Data-Aware Process Networks

Christophe Alias, Alexandru Plesco

Data-Aware Process Networks

Christophe Alias
Laboratoire de l'Informatique du Parallélisme
CNRS, ENS de Lyon, Inria, UCBL, Université de Lyon
Lyon, France
Christophe.Alias@inria.fr

Alexandru Plesco
CEO
XtremLogic SAS
France
Alexandru.Plesco@xtremlogic.com

Abstract

With the emergence of reconfigurable FPGA circuits as a credible alternative to GPUs for HPC acceleration, new compilation paradigms are required to map high-level algorithmic descriptions to a circuit configuration (High-Level Synthesis, HLS). In particular, novel parallelization algorithms and intermediate representations are required. In this paper, we present the data-aware process networks (DPN), a dataflow intermediate representation suitable for HLS in the context of high-performance computing. DPN combines the benefits of a low-level dataflow representation – close to the final circuit – and affine iteration space tiling to explore the parallelization trade-offs (local memory size, communication volume, parallelization degree). We outline our compilation algorithms to map a C program to a DPN (front-end), then to map a DPN to an FPGA configuration (back-end). Finally, we present synthesis results on compute-intensive kernels from the Polybench suite.

CCS Concepts: • **Hardware → High-level and register-transfer level synthesis**; • **Theory of computation → Streaming models**.

Keywords: High-Level Synthesis, Polyhedral Model, FPGA

ACM Reference Format:
Christophe Alias and Alexandru Plesco. 2021. Data-Aware Process Networks. In *Proceedings of the 30th ACM SIGPLAN International Conference on Compiler Construction (CC '21), March 2–3, 2021, Virtual, Republic of Korea.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3446804.3446847

1 Introduction

Since the end of Dennard scaling, transistors can no longer be miniaturized under a constant power density. Hence the rise of power-efficient *hardware accelerators* [19] (Xeon-Phi, GPU, FPGA) which equip most embedded systems and high-performance computers. Recently, reconfigurable FPGA circuits [8] have appeared to be a competitive alternative to GPU [46] in the race for energy efficiency. FPGAs combine both flexibility of programmable chips and energy-efficiency of specialized hardware and appear as a natural solution. However, designing a circuit is far more complex than writing a C program. Disruptive compiler technologies are required to generate automatically a circuit configuration from an algorithmic description [10]. Also, FPGAs come with limited amount of on-chip memories (usually in MBs) that are much smaller than HPC datasets of tens and even hundreds of GBs. Hence, compiler techniques are required to *spill* efficiently the data to an off-chip memory, as would do a register allocation, and to schedule the operations so the *computation hides the communications*. Even though FPGAs with faster HBMs appeared recently, datasets keep growing and are usually much bigger than top of the line 16GB HBM memories. For large datasets slower but much bigger, DDR memories are still mandatory.

In this paper, we outline our experience towards the design of a complete polyhedral-powered approach for high-level synthesis (HLS) of supercomputing kernels to FPGA. We cross fertilize polyhedral process networks (PPN) [42] with our communication synthesis approach [2] to propose a new dataflow intermediate representation which explicits both parallelism and data spilling to the off-chip memory: the *data-aware process networks (DPN)*. We show how DPN and PPN might be view as instances of *regular process networks (RPN)*, a general compilation-oriented dataflow model of computation. RPN induces a *general methodology* of automatic parallelization, that we believe to be appropriate for HLS of supercomputing kernels to FPGA. Specifically, this paper makes the following contributions:

- The Data-aware Process Networks (DPN), a dataflow intermediate representation relevant for high-level circuit synthesis in HPC. DPN explicits the data spilling to an external memory and makes it possible to tune the circuit arithmetic intensity to fit the architecture balance.
- The Regular Process Networks (RPN), a simple dataflow representation for polyhedral programs which captures the partitioning of computations *and communications*. RPN subsumes DPN and PPN and induces a general compilation methodology.

CC '21, March 2–3, 2021, Virtual, Republic of Korea

ACM ISBN 978-1-4503-8325-7/21/03…$15.00
https://doi.org/10.1145/3446804.3446847

- A compilation algorithm to translate a C program to a DPN. In particular, we show how data spilling is generated and how the DPN is parallelized.
- We outline the architectural choices involved in the mapping of a DPN to an FPGA configuration, then we present FPGA synthesis results on linear algebra kernels.

This paper is structured as follows. Section 2 outlines the polyhedral model. Section 4 presents the regular process networks, then the data-aware process networks. Section 5 outlines the algorithms to translate a C program to a DPN. Section 6 outlines the architectural choices involved in the mapping of a DPN to an FPGA configuration. Section 7 presents experimental results. Section 8 presents related work. Finally, Section 9 concludes this paper and draws research perspectives.

2 Preliminaries

This section outlines the concepts of polyhedral compilation used in this paper. In particular, we recall the dynamic single static assignment form, from which the DPN are derived.

2.1 Polyhedral Model

The polyhedral model [14–16, 18, 31, 33] is a general framework to design loop transformations, historically geared towards source-level automatic parallelization [18] and data locality improvement [9]. It abstracts loop iterations as a union of convex polyhedra – hence the name – and data accesses as affine functions. This way, precise – iteration-level – compiler algorithms may be designed (dependence analysis [14], scheduling [16] or loop tiling [9] to quote a few) . The polyhedral model manipulates program fragments consisting of nested `for` loops and conditionals manipulating arrays and scalar variables, such that loop bounds, conditions, and array access functions are *affine expressions* of surrounding loops counters and structure parameters (input sizes, e.g., N). Thus, the control is static and may be analyzed at compile-time. With polyhedral programs, each iteration of a loop nest is uniquely represented by the vector of enclosing loop counters $\vec{i}$. The execution of a program statement S at iteration $\vec{i}$ is denoted by $\langle S, \vec{i}\rangle$. The set $\mathcal{D}_S$ of iteration vectors is called the *iteration domain* of S. Figure 1.(b) depicts the iteration domains $\mathcal{D}_S, \mathcal{D}_T, \mathcal{D}_U$ for our running example, the Jacobi-1D kernel.

2.2 Dependences

Given an operation (*i.e.* an instance of some statement in the program) ω, we write $\mathrm{read}(\omega)$ (resp. $\mathrm{write}(\omega)$) the set of addresses read (resp. written) by ω. There exists a *dependence* from an operation s to an operation t iff $s \prec_{seq} t$, both operations access the same address and one access is a write. When $\mathrm{write}(s) \cap \mathrm{read}(t) \neq \emptyset$ (resp. $\mathrm{read}(s) \cap \mathrm{write}(t) \neq \emptyset$, $\mathrm{write}(s) \cap \mathrm{write}(t) \neq \emptyset$), we have a *flow* dependence and we write: $s \longrightarrow^{\text{FLOW}} t$ (resp. *anti*: $s \longrightarrow^{\text{ANTI}} t$, *output*: $s \longrightarrow^{\text{OUTPUT}} t$). *Direct dependences* are a special case of flow dependences, which relate the production of a value to its consumption. Direct dependences may be specified by a function source_k, mapping an operation $\langle T, \vec{j}\rangle$ to the last operation executed before $\langle T, \vec{j}\rangle$, which writes its k-th read address: $\mathrm{source}_k(\langle T, \vec{j}\rangle) \longrightarrow^{\text{FLOW}} \langle T, \vec{j}\rangle$ for any $\vec{j} \in \mathcal{D}_T$. The source function is always computable on polyhedral programs, the result is a piecewise affine mapping [14]. On our running example, the dependence function for the read 1 ($a[t-1, i-1]$) of assignment T would be:

$$\mathrm{source}_1\langle T, t, i\rangle = \begin{cases} \langle T, t-1, i-1\rangle & \forall (t-1, i-1) \in \mathcal{D}_T \\ \langle S, i-1\rangle & \forall t = 0 \text{ or } i = 0 \end{cases}$$

From dependence functions, a polyhedral program may be turned to *dynamic single assignment form*. It suffices to substitute, for each assignment, the left hand side by $S[\vec{i}]$, where S is a fresh array associated to S and $\vec{i}$ is a vector with enclosing loop counters of S. Then in the right hand side, we substitute each read k by $\mathrm{source}_k(\langle S, \vec{i}\rangle)$ where operations $\langle T, u(\vec{i})\rangle$ are rewritten as array reads $T[u(\vec{i})]$. If we lift the loops and we keep only the assignments, we obtain a *system of affine recurrent equations* (SARE), which is a common intermediate representation in polyhedral compilers:

$$\begin{aligned} &S[i] = \mathit{load}(), \textbf{if } i \in \mathcal{D}_S \\ &T[t,i] = (\textbf{if } (t-1, i-1) \in \mathcal{D}_T : T[t-1, i-1] \textbf{ else } S[i-1]) \\ &\quad +(\textbf{if } (t-1, i) \in \mathcal{D}_T : T[t-1, i] \textbf{ else } S[i]) \\ &\quad +(\textbf{if } (t-1, i+1) \in \mathcal{D}_T : T[t-1, i+1] \textbf{ else } S[i+1]) \\ &U[i] = \textbf{if } i \in \mathcal{D}_U \wedge T \geq 2 : T[T-1, i] \textbf{ else } S[i] \end{aligned}$$

Our main contribution, the data-aware process networks, are derived from this representation.

2.3 Scheduling & Tiling

A *schedule* θ_S assigns each operation $\langle S, \vec{i}\rangle$ with a timestamp $\theta_S(\vec{i}) \in (\mathbb{Z}^d, \ll)$. Intuitively, $\theta_S(\vec{i})$ is the iteration of $\langle S, \vec{i}\rangle$ in the transformed program. A schedule is *correct* if $\langle S, \vec{i}\rangle \rightarrow \langle T, \vec{j}\rangle \Rightarrow \theta_S(\vec{i}) \ll \theta_T(\vec{j})$, the lexicographic order ensuring that the dependence is preserved. The lexicographic ordering over $\mathbb{Z}^d$ is naturally partitioned by depth: $\ll = \ll^1 \uplus \ldots \uplus \ll^d$ where $(u_1 \ldots u_d) \ll^k (v_1, \ldots, v_d)$ iff $\left(\wedge_{i=1}^{k-1} u_i = v_i\right) \wedge u_k < v_k$ for $1 \leq k \leq d$.

Tiling is a reindexing transformation which groups iteration into tiles to be executed atomically. This make possible to tune the grain of computations and to distribute the computations among compute units, while tuning the operational intensity of the program. *Rectangular tiling* reindexes any iteration $\vec{i} \in \mathcal{D}_S$ to an iteration $(\vec{i}_{block}, \vec{i}_{local})$ such that $\vec{i} = \mathcal{T}_S(\vec{i}_{block}, \vec{i}_{local})$, with $\mathcal{T}_S(\vec{i}_{block}, \vec{i}_{local}) = (\mathrm{diag}\,\vec{s})\,\vec{i}_{block} + \vec{i}_{local}$, $0 \leq \vec{i}_{local} < \vec{s}$ where $\vec{s}$ is a vector collecting the tile size across each dimension of the iteration domain. $\vec{i}_{block}$ is called the *outer* tile iterator and $\vec{i}_{local}$ is called the *inner* tile iterator, and we write $\mathcal{B}_S(\vec{i}) = \vec{i}_{block}$. The companion schedule associated to the tiling $\theta_S(\vec{i}_{block}, \vec{i}_{local})$ orders $\vec{i}_{block}$ first

to to ensure the execution tile by tile. The set of tiles obtained by varying the last outer tile iterator is called a *tile band*. To enforce the atomicity (avoid cross dependences between two tiles), it is sometimes desirable to precede the tiling by an injective affine mapping ϕ_S. The coordinates of $\phi_S(\vec{i})$, for $\vec{i} \in \mathcal{D}_S$ are usually called *tiling hyperplanes*. In that case, the transformation $\mathcal{T}_S^{-1} \circ \phi_S$ for some statements S is called an *affine tiling*. Note that rectangular tiling is a particular case of affine tiling where ϕ_S is the identity mapping. Figure 2.(a) depicts a possible tiling on our running example, with $\phi_T(t,i) = (t, t+i)$. In [2], we proposed an optimal communication scheme for HLS based on tiling. For each tile $\vec{i}_{block}$, we derived the minimal data set to load (Load($\vec{i}_{block}$)) and to store (Store($\vec{i}_{block}$)). Then, we executed loads, computations and stores in a pipelined fashion. Data-aware process networks will inherit from this apparatus.

3 Regular Process Networks

Since the early days of high-level synthesis, *partitioning techniques* were designed to distribute the computations across parallel units, while exploring the parallelism tradeoffs [26, 35–37]. On the other hand, dataflow representations are natural candidates to represent the parallelism of an application. For polyhedral programs, the Compaan project propose to represent programs with *polyhedral process networks* (PPN) [25, 34, 39, 42], a dataflow model of computation derived from SARE, with the Kahn process networks semantics [21].

This section presents the *regular process networks* (RPN) (Section 3.1), a simple dataflow representation for polyhedral programs inspired by PPN, which captures the partitioning of computations *and communications*. Although there is nothing fundamentally new behind RPN (a RPN is nothing more than a PPN with a generic partitioning strategy), the cross fertilization between the concept of partitioning and dataflow representation gives us a general HLS design methodology (Section 3.2), which drives the contributions presented in this paper.

3.1 Regular Process Networks

Given a polyhedral program, we build a *regular process network* (RPN) with the following operations:

- We *partition the computation* (iteration domains) into *processes*: if $\mathcal{D}$ denotes the union of the iteration domains of the program, any partition $\mathcal{D} = \mathcal{D}_1 \uplus \ldots \uplus \mathcal{D}_n$ defines a set of processes $\{P_1, \ldots, P_n\}$, each P_i iterating over $\mathcal{D}_i$.
- We provide a *schedule* θ_i for each process P_i.
- Then, we *partition the direct dependences* into *channels*: if $\rightarrow$ denotes the direct dependence relation, any partition $\rightarrow = \rightarrow_1 \uplus \ldots \uplus \rightarrow_m$ defines a set of channels $\{1, \ldots, m\}$ such that the data transferred by $\rightarrow_c$ is conveyed through channel c. We say that $\rightarrow_c$ is *resolved* by channel c.
  With this definition, the channel implementation is completely abstracted. It could be a FIFO, a global/local memory, a link between communicating FPGA, and so on.

Formally:

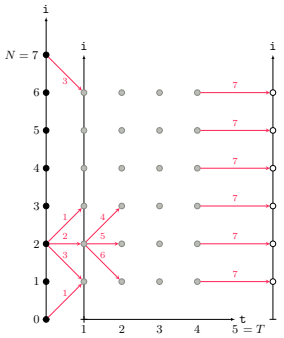
Definition 3.1 (Regular process network (RPN))**.** Given a program, a *regular process network* is a couple $(\mathcal{P}, C, \theta, \omega)$ such that:

- Each process $P \in \mathcal{P}$ is specified by an iteration domain $\mathcal{D}_P$ and a sequential schedule θ_P inducing an execution order $\prec_P$ over $\mathcal{D}_P$. Each iteration $\vec{i} \in \mathcal{D}_P$ realizes the execution instance $\omega_P(\vec{i})$ in the program.
  The processes partition the execution instances in the program: $\{\omega_P(\mathcal{D}_P)$ for each process P of $\mathcal{P}\}$ is a partition of the program computation.
- Each channel $c \in C$ is specified by a producer process $P_c \in \mathcal{P}$, a consumer process $C_c \in \mathcal{P}$ and a dataflow relation $\rightarrow_c$ relating each production of a value by P_c to its consumption by C_c: if $\vec{i} \rightarrow_c \vec{j}$, then execution $\vec{i}$ of P_c produces a value read by execution $\vec{j}$ of C_c. $\rightarrow_c$ is a subset of the direct dependences from the iterations of P_c in the program $(\omega_{P_c}(\mathcal{D}_{P_c}))$ to the iterations of C_c in the program $(\omega_{C_c}(\mathcal{D}_{C_c}))$ and the set of $\rightarrow_c$ for each channel c between two given processes P and C, $\{\rightarrow_c,\ (P_c, C_c) = (P, C)\}$, is a partition of direct dependences from P to C.

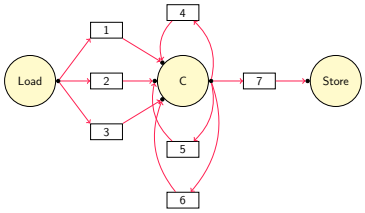
Example. Figure 1.(a) depicts a polyhedral kernel and (b) provides the iteration domains for each assignment (• for assignment *load*, • for assignment *compute* and ∘ for assignment *store*). On Figure 1.(c), each execution $\langle S, \vec{i} \rangle$ is mapped to process P_S: load iterations (•) are mapped to process Load, compute iterations (•) are mapped to process C, and store iterations (∘) are mapped to process Store. With this straightforward partitioning, the instances mapped to a process come from the same statement, hence the input ports are the reads of the statement. The input/output ports are depicted with black points. Dependences labeled by k on the dependence graph in (b) are solved by the channel k. With this dependence partitioning, we have a different channel per couple producer/read reference. This way, the input values can be read in parallel. We assume that, *locally*, each process executes instructions in the same order than in the original program: $\theta_{\text{load}}(i) = i$, $\theta_{\text{compute}}(t,i) = (t,i)$ and $\theta_{\text{store}}(i) = i$. Remark that the leading constant (0 for *load*, 1 for *compute*, 2 for *store*) has disappeared: the timestamps only define an order local to their process: $\prec_{load}$, $\prec_{compute}$ and $\prec_{store}$. The global execution order is driven by the dataflow semantics: the next process operation is executed as soon as its operands are available. □

```
     for i := 0 to N + 1
S(•) load(a[0, i]);
     for t := 1 to T
       for i := 1 to N
T(•)   a[t, i] := a[t − 1, i − 1] + a[t − 1, i]+
                  a[t − 1, i + 1];
     for i := 1 to N
U(∘) store(a[T, i]);
```

(a) Jacobi 1D kernel



(b) Flow dependences



(c) Regular process network

Figure 1. Motivating example: Jacobi-1D kernel. (a) depicts a polyhedral kernel, (b) gives the polyhedral representation of loop iterations (•: load, •: compute, ∘: store) and flow dependences (red arrows), then (c) gives a possible implementation as a regular process network: each assignment (load, compute, store) is mapped to a different process and flow dependences (1 to 7) are solved through channels.

Execution Semantics. The execution of a RPN is *locally sequential* (each process executes its operations sequentially) and *globally dataflow* (for each operation, the process executes once the input data are available). Hence, the execution order is constrained by the producer/consumer dependences between processes $\rightarrow$ for the global dataflow part; and the local sequential execution for each process P_i, $\prec_{\theta_i}$ for the local sequential part:

$$\prec_{RPN} \quad \supseteq \quad \rightarrow \cup \left(\bigcup_i \prec_{\theta_i} \right)$$

Partitioning Strategy. The *partitioning strategy* presented on the example (processes with instances of the same statement, statement instances possibly split across several processes, one channel per couple producer/read) defines the class of *polyhedral process networks* (PPN) [25, 34, 39, 42] developed in the context of the Compaan project. In this paper, *we build on the PPN partitioning strategy* to define an RPN partitioning strategy (referred to as DPN, for data-aware process networks), which explicit the data transfers between the RPN and a remote memory.

Correctness. We may wonder how to constrain the partitioning strategy and the process scheduling to obtain a correct RPN *i.e.* whose execution terminate and produces the same output value than the program given the same inputs. Actually, this is pretty open: for any partition, there exists a process schedule leading to a correct execution. This schedule is simply the original program schedule, or any valid sequential program schedule. We point out that the partitioning does *not* have to split the iterations into atomic processes, it is not necessarily a tiling. At worst, mutual synchronizations may hold between two processes.

Under scheduling constraints, the channels can always be sized to ensure a correct execution, and avoid deadlocks: at worse, the channels can be kept single-assignment. We point out that depending on the partitioning of the channels, the synchronizations may be hard to enforce. This optimization criterion will be discussed later.

3.2 Compilation Methodology

Regular process networks are a very general family of intermediate representations, that we believe to be appropriate and effective for automatic parallelization and specifically high-level synthesis of polyhedral kernels. RPN enforces the following compilation methodology, that drives the contributions discussed in this paper.

Partitioning Strategy. At compiler design time, *choose a partitioning strategy*. This is clearly driven by the features of the target architecture. In the specific context of HLS of high-performance kernels which manipulate a huge volume of data with an important data reuse, we propose the DPN (data-aware process network) partitioning strategy, described in the next section.

Front-End: Derive a RPN from a Program.

1. Apply the computation partitioning and *find a schedule* θ_i for each process P_i. Many criteria can drive the derivation of θ_i: throughput, parallelism, channel size to quote a few.
2. *Compile the process control* (multiplexers for input ports, demultiplexers for output ports, control loop nest or FSM).
3. *Compile the channels* (type of channel, size/allocation function, synchronizations).

In Step 1, scheduling a RPN to maximize the throughput is still an open problem. Though, we propose a partial solution for a *single process*, able reduce bubbles in the arithmetic datapath [3, 4]. Steps 2 and 3 have been partially addressed

in the context of PPN [39]. We show how to extend these algorithms in the context of DPN.

Back-End: Map the RPN to the Target Architecture. In addition to target-specific passes, the back-end should feature simplification passes of the RPN representation. With the PPN partitioning strategy, it is possible to factor the channels [43] and the processes which share a lot of common control. We also proposed a back-end algorithm to compact affine control [6]. Section 6 will outline our back-end.

4 Data-Aware Process Networks

PPN were developed in the context of HLS of data-centric embedded applications [12]. These applications manipulate streams of data and/or reasonably small matrices, while keeping a small number of data alive at the same time [38]. This way, *the total size of the channels stays reasonable*. When it comes to compute-intensive applications for high-performance computing, things are totally different. Indeed, the computation volume and the data reuse are such that *the total channel size would literally explodes*. With PPN, a matrix multiplication $A \times B$ would require to store the whole matrices A and B in channels, which is clearly not possible with FPGA, as the local memory size is at most a few tens of megabytes.

Hence, *spilling strategies* must be found, just as for register allocation. As shown in [2], partitioning strategies must be designed to split the application into *reuse units* (here tile bands) where *into a reuse unit, the data flows through local memory* and *between two reuse units, the data flows through a remote memory*. This general principle inspires the DPN partitioning.

Making Data Transfers Explicit. Given a loop tiling, we consider the execution per tile band along the lines developed in [2]: a tile band acts as a *reuse unit*. Dependences flowing in and out of the tile band are resolved through a remote memory: *incoming dependences are loaded* from the remote memory, *outgoing dependences are stored* to the remote memory. Dependences into the tile band (source and target in the same tile band) are resolved by a channel (in the local memory). Figure 2 gives a possible DPN partition (b) from a loop tiling (a) on the motivating example. The loop tiling is defined by $\phi_T(t, i) = (t, t + i)$. Hence the two tiling hyperplanes have the normal vectors $\vec{\tau}_1 = (1, 0)$ ("t") and $\vec{\tau}_2 = (1, 1)$ ("$t + i$"). By definition, the iteration domain is sliced into tile bands by all the hyperplanes except the last one, hence we obtain vertical bands). (b) depicts the tile band with $4 \leq t \leq 7$. Since tile band acts as a *reuse unit*, incoming dependences (here 1,2,3) are loaded and outgoing dependences (here 13,14,15) are stored to an external storage unit. Dependences inside a band are resolved through channels (here 4 to 12). In other words, with the DPN partitioning strategy, the *reuse units* are made explicit and the ratio computation/data transfers can be tuned, simply by playing on tile size parameters.

Adding Parallelism. Inside a band, the computations may be split in parallel process with the enclosing tiling hyperplanes (all the tiling hyperplanes except the last one). On Figure 2.(a), each band is split in $p = 2$ sub-bands by the hyperplane $\vec{\tau}_1 = (1, 0)^T$, separated by a dotted line. Each sub-band is mapped to separate process on Figure 2.(b): C_1 for the left sub-band, C_2 for the right sub-band. With this partitioning strategy, we can *tune the arithmetic intensity* ($A.I.$) by playing on the band width b (here $A.I. = 2 \times bN/2N = b$) and *select the parallelism independently*. Each parallel process is identified by its outer tile counters $\vec{\ell}$ (all, except the last one). The parallel instance of C of coordinate $\vec{\ell}$ is written $C_{\vec{\ell}}$ (here we have parallel instances C_0 and C_1).

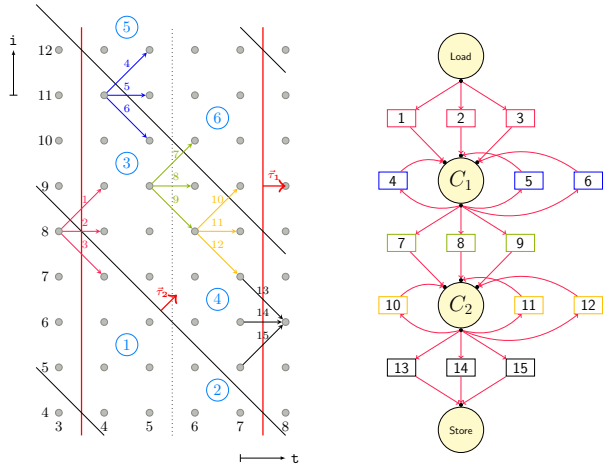
Dependence Partitioning. We distinguish between *I/O dependences*, ($\rightarrow_{I/O}$ source or target outside of the band *e.g.* $1, 2, 3$ or $13, 14, 15$), *local dependences* to each parallel process ($\rightarrow_{local}$, source and target on the same parallel process *e.g.* $4, 5, 6$) and *synchronization dependences* between parallel process ($\rightarrow_{synchro}$, source and target on different parallel process *e.g.* $7, 8, 9$). Again, *I/O dependences* are solved through *remote memory* whereas *local dependences* and *synchronization dependences* are solved through *local memory*.

Connection with the Canonic PPN Partitioning. Finally, the dependences are partitioned in such a way that each new channel c' connects a single producer to a single consumer (here original PPN channel 4 is split into DPN channels $4, 7, 10$): this is the property inherited from PPN. The DPN partitioning will be denoted by $(\mathcal{L}, \mathcal{P}', \mathcal{S}, C')$ where $\mathcal{L}$ is the set of Load process, $\mathcal{P}'$ is the set of parallel processes, $\mathcal{S}$ is the set of store processes Store and C' is the set of channels.

5 Front-End

This section outlines the main compilation steps to translate a C program to a DPN.

Data Multiplexing/Demultiplexing. For each iteration, a process retrieves the data from its input ports, computes the result, and writes the result to the output buffer. For instance process C_1 on figure 2.(b) would execute iteration $(t = 4, i = 9)$ by (i) getting the data from load buffers $1, 2, 3$, compute the sum, and then write the data to buffers $4, 5, 6$. Also, iteration $(t = 5, i = 9)$ would get the data from buffers $4, 5, 6$, compute the sum and write the result to buffers $7, 8, 9$ so they can be read by process C_2. In turn, the execution of process C_2, at iteration $(t = 6, i = 9)$ would read the data from buffers $7, 8, 9$, compute the sum and write the result to buffers $10, 11, 12$. Finally, iteration $(t = 7, i = 9)$ would read



(a) DPN partitioning (b) DPN

Figure 2. Data-aware process network (DPN) for the Jacobi-1D kernel. (a) computations are executed per band (between red thick lines), tile per tile. Incoming dependences (1,2,3) are loaded, outgoing dependences (13,14,15) are stored, internal dependences (4 to 12) are solved through local channels depicted in (b). Parallelization is derived by splitting a band with outer tiling hyperplanes (here the dotted line). With this scheme, arithmetic intensity and parallelism can be tuned easily.

the data from buffers 10, 11, 12, compute the sum and write the results to the store buffers 13, 14, 15.

Each input port must be implemented by a *multiplexor* selecting, from the current iteration, the buffer to read. Similarly, the output port must be realized by a *demultiplexor* selecting the output buffer(s) to write for the current iteration. Each RPN (and in particular, DPN) *compute* process follows a *dynamic single assignment policy*: the buffers are assumed – in a first time – to be fully expanded, just as SARE arrays. In a second time, the buffers are allocated under global scheduling constraints. This way, the multiplexors are just the SARE dependence functions, obtained from array dataflow analysis, as depicted in Section 2. For instance, the multiplexing for the second input port of process C on Figure 1.(c) would be:

$$\text{mux2}(t,i) = \begin{cases} \leftarrow \text{buffer5}[t-1,i] & \forall (t-1,i) \in \mathcal{D}_T \\ \leftarrow \text{buffer2}[i] & \forall t = 0 \end{cases}$$

The demultiplexing may be deduced as follows. For each multiplexing clause $\leftarrow \text{buffer}[u(\vec{i}_C)] \quad \forall \vec{i}_C \in \mathcal{D}$ of a consumer process C, add to the producer process P the following demultiplexing clause:

$$\rightarrow \text{buffer}[\vec{i}_P] \quad \forall \vec{i}_P \in u(\mathcal{D})$$

Communication Synthesis. On the DPN model of computation, the data flowing in/out of a tile band must be resolved by an external memory. Starting from a canonical PPN (e.g. Figure 1.(c)), we add incrementally the multiplexing from the loads and to the store in the following way. For each multiplexing clause $\leftarrow \text{buffer}[u(\vec{i}_C)] \quad \forall \vec{i}_C \in \mathcal{D}$, select the iterations with a source $u(\vec{i}_C)$ from a different tile band:

$$\mathcal{D}_{\text{to\_load}} := \cup_{d=0}^{\dim \text{im}\, \mathcal{B}_C - 1} \{\vec{i}_C \in \mathcal{D}, \mathcal{B}_C(u(\vec{i}_C)) \ll^d \mathcal{B}_C(\vec{i}_C)\}$$

Notice the $\dim \text{im}\, \mathcal{B}_C - 1$ which excludes the last tiling dimension. Then, add a multiplexing clause from the load process, through $\text{buffer}_{\text{Load},C}$:

$$\leftarrow \text{buffer}_{\text{Load},C}[u_{\text{org}}(\vec{i}_C)] \quad \forall \vec{i}_C \in \mathcal{D}_{\text{to\_load}}$$

Where u_{org} denotes the original index function from the program (e.g. for read 2 of process C: $(t,i) \mapsto (t,i)$). This clause will be set as *prioritary*, meaning that in case of ambiguity with another multiplexing clause, this clause must be chosen. Demultiplexing to the store is computed symmetrically: each cut source iteration $\vec{i}_P$ of the original producer process P must redirect its result to the store process. Hence, we add the P demultiplexing clause:

$$\rightarrow \text{buffer}_{\text{Store},P}[u_{\text{org}}(\vec{i}_P)] \quad \forall \vec{i}_P \in u(\mathcal{D}_{\text{to\_load}})$$

Again, $u_{\text{org}}(\vec{i}_P)$ denotes the original index written in original program. Finally, the load and store process are synthesized and synchronized along the lines of [2].

Parallelization. The parallelization consists in splitting the compute process iterations with the outer tiling hyperplanes (excluding the last one). For each process C and each sub-tile coordinate $\vec{\ell}$, we generate the process $C_{\vec{\ell}}$ with the iteration domain:

$$\mathcal{D}_{C_{\vec{\ell}}} := \{\vec{i} \in \mathcal{D}_C, \phi_C(\vec{i}) = \mathcal{T}_C(\vec{i}_{block}, \vec{i}_{local}, i_{last}) \wedge \vec{\ell} = \vec{i}_{local}/\vec{s}\}$$

Where the local indices $\vec{i}_{local}$ are separated from the last local index i_{last} and $\vec{s}$ gathers the "cut step" across each tiling dimension (on our example, $s_0 = 2$). Each original multiplexing clause $\leftarrow \text{buffer}[u(\vec{i}_C)] \quad \forall \vec{i}_C \in \mathcal{D}$ from a producer process P is rephrased as follow. For each parallel instances $P_{\vec{\ell}'}$ and $C_{\vec{\ell}}$, add the multiplexing clause:

$$\leftarrow \text{buffer}_{P_{\vec{\ell}'},C_{\vec{\ell}}}[u(\vec{i}_C)] \quad \forall \vec{i}_C \in \mathcal{D} \cap \mathcal{D}_{C_{\vec{\ell}}} \text{ s.t. } u(\vec{i}_C) \in \mathcal{D}_{P_{\vec{\ell}'}}$$

And the symmetric demultiplexing clause to $P_{\vec{\ell}'}$:

$$\rightarrow \text{buffer}_{P_{\vec{\ell}'},C_{\vec{\ell}}}[\vec{i}_P] \quad \forall i_P \in u(\mathcal{D}_{P_{\vec{\ell}'},C_{\vec{\ell}}})$$

With $\mathcal{D}_{P_{\vec{\ell}'},C_{\vec{\ell}}}$ the set computed for the multiplexing clause.

Channel Selection and Sizing. So far, the DPN verifies the dynamic single assignment property: the channels are completely expanded, just like SARE arrays. For each channel, we analyze the communication pattern to select the channel type (FIFO, FIFO+register, scratchpad). We apply the method proposed in [40], briefly described thereafter. When the consumer reads the data in the same order as the

writes, we have an *in-order* communication pattern. When each data is read only once, we have a *read-once* communication pattern. When we have both *in-order* and *read-once*, the channel may be implemented by a FIFO. When *in-order* but not *read-once*, the channel may be implemented by a FIFO connected to register containing the read data (possibly several times). When *in-order* does not hold, the channel is implemented with an addressed buffer. Additional synchronization mechanisms are then required [5].

Finally, we size the channel using an array contraction technique [23] from our exact liveness analysis [1]. The result is the buffer size along each dimension as well as an allocation function, mapping each single assignment channel cell $\vec{i}$ to a buffer cell $\sigma(\vec{i})$. This mapping is used only when the channel is implemented with an addressed buffer.

6 Back-End

This section outlines the architectural choices involved in the mapping of a DPN to an FPGA configuration.

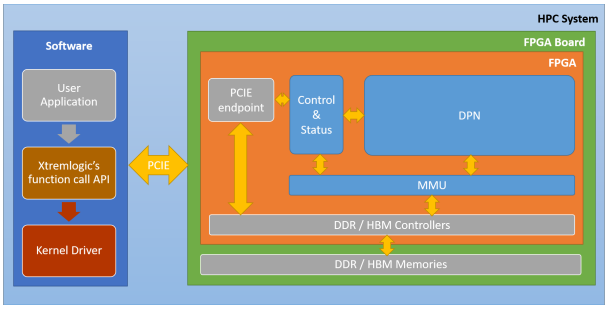


Figure 3. BSP integration.

Figure 3 depicts the high-level HPC system that uses PCI-Express attached FPGA boards. The user interacts with the accelerator through a provided high level function call API wrapper over the low level data transfer and handshaking with the FPGA board. The FPGA board is composed of an FPGA and multiple DDR or HBM modules depending on the FPGA board model. The PCIE endpoint is used in a high level passthrough interface to the DDR memory and to the kernel Control and Status module. The MMU is responsible for data fetch and store from and to the main DDR memory. Control and Status module interacts with the software API and is responsible for getting base pointers of the data and parameters, and signals the end of computation. The DPN in hardware is implemented directly with processes responsible for iteration and execution on data and channels used for data transfer and synchronization between processes.

The hardware implementation of the DPN is presented in Figure 4 and is composed of processes (in blue) communicating through channels (in green).

Processes. have a control unit that computes the iterations and control signals to the Inmuxes and Demuxes. The

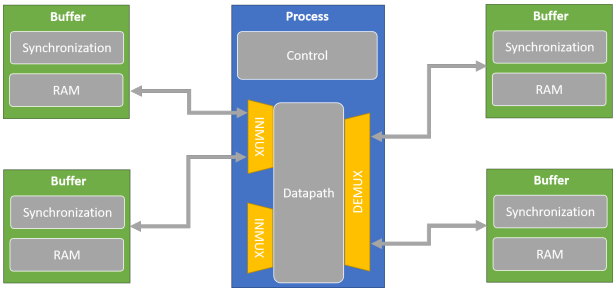


Figure 4. High level diagram of the backend implementation of a DPN.

Inmux is responsible of fetching data from the Buffers and Demuxes to store the data into output Buffers. All these units make an extensive use of piece-wise affine functions, which are efficiently synthesized as DAGs, using the technique described in [6]. These units communicate over a long pipelined bus with channels. This bus allows for better operating timings of the circuit. The communication protocol is a buffered handshake. The datapath as an instantiation of a statement using using FloPoCo automatic operator generation tool [11]. The tool allows for custom arithmetic operator generation based on target, required frequency and arithmetic precision. Custom pipeline improves timing and custom precision improve the resource usage of the FPGA. Based on the required operator the tool will instantiate it using LUT, RAMB and DSP resource available on an FPGA.

Channels. When the channels are not typed as a FIFO by the front-end, they are implemented as an addressable buffer. We synthesize a RAM memory and a synchronization unit [5]. Thanks to an affine predicate using the producer and the consumer schedule, the synchronization unit receives the iterations from the producer and the consumer, and unlocks the read and the writes transactions. The RAM unit is implemented in FPGA using RAMB resources.

7 Experimental Results

This section presents the experimental results obtained from our HLS compiler, implementing the ideas described in this paper.

Experimental Setup. As experimental setup we use as target the XCVU9P-L2FSGD2104E FPGA found on VCU1525 board from Xilinx with a 64 GB of quad DIMM DDR memory with a maximum theoretical bandwidth of 76.8 GB/s. For synthesis and simulation, we use Xilinx Vivado version 18.3. We can generate a FPGA specific hardware system for Intel FPGAs as well and for many targets, Arria 10, Stratix10 and other generations. However for clearness purpose, we use just one vendor. In Table 1, we present the simulation and synthesis results. The examples presented here were taken from the Polybench library [28]. We use small dataset

and parallelization levels for reasonable low level hardware simulation times (less than one hour per example). Even thought our compiler generates the hardware compute accelerator under 10 minutes, hardware simulation of Polybench Large dataset on a highly parallel system, i.e. full FPGA usage takes weeks. The examples use single precision floating point format. The synthesis results includes only the light blue hardware modules from figure 3. The rest of the modules are assumed to be the Board Support Package (BSP) related and use a constant amount of FPGA resources for all the kernels. The Kernel/par16 means that the kernel is parallelized by 16: when the tiles are 3-dimensional (GEMM), the two first dimensions are split by 4, when the tiles are 2-dimensional (BICG, Jacobi1D) the first dimension is split by 16.

Analysis. The purpose of this section is to present how the DPN model implemented in hardware behaves in a real HPC hardware environment. As opposed to embedded environment, when most of the time data is streamed or is present in the local chip SRAM (called Block RAM in FPGAs), an HPC integration requires addressing and testing integration issues related to bus and memory turnaround latencies, wide burst transfers, compute process location and data distribution in the FPGA. All these require data flow synchronization protocols to be included deep inside the DPN model. As compared to manual implementation of an embedded kernel, our system aware implementation will use more hardware resources due to complex control flow and synchronization and will have some performance penalty as compared to ideal no control flow implementation. For example a GEMM systolic local memory access only implementation will use less FPGA logic resources such as LUTs (no control flow) and similar amount of DSP resources (multiply accumulate part).

We will continue with the analysis of the experimental results table. We chose small tile sizes for our experimentation limited by our parallelization levels and data set size. Modern FPGAs can fit tiles sizes hundreds of time bigger. One can implement a DPN on most FPGAs, even the smallest ones by parametrizing the resource vectors in the input of the compiler. The smallest DPN would be one with a tile of 1 element and a parallelization of 1. However, this will have high performance impact due to startup latency and handshaking ping-pong. The model can be theoretically deployed on non-FPGA targets, but this is for now an open research question.

In order to asses the scalability penalty of parallelization of the system, we benchmark the sequential (one parallel unit) and a parallelized by 16 version. We measure multiple performance indicators such as clock cycles from start to finish, the amount of data read and written to/from the external DDR memory, clock period, and FPGA resources such as LUTs (small lookup tables used in implementing boolean logic function – mostly used for control), RAMB36/18 - number of Block RAMS of 36kbit and 18kbit, URAM - 288kbits 72bit wide block RAMs and DSP (Digital Signal Processing blocks) that implement multiplications. We use all the parameters to asses the scalability of the system as the full FPGA system performance will be limited by all of them.

One can chose the number of clock cycles in order to asses the performance effectiveness of the parallelization. For example BIGC shows almost linear level of parallelization scalability for both tiled and non tiled versions. On the other hand GEMM and Jacobi1D does not scale linearly. This is due to radically different model and implementation of our compiler compared to embedded system ones. The analysis of the signal traces in the hardware simulation shows that the problem lies in the handshacking protocol used to synchronize and transfer data between processes through a buffer. The hardware was designed for big tiles (big tile sizes in full FPGA design) and since the parallelization cuts the original tile in small pieces for each parallel process, the data throughput through buffers suffers due to ping pong synchronization of control flow introducing many voids in the pipelined communication bus. However, this should not be the case for much bigger per parallel process tiles (starting at $128 \times 128 \times 128$).

One of the highly limiting performance factors of HPC systems is the memory wall problem. By comparing our data reads and writes to original non tiled version of the application, we can observe that our automatic tiling techniques reduce significantly the memory bandwidth required. For a large system with large domains such optimizations are mandatory, as an embedded low/no control flow kernel will stall most of the time waiting for the data from the DDR. Such techniques can be applied manually for simple examples such as GEMM before writing the code for example for OpenCL, however algorithms such as Jacobi1D are much more difficult to tile manually, incurring significant debugging and tuning times from the FPGA programmer.

We estimate the clock period (frequency) of the circuit. As can be seen highly pipelined busses scale well with parallelization and induce only a fraction of the penalty even in highly complex examples such as Jacobi1D that have many data dependences (*i.e.* many communication busses) that put pressure on the routing resources of the FPGA (configurable communication wires present on the FPGAs). We estimate that the clock frequency will remain stable even on very large FPGAs such as VU57p from Xilinx.

FPGA usage of resources is another limiting factor in parallelization. Generally the parallelization level will be limited by the first resource that achieve saturation of the FPGA usage. Our compiler can generate systems that use most of the resources by implementing for example FPGA resources such as DSP using LUTs, or DSP using LUTs. If we analyze the LUTs usage between a parallelized and non parallelized

Table 1. Resource usage and timings.

Kernel	Tile	Dataset	Clock cycles	Our data RW GB	Orig data RW GB	Clock period ns	LUT	RAMB36	RAMB18	URAM	DSP
GEMM	$32 \times 32 \times 32$	SMALL	484770	0.000147	0.005038	4.5	10275	97	13	0	41
GEMM/par16	$32 \times 32 \times 32$	SMALL	110044	-	-	5.2	74531	425	104	0	131
BIGC	32×32	SMALL	295224	0.000132	0.000375	3.4	13237	88	17	1	4
BIGC/par16	32×32	SMALL	20816	-	-	4.1	104992	538	168	0	64
Jacobi1D	64×64	MEDIUM	45010	0.000016	0.000890	2.4	12987	112	11	0	0
Jacobi1D/par16	64×64	MEDIUM	19328	-	-	2.9	94136	322	191	0	0

version, we use mostly $\times$ 7 times only for a 16 level of parallelization. This is due to a fixed part managing the tiles present in both parallelized and non parallelized versions, due to logarithmic resource usage in all the data transfer between the tile managing unit and the independent parallel units and due to the low resource usage in communication between adjacent parallel processes. This proves the implementation of the DPN is highly scalable with sub linear overhead on the LUT resources. The RAM resources of a parallel system increase by a factor between 3 and 7 and this is counter-intuitive, as we divided the tiles between parallel processes. Some of it is related to the distribution of data to parallel processes and the limitations of the FPGAs. Compared to GPUs that have true multi-port SRAM memories, BRAM on FPGA are only dual-port. Let us take the example of GEMM that requires 131kb of BRAM per tile for the array A. The distribution of this RAM for the input array A to parallel processes along the direction of the j loop is in the broadcast mode, *i.e.* data must be replicated, as no multi-reader is allowed on a dual port memory. The distribution of B in the direction of the i loop, data is distributed as smaller chunks assigned to individual processes. With a parallelization factor of 4 in the direction of i and j loops, the overhead in data usage should be 4. We observe a little bit more than that due to sub-optimal data placement of the Xilinx placer that chose to use 18kb memories for 32kbit of data. For a full design, the Xilinx placer chooses better placing and packing of resources. DSP (multiplication) resource usage increase is linear for BIGC and smaller than linear for the rest of the examples. In our implementation the DSP usage increase logarithmically, as they are not only used for computation on data, but also for computation of the load/store address from the DDR memory. The current implementation of our backend is not yet fully optimized: compared to a manual implementation it would consume more LUT resources. However, the overall performance for most applications will not suffer as FPGAs have hundreds of times more LUTs than DSP blocks used for multiplication usually required in HPC applications. In the long term our performance should be similar to a manual implementation and this is a topic of future research.

We analyzed the performance scalability on all the important performance metrics for an HPC application acceleration. Such an analysis is much more complex than on other systems due to the heterogenity of the FPGA architecture and the diversity of the applications. It is much more complex than a standard CPU roofline model. We conclude that the DPN model with presented implementation has good scalability for HPC application acceleration on FPGA targets.

8 Related Work

This section presents the related work. First, we discuss the dataflow models of interest for HLS. Then, we discuss the approaches for HLS using the polyhedral model.

Dataflow Models and HLS. Dataflow models of execution express naturally the parallelism and serve as parallel programming languages or intermediate representation for parallelizing compilers. Since Kahn process networks [21], many domain-specific variants were introduced to target synchronous systems [22] and streaming applications for embedded systems [17, 20, 27]. Unlike RPN, none of these models are defined as a translation of a program. In particular, there is no direct way to "tile" these models and define a data spilling system as for DPN. This flexibility is a major added-value.

Polyhedral process networks [42] is the first attempt to propose a compilation-oriented dataflow model of computation *for polyhedral programs*. PPN were initially designed as a coarse-grain model of computation for heterogeneous systems [34]. Then, as a fine-grain intermediate representation for HLS [39, 41] with a direct parallel partitionning strategy. As PPN target embedded applications with a small footprint, no spilling mechanism was proposed. In the seminal definition, polyhedral process networks are restricted to one statement per process. Ad-hoc transformations (process splitting/merging) were also proposed [25]. RPN might be view as a general formalism to express the partitioning of computations into processes and the partitioning of communications (direct dependences) into channels.

Polyhedral HLS. Since the seminal work on systolic circuit synthesis [13, 30, 32], most polyhedral approaches for HLS focus on source-level transformations in front of an HLS tool – typically VivadoHLS [44]. This includes polyhedral loop transformations to enable parallelism, pipelining and fine-grain communication [48], throughput improvement [24], off-chip memory transfer minimisation [2, 7, 29] or more recently systolic array generation [45].

These approaches are inherently bounded by the HLS tool, the lack of clear semantic for the input langage, where prescribing a dataflow execution can be very difficult; and the lack of documentation for the translation schemes. Also, polyhedral code generators need to be improved, in particular to reduce the control overhead [47]. Another subtle limitation is inherent to the polyhedral model itself. Affine scheduling, when considered globally, prescribes fork-join parallelism, but not dataflow parallelism. For instance, it is not possible to prescribe a dataflow execution for a simple producer/consumer on a multidimensional array. With our DPN process networks, we overcome that issue *structurally* at the langage level.

9 Conclusion

In this paper, we have proposed the data-aware process networks, a dataflow low-level intermediate representation for high-level synthesis of HPC kernels, using polyhedral compilation techniques. We show how both data-aware process networks and polyhedral process networks might be view as specializations of a more general compilation oriented dataflow model of computation: the regular process networks. In particular, we show that regular process networks induces a general compilation methodology, and we show how to instanciate it on the DPN case to compile an FPGA circuit configuration from a C-level algorithmic description. Finally, experimental results show the effectiveness of our approach.

In the future, we plan to extend our compilation scheme with multibanking, as required for HBM and novel resource factorization schemes. Also, we plan to investigate new RPN instances to target heterogeneous systems with multiple hardware accelerators (including FPGAs and GPUs).

References

[1] Christophe Alias, Fabrice Baray, and Alain Darte. 2007. Bee+Cl@k : An Implementation of Lattice-Based Array Contraction in the Source-to-Source Translator ROSE. In *ACM SIGPLAN/SIGBED Conference on Languages, Compilers, and Tools for Embedded Systems (LCTES'07)*.

[2] Christophe Alias, Alain Darte, and Alexandru Plesco. 2013. Optimizing Remote Accesses for Offloaded Kernels: Application to High-Level Synthesis for FPGA. In *ACM SIGDA Intl. Conference on Design, Automation and Test in Europe (DATE'13)*. Grenoble, France.

[3] Christophe Alias, Bogdan Pasca, and Alexandru Plesco. 2011. Automatic Generation of FPGA-Specific Pipelined Accelerators. In *International Symposium on Applied Reconfigurable Computing (ARC'11)*.

[4] Christophe Alias, Bogdan Pasca, and Alexandru Plesco. 2012. FPGA-Specific Synthesis of Loop-Nests with Pipeline Computational Cores. *Microprocessors and Microsystems* 36, 8 (November 2012), 606–619.

[5] Christophe Alias and Alexandru Plesco. 2014. Method of Automatic Synthesis of Circuits, Device and Computer Program associated therewith. Patent FR1453308.

[6] Christophe Alias and Alexandru Plesco. 2017. Optimizing Affine Control with Semantic Factorizations. *ACM Transactions on Architecture and Code Optimization (TACO)* 14, 4 (Dec. 2017), 27.

[7] Samuel Bayliss and George A Constantinides. 2012. Optimizing SDRAM bandwidth for custom FPGA loop accelerators. In *Proceedings of the ACM/SIGDA international symposium on Field Programmable Gate Arrays*. ACM, 195–204.

[8] Michaela Blott. 2016. Reconfigurable future for HPC. In *High Performance Computing & Simulation (HPCS), 2016 International Conference on*. IEEE, 130–131.

[9] Uday Bondhugula, Albert Hartono, J. Ramanujam, and P. Sadayappan. 2008. A practical automatic polyhedral parallelizer and locality optimizer. In *Proceedings of the ACM SIGPLAN 2008 Conference on Programming Language Design and Implementation, Tucson, AZ, USA, June 7-13, 2008*. 101–113. https://doi.org/10.1145/1375581.1375595

[10] Philippe Coussy and Adam Morawiec. 2008. *High-Level Synthesis: From Algorithm to Digital Circuit*. Springer.

[11] Florent de Dinechin and Bogdan Pasca. 2011. Designing Custom Arithmetic Data Paths with FloPoCo. *IEEE Design and Test* (2011).

[12] E. F. Deprettere, E. Rijpkema, P. Lieverse, and B. Kienhuis. 2000. Compaan: Deriving Process Networks from Matlab for Embedded Signal Processing Architectures. In *8th International Workshop on Hardware/-Software Codesign (CODES'2000)*. San Diego, CA.

[13] Steven Derrien, Sanjay Rajopadhye, Patrice Quinton, and Tanguy Risset. 2008. High-level synthesis of loops using the polyhedral model. In *High-level synthesis*. Springer, 215–230.

[14] Paul Feautrier. 1991. Dataflow analysis of array and scalar references. *International Journal of Parallel Programming* 20, 1 (1991), 23–53. https://doi.org/10.1007/BF01407931

[15] Paul Feautrier. 1992. Some Efficient Solutions to the Affine Scheduling Problem. Part I. One-dimensional Time. *International Journal of Parallel Programming* 21, 5 (Oct. 1992), 313–348. https://doi.org/10.1007/BF01407835

[16] Paul Feautrier. 1992. Some Efficient Solutions to the Affine Scheduling Problem. Part II. Multidimensional Time. *International Journal of Parallel Programming* 21, 6 (Dec. 1992), 389–420. https://doi.org/10.1007/BF01379404

[17] Paul Feautrier. 2006. Scalable and Structured Scheduling. *International Journal of Parallel Programming* 34, 5 (Oct. 2006), 459–487.

[18] Paul Feautrier and Christian Lengauer. 2011. Polyhedron Model. In *Encyclopedia of Parallel Computing*. 1581–1592.

[19] Al Geist and Daniel A Reed. 2015. A survey of high-performance computing scaling challenges. *International Journal of High Performance Computing Applications* (2015), 1094342015597083.

[20] Michael I Gordon, William Thies, and Saman Amarasinghe. 41. Exploiting coarse-grain task, data, and pipeline parallelism in stream programs. *ACM SIGPLAN Notices* 11 (41), 2006.

[21] Gilles Kahn. 1974. The Semantics of Simple Language for Parallel Programming. In *IFIP Congress 74*. 471–475.

[22] Edward A Lee and David G Messerschmitt. 1987. Synchronous data flow. *Proc. IEEE* 75, 9 (1987), 1235–1245.

[23] Vincent Lefebvre and Paul Feautrier. 1998. Automatic Storage Management for Parallel Programs. *Parallel Comput.* 24 (1998), 649–671.

[24] Peng Li, Louis-Noël Pouchet, and Jason Cong. 2014. Throughput optimization for high-level synthesis using resource constraints. In *Int. Workshop on Polyhedral Compilation Techniques (IMPACTâĂŹ14)*.

[25] Sjoerd Meijer. 2010. *Transformations for Polyhedral Process Networks*. Ph.D. Dissertation. Leiden Institute of Advanced Computer Science (LIACS), and Leiden Embedded Research Center, Faculty of Science, Leiden University.

[26] Dan I. Moldovan and Jose A. B. Fortes. 1986. Partitioning and mapping algorithms into fixed size systolic arrays. *IEEE transactions on computers* 1 (1986), 1–12.

[27] Antoniu Pop and Albert Cohen. 2013. OpenStream: Expressiveness and data-flow compilation of OpenMP streaming programs. *ACM Trans. Archit. Code Optim.* 9, 4, Article 53 (Jan. 2013), 25 pages.

[28] Louis-Noël Pouchet. 2012. Polybench: The polyhedral benchmark suite. *URL: http://www. cs. ucla. edu/˜ pouchet/software/polybench/[cited July,]* (2012).

[29] Louis-Noel Pouchet, Peng Zhang, P. Sadayappan, and Jason Cong. 2013. Polyhedral-based Data Reuse Optimization for Configurable Computing. In *Proceedings of the ACM/SIGDA International Symposium on Field Programmable Gate Arrays* (Monterey, California, USA) *(FPGA '13)*. ACM, New York, NY, USA, 29–38. https://doi.org/10.1145/2435264.2435273

[30] Patrice Quinton. 1984. Automatic Synthesis of Systolic Arrays from Uniform Recurrent Equations. In *Proceedings of the 11th Annual Symposium on Computer Architecture, Ann Arbor, USA, June 1984*. 208–214. https://doi.org/10.1145/800015.808184

[31] Patrice Quinton and Vincent van Dongen. 1989. The mapping of linear recurrence equations on regular arrays. *Journal of VLSI signal processing systems for signal, image and video technology* 1, 2 (1989), 95–113.

[32] Sanjay V Rajopadhye and Richard M Fujimoto. 1990. Synthesizing systolic arrays from recurrence equations. *Parallel computing* 14, 2 (1990), 163–189.

[33] Sanjay V. Rajopadhye, S. Purushothaman, and Richard M. Fujimoto. 1986. On synthesizing systolic arrays from Recurrence Equations with Linear Dependencies. In *Foundations of Software Technology and Theoretical Computer Science*, Kesav V. Nori (Ed.). Lecture Notes in Computer Science, Vol. 241. Springer Berlin Heidelberg, 488–503.

[34] Edwin Rijpkema, Ed F Deprettere, and Bart Kienhuis. 2000. Deriving process networks from nested loop algorithms. *Parallel Processing Letters* 10, 02n03 (2000), 165–176.

[35] Robert Schreiber, Shail Aditya, B Ramakrishna Rau, Vinod Kathail, Scott Mahlke, Santosh Abraham, and Greg Snider. 2000. High-level synthesis of nonprogrammable hardware accelerators. In *Application-Specific Systems, Architectures, and Processors, 2000. Proceedings. IEEE International Conference on*. IEEE, 113–124.

[36] Weijia Shang and Jose A. B. Fortes. 1992. Independent partitioning of algorithms with uniform dependencies. *IEEE Trans. Comput.* 41, 2 (1992), 190–206.

[37] Jürgen Teich, Lothar Thiele, and Lee Z Zhang. 1997. Partitioning processor arrays under resource constraints. *Journal of VLSI signal processing systems for signal, image and video technology* 17, 1 (1997), 5–20.

[38] William Frederick Thies. 2009. *Language and compiler support for stream programs*. Ph.D. Dissertation. Massachusetts Institute of Technology.

[39] Alexandru Turjan. 2007. *Compiling nested loop programs to process networks*. Ph.D. Dissertation. Leiden Institute of Advanced Computer Science (LIACS), and Leiden Embedded Research Center, Faculty of Science, Leiden University.

[40] Alexandru Turjan, Bart Kienhuis, and Ed Deprettere. 2007. Classifying interprocess communication in process network representation of nested-loop programs. *ACM Transactions on Embedded Computing Systems (TECS)* 6, 2 (2007), 13.

[41] Sven van Haastregt. 2013. *Estimation and Optimization of the Performance of Polyhedral Process Networks*. Ph.D. Dissertation. Leiden Institute of Advanced Computer Science (LIACS), and Leiden Embedded Research Center, Faculty of Science, Leiden University.

[42] Sven Verdoolaege. 2010. *Polyhedral Process Networks*. 931–965.

[43] Sven Verdoolaege, Hristo Nikolov, and Todor Stefanov. 2007. PN: a tool for improved derivation of process networks. *EURASIP journal on Embedded Systems* 2007, 1 (2007), 19–19.

[44] vivado [n.d.]. Xilinx Vivado HLS. http://www.xilinx.com/products/design-tools/vivado/integration/esl-design.html.

[45] Jie Wang, Licheng Guo, and Jason Cong. 2021. AutoSA: A Polyhedral Compiler for High-Performance Systolic Arrays on FPGA. In *Proceedings of the 2021 ACM/SIGDA international symposium on Field-programmable gate arrays*.

[46] Chen Zhang, Di Wu, Jiayu Sun, Guangyu Sun, Guojie Luo, and Jason Cong. 2016. Energy-Efficient CNN Implementation on a Deeply Pipelined FPGA Cluster. In *Proceedings of the 2016 International Symposium on Low Power Electronics and Design*. ACM, 326–331.

[47] Wei Zuo, Peng Li, Deming Chen, Louis-Noël Pouchet, Shunan Zhong, and Jason Cong. 2013. Improving polyhedral code generation for high-level synthesis. In *2013 International Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS)*. 1–10. https://doi.org/10.1109/CODES-ISSS.2013.6659002

[48] Wei Zuo, Yun Liang, Peng Li, Kyle Rupnow, Deming Chen, and Jason Cong. 2013. Improving high level synthesis optimization opportunity through polyhedral transformations. In *Proceedings of the ACM/SIGDA international symposium on Field programmable gate arrays*. 9–18.