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 2

Preliminaries

2.1 Compiler Techniques for Automatic Parallelization

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

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

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

- for-loop bounds are affine expressions of the enclosing loop iterators, static program parameters, and constants,
- for-loop iterators have a constant step size,
2.1. COMPILER TECHNIQUES FOR AUTOMATIC PARALLELIZATION

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

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

```c
#define M 4
#define N 3
int i, j;

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

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

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

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

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

The static program parameters \( M \) and \( N \) in the program above are defined at lines 1-2. Static program parameters can be also used in expressions that define for-loop bounds and array indices. For example, the upper bound of the for-loop with iterator \( i \) at line 5 is given by the program parameter \( M \) and the lower bound is given by the constant 1. Thus, the domain of for-loop iterator \( i \) can be described by linear inequality \( 1 \leq i \leq M \). At lines 5-6, there is a doubly-nested for-loop surrounding the program statement \( S \) at line 7. A program statement can take different forms; a program statement could be for example an assign statement, if condition, or a function call. The statement \( S \) at line 7 corresponds to the function call \( f \) with input...
argument \( \text{in}[i][j] \) and output argument \( \text{tmp}[i][j] \), and the statement \( T \) at line 8 corresponds to the function call \( g \) with input argument \( \text{tmp}[i][j] \) and output argument \( \text{out}[i][j] \). The indexes of function arguments \( \text{in}, \text{tmp}, \) and \( \text{out} \) are affine forms of loop indices \( i \) and \( j \). Data flow between statements \( S \) and \( T \) is made explicit through array variable \( \text{tmp} \). Below, we define several concepts required for program modeling and compile-time analysis:

**Definition 2 (Iteration vector)**

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

**Definition 3 (Program parameters vector)**

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

**Definition 4 (Iteration domain)**

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

In the polyhedral model, the iteration domain of a statement \( S \) is represented by a polyhedron \( D_S \).

**Definition 5 (Polyhedron)**

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

\[
P = \{ x \in \mathbb{Q}^d \mid Ax \geq b \}
\]  

(2.1)

where \( A \) is an integral \( m \times d \) matrix and \( b \) is an integral vector of size \( m \).

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

In this thesis, we use both the terms polyhedron and polytope interchangeably to refer to bounded, parametric \( \mathbb{Z} \)-polyhedra. Note that polyhedra can be parametrized in program parameters (Definition 2 in [134]).
Let us illustrate the notation introduced above on the simple SANLP code snippet given in Listing 2.1. Figure 2.1 shows the polytope $D_S$ associated with statement $S$ nested within for-loops at lines 5-7.

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

\begin{align*}
  i &\geq 0 \\
  i &\leq M - 1 \\
  j &\geq i \\
  j &\leq N - 1.
\end{align*}

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

\[
\begin{bmatrix}
  1 & 0 & 0 & 0 & 0 \\
  -1 & 0 & 1 & 0 & -1 \\
  0 & 1 & 0 & 0 & 0 \\
  0 & -1 & 0 & 1 & -1
\end{bmatrix}
\begin{bmatrix}
  i \\
  j \\
  M \\
  N \\
  1
\end{bmatrix} \geq 0
\]
2.1. COMPILER TECHNIQUES FOR AUTOMATIC PARALLELIZATION

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

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

Below, we introduce concepts related to SANLP execution:

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

![Listing 2.2](image)

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

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

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

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

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

$$
\mathbf{a} < \mathbf{b} \equiv \bigvee_{i=1}^{n} ( a(i) < b(i) \land \bigwedge_{j=1}^{i-1} a(j) = b(j) )
$$

(2.7)
\end{definition}
2.1. **COMPILER TECHNIQUES FOR AUTOMATIC PARALLELIZATION**

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

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

\[
(S, \vec{x}_S) \lt_{seq} (T, \vec{x}_T) \Leftrightarrow (\vec{x}_S \lt_{lex} \vec{x}_T) \lor (\vec{x}_S = \vec{x}_T \land S \lt_{text} T)
\]

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

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

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

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

is **loop-independent** if it occurs for a given iteration of all loops that surround both statements \( S \) and \( T \), i.e. \( \vec{x}_S = \vec{x}_T \), where \( \vec{x}_S \) and \( \vec{x}_T \) are the vector components up to the common nesting level. Otherwise, a dependence is **loop-carried**, which means that the dependence occurs between operations with different values of the loop counter, i.e. \( \vec{x}_S \not< \vec{x}_T \).

**Definition 8 (Distance vector)**

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

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

We classify data dependences in four types. The first three types of dependences, i.e. **dataflow dependences**, **output dependences**, and **anti-dependences** are dependences
2.1. Compiler Techniques for Automatic Parallelization

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

- **Dataflow dependence**: Read-after-write (RAW). If a write operation to a shared variable is followed by a read operation from the same location, the value read depends on the value written. This type of dependence is also known as a true dependence, and it is the main dependence type considered in PPN construction.

- **Output dependence**: Write-after-write (WAW). If two operations write to the same location, the value of memory location will have the wrong value after both operations are performed if the operations are permuted.

- **Anti-dependence**: Write-after-read (WAR). If a read operation from a shared variable is followed by a write operation to the same location, the value read from the memory location will be wrong if the operations are permuted.

- **Read dependence**: Read-after-read (RAR). A read dependence exists if one operation reads the same location as the other.

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

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

In PPN construction, only dataflow dependences are considered, as there is no logical re-use of memory cells for writing. More specifically, PPN dependence edges are annotated with the exact dependence specification, which for each dataflow dependence directed to operation \( T(\vec{j}) \) provides the iteration vector of the last operation that writes to a memory location read by \( T(\vec{j}) \). The exact dependence specification is determined using linear programming techniques for finding lexicographical the maximum [54] as described in [55] and implemented in the pipLib library [59].
addition, read dependences are considered through multiplicity analysis introduced by Turjan [130, 132].

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

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

Definition 9 (Loop nest depth)
The depth of a loop nest $d_L$ is the number of for-loops in the given loop nest $L$.

Definition 10 (Loop nesting level, Loop depth)
The nesting level $l_i$ (loop depth) of loop $i$ in loop nest $L$ is one plus the number of the loops in the surrounding loop nest. The nesting level of the outermost loop thus equals one, i.e.

$$1 \leq l_i \leq d_L$$

Definition 11 (Common nesting level)
We define the common nesting level $n_{ST}$ of two statements $S$ and $T$ as the number of for-loops that surround both $S$ and $T$. Vectors $\vec{x}_S$ and $\vec{x}_T$ are the vectors formed by the first $n_{ST}$ components of iteration vectors $\vec{x}_S$, and $\vec{x}_T$ respectively. If statements $S$ and $T$ do not share any enclosing loops, then $n_{ST} = 0$.

Definition 12 (Level of dependence)
Level of dependence $l(e)$ is the depth (one plus the number of surrounding loops) of the outermost loop for which the loop counters of two dependent operations are different.

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

- **loop-independent dependence**: $l(e) = \text{inf}$ if $S(\vec{x}_S) \Rightarrow T(\vec{x}_T)$ with $\vec{x}_S = \vec{x}_T$.

This case corresponds to a loop-independent dependence, where statements
2.1. COMPILER TECHNIQUES FOR AUTOMATIC PARALLELIZATION

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

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

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

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

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

\[
V = \{S_i(\vec{x}_S_i) \mid \forall i, S_i \in \mathcal{P} \land \vec{x}_S_i \in D_{S_i} \land \bigcup_{S_i} = \mathcal{P}\}
\]

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

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

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

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

Definition 13 (Polyhedral Reduced Dependence Graph (PRDG))

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

- The set of vertices \( V \) of a PRDG contains \( s \) nodes, where \( s \) is the number of statements in program \( P \), i.e. \( |P| = s \). Each node \( N_{S_i} \) in the PRDG is defined by the program statement \( S_i \), dimensionality of the statement \( d_{S_i} \) (which corresponds to the number of surrounding for-loops), its iteration vector, and its iteration domain \( D_{S_i} \in \mathbb{Z}^{d_{S_i}} \).

- For each set \( R_{S,T} \), where \( R_{S,T} \) is a non-empty set of dependence pairs between operations of statements \( S \) and \( T \), there is a (directed) edge \( e_{S,T} \in E \) from node \( S \) to node \( T \) in the graph.

- Each edge \( e_{S,T} \in E \) is annotated with the exact mapping between dependent operations. The mapping between producer (write) and consumer (read) operations is an affine function in iterators, program parameters, and constants.

2.2 Polyhedral Process Networks

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

In a PPN, program statements, dependence edges, and the input and output arguments of the statements are described as polytopes obtained by polyhedral analysis of static affine nested loop programs (SANLPs) [3,134]. Extraction of a SANLP’s polyhedral model enables exact dataflow analysis of scalar and array references, which is the fundamental requirement for derivation of a Polyhedral Process Network (PPN) model [93]. For the details on analytical derivation of a PPN model, interested reader is referred to [130, 131, 135]. The PPN processes are structured in a particular way,
which is described below. The producer-consumer relationships between processes are also described by polyhedra. In this section, we give fundamental PPN definitions and explain its execution model on a simple example.

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

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

1. **Read (R)** - in this phase, a process reads input data from incoming channels into local variables. If input data is not available, the process blocks.

2. **Execute (E)** - in this phase, a process executes the process function (program statement) on the input data in local variables, and produces output data.

3. **Write (W)** - in this phase, a process writes the output data from local variable into outgoing channels.

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

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

**Definition 14 (PPN Process)**

A PPN process is an autonomous execution entity specified by:

1. A **process domain** that specifies all process iterations,

2. A specification of **input port domains** to read all the function input arguments from the corresponding input channels,
2.2. POLYHEDRAL PROCESS NETWORKS

3. A process function to processes the input arguments and to produce output arguments, and

4. A specification of output port domains to write the function output arguments to the corresponding output channels.

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

Definition 15 (Process function)
A process function represents the computational part of a process. It corresponds to a function call statement in the sequential application that is a pure function without side-effects which only reads/writes through its input/output arguments.

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

Definition 16 (Process body)
The body of a process (process body) includes communication and computation parts of the process. The process body is structured in three phases: read phase, execute phase, and write phase. First all input data is read from incoming channels (read phase), the process function is executed (execute phase), and subsequently all output data is written to outgoing channels (write phase).

Definition 17 (Process iteration)
A process iteration of a process $P$ is defined as a (single) execution of the process body, i.e. the evaluation of the process function on a single set of input and output arguments.

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

Definition 18 (Process iteration domain (Node domain))
The process iteration domain (also known as node domain) of a process $P$, denoted by $D_P$, is defined as a set of all process iterations of process $P$.

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

The PPN processes communicate via single-producer single-consumer channels, typically implemented as FIFO buffers. The PPN model imposes blocking read of input data from the channel, and blocking write of output data to the channel. This means that the process stays in blocked state in the read phase, as long as there is no input data on the incoming channels. As soon as the input arguments of the function...
appear on the input ports of the process, the process executes the process function. Then it needs to write the output data from a local variable into the outgoing PPN channel connected to the relevant consumer node. The write can also block if there is no space in the PPN channel.

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

**Definition 19 (Input Port Domain (IPD))**

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

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

**Definition 20 (Output Port Domain (OPD))**

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

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

**Definition 21 (Mapping)**

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

\[
OPD_{P_P,i} = M^k(IPD_{P_C,k}).
\]

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

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

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

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

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

A multiprocessor is designed to execute hundreds of CUDA threads concurrently. To manage such a large amount of threads, it employs the single-instruction, multiple-
2.3. PARALLEL COMPUTING WITH GPU ACCELERATORS

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

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

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

Programming Model The Compute Unified Device Architecture (CUDA) is a programming model and software architecture for the GPU accelerators that allows
2.3. PARALLEL COMPUTING WITH GPU ACCELERATORS

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

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

The key programming abstractions that are introduced for mapping onto GPU hardware are the following: (CUDA) threads, shared memories, and barrier synchronization. The abstractions are simply exposed to the programmer as a set of C language extensions. CUDA threads executing a kernel are structured into grid of thread blocks, which is shown in Figure 2.3(b). The GPU executes a grid of thread blocks, as indicated in Figure 2.3(a). An SMP executes one or more thread blocks from the grid, as indicated in Figure 2.3(a). Each thread block (TB) (or simply a block) consists of multiple CUDA threads. CUDA threads are executed by SPs on the SMP. Threads are organized in one, two or three-dimensional blocks. Threads have private
memory, which is implemented as (partitioned) register file. Threads within a thread block can access a shared partition of on-chip memory, which is in CUDA terminology called shared memory. All threads can access shared global memory, which is realized as device memory (DRAM). The thread blocks in a grid express coarse-grain data parallelism. Thread blocks must be independent. The concurrent threads in a thread block express fine-grain data parallelism. Threads within a block are not necessarily independent. Independent grids express coarse-grain task parallelism. As a reference, the mapping between CUDA and OpenCL concepts is given in Table 2.3.

<table>
<thead>
<tr>
<th>CUDA Concepts</th>
<th>OpenCL Concepts</th>
</tr>
</thead>
<tbody>
<tr>
<td>Thread (Thread) Block Grid</td>
<td>Work-Item Work-Group NDRange</td>
</tr>
<tr>
<td>global memory shared memory (scratchpad) local memory registers</td>
<td>global memory local memory (scratchpad) - private memory</td>
</tr>
</tbody>
</table>

Table 2.1: CUDA and OpenCL programming models

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

**Execution Model** The GPU architecture implements hardware management and scheduling of threads and thread blocks. GPUs have an efficient mechanism for fine-grain hardware multi-threading. The GPU scheduler time-slices execution of threads on an SMP. The scheduling unit is a warp of 32 threads. The execution context (program counters, registers, etc) for each warp processed on an SMP is maintained on-chip during the entire lifetime of the warp. The multiprocessors tracks which warps are ready to execute (i.e. which warps have all operands available for the next instruction) using a mechanism called *scoreboarding*. The warps executed on a single SMP do not have to belong to the same thread block, but they can belong to any thread block scheduled for execution on that particular multiprocessors. The advantage of this approach is that after issuing a long latency instruction, such as memory
load or store, the GPU does not need to wait idle until that particular instruction is completed. Instead, at every instruction issue time, a warp scheduler selects a warp that has threads ready to execute its next instruction and issues the instruction to those threads. Having a large number of independent instructions in flight is required to hide latencies on the GPU. The common way to achieve this is by finding fine-grain data parallelism in the problem and exposing it to the GPU as large number of light-weight CUDA threads.

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