers of boxes. Each pseudocode box is encountered at a deeper level, i.e. increasing nesting depth. The notion of depth (level) in the program is the key concept that we use to derive the hierarchical program model out of the standard (flat) polyhedral program model, and it will be explained in Section 4.2.1.

The HiPRDG model is obtained by a set of structured transformations directly in the polyhedral model, without a need for manual code modifications. In the next section, we present a structured graph slicing method that enables us to derive a HiPRDG from a standard, flat PRDG.

### 4.2 The Slicing Transformation

The slicing transformation takes as input a polyhedral program specification, e.g. expressed in form of an annotated dependence graph (PRDG) and outputs a hierarchical program model spanning two or more layers (HiPRDG), described in Section 4.1. Intuitively, slicing corresponds to the separation of a flat program model into two or more layers stacked on top of each other. Transformation of a SANLP into multiple levels based on the level of program components in a loop nest, or equivalently its polyhedral representation extended with concepts presented in previous sections, is what we call slicing. Before we proceed to with the method, let us explain the concept of depth (level) which has a central place in the slicing analysis.

#### 4.2.1 Concepts

In compiler analysis, the nesting level \( l_i \) (depth) of some loop \( i \) in loop nest \( \mathcal{L} \) is considered to be equal to the number of the enclosing loops plus one. A statement in a SANLP with imperfectly nested loops is fully defined by its itera-
tion domain and the textual order (position) within the loop nest. To encode the information on the textual order, we extend the range of the statement level to \( l \in \{1 \ldots n_S \} \cup n_S^* \).

The special value \( n_S^* \) is introduced to encode the presence of the textual order at the loop level \( n \). This allows us to consider imperfectly nested loops.

Following the definition by Allen and Kennedy [14], the dependence level \( dl(e) \) of an edge \( e : S(x_S) \rightarrow T(x_T) \) corresponds to the depth of the first (outermost) loop for which the iterator values are different. The dependence level of a loop-carried dependence takes value from 1 up to \( n_{S,T} \). 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 valid range of dependence levels is in the set \( \{0, 1, \ldots, n_{S,T} \} \cup \{n_{S,T}^* \} \).

The dependence level of a dependence edge \( e \) in PRDG can be derived directly from its dependence polyhedron \( P_e \). The equality in dependence polyhedron specifies the affine mapping between the pair of P/C operations, i.e. it corresponds to the distance vector \( \vec{d} = \vec{x}_T - \vec{x}_S \). We derive the dependence level \( dl \) of the edge \( e \) as the first positive component of the distance vector. Dependence is carried at depth \( k \) of a SANLP, if its dependence level equals \( k \), i.e. \( dl = k \). The zero value \( dl = 0 \) 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.

4.2.2 Slicing

Slicing at slicing level \( sl \) means that components of the program model, such as loops at and above the slicing level \( sl \) are assigned to hierarchy level \( L_h \), whereas loops strictly below the slicing level \( sl \), i.e. the loops with depths \( d > sl \) are assigned to the lower hierarchy level denoted as \( L_l \). Given a polyhedral program model, the slicing of the model into two layers is performed by comparing 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 (depths):

- level \( l_m \) above/at level \( l_k \), i.e. \( 0 \leq l_m \leq l_k \)
- level \( l_m \) below level \( l_k \), i.e. \( l_k < l_m \leq l_n \)

where \( l_n \) is the total number of levels. Let us illustrate the slicing transformation on a P/C pair of statements in M-JPEG encoder pseudocode (Listing 2). The default single-layer PRDG corresponding to this pseudocode is shown in Figure 2(a). It contains two nodes \( S \) and \( T \) that are connected via dependence edge \( e \) representing a dataflow dependence induced by read-write accesses to program variable \( \text{block}[j][i] \). The iteration vector of statement \( S \) is given as \( \vec{x}_S = [f,i,s,j] \), and the iteration vector of statement \( T \) is given as \( \vec{x}_T = [f,i,t,j] \). The common nesting level of statements \( S \) and \( T \) is \( n_{S,T} = 1 \). The iteration vector representing common dimensions are \( \vec{x}_S^c = [f] \) and \( \vec{x}_T^c = [f] \), and the distance vector is \( \vec{d} = \vec{x}_T - \vec{x}_S = \vec{0} \).

Listing 2. An example of a P/C pair (M-JPEG Encoder)

Zero-valued distance vector means that the dependence is determined by the textual order of statements at the common loop nesting level \( n_{S,T} = 1 \). According to the textual order encoding scheme, the level of this dependence is \( dl(e_{S,T}) = 1 + \).

Node Splitting First step of the slicing transformation consist of splitting of the PRDG nodes according to the slicing level \( sl \). Statement \( S \) with iteration vector \( \vec{x}_S = [x_1, x_2, \ldots, x_n] \) is split into its higher part \( S^H \) and its lower part \( S^L \):

- \( S^H \) with the iteration vector \( \vec{x}_S^H = [x_1, \ldots, x_d] \)
- \( S^L \) with the iteration vector \( \vec{x}_S^L = [x_{d+1}, \ldots, x_n] \)

In the polyhedral model, this is realized by splitting the statement nodes and modifying their iteration vectors. As illustrated in Figure 2(b), the iterators denoting node dimensions above the slicing level, i.e. iteration vector components from the 1-st to the \( sl \)-th are assigned to the iteration domain of the node \( S^H \) in the higher level of hierarchy \( L_H \), while the node dimensions below the slicing level, i.e. iteration vector components from the \( sl + 1 \)-th to the \( n_S \)-th are assigned to the iteration domain of the node’s lower slice \( S^L \) in the lower level of hierarchy.

If two or more statements are at the same level in the original program, we introduce a derived statement to represent them in the higher layer. Derived statements are represented by shadowed boxes encapsulating parts of SANLP in Figure 1(a). Let us explain this on the example of statements \( H \) and \( I \) enclosed by a doubly nested loop. Statements \( H \) and \( I \) are located at level 2\(^\circ\) (loop nest level 2 plus textual order). If we slice at the boundary of the second for-loop \( (sl = 2) \), it is necessary to introduce a derived statement \( D \) to represent both \( H^H \) and \( I^H \). Derived statement \( D \) has an iteration vector \( \vec{l}_D = [x_1, x_2] \). The lower parts of statements (below level \( sl = 2 \)) \( H^L \) and \( I^L \) are generated in the same manner as in the previous example, and assigned to PRDG \( G_{12,Z} \), which defines tree node \( Z \).

Slicing of a PRDG node result in two nodes with lower-dimensional iteration domains than the original
node. If two statements are at the same level (including textual order), their higher level parts are encapsulated into a derived statement represented by a single node in a top PRDG. The nodes representing statements obtained by slicing are assigned to PRDGs at different layers of hierarchy.

**Dependence Placement** Second, the dependence edges must be analyzed, and assigned to the appropriate HiPRDG layer and the appropriate component at that 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 1: 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 2: 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. Furthermore, the specifications of dependence edges (dependence polyhedra) are reduced to contain only the linear (in)equality in the iteration space dimensions of the given program layer.

The result of the slicing transformation is a set of stand-alone PRDGs. Each PRDG is associated with some program layer, and assigned accordingly to a HiPRDG node in that layer, i.e. a HiPRDG node statement is defined by its polyhedral graph. Nodes at each layer of HiPRDG represents a subset of (iteration space) dimensions of the original program. For example, in case of a two-layer HiPRDG, the top-layer contains all iteration space dimensions above the slicing level, and the bottom-layer contains all iteration space dimensions below the slicing level.

**4.3 Encapsulation**

In the parallel program construction phase, each connected component (a stand-alone PRDG) in a HiPRDG node specification is transformed into a program module. The general steps in program module construction are:


2. **Code-generation** for the given model.

3. **Encapsulation** into stand-alone program modules.

First, it is necessary to find a schedule that satisfies the partial order imposed by data dependences in the original program [2]. At this stage numerous transformations in polyhedral model [1,5,7,10] can be applied in order to optimize, or further parallelize the given component. Additionally, it is possible to build an intermediate parallel program model for the given component, such as a PPN [15], which can aid generation of a parallel program at a given layer with specific characteristics such as task and pipeline parallelism.

Second, we perform code generation for the given program model. Program code can be generated by running a code generator tool on the PRDG provided with a schedule [4]. There are several code generation tools available. For example, the Compaan compiler that we use to construct the PPN model is capable of generating code for wide-variety of programming models and APIs, such as SystemC, VHDL, and CUDA [11, 16].

Third, it is necessary to encapsulate the generated code into a function. The function arguments correspond to the tokens used for inter-layer communication in HiPRDG. During program slicing we assume that all accesses (memory references) at some level have the same access granularity. In M-JPEG encoder given in Listing 2, loops \(js\) and \(jt\) at level 3 access individual image blocks, loops \(is\) and \(it\) at level 2 work on image rows, and the \(f\)-loop at level 1 works on entire frames (two-dimensional array of blocks). If we slice M-JPEG encoder at level \(sl = 2\), the bottom layer components would still process image blocks, but the components in the top layer would be processing...
data at coarser granularity corresponding to image rows (one-dimensional array of blocks). Thus, the tokens passed between two layers are image rows. The tokens passed between two adjacent layers correspond to the access granularity at the higher layer. The formal arguments of the program module are then generated using the specification of token data structure.

5 Results

The HiPRDG model has been introduced with several usage scenarios in mind: HiPRDG enables generation of multi-level parallel programs targeting multi-level platforms, it enables hybrid parallelization by making it possible to use different transformation tools on HiPRDG nodes, and it enables granularity tuning in pipeline parallel programs. In this section, we show how the concepts and techniques introduced in the previous sections improve parallelization of streaming multimedia applications and mapping onto heterogeneous platforms with accelerators on the M-JPEG encoder case study. First, we give a brief overview of the M-JPEG encoder workflow. Second, we describe the default singe-layered M-JPEG PPN model featuring task and pipeline parallelism that can be automatically obtained from M-JPEG SANLP. Then, we describe two improvements achieved by using our model for hierarchical parallelization.

5.1 M-JPEG Overview

Various standards have been developed for compression of digital video signals. The M-JPEG standard specifies a video codec in which each frame of the video stream is encoded independently as a still image using the JPEG standard for image compression. Figure 3 shows the block diagram of the JPEG encoder. The JPEG encoder partitions the input image into 8 × 8 blocks of pixels (macroblocks). Each macroblock is processed independently. Each macroblock of the image is passed through the DCT module after loading and pre-processing. The DCT module performs a 2-dimensional DCT transform to decorrelate 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. This operation is equivalent to applying different quantizers to different frequency bands of the image. 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 image.

5.2 M-JPEG PPN

We converted the M-JPEG SANLP to a PPN using COMPAN [11]. The M-JPEG PPN is a single-layered model containing four processes (corresponding to four JPEG tasks) that form a simple processing pipeline. The processes are connected by channels implemented as FIFO buffers. The processes exchange data via tokens corresponding to 8 × 8 pixel macroblocks. A task is generated for each process. The code within each task is executed sequentially. The parallel program mjpeg – ppn obtained from the default single-layer PPN is depicted as a four-task pipeline in Figure 4. The default PPN does exploit task parallelism (four nodes represent autonomous tasks), and also pipeline parallelism (through pipelining of task computations by using FIFO buffers for channels). However, it is a single-layered model, and as such cannot take advantage of further parallelizing nodes for vector or data parallelism, for example. This could be improved by constructing a two-layer model, where the top-layer still contains a PPN with task and pipeline parallelism, but the components at the lower level may be further processed for data parallelism. This is achieved by slicing the program model into two layers, as described in Section 4.2.2 on the partial M-JPEG code snippet. After obtaining the two-layer HiPRDG, similar to the pipeline in Figure 2, the top-level graph G₀ is transformed into a PPN. To demonstrate benefits of the two-level parallelization, we selected the computationally intensive mainDCT node for further parallelization. To extract data parallelism and generate CUDA code for the GPU, we applied the KPN2GPU tool [16] on the bottom-level mainDCT node. The other bottom-level components were transformed into sequential code. The program represented by a pipeline in Figure 4 can take advan-
tage of two-levels of parallelism: at the top-level there is
still task and pipeline parallelism just at a coarser granular-
ity, and at the bottom-level, we perform data parallelization
and computation offloading of selected nodes to a GPU. By
adjusting the slicing level, we also adjust the granularity of
tasks and tokens.

5.3 Experiments

The test platform used for experiments is a desktop PC fea-
turing an Intel Core i7-920 Nehalem architecture 2.66GHz
multicore processor, and an NVIDIA Tesla C2050 GPU.
connected via PCIe 2.0 x16 bus. To obtain the baseline
performance, we measured the performance of the default
PPN generated by Compaa. The PPN was executed on the
multicore CPU as a set of autonomous tasks (implemented
as POSIX threads) communicating via FIFO bu-
ff
ers. The
experiments were performed on a stream of frames, corre-
sponding to color images of 128 × 128 pixels each. The
default PPN working on tokens of the macroblock size has
an average throughput of 470 KB/s. Analysis revealed
that the computationally-intensive mainDCT node domi-
nates the execution time of the M-JPEG encoder, thus mak-
ing it the bottleneck node of the M-JPEG pipeline. The
overall performance could be improved with acceleration
of DCT node’s computations. In addition, using small size
tokens (macroblocks), there is frequent synchronization be-
tween each pair of P/C nodes. In the subsequent section,
we show how the techniques and methods proposed in this
token size, the
paper alleviate some of the issues above.

Figure 5. Token Granularity Tuning

Token Granularity Synchronization in the PPN is in-
duced by blocking read and write operations performed in
each process iteration. The number of blocking operations
can be reduced by increasing the token size, and thus re-
ducing the number of process iterations that are required
to process the same amount of input data. The coarsen-
ing of the token granularity is achieved by using the slic-
ing method to construct a two-level HiPRDG, and adjusting

Figure 6. Two-level parallelization: platform-level task and
pipeline parallelism, extended with data parallel DCT.

Multi-Level Parallelization Figure 6 shows compari-
son of M-JPEG throughput when executed with a default
(single-level) PPN denoted as mjpeg-ppn-default, and
when executed with a two-level PPN denoted as mjpeg-
ppn-twolevel. The x-axis show token size in terms of im-
age macroblocks. The top-level graph in HiPRDG model
was used to obtain a coarser-grain PPN. Zooming into DCT
node reveals that it can be o
ffl
oaded onto GPU for data par-
allel processing. Speedup of DCT computation alone is up
to 87× using the KPN2GPU tool. The overall performance
of the parallel M-JPEG increases from 480 KB/s to 1826
KB/s. The use of two-level parallelization enables us ex-
plote data parallelism on GPU in addition to the task and
pipeline parallelism at the platform level.

6 Conclusion

We believe that in order to exploit rich parallelism op-
portunities provided by heterogeneous platforms, having
a multi-level program model is highly advantageous. We
make a step in this direction by introducing a novel in-
termediate model (HiPRDG) used to derive and repre-
sent multi-level parallel programs directly in the polyhe-
dral framework. In addition, we discuss use cases for
HiPRDG including the generation of a two-level parallel
program featuring task, pipeline, and data parallelism, and
Figure 4. M-JPEG PPN is a pipeline composed of four tasks. Using HiPRDG, we can perform two-level hierarchical parallelization resulting in top-level task and pipeline parallelism (as in the original PPN), with additional data parallel processing for the selected node. The use of HiPRDG allows adjustment of the token granularity (illustrated with the multirate factor mr).

demonstrate performance improvements over a single-level model when mapping M-JPEG onto a heterogeneous platform with a GPU. By introducing the possibility to represent the concept of hierarchy directly in the polyhedral model, this paper opens doors for future research on automated generation of multi-level parallel programs that can more efficiently exploit features of different accelerators and the abundant parallelism on heterogeneous platforms.

References