Compiling Circuits with Polyhedra

A geometric perspective for HLS

Christophe Alias

INRIA, CNRS, ENS de Lyon, Université de Lyon, France

Scientific Computing Accelerated on FPGAs
Saclay, July 7, 2022

Advertisement

Research interests:

- High-level compilation, focused on automatic parallelization
- Application to HLS for FPGA

Non-academic interests:

- Proud co-founder of the XtremLogic start-up (part of this talk)

!!!Job offering on compilers and/or HLS (Lyon)!!!

- PhD, postdocs
- Faculty positions (MCF/CR)

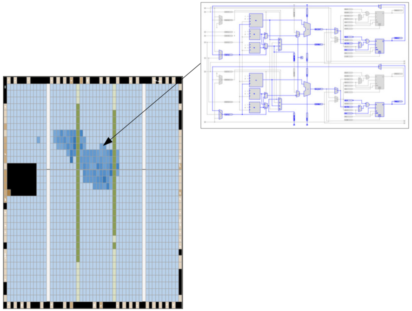
Contact: `christophe.alias@inria.fr`

Outline

Target: FPGA accelerators (1/2)

Field-**P**rogrammable **G**ate **A**rray:

- Look-up tables (LUT), multiplexers, registers
- DSPs
- BRAMs



- Programming an FPGA = designing a circuit!
- Existing C-to-circuit compilers, not out-of-the-box!

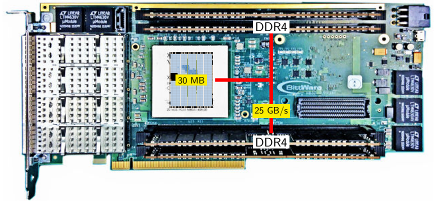
Target: FPGA accelerators (2/2)

Promises:

- Teraflops, with a better energy efficiency than GPUs
- Flexible programming model (not limited to data parallelism)

Challenges:

- Few local memory (tens of MB)
- No OS, no parallel runtime. Everything must be compiled!
- High-level translation required!



High-Level Synthesis at a glance

High-Level Synthesis (HLS): Program → Hardware

- Typically: compute-intensive kernel → hardware accelerator IP
- Target: ASIC or FPGA

Typical flow: $C \xrightarrow{HLS} RTL \xrightarrow{synthesis} Hardware$

Architecture model:

- Von-Neumann (VivadoHLS)
- Synchronous dataflow (e.g. systolic networks) (AlphaZ)
- Asynchronous dataflow (e.g. KPN, RPN partitioning) (Dcc)

Focus: VivadoHLS

HLS tool: C program $\rightarrow$ specialized von-Neumann architecture

Translation schemes: undocumented, but based on J. Cong's Autopilot HLS tool: loop/array transformations, then classical CFG-based C-to-Hardware.

Code optimizations: triggered with user-defined pragmas, and through GUI configuration options.

Low-level and conservative, as the dependence analysis!

Loop transformations: back to the 80s

Loop unroll: exposes cross-iteration parallelism

- Decrease loop overhead, increase parallelism
- More operations ⇒ area and power.

```
for(i=0; i<N; i++)
#pragma HLS UNROLL
  C[i] = A[i] + B[i];
```

```
C[0] = A[0] + B[0];
C[1] = A[1] + B[1];
...
```

Loop transformations: back to the 80s

Loop unroll: exposes cross-iteration parallelism

- Decrease loop overhead, increase parallelism
- More operations ⇒ area and power.

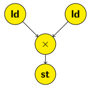
```
for(i=0; i<N; i++)
#pragma HLS UNROLL
   C[i] = A[i] + B[i];
```

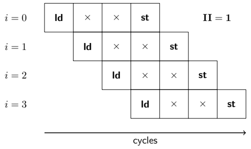
```
C[0] = A[0] + B[0];
C[1] = A[1] + B[1];
...
```

Loop pipelining: Consecutive iterations executed in a pipelined fashion. *Key metric:* initiation interval, II.

- Controlled parallelism and footprint, through II
- Spoilt by conservative dependence analysis!

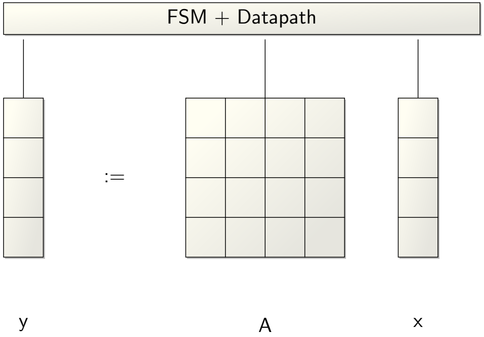
```
for(i=0; i<N; i++)
#pragma HLS PIPELINE II=1
   C[i] = A[i]*B[i];
```





The trouble with arrays

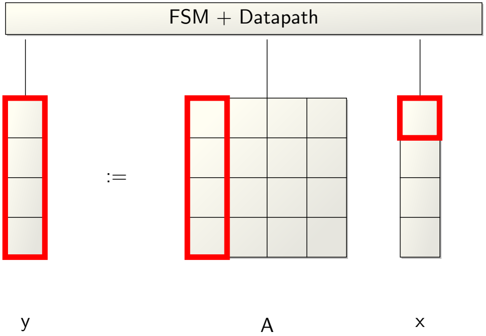
```
for(i=0; i<N; i++) //parallel
  for(j=0; j<N; j++)
    y[i] += A[i][j]*x[j];
```



Arrays are implemented as block RAM, two data ports max.

The trouble with arrays

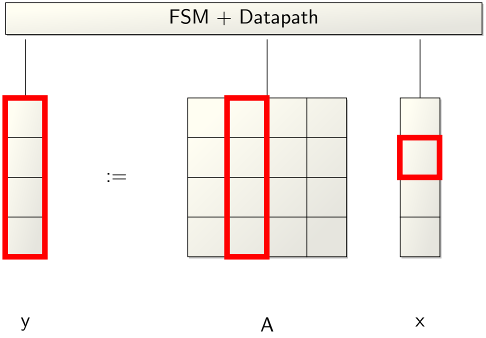
```
for(i=0; i<N; i++) //parallel
  for(j=0; j<N; j++)
    y[i] += A[i][j]*x[j];
```



Arrays are implemented as block RAM, two data ports max.

The trouble with arrays

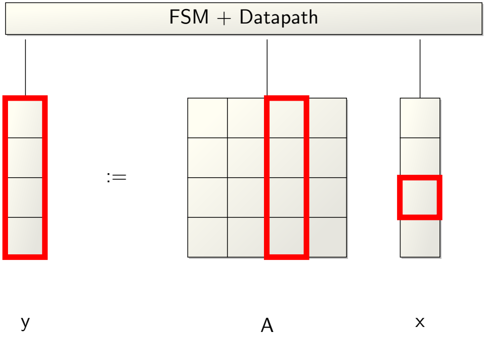
```
for(i=0; i<N; i++) //parallel
  for(j=0; j<N; j++)
    y[i] += A[i][j]*x[j];
```



Arrays are implemented as block RAM, two data ports max.

The trouble with arrays

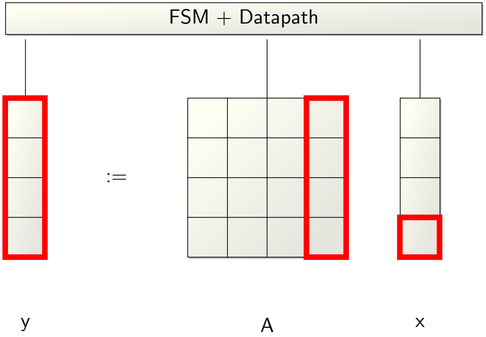
```
for(i=0; i<N; i++) //parallel
  for(j=0; j<N; j++)
    y[i] += A[i][j]*x[j];
```



Arrays are implemented as block RAM, two data ports max.

The trouble with arrays

```
for(i=0; i<N; i++) //parallel
  for(j=0; j<N; j++)
    y[i] += A[i][j]*x[j];
```



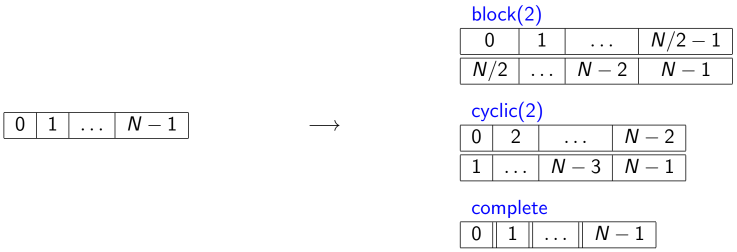
Arrays are implemented as block RAM, two data ports max.

Solution: Multibanking

Multibanking: Partition data across memory banks readable in parallel

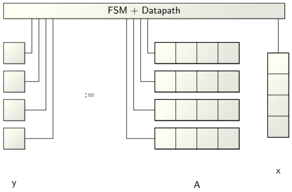
Vivado HLS: language-level array partitioning

- Array dimension(s) to be partitioned
- Array partitioning:
    - (cyclic or block) + factor
    - complete



Use case 1: matrix-vector product

```
#pragma HLS ARRAY_PARTITION
  variable=y complete dim=1
#pragma HLS ARRAY_PARTITION
  variable=A complete dim=1
for(i=0; i<N; i++)
#pragma HLS PIPELINE
  for(j=0; j<N; j++)
    y[i] += A[i][j]*x[j];
```

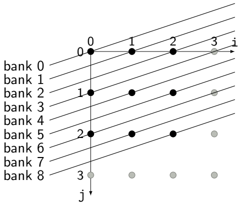


Synthesis results: (VivadoHLS 2019.1, Kintex 7 FPGA)

Kernel	Version	Latency	**Speed-up**	Period (ns)	DSP	FF	LUT
matvec	Baseline	532	**10.2**	6.9	320	4799	4423
	With banking	52		6.3	320	67618	13518

Use case 2: 2D convolution

```
for(i=1; i<N-1; i++)
  for(j=1; j<N-1; j++)
    out[i,j] =
      in[i-1,j-1]+in[i-1,j]+in[i-1,j+1]+
      in[i,j-1]  +in[i,j]  +in[i,j+1]  +
      in[i+1,j-1]+in[i+1,j]+in[i+1,j+1]; //S
```



$$\text{BANK}_{in}(i,j) = i + 3j \bmod 9 \quad \text{OFFSET}_{in}(i,j) = j \bmod N$$

Methodology

- $in[u(\vec{i})] \mapsto \hat{in}[\text{BANK}_{in}(u(\vec{i}))][\text{OFFSET}_{in}(u(\vec{i}))]$
- Add pragmas to partition the bank dimensions: option cyclic, factor=9
- Automation and results: later on the talk!

(Negative) conclusion on mainstream HLS

Promises:

- Full fledge C-to-circuit: programming FPGA as any HA.
- No circuit skills required

Reality:

- Lack of high-level parallelization algorithms: 80's code optimizations, triggered by hand.
- Undocumented compilation scheme: hard to tune
- Poor performances

How to improve the HLS process?

Towards High*er*-Level Synthesis

Approaches:

- **integrated:** throw away mainstream HLS tools, write our own.
- **source-to-source:** consider HLS tools as assemblers, recycle source-level code transformations

Challenges:

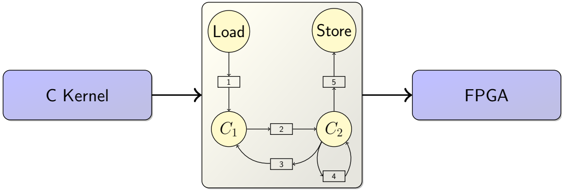
- **all-static**: parallelisation decisions at compile time
- **(very large) scalability**: thousands of parallel units must be considered.

Outline

Contributions

A complete, fully automated, C-to-FPGA approach:

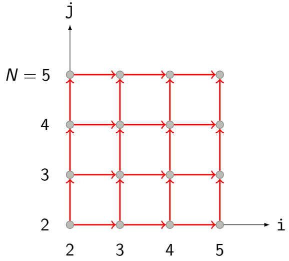
- Data-aware Process Networks (DPN), a dataflow intermediate representation for high-level synthesis of HPC kernels.
- Explicit data spilling, tunable parallelism and arithmetic intensity.
- A front-end $C \to DPN$, and a back-end $DPN \to FPGA$.
- Regular Process Networks (RPN), a generalization of DPN inducing a general HLS methodology.

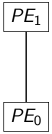


DPN

Focus: polyhedral programs

```
**for** $i$ := 2 **to** $N$
  **for** $j$ := 2 **to** $N$
    a[i,j] := a[i-1,j] + a[i,j-1];
```

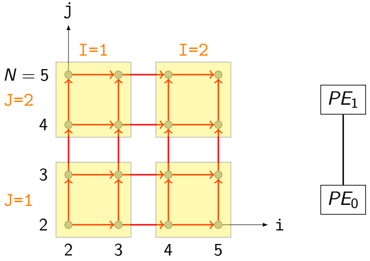




- Loop nests manipulating arrays, all-affine

Focus: polyhedral programs

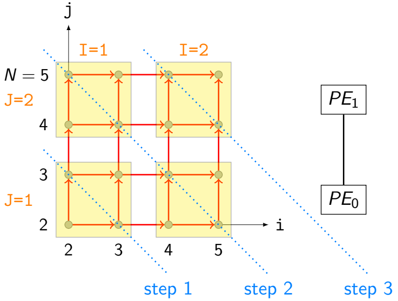
```
for $i$ := 2 to $N$
  for $j$ := 2 to $N$
    a[i,j] := a[i-1,j] + a[i,j-1];
```



- Loop nests manipulating arrays, all-affine
- All-affine world:
  $\phi(i, j) = (i, j)$

Focus: polyhedral programs

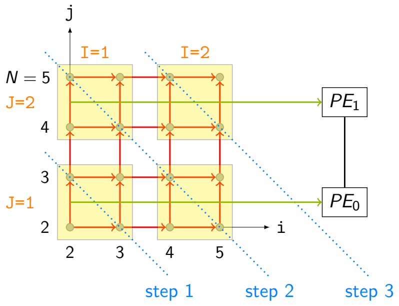
```
**for** $i$ := 2 **to** $N$
  **for** $j$ := 2 **to** $N$
    a[i,j] := a[i-1,j] + a[i,j-1];
```



- Loop nests manipulating arrays, all-affine
- All-affine world:
  $\phi(i,j) = (i,j)$ $\theta(I,J,i,j) = (I+J,i,j)$

Focus: polyhedral programs

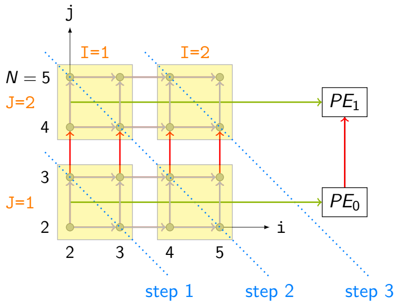
```
**for** $i$ := 2 **to** $N$
  **for** $j$ := 2 **to** $N$
    a[i,j] := a[i-1,j] + a[i,j-1];
```



- Loop nests manipulating arrays, all-affine
- All-affine world:
  $\phi(i,j) = (i,j)$ $\theta(I,J,i,j) = (I+J,i,j)$ $\Pi(I,J,i,j) = J$

Focus: polyhedral programs

```
for $i$ := 2 to $N$
  for $j$ := 2 to $N$
    a[i,j] := a[i-1,j] + a[i,j-1];
```

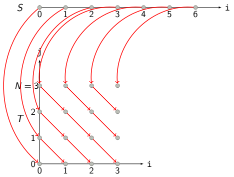


- Loop nests manipulating arrays, all-affine
- All-affine world:
  $\phi(i,j) = (i,j)$ $\theta(I,J,i,j) = (I+J,i,j)$ $\Pi(I,J,i,j) = J$

Dataflow representation of polyhedral programs

```
       for $i$ := 0 to $2N$
S:       c[i] := 0;

       for $i$ := 0 to $N$
         for $j$ := 0 to $N$
T:         c[i+j] := c[i+j] + a[i]*b[j];
```



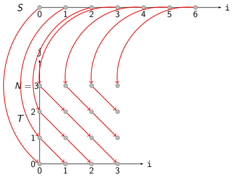
Exact dataflow is **computable:**

$$\sigma(\langle T, i, j \rangle, 1) = \begin{cases} \langle T, i-1, j+1 \rangle & 0 \leq i-1, j+1 \leq N \\ \langle S, i+j \rangle & \text{otherwise} \end{cases}$$

Dataflow representation of polyhedral programs

```
     for $i$ := 0 to $2N$
S:     c[i] := 0;

     for $i$ := 0 to $N$
       for $j$ := 0 to $N$
T:       c[i+j] := c[i+j] + a[i]*b[j];
```



Exact dataflow is **computable:**

$$\sigma(\langle T, i, j\rangle, 1) = \begin{cases} \langle T, i-1, j+1\rangle & 0 \leq i-1, j+1 \leq N \\ \langle S, i+j\rangle & \text{otherwise} \end{cases}$$

Polyhedral programs might be translated to a dataflow compliant **equational representation (SARE):**

$$\begin{cases} S[i] = 0 & 0 \leq i \leq 2N \\ T[i,j] = T[i-1, j+1] + a[i] * b[j] & 0 \leq i-1, j+1 \leq N \\ T[i,j] = S[i+j] + a[i] * b[j] & \text{otherwise} \end{cases}$$

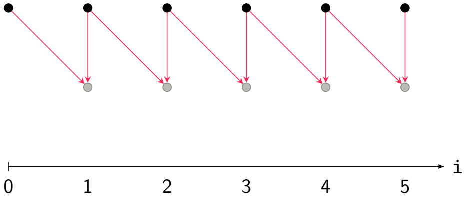
Outline

1) Regular process networks: We combine that representation with partitionings computation $\mapsto$ processes and data $\mapsto$ channels

2) Data-aware process network: We define DPN as a RPN partitioning induced by a loop tiling

Regular Process Networks

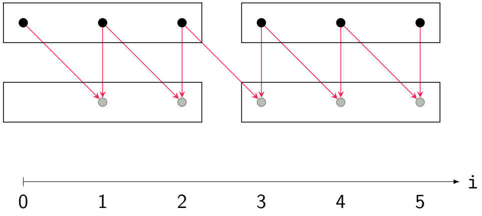
```
  for $i := 0$ to $N$
● $a[i] = f(i);$
  for $i := 1$ to $N$
● $b[i] := a[i-1] + a[i];$
```

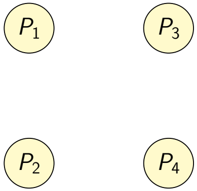


- Partition of the computation: processes
- Partition of $\rightarrow_{pc}$: channels $\{\rightarrow_1, \rightarrow_2, \ldots\}$
- A schedule θ_P for each process P

Regular Process Networks

```
for $i := 0$ to $N$
●  $a[i] = f(i);$
for $i := 1$ to $N$
●  $b[i] := a[i-1] + a[i];$
```

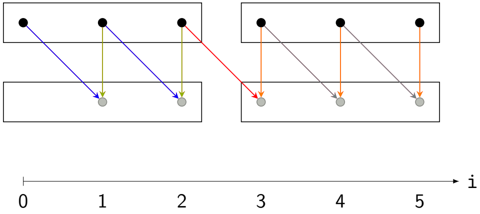


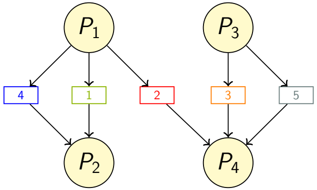


- Partition of the computation: processes
- Partition of $\rightarrow_{pc}$: channels $\{\rightarrow_1, \rightarrow_2, \ldots\}$
- A schedule θ_P for each process P

Regular Process Networks

```
  for $i := 0$ to $N$
● $a[i] = f(i)$;
  for $i := 1$ to $N$
● $b[i] := a[i-1] + a[i]$;
```

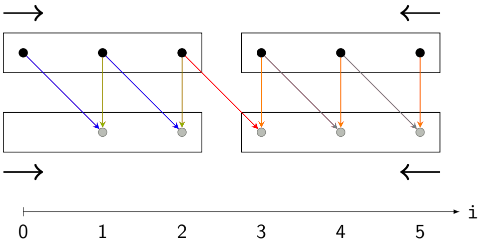


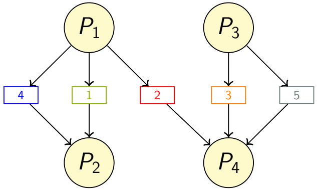


- Partition of the computation: processes
- Partition of $\rightarrow_{pc}$: channels $\{\rightarrow_1, \rightarrow_2, \ldots\}$
- A schedule θ_P for each process P

Regular Process Networks

```
  for $i := 0$ to $N$
● $a[i] = f(i);$
  for $i := 1$ to $N$
● $b[i] := a[i-1] + a[i];$
```



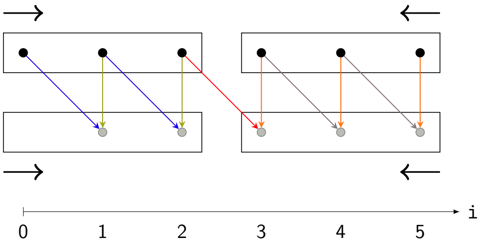


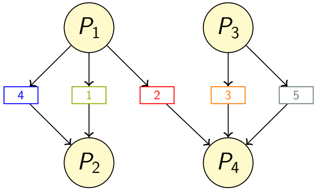
- Partition of the computation: processes
- Partition of $\rightarrow_{pc}$: channels $\{\rightarrow_1, \rightarrow_2, \ldots\}$
- A schedule θ_P for each process P

Regular Process Networks

```
    for $i := 0$ to $N$
●     $a[i] = f(i)$;
    for $i := 1$ to $N$
●     $b[i] := a[i-1] + a[i]$;
```

Locally sequential
Globally dataflow



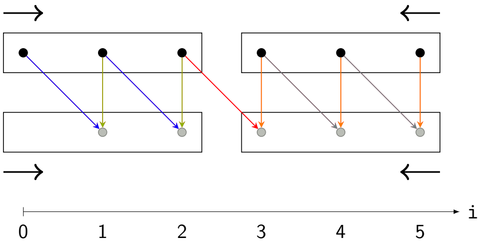


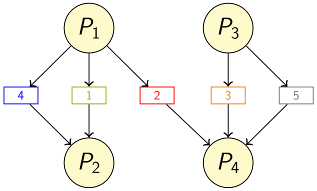
- Partition of the computation: processes
- Partition of $\rightarrow_{pc}$: channels $\{\rightarrow_1, \rightarrow_2, \ldots\}$
- A schedule θ_P for each process P

Regular Process Networks

```
    for $i := 0$ to $N$
●     $a[i] = f(i)$;
    for $i := 1$ to $N$
●     $b[i] := a[i-1] + a[i]$;
```

Locally sequential
Globally dataflow



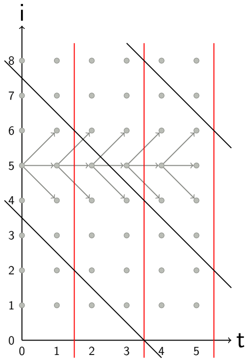


- Process and channel implementation abstracted away
- Bridge polyhedral model $\leftrightarrow$ dataflow models

Data-Aware Process Networks

```
for $t := 1$ to $T$
  for $i := 1$ to $N - 2$
    $a[t, i] := a[t - 1, i - 1] +$
                $a[t - 1, i] +$
                $a[t - 1, i + 1];$
```

Tiling $\phi_S(t, i) = (t, t + i)$

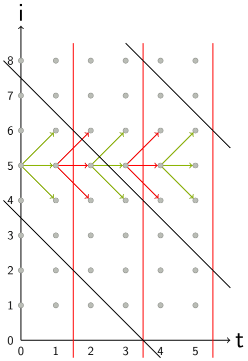


- **RPN partitioning induced by a loop tiling**
- **Communications:** consider tile bands as reuse units
  $\rightsquigarrow$Pipeline: Load(T) $\rightarrow$ C(T) $\rightarrow$ Store(T)

Data-Aware Process Networks

```
for $t := 1$ to $T$
  for $i := 1$ to $N-2$
    $a[t,i] := a[t-1,i-1]+$
               $a[t-1,i]+$
               $a[t-1,i+1];$
```

Tiling $\phi_S(t,i) = (t, t+i)$

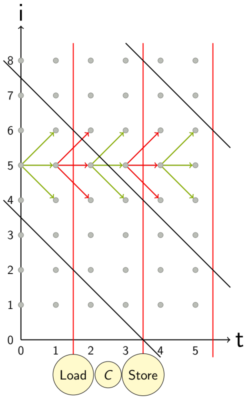


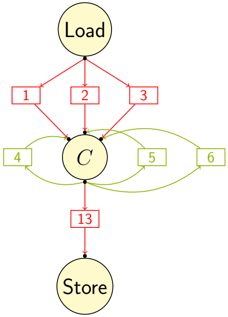
- **RPN partitioning induced by a loop tiling**
- **Communications:** consider tile bands as reuse units $\rightsquigarrow$Pipeline: Load(T) $\to$ C(T) $\to$ Store(T)

Data-Aware Process Networks

```
for $t := 1$ to $T$
  for $i := 1$ to $N-2$
    $a[t,i] := a[t-1,i-1]+$
               $a[t-1,i]+$
               $a[t-1,i+1];$
```

Tiling $\phi_S(t,i) = (t, t+i)$



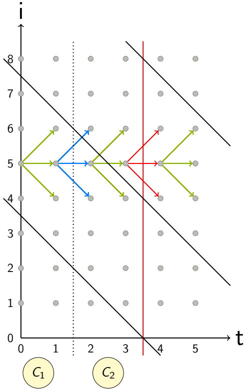


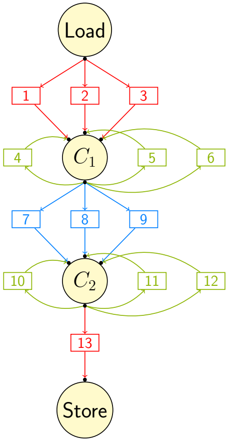
- **RPN partitioning induced by a loop tiling**
- **Communications:** consider tile bands as reuse units
  $\rightsquigarrow$Pipeline: Load(T) $\rightarrow$ C(T) $\rightarrow$ Store(T)

Data-Aware Process Networks

```
for $t := 1$ to $T$
  for $i := 1$ to $N-2$
    $a[t,i] := a[t-1,i-1]+$
               $a[t-1,i]+$
               $a[t-1,i+1];$
```

Tiling $\phi_S(t,i) = (t, t+i)$





Parallelism: split tile band with outer tiling hyperplanes

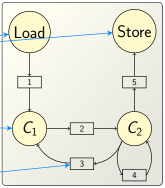
Outline

Data-aware process networks (DPN) [CC21]

Communication synthesis [ASAP10,PPoPP12,IMPACT12,DATE13]

- Control scheduling [ARC11,MICPRO12]
- Control synthesis [TACO17]

- **Channel typing** **[HiP3ES18]**
- Channel allocation [LCTES07]
- Synchronizations [Patent14]



Software: `dcc` (DPN C Compiler)
Transferred to XtremLogic under an Inria license

Focus: channel typing

Goals

- Compile DPN channels
- Preferably with FIFO
  $\rightsquigarrow$ light silicon surface, less synchronization overhead

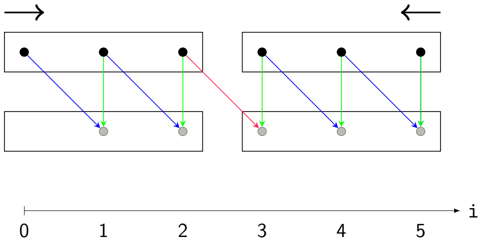
Challenge

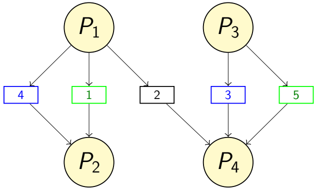
DPN relies on loop tiling, which breaks most FIFO patterns

Approach

- Restructure the channels so most FIFO patterns are recovered
- $\boxed{\textbf{Theorem:}\ \text{the recovery is complete on DPN}}$

Communication patterns

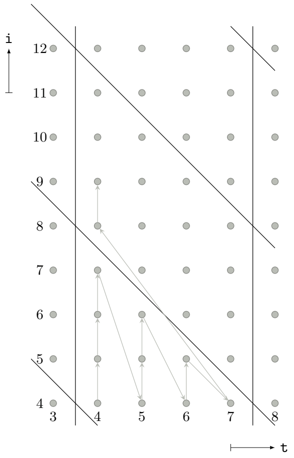


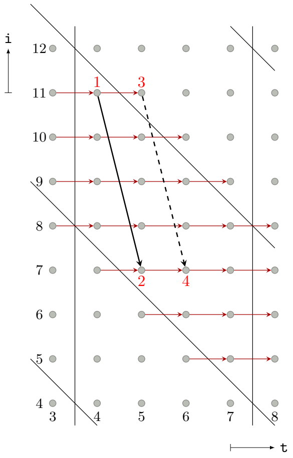


A channel might be implemented by a FIFO iff

- the values are read in the production order (*in-order*)
- each value is read exactly once (*unicity*)

How loop tiling breaks FIFO channels

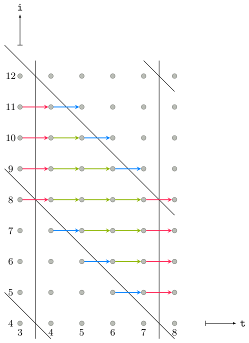


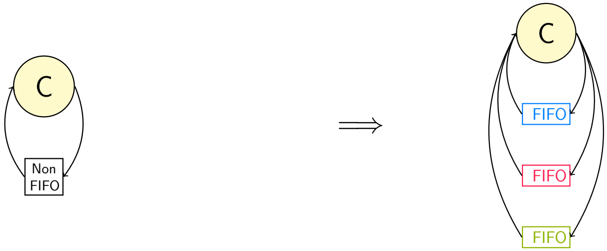


Solution: channel restructuring

```
$\text{SPLIT}(\rightarrow_c, \theta_P, \theta_C)$
  for $k := 1$ to $n$
    $\text{ADD}(\rightarrow_c \cap \{(x,y),\ \theta_P(x) \ll^k \theta_C(y)\})$;
  $\text{ADD}(\rightarrow_c \cap \{(x,y),\ \theta_P(x) \approx^n \theta_C(y)\})$;

$\text{FIFOIZE}((\mathcal{P}, \mathcal{C}))$
  for each channel $c$
    $\{\rightarrow_c^1, \ldots, \rightarrow_c^{n+1}\} := \text{SPLIT}(\rightarrow_c, \theta_{P_c}, \theta_{C_c})$;
    if fifo$(\rightarrow_c^k, \prec_{\theta_{P_c}}, \prec_{\theta_{C_c}})\ \forall k$
      $\text{REMOVE}(\rightarrow_c)$;
      $\text{INSERT}(\rightarrow_c^k)\ \forall k$;
```



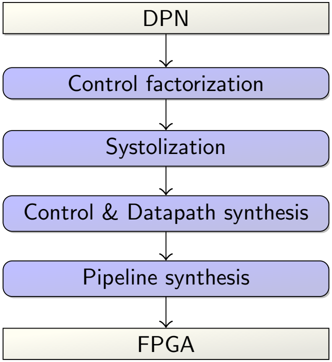


Experimental evaluation

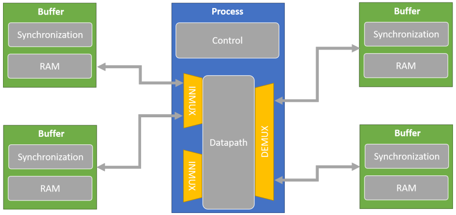
Kernel	#buffers	#fifos	total fifo size	total size	#fifo basic	#fifo passed	#fifo fail	#fifo restored	%fail
trmm	12	12	516	516	2	1	1	1	0
gemm	12	12	352	352	2	1	1	1	0
syrk	12	12	8200	8200	2	1	1	1	0
symm	30	30	1644	1644	6	5	1	1	0
gemver	15	13	4180	4196	4	3	1	1	0
gesummv	12	12	96	96	6	6	0	0	0
syr2k	12	12	8200	8200	2	1	1	1	0
lu	45	22	540	1284	3	0	3	3	0
trisolv	12	9	23	47	4	3	1	1	0
cholesky	44	31	801	1129	6	4	2	2	0
doitgen	32	32	12296	12296	3	2	1	1	0
bicg	12	12	536	536	4	2	2	2	0
mvt	8	8	36	36	2	0	2	2	0
3mm	53	43	5024	5664	6	3	3	3	0
2mm	34	28	1108	1492	4	2	2	2	0
covariance	45	24	542	1662	7	4	3	3	0
correlation	71	38	822	2038	13	9	4	4	0
fdtd-2d	120	120	45696	45696	12	5	7	7	0
jacobi-2d	123	123	10328	10328	10	2	8	8	0
seidel-2d	102	102	60564	60564	9	2	7	7	0
jacobi-1d	23	23	1358	1358	6	2	4	4	0
heat-3d	95	95	184864	184864	20	2	18	18	0

- PolyBench/C v3.2 kernels
- Completeness of recovery on DPN partitioning

Back-end

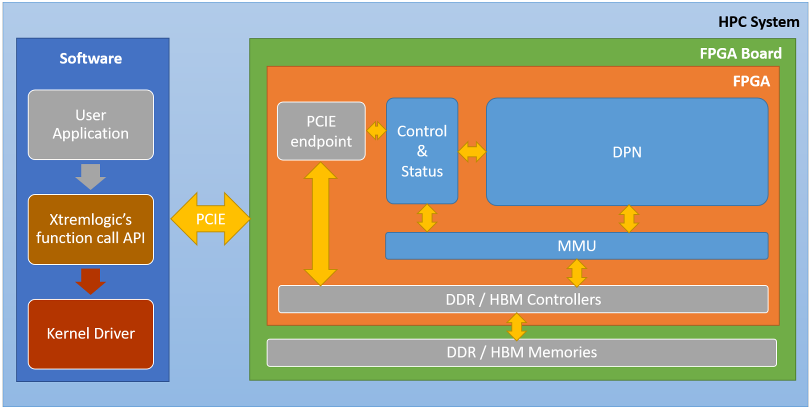


DPN
Control factorization
Systolization
Control & Datapath synthesis
Pipeline synthesis
FPGA

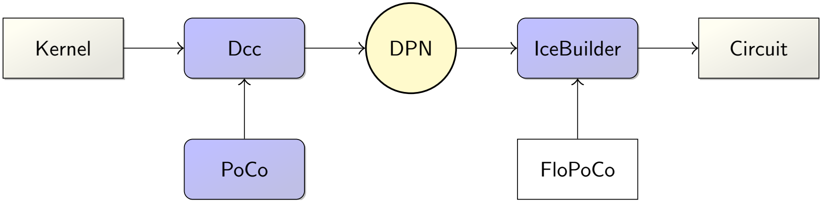


Buffer
Synchronization
RAM
Buffer
Synchronization
RAM
Process
Control
INMUX
Datapath
INMUX
DEMUX
Buffer
Synchronization
RAM
Buffer
Synchronization
RAM

Integration



Experimental setup



Kernels: PolyBench/C v3.2

Target:

- Xilinx VCU1525 board, Virtex Ultra Scale+ FPGA
- 64 GB DDR quad DIMM, maximum bandwidth: 76.8 GB/s

Synthesis: Xilinx Vivado version 18.3

Simulation: XSim (light) + SDAccel (complete)

Experimental results

Kernel	Cycles	Comms (MB)	Orig Comms (MB)	Period (ns)	LUT	RAMB36	RAMB18	URAM	DSP
gemm	484770	0.147	5.038	4.5	10275	97	13	0	41
gemm $\times 16$	110044	-	-	5.2	74531	425	104	0	131
bicg	295224	0.132	0.375	3.4	13237	88	17	1	4
bicg $\times 16$	20816	-	-	4.1	104992	538	168	0	64
jacobi-1d	45010	0.016	0.890	2.4	12987	112	11	0	0
jacobi-1d $\times 16$	19328	-	-	2.9	94136	322	191	0	0

- Cycles ratio: `bicg`: 14, `gemm`: 4, `jacobi-1d`: 2 (tight schedule, process synchronization)
- Data reuse properly exploited
- Stable frequency
- $\times 7$ LUT, $\times 4 - 6$ BRAM (common MMU)

Conclusion (for this part)

Contributions:

- The data-aware process networks, a dataflow model for HLS in the polyhedral model
- A general HLS methodology based on regular process networks
- A complete front-end $C \rightarrow DPN$ and back-end $DPN \rightarrow circuit$
- Industrial transfer: XtremLogic

http://www.xtremlogic.com

Conclusion (for this part)

Contributions:

- The data-aware process networks, a dataflow model for HLS in the polyhedral model
- A general HLS methodology based on regular process networks
- A complete front-end $C \to DPN$ and back-end $DPN \to circuit$
- Industrial transfer: XtremLogic

Future work:

- Increase the abstraction of the input language
- Multibanking support to handle HBM memory
- RPN partionning strategies for heterogeneous systems

`http://www.xtremlogic.com`

Outline

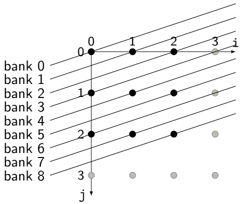
Motivations

Challenge: arrays are stored in memories with few read ports, this hinders parallelism

Approach:

- map memory cells accessed in parallel to distinct memory banks.
- automatically, as a source-to-source polyhedral code transformation

```
for(i=1; i<N-1; i++)
  for(j=1; j<N-1; j++)
    out[i,j] =
      in[i-1,j-1]+in[i-1,j]+in[i-1,j+1]+
      in[i,j-1]  +in[i,j]  +in[i,j+1]  +
      in[i+1,j-1]+in[i+1,j]+in[i+1,j+1]; //S
```



$$\mathrm{BANK}_{in}(i,j) = 3i + j \bmod 9 \quad \mathrm{OFFSET}_{in}(i,j) = i \bmod N$$

Affine multibanking

Global (inter-array) allocation:

- $\text{BANK}_a(\vec{i})$: bank number of $a[\vec{i}]$ (can be a vector)
- $\text{OFFSET}_a(\vec{i})$: offset of $a[\vec{i}]$ into his bank (can be a vector)

source-to-source transformation:

- $a[u(\vec{i})] \mapsto \hat{a}[\text{BANK}_a(u(\vec{i}))][\text{OFFSET}_a(u(\vec{i}))]$
- Add pragmas to partition the bank dimensions

Focus: affine transformations (easier to derive)

- $\text{BANK}_a(\vec{i}) = \phi_a(\vec{i}) \mod \sigma(\vec{N})$
- $\text{OFFSET}_a(\vec{i}) = \psi_a(\vec{i}) \mod \tau(\vec{N})$

Methodology: Write the correctness/efficiency constraints as affine constraints, then give to an ILP solver.

Banking contraints

Correctness: enforce distinct banks for concurrent access

$$a(\vec{i}) \parallel_\theta b(\vec{j}) \wedge (a, \vec{i}) \neq (b, \vec{j}) \Rightarrow \textsc{bank}_a(\vec{i}) \neq \textsc{bank}_b(\vec{j})$$

Relaxed as:

$$a(\vec{i}) \parallel_\theta b(\vec{j}) \wedge (a, \vec{i}) \neq (b, \vec{j}) \Rightarrow \phi_a(\vec{i}) \ll \phi_b(\vec{j})$$

Efficiency: reduce bank numbers

$$\phi_b(\vec{j}) - \phi_a(\vec{i}) \leq \sigma(\vec{N}) \text{ then minimize } \sigma(\vec{N})$$

Analogous to **affine scheduling:**

operation	array cell
dependence	concurrent access
latency	number of banks

Banking algorithm

Input: Program (P, θ)
Output: Bank mapping $\textsc{bank}_a : (\vec{i}, \vec{N}) \mapsto \phi_a(\vec{i}) \mod \sigma(\vec{N})$, for each array a

1. $\mathcal{C} \leftarrow \{(a(\vec{i}), b(\vec{j})) \mid a(\vec{i}) \parallel_\theta b(\vec{j}) \wedge \vec{i} \ll \vec{j} \wedge \vec{i} \in \mathcal{D}_a \wedge \vec{j} \in \mathcal{D}_b\}$

Banking algorithm

Input: Program (P, θ)
Output: Bank mapping $\textsc{bank}_a : (\vec{i}, \vec{N}) \mapsto \phi_a(\vec{i}) \mod \sigma(\vec{N})$, for each array a

```
1. $\mathcal{C} \leftarrow \{(a(\vec{i}), b(\vec{j})) \mid a(\vec{i}) \parallel_\theta b(\vec{j}) \wedge \vec{i} \ll \vec{j} \wedge \vec{i} \in \mathcal{D}_a \wedge \vec{j} \in \mathcal{D}_b\}$
2. $d \leftarrow 0$
3. while $\mathcal{C} \neq \emptyset$
    1. $\min_\ll \sigma^d$ coefficients s.t.
       $\textsc{correct}(\mathcal{C}, \phi^d) \wedge \textsc{efficient}(\mathcal{C}, \phi^d, \sigma^d) \wedge \phi^d$ non-constant
    2. $\mathcal{C} \leftarrow \mathcal{C} \cap \{(a(\vec{i}), b(\vec{j})) \mid \phi_a^d(\vec{i}) = \phi_b^d(\vec{j})\}$
    3. $d \leftarrow d + 1$
4. return $\textsc{bank}$
```

$\textsc{correct}(\mathcal{C}, \phi) : (a(\vec{i}), b(\vec{j})) \in \mathcal{C} \wedge \vec{i} \ll \vec{j} \Rightarrow \phi_a(\vec{i}) \leq \phi_b(\vec{j})$
$\textsc{efficient}(\mathcal{C}, \phi, \sigma) : (a(\vec{i}), b(\vec{j})) \in \mathcal{C} \wedge \vec{i} \ll \vec{j} \Rightarrow \phi_b(\vec{j}) - \phi_a(\vec{i}) \leq \sigma(\vec{N})$

Offset contraints

Correctness: enforce distinct offsets for conflicting array cells

$$\text{BANK}_a(\vec{i}) = \text{BANK}_b(\vec{j}) \wedge a(\vec{i}) \bowtie_\theta b(\vec{j}) \wedge (a, \vec{i}) \neq (b, \vec{j}) \Rightarrow \text{OFFSET}_a(\vec{i}) \neq \text{OFFSET}_b(\vec{j})$$

Relaxed as:

$$\phi_a(\vec{i}) = \phi_b(\vec{j}) \wedge a(\vec{i}) \bowtie_\theta b(\vec{j}) \wedge (a, \vec{i}) \neq (b, \vec{j}) \Rightarrow \psi_a(\vec{i}) \ll \psi_b(\vec{j})$$

Efficiency: minimize the number of offsets (into a same bank)

$$\phi_a(\vec{i}) = \phi_b(\vec{j}) \Rightarrow \psi_b(\vec{j}) - \psi_a(\vec{i}) \leq \tau(\vec{N})$$

Again, analogous to affine scheduling:

operation	array cell
dependence	liveness conflict
latency	number of offsets

Experimental setup

Setup:

- VivadoHLS 2019.1
- Target: Kintex 7 FPGA (XC6K70T-FBV676-1)

Benchmarks:

- *Linear algebra:* matvec, matmul
- *Stencils:* jacobi2d, seidel2d
- *Convolutions:* conv2d, canny, gaussian, median, prewitt

Preliminary prototyping, using fkcc

Experimental results

Kernel	Version	Latency	**Speed-up**	Period (ns)	DSP	FF	LUT
matvec	Baseline	532	**10.2**	6.9	320	4799	4423
	With banking	52		6.3	320	67618	13518
matmul	Baseline	1555	**29.9**	6.8	10240	135581	123129
	With banking	52		6.3	10240	196648	152161
conv2d	Baseline	1442	**29.4**	6.1	0	923	4290
	With banking	49		6.9	0	65562	33043
jacobi2d	Baseline	11011	**1.6**	6.1	0	117140	96019
	With banking	6851		7.0	0	192295	137499
seidel2d	Baseline	6914	**2.0**	6.5	0	452	1280
	With banking	3458		6.6	0	574	2903
canny	Baseline	10194	**4.3**	6.8	0	669	1837
	With banking	2355		6.6	0	6616	6085
gaussian	Baseline	3922	**1.7**	5.8	0	449	1012
	With banking	2354		5.8	0	2367	2811
median	Baseline	3362	**1.3**	6.1	0	373	846
	With banking	2522		5.8	0	2367	2501
prewitt	Baseline	3846	**2.0**	6.1	0	371	906
	With banking	1924		6.9	0	2249	2142

- **Trade-off** surface $\leftrightarrow$ performance still to be explored

Conclusion

Contributions:

- A general formalization & algorithm for affine multibanking
- Our approach reduces the number of banks and the maximal bank size, without hindering parallel accesses.
- Promising (but still preliminary) experimental validation

Perspectives:

- Common bank size, minimize each bank size
- Investigate the trade-off circuit size/latency (through tiling?)

Outline

Conclusion

Models and algorithms for polyhedral HLS:

- **Integrated:**
  - DPN, a dataflow intermediate representation cross fertilizing dataflow models and partitioning
  - Benefits: explicit data spilling, natural tuning of arithmetic intensity and parallelism
- **Source-to-source:**
  - Affine multibanking, as a VivadoHLS preprocessing
  - Benefits: general, HLS independent

Major concepts:

- (affine) tiling, **the** key transformation
- dataflow models, **the** key representation

Perspectives

Partial compilation

- Let parameters (parallelism, local footprint) survive the compilation
- Application: HLS/FPGA: tune the parallelism of a circuit
- Challenges: How to parametrize a DPN? What would be a generic parallel process?

Perspectives

Partial compilation

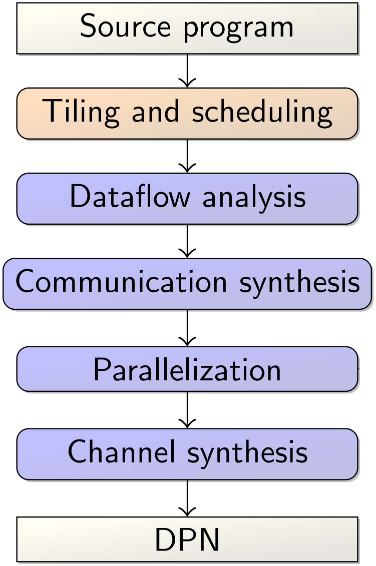
- Let parameters (parallelism, local footprint) survive the compilation
- Application: HLS/FPGA: tune the parallelism of a circuit
- Challenges: How to parametrize a DPN? What would be a generic parallel process?

Lazy compilation

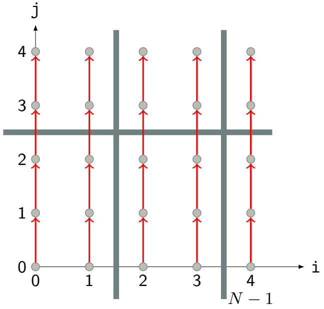
- Complexity: set subtraction, min/max of piecewise affine mappings
- Idea: hide complexity with lazy values, evaluated dynamically (e.g. $R := P \setminus Q$)
- Challenges: How to compose/simplify lazy values? How to rephrase compiler analysis with lazy values?

Questions?

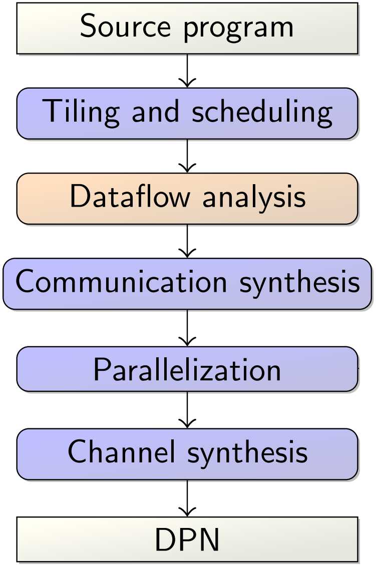
Front-end

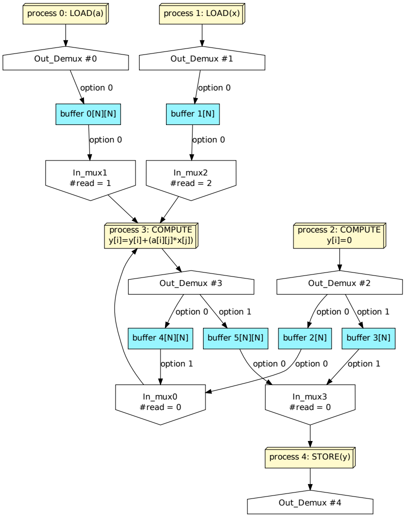


```
for $i$ := 0 to $N-1$
  $y[i] := 0$; //S
  for $j$ := 0 to $N-1$
    $y[i] := y[i] + a[i,j] * x[j]$ //T
```

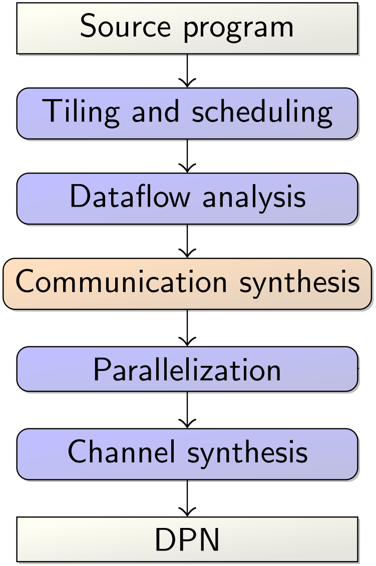


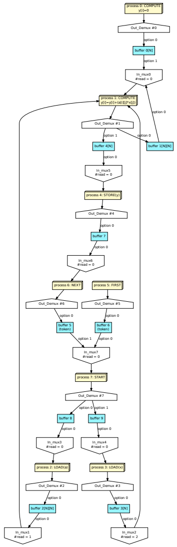
Front-end


Source program
Tiling and scheduling
Dataflow analysis
Communication synthesis
Parallelization
Channel synthesis
DPN

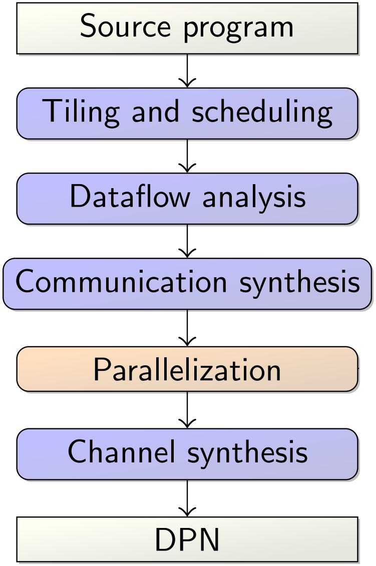

process 0: LOAD(a)
process 1: LOAD(x)
Out_Demux #0
Out_Demux #1
option 0
option 0
buffer 0[N][N]
buffer 1[N]
option 0
option 0
In_mux1
#read = 1
In_mux2
#read = 2
process 3: COMPUTE
y[i]=y[i]+(a[i][j]*x[j])
process 2: COMPUTE
y[i]=0
Out_Demux #3
Out_Demux #2
option 0
option 1
option 0
option 1
buffer 4[N][N]
buffer 5[N][N]
buffer 2[N]
buffer 3[N]
option 1
option 0
option 0
option 1
In_mux0
#read = 0
In_mux3
#read = 0
process 4: STORE(y)
Out_Demux #4

Front-end

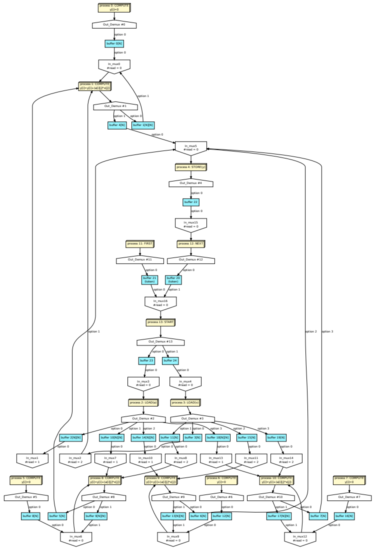

Source program
Tiling and scheduling
Dataflow analysis
Communication synthesis
Parallelization
Channel synthesis
DPN


process 0: COMPUTE
y[i]=0
Out_Demux #0
option 0
buffer 0[N]
option 1
In_mux0
#read = 0
process 1: COMPUTE
y[i]=y[i]+(a[i][j]*x[j])
option 0
Out_Demux #1
option 1
option 0
buffer 4[N]
buffer 1[N][N]
option 0
In_mux5
#read = 0
process 4: STORE(y)
Out_Demux #4
option 0
buffer 7
option 0
In_mux6
#read = 0
process 6: NEXT
process 5: FIRST
Out_Demux #6
Out_Demux #5
option 0
option 0
buffer 5
(token)
buffer 6
(token)
option 1
option 0
In_mux7
#read = 0
process 7: START
Out_Demux #7
option 0
option 1
buffer 8
buffer 9
option 0
option 0
In_mux3
#read = 0
In_mux4
#read = 0
process 2: LOAD(a)
process 3: LOAD(x)
Out_Demux #2
Out_Demux #3
option 0
option 0
buffer 2[N][N]
buffer 3[N]
option 0
option 0
In_mux1
#read = 1
In_mux2
#read = 2

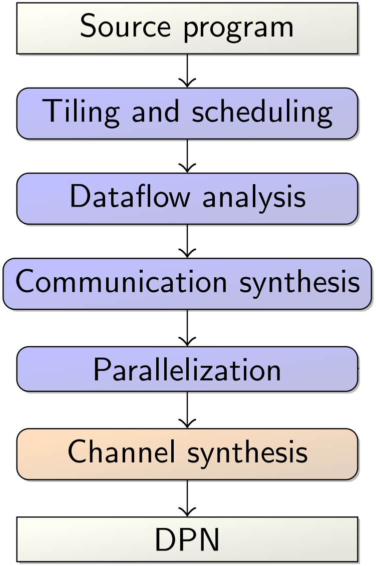
Front-end



Source program
Tiling and scheduling
Dataflow analysis
Communication synthesis
Parallelization
Channel synthesis
DPN



Front-end


Source program
Tiling and scheduling
Dataflow analysis
Communication synthesis
Parallelization
Channel synthesis
DPN