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
Chapter 3

Identification and Exploitation of Data Parallelism in PPNs

3.1 Introduction

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

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

![Diagram of process network node vs. data parallel processing on GPU](image)

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

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

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

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

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

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

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

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

3.3 Overview of the Solution Approach

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

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

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

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

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

3.4.1 Data Parallelism

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

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

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

Data dependences between operations are the only source of causal constraints on the execution order. To preserve the semantics, the operation \( c \), which is the source of the data dependence, must execute before operation \( d \), which is the destination of the data dependence.
Let us define a function \( t \) that assigns a time step to each iteration point of the iteration domain \( D \) as follows:

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

\[
\forall x, y \in D \land (x \Rightarrow y) : t(x) < t(y).
\]

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

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

### 3.4.2 Space-Time Mapping of PPN Processes

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

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

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

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

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

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

- Predictor
- Grid
- Parallel2D

We now discuss each pattern in detail.

**Predictor** The source code of the predictor example is given in Appendix A.1. Let us analyze the transformer node \( T \), which evaluates the function \( \text{predict} \). The transformer process computes a new pixel value in a 2D-image on the basis of its two

![Figure 3.4: Predictor Node T with domain D: (a) Data dependences, (b) Sets of independent operations, (c) Data Parallel Target Domain (DII).](image)
neighbouring pixels. The statement executed by process $T$ is $A[i][j] = pred(A[i-1][j], A[i][j-1])$. This statement is executed by all operations of the iteration domain shown in Figure 3.4(a). As a result of data flow analysis, the PPN process $T$ contains two self-links representing the two data flow dependences:

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

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

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

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

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

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

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

3.4.3 Target Domain Characterization

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

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

**Definition 24 (\((W, D)\) parameters)** Given a target domain \(D_p^{II}\) that is the image of the node \(P\)’s domain \(D_p\) with index vector \(\tilde{x}_p\) under a space-time mapping \(T\), which
3.5 Intermediate Model: A Data Parallel View (DPV)

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

The PPN processes execute process iterations one by one. The operations are executed following the lexicographic order in the source code from which the PPN node
3.5. **INTERMEDIATE MODEL: A DATA PARALLEL VIEW (DPV)**

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

![Diagram](image.png)

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

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

In a PPN, the processes read and write single tokens into the channels. To support data parallel execution, we introduce a *Data Parallel Channel* (DPC). A data parallel channel supports concurrent accesses by parallel processing elements, i.e. a process can read multiple tokens at once, or write multiple tokens at once. The ports are transformed in a similar manner, thus instead of being one token wide, now they are \( W \) tokens wide, where \( W \) corresponds to the amount of data parallelism in the process.
domain. For example, the DPP process $P'_2$ at time step $t = 3$ in the predictor example above reads 4 tokens in the read phase, evaluates the predict function 4 times in the execute phase, and writes 4 results to the output channels in the write phase.

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

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

### 3.5.1 Data Parallel Process

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

**Definition 25 (Parallel Process Iteration (PPI))**

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

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

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

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

49
3.5. INTERMEDIATE MODEL: A DATA PARALLEL VIEW (DPV)

Definition 26 (Parallel Node Domain (PND))

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

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

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

Definition 27 (DPP Process)

A DPP process is an autonomous execution entity specified by:

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

where $(W,D)$ parameters introduced in Definition 24 specify the maximal number of parallel processing entities\(^1\) that can process the PND, and the number of time steps required to process the PND with $W$ processing entities.

\(^1\)A processing entity can be implemented, for example, as a process or a thread.
3.5. INTERMEDIATE MODEL: A DATA PARALLEL VIEW (DPV)

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

3.5.2 Data Parallel Channel

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

Let us explain the construction of the DPC address function step by step on the predictor example’s node P_2 in Figure B.1. For the purpose of the explanation, let us suppose that a PPN channel C_2 is represented as a memory array a_1 as illustrated in Figure 3.8. At each process iteration, producer P_1 writes an element to the location that corresponds to the current value of its iteration vector. A PPN channel is annotated with an affine mapping that specifies the exact producer-consumer relation between the nodes, i.e. given a specific value of the iteration vector of the consumer, it is possible to trace back the exact iteration at which the token was produced by P_1. To read a token into process P_2, we construct the array index as the read vector \( \vec{r} \) using the current value of the process iteration vector and the channel mapping specification as \( \vec{r} = \vec{x}_P = M \vec{x}_C \). In the case of a read access from P_2 to channel C_2 with the mapping \( M_{C_2} = \begin{bmatrix} 1 & 0 & -1 \\ 0 & 1 & 0 \end{bmatrix} \), we obtain read vector \( \vec{r}_{P_2} = M_{C_2} \vec{x}_{P_2} = [i - 1, j]^T \), since \( \vec{x}_C = \vec{x}_{P_2} = [i, j]^T \). This means that at iteration \( i = 3, j = 1 \), the process P_2 reads input argument \( i_{n_0} \) from channel C_2 from the location a_1[2, 1].

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

![Diagram](image)

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

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

\[
\vec{x}_P = T_P \vec{x}_P \land \vec{x}_C = T_C \vec{x}_C \land \vec{x}_P = M \vec{x}_C \Rightarrow \vec{x}_P = T_P M T_C^{-1} \vec{x}_C.
\]

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

**Definition 28 (DPC Mapping)**

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

\[
OPD_{P',k} = M'(IPD_{C',k}).
\]

The mapping \( M' \) is a composite function:

\[
M' = T_P M T_C^{-1}.
\]

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

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

### 3.5.3 Synchronous Data Parallel Execution

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

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

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

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

3.6 Mapping and Code Generation for GPU Accelerators

3.6.1 Introduction

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

In this section, we provide a structured way of mapping a process under DPV (i.e. a DPP) onto a CUDA kernel. We show how to generate all necessary components of a CUDA kernel from the DPV model. A simple, functionally correct CUDA kernel that exposes maximal data parallelism is obtained by traversing the data model behind the Data Parallel View in a straight-forward manner. This approach is implemented in the KPN2GPU tool, which is a Java-based extension of the Compaan compiler.
The same mapping approach can be applied for obtaining parallel programs according to the OpenCL standard. OpenCL kernels can easily be obtained by writing an OpenCL backend according to CUDA to OpenCL mapping scheme, or by using source-to-source code conversion tools [71, 91].

3.6.2 CUDA Code Generation

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

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

$$p_0 = threadIdx.x + 1.$$ 

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

for($t_0 = 0; t_0 < ND_2_D; t_0 += +)$....

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

---

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

3The offset 1 is necessary only out of technical reasons. The scheduler returns $t$ values starting from 0, but the allocator tool returns $p$ values starting from 1.
Now that we know how to parametrize kernel execution, let us show on the predictor example how to automatically generate a CUDA kernel from the specification of a DPP. A detailed specification of the DPP and the DPCs used to generate $ND_2_{\text{Kernel}}$ from the DPP node $P'_2$ can be found in Appendix B.1.3, and its complete CUDA kernel can be found in Listing B.2. For reference, we also illustrate the mapping between DPP components for the transformer node $P'_2$ in the predictor example and the programming constructs in the CUDA kernel in Figure 3.10.

We generate the number of CUDA threads $W$ executing the kernel (i.e., 4 threads shown in Figure 3.10(c)) from the $W$ value in the $(W, D)$ characterization of DPP $P'_2$. We generate the number of time steps executed by the DPP from the $D$ value in the $(W, D)$ characterization of DPP $P'_2$. As a result, the kernel $ND_2_{\text{Kernel}}$ that implements DPP $P'_2$ is executed by $W = 4$ parallel threads in $D = 7$ synchronous time steps. The 4 threads are organized in one thread block (TB), as shown in Figure 3.10(c).

We transform the DPP control unit $ctrl$ into the kernel control code. The kernel control code consists of the for-loop that goes through all instances of the time-dimension $t_0$ and invokes process iterations, and the synchronization to control transitions between the R/E/W phases of process iterations. To guide the execution of 4 CUDA threads through 7 time steps, we introduce the control loop in the $ND_2_{\text{Kernel}}$ as illustrated in Figure 3.10(g). The upper bound of the control loop $ND_2_D$ corresponds to the number of time steps executed by the DPP, i.e., the value of the $D$ parameter. At each iteration of the loop, one process iteration (See Figure 3.10(e)) is executed by each thread. Since all threads execute the kernel body in parallel, this means that at each time step $t_0$ multiple process iterations are executed (one per each active thread). As in the PPN process, each process iteration contains the three
3.6. MAPPING AND CODE GENERATION FOR GPU ACCELERATORS

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

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

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

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

Listing 3.1: CUDA Kernel: Process Iteration

```c
Listing 3.1: CUDA Kernel: Process Iteration
```
instance of \( t_0 \). For reference, we show the parametrized CUDA code that implements the process iteration from Figure 3.10(e) also in Listing 3.1.

In the read phase, each active thread reads the input arguments \( \text{in}_0 \) and \( \text{in}_1 \) from incoming channels into the local variables. In the predictor example, the actual values of each argument can be provided by two channels, e.g. they could be read from either Data Parallel Channel \( \text{DPC}_1 \) or \( \text{DPC}_2 \) for the input argument \( \text{in}_0 \). From which of the two channels the thread gets the value at a given iteration is determined by evaluation of the guard conditions at lines 7 and 9 for the input argument \( \text{in}_0 \) and by evaluation of the guard conditions at lines 13 and 15 for the input argument \( \text{in}_1 \).

The CUDA definition of the guard conditions is generated automatically from the polytopes describing the input port domains (IPDs) which the DPP uses to read data from the channels \( \text{DPC}_1 \) and \( \text{DPC}_3 \). The resulting guard conditions are parametrized in \((p_0, t_0)\), i.e. thread index and control-loop counter. For more details, see the guard condition definitions on lines 5-14 in Listing B.2.

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

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

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

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

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

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

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

3.7 Scaling

3.7.1 Introduction

In Section 3.6 we presented an approach for mapping a DPV network onto a GPU. The parallelism in a DPP conceptually matches very well a single CUDA thread block. Let us see how this approach can be scaled up. In the previous sections, we have shown how to convert a 4×4 predictor domain into a DPP processed by 4 threads in parallel. If we increase the size to 512 threads, the code can be still obtained and processed in the same manner as explained earlier. By further increasing the domain size to 4000×4000, we exceed the number of threads supported by CUDA for a single thread block. Taking care of this problem is important, since to fully fill up the GPU,
it is necessary to provide work for all of its streaming multiprocessors (SMPs). For example, our Tesla C2050 GPU with 14 SMPs needs more than 14 thread blocks to make all SMPs busy. Furthermore, even larger number of thread blocks is desired in order to process work efficiently on the GPU. Having a larger number of thread blocks enables the scheduler on each SMP to overlap independent instructions from different thread blocks. It also enables better load balancing on the GPU, since different blocks can take different time to complete.

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

### 3.7.2 Tiling for Coarse-Grain Data Parallelism

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

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

\begin{align}
TY \cdot jt &\leq p_0 \leq TY \cdot jt + TY - 1, \\
TX \cdot it &\leq t_0 \leq TX \cdot it + TX - 1,
\end{align}

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

\begin{align}
0 \leq it &\leq 3, \\
0 \leq jt &\leq 1, \\
it - 2 \leq jt, \\
jt &\leq it.
\end{align}

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

Dependences between tiles must be satisfied during the execution. In the example above, this means that tiles C, D, and E must be executed after tile B. Since there are no dependences between tiles C and D, these two tiles can be executed in parallel on a GPU. Finding a valid execution order of the tiles is the same problem as finding data parallelism in a domain, and it can be solved following the approach in Section 3.4.2.

We optimize for maximal data parallelism in order to maximize the number of independent tiles in each time step $t_1$. For the example above, the tile scheduling function is $t_1(it, jt) = it + jt$, and the tile allocation function is $p_1(it, jt) = jt$.

### 3.7.3 Consequences for GPU Mapping

Recall from Section 2.3 that CUDA specifies a two-level architecture: an array of simple streaming processors organized in streaming multiprocessors (SMPs). The CUDA programming model provides the concept of a thread block as a unit of execution on an SMP, and the concept of a thread as a unit of execution on a single
streaming processor (SPs). Section 3.6, illustrates mapping of a Data Parallel Process to fine-grain threads within a single CUDA thread block. Using the approach in Section 3.7.2 to tile the node domain, we obtain the coarse-grain parallelism which enables the generation of a CUDA grid of blocks, and scaling the DPP to multiple GPU SMPs.

**Grid Specification**

The amount of coarse-grain data parallelism that directly corresponds to the number of thread blocks mapped onto SMPs, is specified as a parameter of the CUDA grid using the \( <<< \text{GridDim}, \text{BlockDim} >>> \) notation, where the \text{GridDim} variable corresponds to the number of thread blocks, and the \text{BlockDim} variable corresponds to the number of threads within a thread block. In the ideal case, all tiles are independent, and can be processed by different thread blocks during a single kernel execution. In the general case, there can be dependences between tiles, as in the \textit{predictor} example depicted in Figure 3.11(b). The inter-tile dependences determine the precedence order of the tiles. We satisfy the precedence order by processing the tiles according to the tile schedule and allocation function. At each time step \( t_1 \), we launch a kernel which processes up to \( W_2 \) independent tiles in parallel. In the example above, the maximal width \( W_2 = \text{width}(p_1) = 2 \), since only the tiles \( C \) and \( D \) can be processed in parallel. The domain in Figure 3.11(a) is only given for the illustration. In practice, domains offloaded to the GPU for acceleration are much larger. As a consequence the starting domain can be partitioned into larger number of tiles that can be executed in parallel. For example, a \( 8000 \times 8000 \) \textit{predictor} could be processed with 32 thread blocks of 256 threads, resulting in much higher utilization of a Tesla C2050 GPU than with a single thread block. The 32 thread blocks are distributed to 14 GPU SMPs by the GPU block scheduler. The parallelism on the tile level is specified through the grid size \text{ND}_2\_\text{GridDim} parameter, which corresponds to the number of CUDA thread blocks. The scalable processing of the \textit{predictor} example on GPU is realized with the following CUDA host code:

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

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

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

Parametrized SPMD Code Generation

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

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

Mapping Two-Level Thread Hierarchy  Let us illustrate the mapping of the CUDA thread hierarchy on PND with the predictor example. Let us consider the third kernel launch that processes tiles C and D on the GPU. The third kernel call is executed by two CUDA thread blocks with unique identifiers blockIdx.x = 0 and blockIdx.x = 1. Each of the thread blocks contains 4 CUDA threads with thread identifiers threadIdx.x = 0 to 4. The first thread block blockIdx.x = 0 processes tile C with tile index vector (it, jt) = (2, 0). The threads of the first block map to processing entities p_0 = 1..4 in space-time coordinates, and execute time steps t_0 = 8..11 of the DPP’s PND. The second thread block blockIdx.x = 1 processes tile D with tile index vector (it, jt) = (1, 1). The threads of the second block map to processing entities p_0 = 5..8 in space-time coordinates, and execute time steps t_0 = 8..11 of the DPP’s PND. There is a one-to-one mapping between the data parallel operations in the transformed tile domain and thread blocks, i.e. p_1 = blockIdx.x. This enables us to calculate which tiles in the original PND are processed by which thread block.
at each time step $t_1$ by using the inverse space-time mapping to calculate the vectors. Using the schedule $t_1$ and the allocation $p_1$ for the predictor example, we obtain the tile index components $it = t_1 - p_1$ and $jt = p_1$. As a result, for each thread block launched at some time step $t_1$, we can determine which tile $(it, jt)$ is processed as $(t_1 - blockIdx.x, blockIdx.x)$.

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

$$p_0 = 4 \cdot blockIdx.x + threadIdx.x.$$  

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

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

$$4 \cdot blockIdx.x \leq p_0 \leq 4 \cdot blockIdx.x + 3,$$

$$4 \cdot (t - 1 - blockIdx.x) \leq t_0 \leq 4 \cdot (t_1 - blockIdx.x) + 3,$$

$$0 \leq (t_1 - blockIdx.x) \leq 3,$$

$$0 \leq blockIdx.x \leq 1,$$

$$(t_1 - blockIdx.x) - 2 \leq blockIdx.x,$$

$$blockIdx.x \leq (t_1 - blockIdx.x).$$

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

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

3.8.1 Memory Optimizations

DPV Channel Merging

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

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

![Figure 3.12: DPV: Channel merging optimization.](image)
3.8. EXTENSIONS AND OPTIMIZATIONS

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

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

**Buffer Size Optimization**

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

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

Let us analyze how the transformer node in the predictor example executes. The execution of the transformer node is depicted in Figure 3.13. The transformer node $P'_2$ writes up to 4 results in parallel to the channels $C_{34}$ and $C_5$, and reads up to 4 tokens per input argument in parallel from channels $C_{12}$ and $C_{34}$. The guard conditions constructed from IPDs determine for each input argument which channel needs to be accessed. At time step $t = 0$, only a single value is read from the incoming channel $C_{12}$. At time step $t = 1$, one value is read from the incoming channel $C_{12}$, and one value from the self-link $C_{34}$. This value has been produced in the previous data parallel iteration of process $P'2_2$. Each of the values produced by the process $P'_2$

---

66Lifetime analysis is a compiler techniques that determines how long a value is actually used and thus how long it needs to be kept alive.
is used as an input argument only in the first subsequent time step. After the read phase of the next step is complete, the value is not needed any more, and it is safe to overwrite the memory location. This observation can be used to reduce the size of the memory buffer for $C_{34}$. Instead of reserving $D \times W$ locations for the channel, only $W$ locations are necessary.

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

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

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

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

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

Channel Mapping Optimizations

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

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

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

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

The inter-node channels must be mapped to the global memory space, since it is the only memory that is guaranteed to preserve the data between kernel invocations, and thus executions of different DPPs. However, we can leverage the rich GPU memory hierarchy to improve the mapping of self-links. In most cases, self-links can be mapped onto the shared memory of the GPU (i.e. fast on-chip scratchpad). The scratchpad can be accessed by multiple threads in parallel, and it is also frequently used for inter-thread communication. However, the on-chip shared memory is a limited resource. The size of the shared memory available on each streaming multiprocessor is typically several orders of magnitude smaller than the size of the device’s global memory. Thus, an important consideration is whether the buffer allocated for the channel can fit into the shared memory of the GPU. Before channel mapping optimizations, the buffer sizes should be optimized, e.g. using techniques described in Section 3.8.1. The results of the channel mapping optimizations are given for the predictor example in Listing B.3. The lines 10-14 of Listing B.3 illustrate mapping of self-links \( DPC_1 \), \( DPC_3 \), and \( DPC_{13} \) into shared memory array, which is defined at line 20 of Listing B.3.

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

3.8.2 Task Parallelism

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

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

![Diagram of PPN under DPV](Figure 3.14)

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

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

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

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

---

6Feedback cycles are undefined on GPU and cause a GPU to deadlock.
a DPP as a data-driven GPU task containing synchronization and kernel launch capabilities. We designate an independent CUDA stream to each DPP in the process network. Given that the CUDA programming model allows GPU operations in different streams to execute concurrently, this makes it possible for two independent DPPs $P'_2$ and $P'_3$ in Figure 3.14 (Case B) to run in task-parallel manner on a single GPU. We realize the inter-task dependences (Case A) with an event-based synchronization mechanism.

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

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

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

![Figure 3.15: An event-based protocol for data-driven execution on the GPU.](image)

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

\( P'_2 \) and \( P'_3 \) that the output of process \( P'_1 \) is ready to be consumed. Tasks implementing processes \( P'_2 \) and \( P'_3 \) transition into the EXECUTE state. Each process then launches its kernel. The two kernels are free to execute in parallel on the GPU. Upon completion of each kernel, the GPU task issues the kernel completion event. A GPU task may block on one or more data events. Links between DPP processes denote dataflow dependencies. The event sensitivity list of each GPU task is derived from the list of incoming channels. A GPU task can be blocked waiting on one or more events. An example of a GPU task with multiple events in the sensitivity list is Task4. Task4 must wait on event \( e_2 \) and \( e_3 \) before it is allowed to launch kernel \( P'_{4} \). When all events in the sensitivity list of a process are set, a kernel can start its execution. The protocol according to which Task4 executes is given in Figure 3.15(b).

As a result, a GPU can leverage not only data parallelism within DPPs, but also task parallelism between independent DPPs, such as \( P'_2 \) and \( P'_3 \), as illustrated in Figure 3.16. We did some experiments with this technique on the sobel example, but were not able to achieve significant benefits. This is primarily due to the short processing time of the sobel tasks and already sufficient amount of data parallelism in each task. To benefit from the task-parallelism on the GPU, the tasks must have sufficiently long running time and be sufficiently intensive to amortize overheads incurred by kernel launches, the use of CUDA streams, and event-based synchronization. Further research is needed to determine the threshold and application characteristics for efficient task parallel execution.

3.8.3 Token Composition and Reuse

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

Figure 3.16: Task-parallel execution of DPP nodes on a Fermi-architecture GPU.
So far, we have considered only PPN communication, in which the producer writes and the consumer reads tokens of the same data type, e.g., a pixel. However, there are situations possible where the producer writes a token of data type pixel while the consumer reads a collection of these tokens, for example a block of pixels. In this case, we introduce the notion of a composite token to designate the collection of the tokens. The concept of the composite token is similar to the multirate concept found in dataflow formalisms, where a process (actor) can produce and consume multiple tokens from a stream on a single firing [83, 104]. A more extensive and formal discussion on the concept of the composite token will be given in Section 4.3.2.

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

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

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

```c
mainDCT(&blockIn, &blockOut);
for (i=0; i<8; i++)
    for (j=0; j<8; j++)
        shift(&blockIn[i][j]);
for (i=0; i<8; i++)
    for (j=0; j<8; j++)
        tmp[i][j] = dotProduct1(blockIn, c);
for (i=0; i<8; i++)
    for (j=0; j<8; j++)
        bound(blockOut[i][j]);
```

72
To see how token composition and multiplicity help to parallelize the DCT block in the M-JPEG code, let us have a look at the pseudocode for this calculation (the function mainDCT) shown in Figure 3.17 (See Appendix C for more a complete description of the MJPEG application). The pseudo code shows the function DCT that obtains an $8 \times 8$ block $\text{blockIn}$ and which produces an $8 \times 8$ $\text{blockOut}$. On the incoming block of data the DCT function is performed in a sequence of computations: a normalization of values (shift), two integer arithmetic passes: (DCT1) and (DCT2), and bounding of values (bound).

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

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

### 3.9 Results

We implemented the three-phase approach presented in this Chapter as an extension to the Compaan compiler, called KPN2GPU. The KPN2GPU takes as input a PPN, transforms it into an intermediate model with data parallelism, and generates CUDA code for GPUs. As a result, we present the execution time for kernels generated by KPN2GPU from the PPNs of the two running examples: the predictor example and the grid example. For reference, we also add the execution time of an absolutely parallel code captured within the parallel2d example. The main computational ker-
Figure 3.18: GPU Accelerated DCT Computation: Composite Tokens and Multiplicity.

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

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

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

<table>
<thead>
<tr>
<th>Case Name</th>
<th>Target Domain(s)</th>
<th>(Space,Time) Dim</th>
<th>W</th>
<th>D</th>
</tr>
</thead>
<tbody>
<tr>
<td>Predictor</td>
<td>1xPND (1D, 1D)</td>
<td></td>
<td>N</td>
<td>$M + N - 1$</td>
</tr>
<tr>
<td>Grid</td>
<td>2xPND (merged)</td>
<td>(2*1D, 1D)</td>
<td>$M + N - 1$</td>
<td>$\text{max}(N, M - 1)$</td>
</tr>
<tr>
<td>Parallel2D</td>
<td>1xAPND (2D, t=0)</td>
<td>(2D, t=0)</td>
<td>$M \times N$</td>
<td>1</td>
</tr>
</tbody>
</table>

Table 3.1: Data Parallel View on The Transformer Node

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

![Figure 3.19](image)

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

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

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

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

The measurements in Figure 3.19 are the result of a default PPN mapping onto the CUDA architecture. However, the channel classification in DPV can be used to better exploit features of the rich CUDA memory hierarchy. CUDA provides not only global memory, but also low latency on-chip shared memory (SM), and on-chip register file. Figure 3.20 shows performance improvements achieved by channel mapping optimizations introduced in Section 3.8.1. As part of the predictor example optimizations, we mapped intra-node channels to the shared-memory of the GPU as proposed in Section 3.8.1. However, this approach (illustrated by pred-kpngpu-sm) does not scale, since the buffer sizes quickly exceed the rather small shared memory size (16 KB). Moreover, the large shared memory requirements limit the number of thread blocks processed in parallel. We alleviate both issues by combining the optimized channel mapping with lifetime analysis presented in Section 3.8.1 to re-
duce buffer sizes. As illustrated in Figure 3.20(a), combining these two optimizations results not only in performance improvements, but also in much better scalability. Although `pred-kpngpu-sm-optsize` requires introduction of an additional barrier to ensure that all threads have finished loading shared memory data before another round of writing starts, it results in a speedup of 3x compared to the baseline version. In the grid example the intra-node channels can be implemented as registers, resulting in considerable performance improvements shown in Figure 3.20(b).

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

### 3.10 Conclusions

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