Data-flow Compiler for Stream Computing Hardware on FPGA

RIKEN
Center for Computational Science (R-CCS)
Kentaro Sano

Jan 29, 2020

Outline

- **Introduction**
  - ✓ Why computing with FPGAs? **>> Spatial custom computing!**
- **FPGA cluster prototype**
- **Data-flow stream computing its compiler : SPGen**
- **Upgrade plan**
- **Summary**

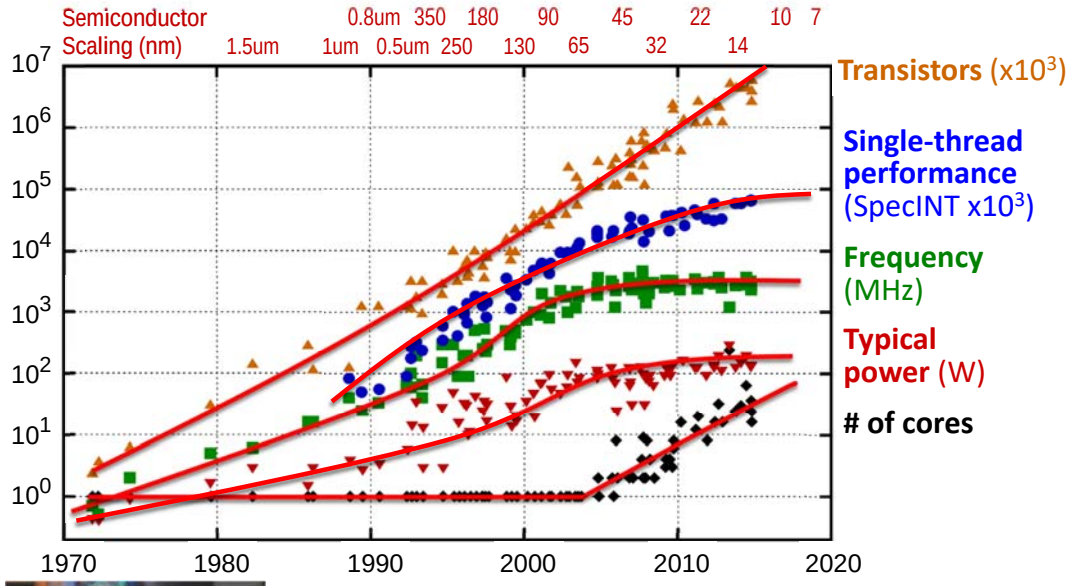
Stratix10 FPGA board (PAC)

Outline

- **Introduction**
  - ✓ Why computing with FPGAs? **>> Spatial custom computing!**
- FPGA cluster prototype
- Data-flow stream computing
  its compiler : SPGen
- Upgrade plan
- Summary

Stratix10 FPGA board (PAC)

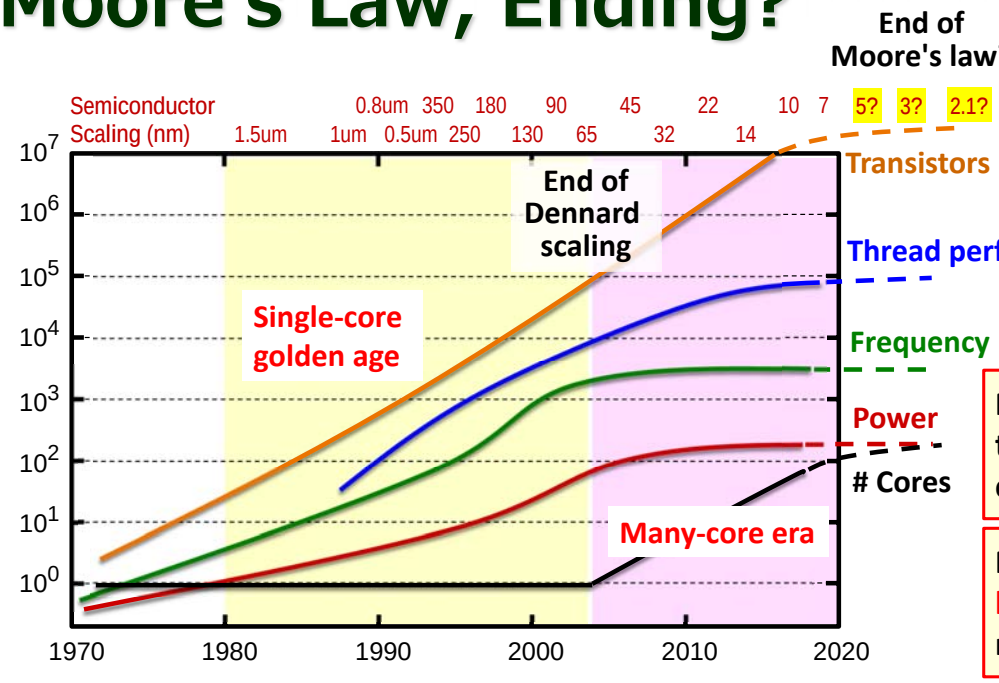
Microprocessor Trend in 40 Years



Original data up to the year 2010 collected and plotted by M. Horowitz, F. Labonte, O. Shacham, K. Olukotun, L. Hammond, and C. Batten,
New plot and data collected for 2010-2015 by K. Rupp, and Trend lines drawn by K.Sano

- ✓ **Moore's law** (Feature size scaling)
  - ➢ *Transistors double every generation.*
- ✓ **Dennard scaling** (MOSFET scaling)
  - ➢ *Same power for x2 transistors at x1.4 frequency*

Moore's Law, Ending?



Moore is slowing down, and about to end.
- Fake scaling
- Increasing cost / Tr. (new Fab costs much)

3D integration saves us for More Moore to give more Transistors.

Power / Tr. not decrease, then Dark silicon problem continues.

Need to allow relatively higher latency to drive more transistors.

- ✓ **3D integration as Saviour in Post-Moore era**
  - ➢ *More transistors available, but Dark silicon continues (no high freq.)*
  - ➢ *Higher latency to drive more transistors (due to no size scaling)*
- ✓ **What architecture is appropriate in Post-Moore era?**

Answer: Spatial Custom Computing

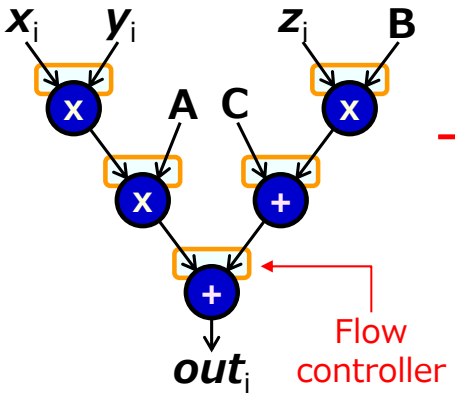
Customization

- ✓ Reconfigurable computing (with FPGAs)

→ **More efficient use of transistor & switching for computation**

Spatial compt. w/ Data-flow

- ✓ Flow instead of cycles



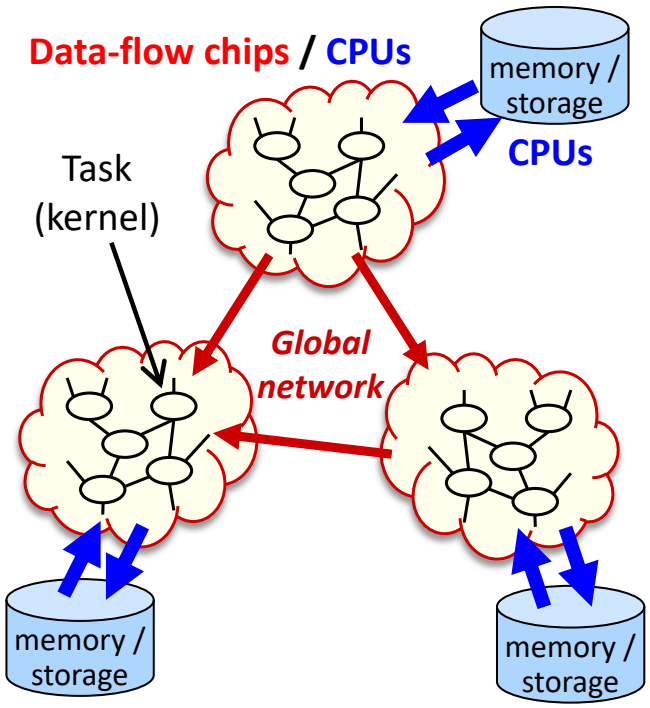
→ **Latency-tolerant architecture w/o cycles**

→ **Data-movement w/o memory access**

We consider data-movement first.

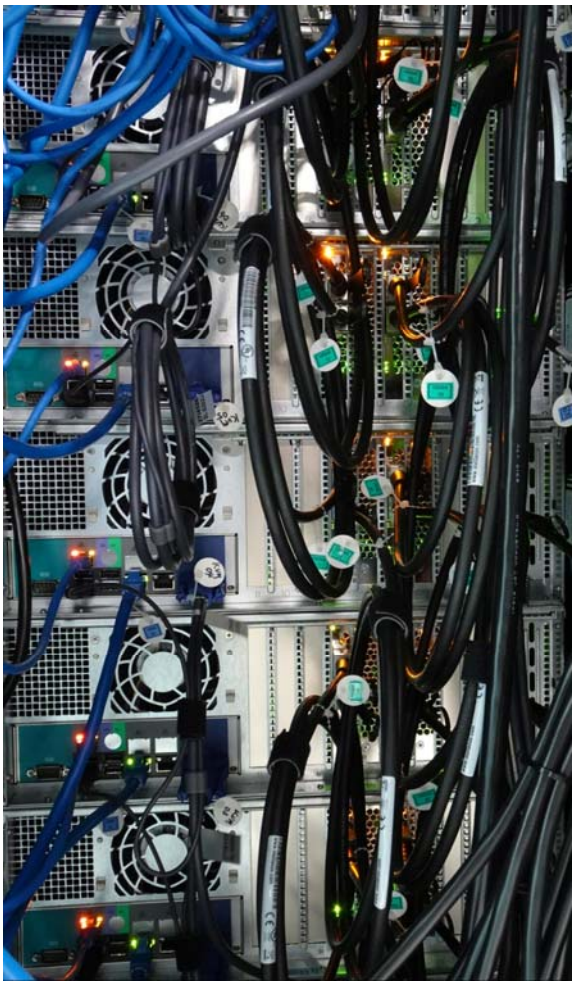
System-wide Spatial Custom Computing

- **Computation**
  = Update of memories
- **Stream computing based on data-flow**
  - ✓ Data-flow circuits on FPGA cluster
  - ✓ Stream data from/to memories through FPGAs
- **Appropriate architecture?**



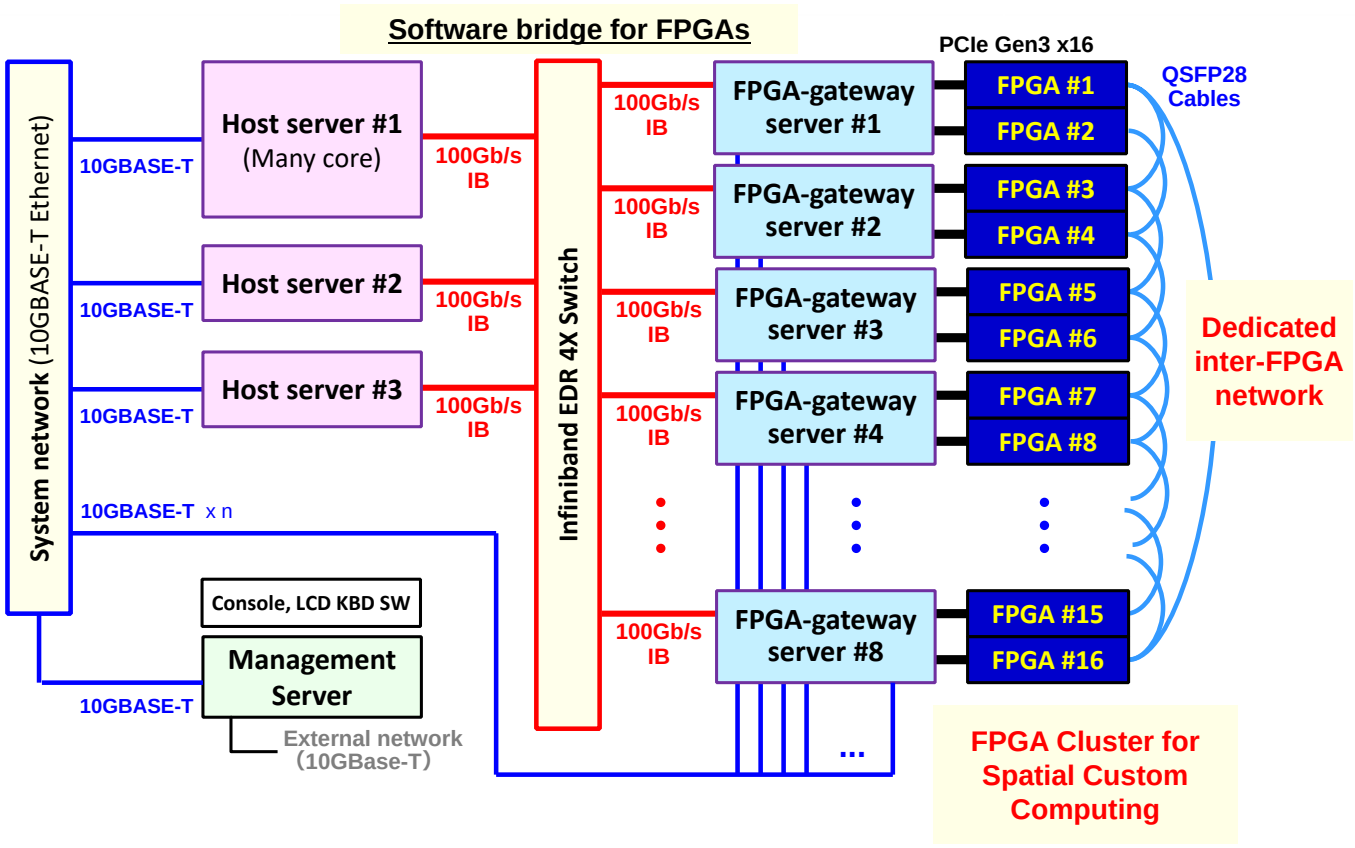
Outline

- Introduction
  - ✓ Why computing with FPGAs?
- **FPGA cluster prototype**
- Data-flow stream computing its compiler : SPGen
- Upgrade plan
- Summary



Experimental Prototype System

Experimental Prototype System



Intel PAC (Programmable Acceleration Card)

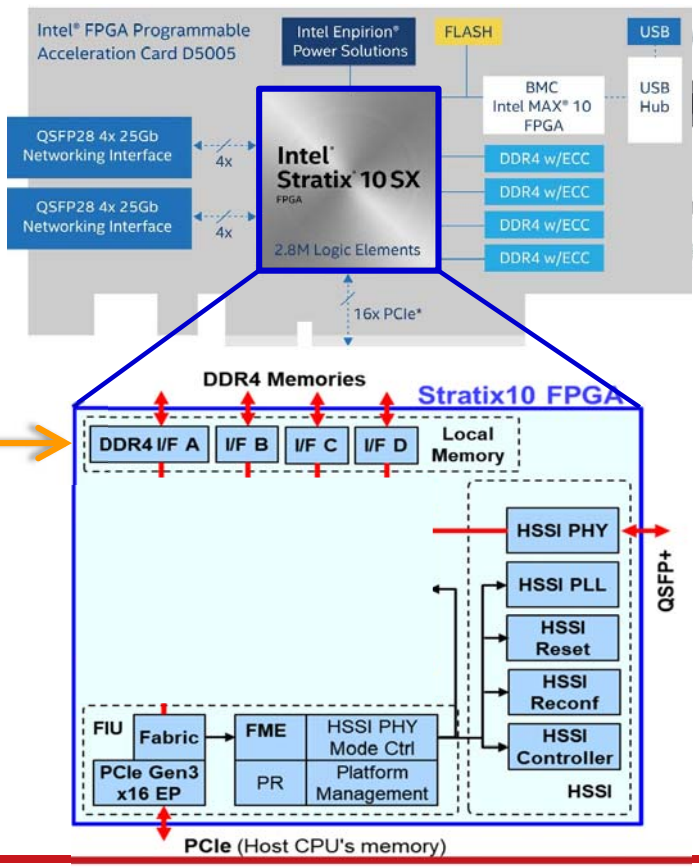
Stratix10: 1SX280HN2F43E2VG

Darby Creek

- ✓ **Stratix10 FPGA** (14nm)
  - ➢ *2753K LEs, 229 Mb BRAMs*
  - ➢ *5760 FP DSPs (7TF @ 600MHz)*
- ✓ 8GB DDR4 x 4ch
- ✓ PCIe Gen3 x16
- ✓ 2x QSFP28 (100Gb/s)

FIM (FPGA Interface Manager)

- ✓ Fixed HW made by Intel



Intel PAC (Programmable Acceleration Card)

Stratix10: 1SX280HN2F43E2VG

Darby Creek

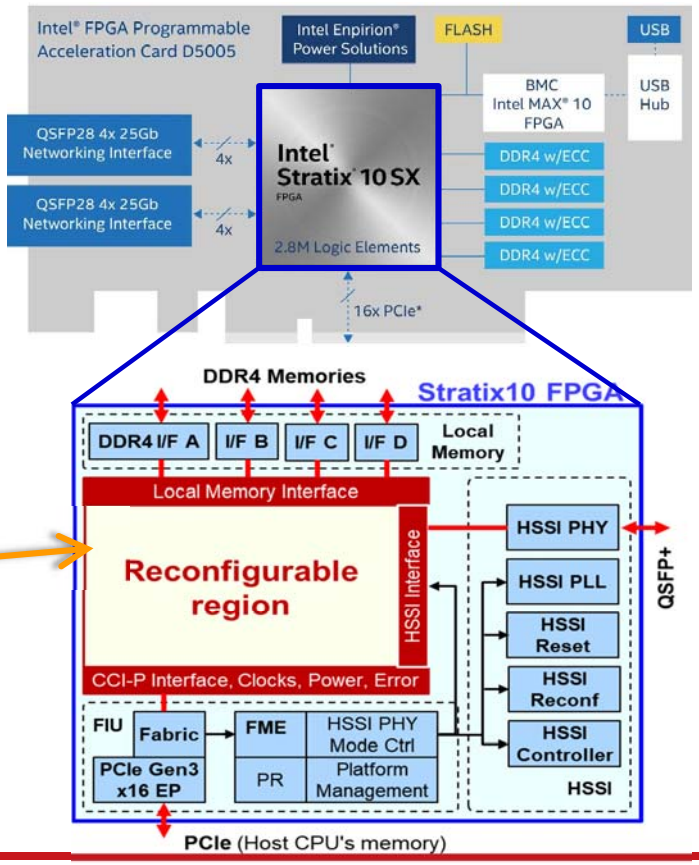
- ✓ **Stratix10 FPGA** (14nm)
  - ➢ *2753K LEs, 229 Mb BRAMs*
  - ➢ *5760 FP DSPs (7TF @ 600MHz)*
- ✓ 8GB DDR4 x 4ch
- ✓ PCIe Gen3 x16
- ✓ 2x QSFP28 (100Gb/s)

FIM (FPGA Interface Manager)

- ✓ Fixed HW made by Intel

AFU (Acceleration Function Unit)

- ✓ Reconfigurable region



Intel PAC (Programmable Acceleration Card)

Stratix10: 1SX280HN2F43E2VG

Darby Creek

- ✓ **Stratix10 FPGA** (14nm)
  - ➢ *2753K LEs, 229 Mb BRAMs*
  - ➢ *5760 FP DSPs (7TF @ 600MHz)*
- ✓ 8GB DDR4 x 4ch
- ✓ PCIe Gen3 x16
- ✓ 2x QSFP28 (100Gb/s)

FIM (FPGA Interface Manager)

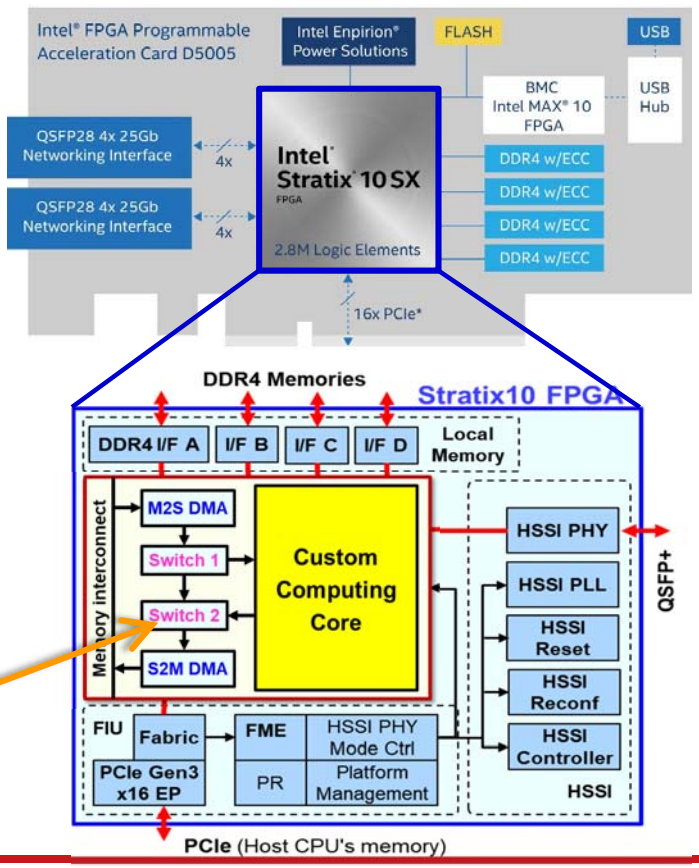
- ✓ Fixed HW made by Intel

AFU (Acceleration Function Unit)

- ✓ Reconfigurable region

AFU Shell : **Our own HW shell**

- ✓ DMAs, Interconnect, and custom computing cores



Outline

- Introduction
  - ✓ Why computing with FPGAs? >> Spatial custom computing!
- FPGA cluster prototype
- **Data-flow stream computing its compiler : SPGen**
- Upgrade plan
- Summary

Stratix10 FPGA board (PAC)

Data-Flow Stream Computing

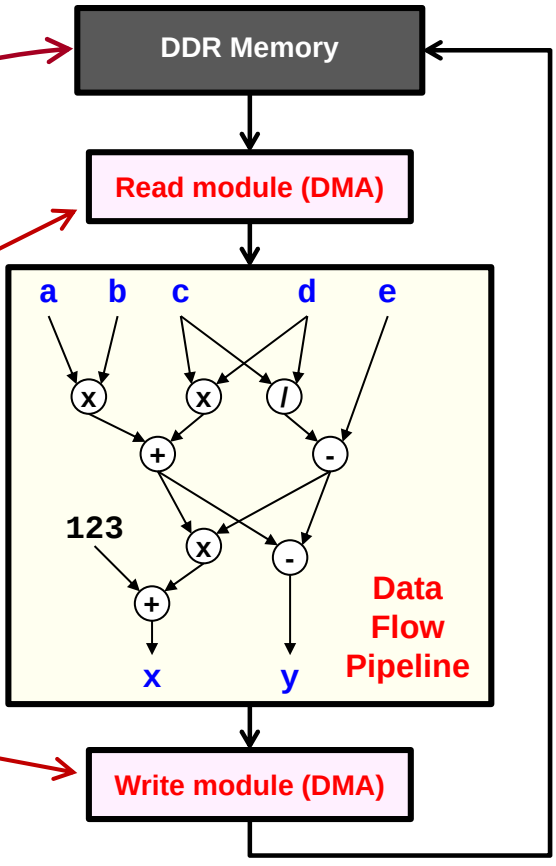
Transform of data structure to stream-friendly one

```
for (int j=0; j++; j<HIGHT)
 {
  for (int i=0; i++; i<WIDTH)
   {
    a = A[j][i];
    b = B[j][i];
    c = C[j][i];
    d = D[j][i];
    e = E[j][i];

    t1  = a*b + c*d;
    t2  = c/d - e;
    x   = t1*t2 + 123;
    y   = t1 - t2;

    X[j][i] = x;
    Y[j][i] = y;
   }
 }
```

Read memory

Compute

Write memory



Loop kernel code

SPGen : Stream Processor Generator

- **Compiler to generate IP core for stream computing**
  - ✓ Based on static data-flow model
  - ✓ Easy and precise description of what programmers want to make as data-flow hardware
- **Stream computing can be programmed easily.**
  - ✓ Own description language : Stream-processing description (SPD)
    - + *Formulae* *abstract description of computing*
    - + *Module-calls* *hierarchical description of structure, and extension of functions*
- **Can be utilized for design space exploration**
  - ✓ Case study : iterative stencil computing

Related work

HDL-like framework

- ✓ Extended HDL by introducing other languages

Bluespec System Verilog (BSV)

C-based

- ✓ IP co … n of existi …

LegUp … *CWB*

Heterog… rk

- ✓ FPGA … s envir…
- ✓ Gene… bitstrea… va

ALTERA OpenCL for FPGA, MaxCompiler

Model-based framework

- ✓ Predefined model for higher productivity in implementing HW

LabVIEW, Matlab HDL coder

SPGen

Computing description with higher abstraction

&

Description capability of other HW than computing

☺ Can design **any HW**

☹ **Not specialized for HPC with FP numbers**

☹ **No system generation. Needs description of HW components other than computing one such like peripheral & interconnects**

☺ **Highly abstracted description** for HW of **HPC systems**

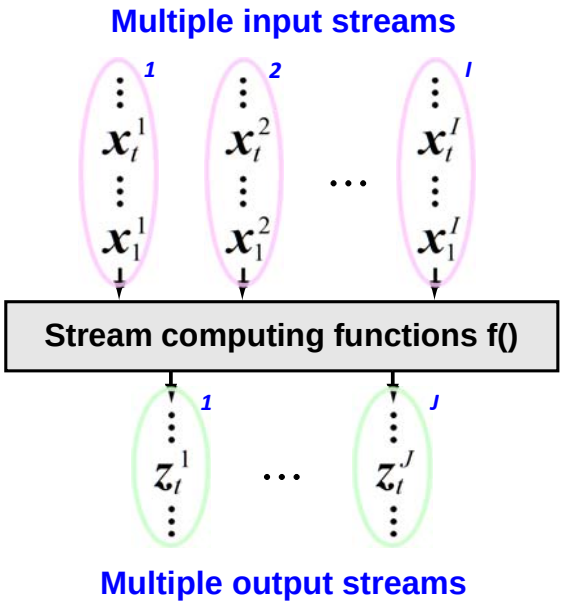
☹ **Available only for specific platforms**

☹ **Black-boxed. Difficult to change and modify**

Stream Computing

Data stream: time series of incoming/outgoing scalar elements

- **Model of stream computing**
  - ✓ Functions to compute a set of elements in multiple input streams
  - ✓ Output multiple streams
- **Various types of computing as stream computing (pipeline)**
  - ✓ ex) signal & image processing
  - ✓ ex) iterative stencil computing
  - ✓ ex) Full N-body computing
- **We target Stream Computing.**
  - ✓ Good at DDR memory access
  - ✓ Higher throughput by pipelining



Requirement for SPGen

- **Users can use SPGen w/o awareness of HW**
  - ✓ Not necessary to know pipelining, clocks, resets, …

Static data-flow architecture, automatic pipelining of DFG

- **Users should simply write just formulae, but can implement other processing.**

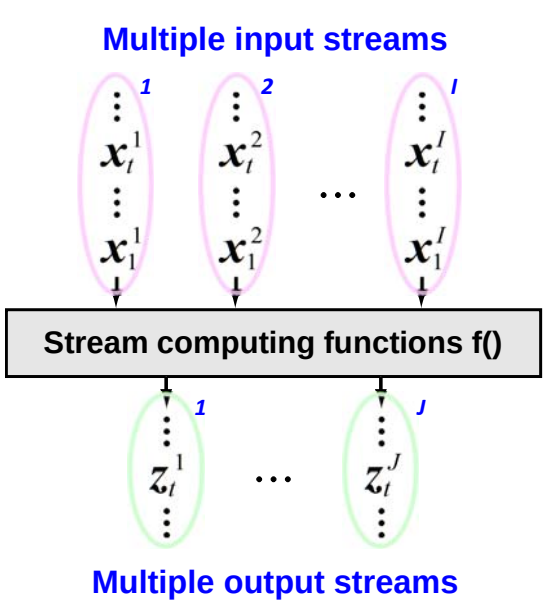
Stream description description format (SPD)

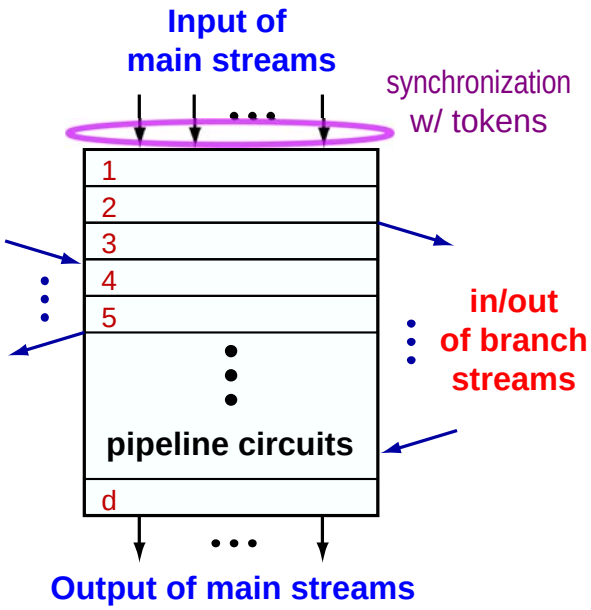
DFG node : formulae or HDL module

- **Hierarchical description allowed for large/complex HW with efficient reuse**

HW model allowing sub DFG to be node of DFG

Hardware Model for Stream Computing





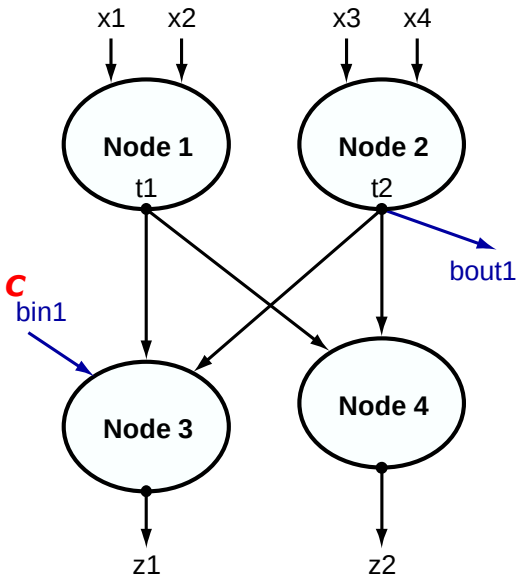
Stream computing

$$v_i^{\text{out}} = f\left(v_{i-M}^{\text{in}}, \ldots, v_i^{\text{in}}, \ldots, v_{i+N}^{\text{in}}\right)$$

Hardware model

(for DFG node)

Example DFG of Stream Computing



Data-flow graph (DFG)

- ✓ Each node : each formula
- ✓ Connection via common variable

Formulae

- ✓ Single assignment statement

Node 1 $t1 = x1 \times x2$

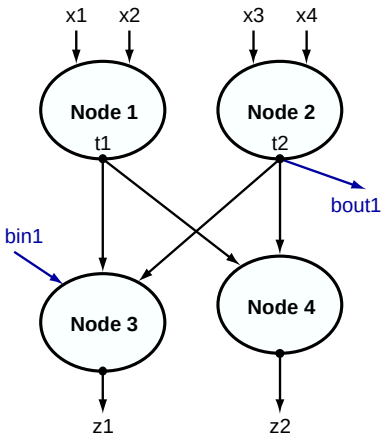
Node 2 $t2 = x3 + x4$

Node 3 $z1 = t1 - t2 \times c$

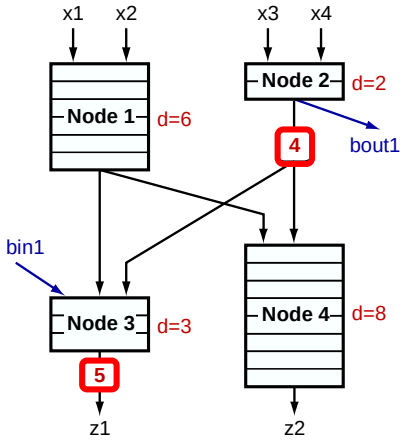
Node 4 $z2 = t1 \;/\; t2$

Generate Pipeline for DFG

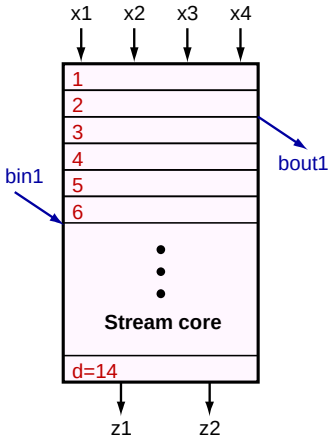
- **Stream-computing core from DFG**
  - ✓ Generate pipelined module for each node
  - ✓ Equalize delays of all paths by inserting delay nodes
  - ✓ Finally the entire DFG is pipelined (satisfying HW model).



DFG



Node pipelines and delay nodes

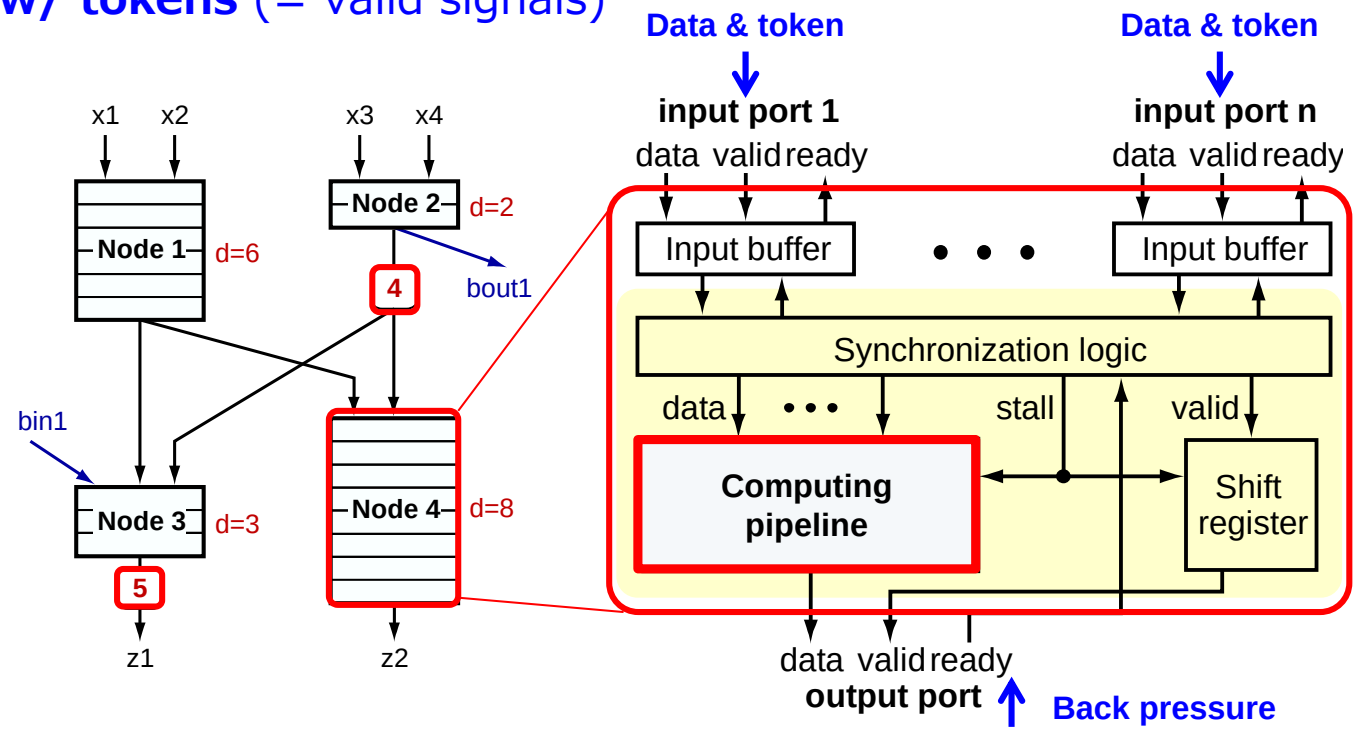


Stream computing core

Logic for Synchronization w/ Tokens

Completely localized control at node

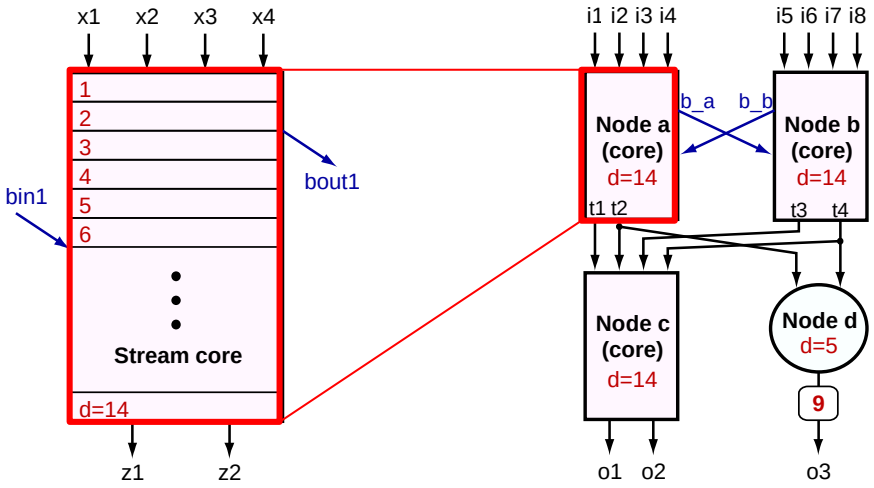
w/ tokens (= valid signals)



Hierarchical Design

Generated core can be used as a Node.

- ✓ *Examples) low-level: local-data path of core*
  high-level: global array structure with cores



Stream computing core
(generated by SPGen)

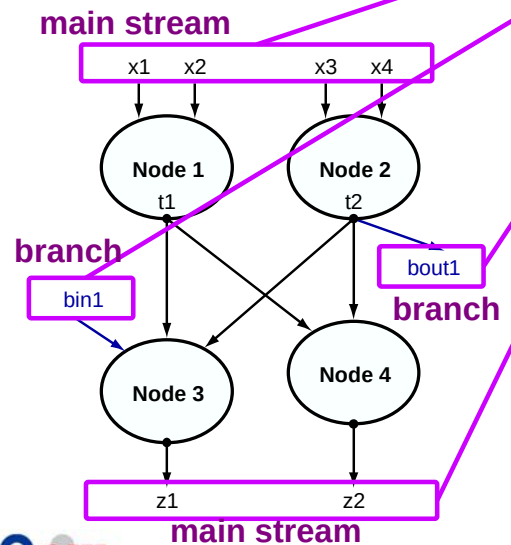
Larger-scale hardware with cores interconnected

Stream Processing Description (SPD)

Intuitive description with formulae and module calls

Definition of Core input/output

```
Name        Core:                          # core name
Main_In     {main_i::x1,x2,x3,x4};         # main stream in
Main_Out    {main_o::z1,z2};               # main stream out
Branch_In   {brch_i::bin1};                # branch inputs
Branch_Out  {brch_o::bout1};               # branch outputs

Param       cnst = 123.456789;             # define parameter
EQU  Node1,   t1 = x1 * x2;                # equation (Node1)
EQU  Node2,   t2 = x3 + x4;                # equation (Node2)
EQU  Node3,   z1 = t1 - t2 * bin1;         # equation (Node3)
EQU  Node4,   z2 = t1 / t2 + cnst;         # equation (Node4)
DRCT        (bout1) = (t2);                # port connection
```



Node 1 $t1 = x1 \times x2$

Node 2 $t2 = x3 + x4$

Node 3 $z1 = t1 - t2 \times c$

Node 4 $z2 = t1 \;/\; t2$

Definition of computing (nodes)

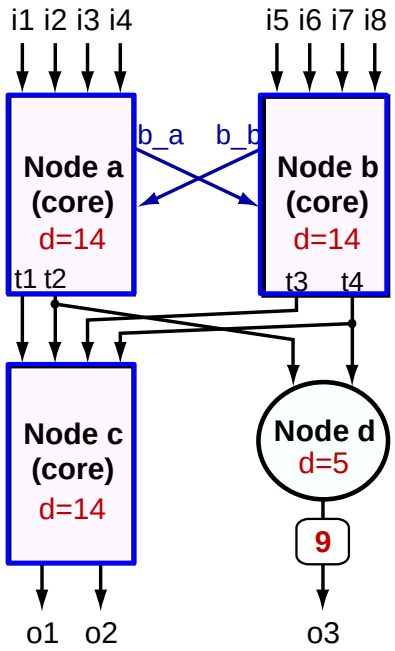
All operations in single-precision FP.

HDL Module Call

Use existing modules with such description like function call

```
(main stream)(branch) = MODULE_NANE (main stream)(branch)
```

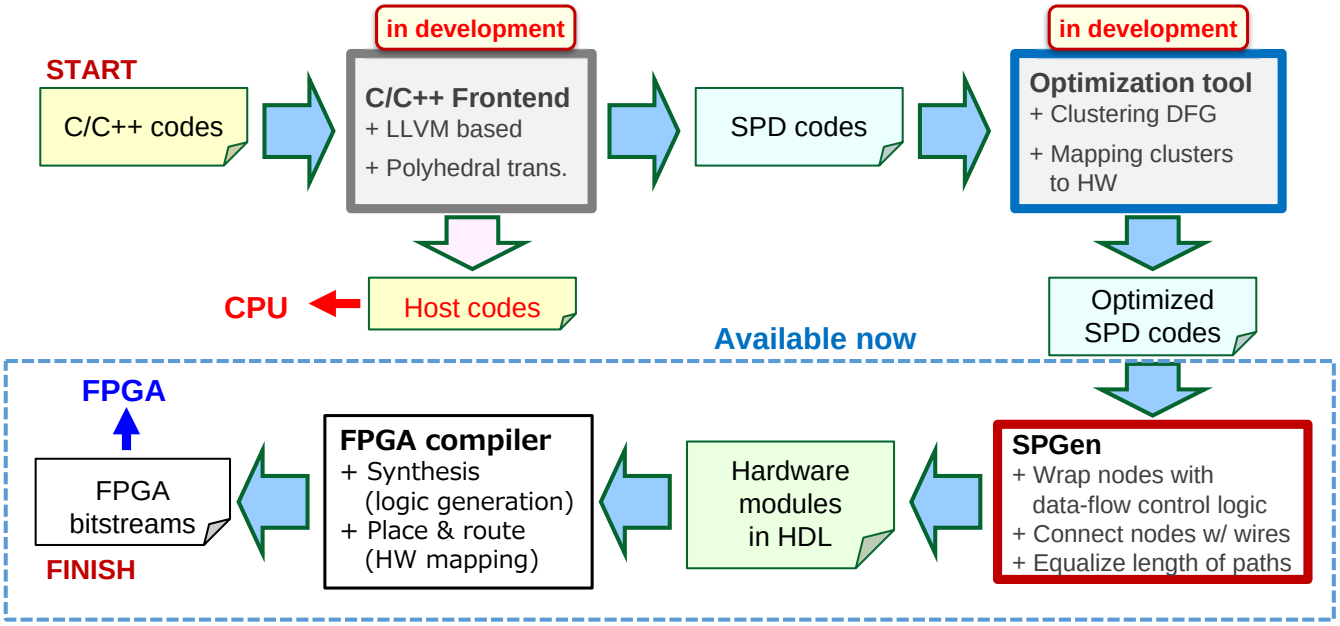
```
Name          Array;
Main_In       {main_i::i1,i2,i3,i4,i5,i6,i7,i8};
Main_Out      {main_o::o1,o2,o3};

HDL Node_a, 14, (t1,t2)(b_a) = Core(i1,i2,i3,i4)(b_b);
HDL Node_b, 14, (t3,t4)(b_b) = Core(i5,i6,i7,i8)(b_a);
HDL Node_c, 14, (o1,o2)      = Core(t1,t2,t3,t4);
EQU Node_d,          o3      = t2 * t4;
```



HDL Module Library

- **Common libraries for primitives**
  - ✓ *Sync. multiplexor* — out = mMux_syn(in1, in2, sel[0])
  - ✓ *Comparator* — out = mCompare(in1, in2)
  - ✓ *Elimination of stream* — mEliminate(in)
  - ✓ *Constance generation* — out = mConst(), <.pConstData(32'h01234)>
  - ✓ *Delay* — out = mDelay(in), <.pDelay(40)>
  - ✓ *Stencil buffer* — out = mStencilBuff_2D(in, sop, eop)
  - ✓ *Forwarding stream* — out = mStreamFwd(in), <.pFwdCycle(12)>
  - ✓ *Backwarding stream* — out = mStreamBwd(in), <.pBwdCycle(12)>
  - ✓ *etc...*
- **You can add your own HDL modules for your application.**
  - ✓ ex) 3D stencil buffer
  - ✓ ex) buffer for convolutional computing
  - ✓ etc.

SPGen Tool Chain



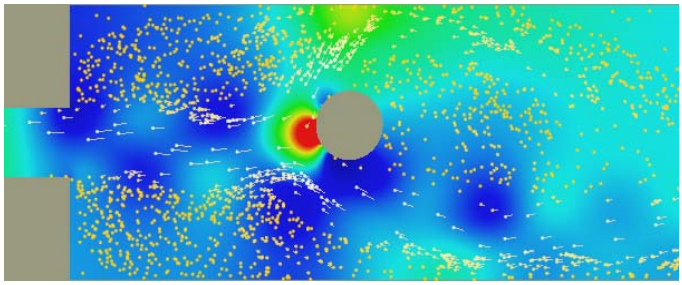
Stream Computing with Stratix10 FPGA

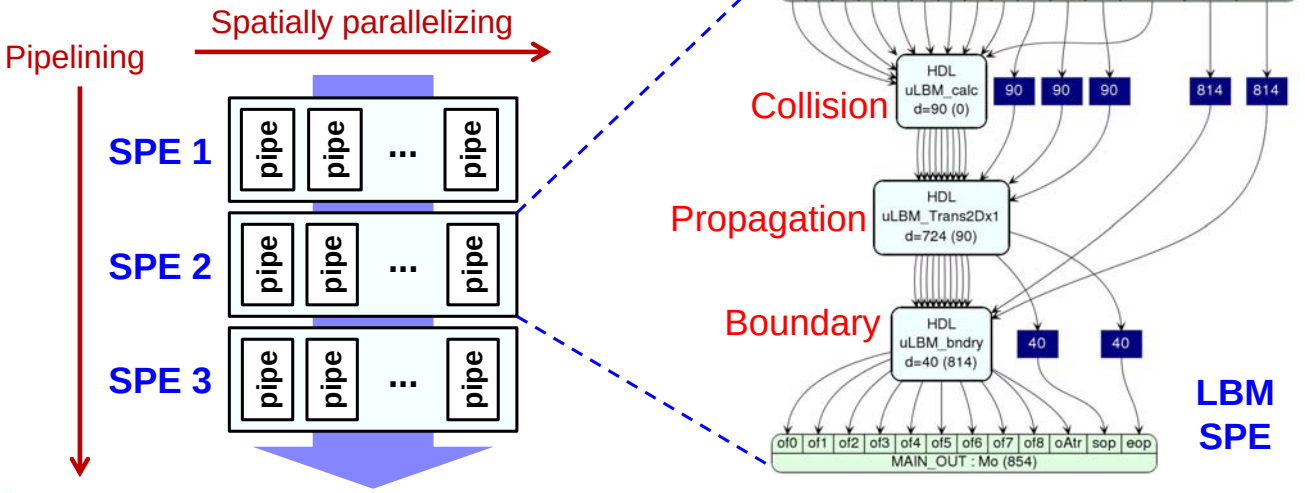
Fluid dynamics simulation

- ✓ **LBM** (lattice Boltzmann method)

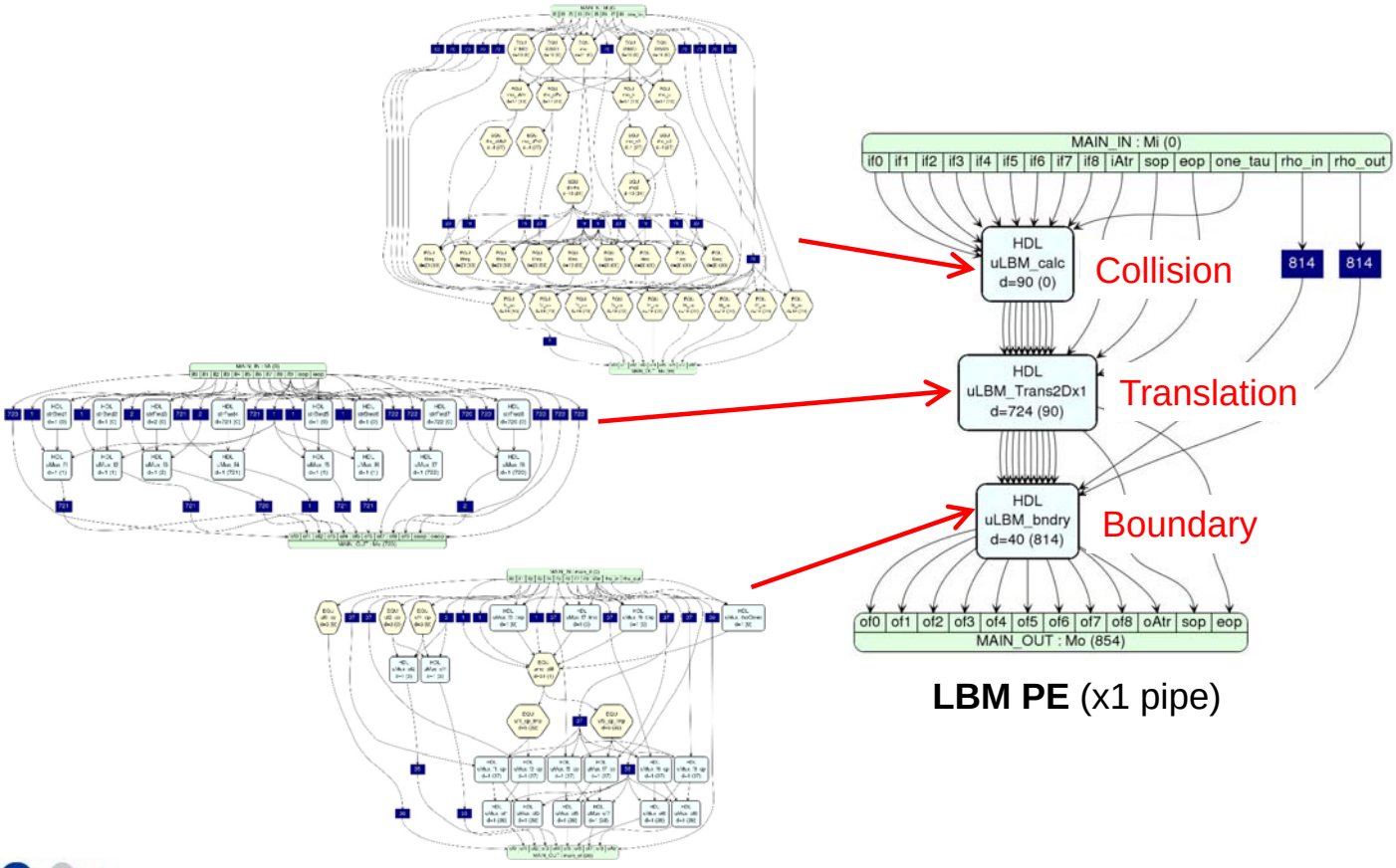
Array architecture of SPEs

- ✓ **S**tream **P**rocessing **E**lements
- ✓ Generated by SPGen compiler



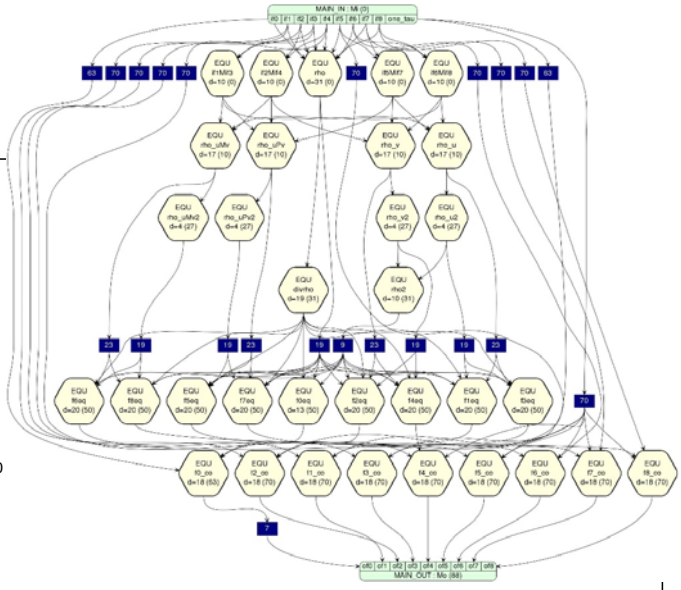


Modules of LBM Processing Element



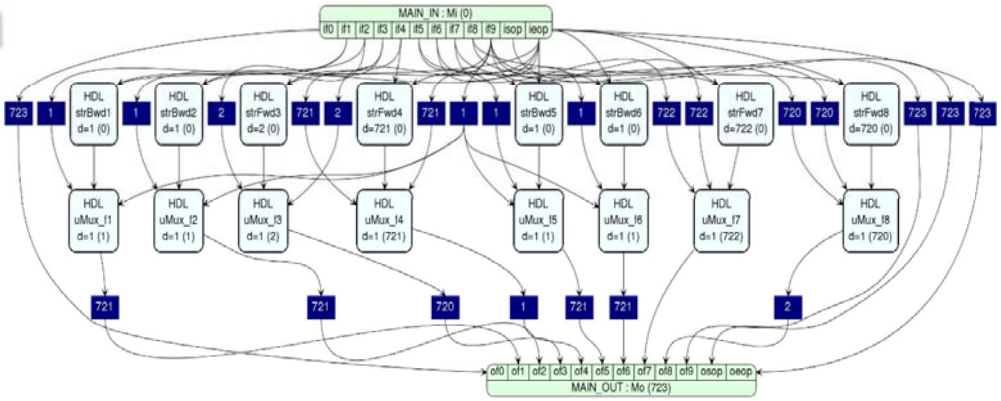
LBM PE (x1 pipe)

Collision Module



```
Name mLBM_calc;
Main_In   {Mi::if0,if1,if2,if3,if4,if5,if6,if7,if8, one_tau};
Main_Out  {Mo::of0,of1,of2,of3,of4,of5,of6,of7,of8};

Param P_2_3          = 0.66666666666667;
Param P_1_6          = 0.16666666666667;
Param one_two        = 0.50000000000000;
Param one_three      = 0.33333333333333;
...
Param two_three      = 0.66666666666667;

EQU if5Mif7, if5Mif7 = ( if5 - if7 );
EQU if6Mif8, if6Mif8 = ( if6 - if8 );
EQU if1Mif3, if1Mif3 = ( if1 - if3 );
EQU if2Mif4, if2Mif4 = ( if2 - if4 );
EQU rho,         rho = (((if5+if7) + (if6+if8)) + ((if1+if3) + (if2+if4) + if0
EQU rho_u,     rho_u = ((if5Mif7 - if6Mif8) + if1Mif3);
EQU rho_v,     rho_v = ((if5Mif7 + if6Mif8) + if2Mif4);
EQU rho_uMv, rho_uMv = ( if1Mif3 - if2Mif4) - ((2.0)*if6Mif8);
EQU rho_uPv, rho_uPv = ( if1Mif3 + if2Mif4) + ((2.0)*if5Mif7);

EQU rho_u2,     rho_u2 = ( rho_u   * rho_u   );
EQU rho_v2,     rho_v2 = ( rho_v   * rho_v   );
EQU rho_uPv2, rho_uPv2 = ( rho_uPv * rho_uPv );
EQU rho_uMv2, rho_uMv2 = ( rho_uMv * rho_uMv );
EQU rho2,         rho2 = ( rho_u2  + rho_v2  );
EQU divrho,     divrho = ( 1.0 / rho );
EQU f0eq,       f0eq = ( (four_nine     * rho) -                                                    (two_three      * rho2)   * divrho);
EQU f1eq,       f1eq = ( (one_nine      * rho) + (one_three  * rho_u)   ) + ( ( (one_two   * rho_u2  ) -  (one_six        * rho2) ) * divrho);
EQU f3eq,       f3eq = ( (one_nine      * rho) - (one_three  * rho_u)   ) + ( ( (one_two   * rho_u2  ) -  (one_six        * rho2) ) * divrho);
EQU f2eq,       f2eq = ( (one_nine      * rho) + (one_three  * rho_v)   ) + ( ( (one_two   * rho_v2  ) -  (one_six        * rho2) ) * divrho);
EQU f4eq,       f4eq = ( (one_nine      * rho) - (one_three  * rho_v)   ) + ( ( (one_two   * rho_v2  ) -  (one_six        * rho2) ) * divrho);
EQU f5eq,       f5eq = ( (one_thirtysix * rho) + (one_twelve * rho_uPv) ) + ( ( (one_eight * rho_uPv2) -  (one_twentyfour * rho2) ) * divrho);
EQU f7eq,       f7eq = ( (one_thirtysix * rho) - (one_twelve * rho_uPv) ) + ( ( (one_eight * rho_uPv2) -  (one_twentyfour * rho2) ) * divrho);
EQU f6eq,       f6eq = ( (one_thirtysix * rho) - (one_twelve * rho_uMv) ) + ( ( (one_eight * rho_uMv2) -  (one_twentyfour * rho2) ) * divrho);
EQU f8eq,       f8eq = ( (one_thirtysix * rho) + (one_twelve * rho_uMv) ) + ( ( (one_eight * rho_uMv2) -  (one_twentyfour * rho2) ) * divrho);

EQU f0_co,   of0 = if0 - one_tau * (if0 - f0eq);
EQU f1_co,   of1 = if1 - one_tau * (if1 - f1eq);
EQU f2_co,   of2 = if2 - one_tau * (if2 - f2eq);
EQU f3_co,   of3 = if3 - one_tau * (if3 - f3eq);
EQU f4_co,   of4 = if4 - one_tau * (if4 - f4eq);
EQU f5_co,   of5 = if5 - one_tau * (if5 - f5eq);
EQU f6_co,   of6 = if6 - one_tau * (if6 - f6eq);
EQU f7_co,   of7 = if7 - one_tau * (if7 - f7eq);
EQU f8_co,   of8 = if8 - one_tau * (if8 - f8eq);
```

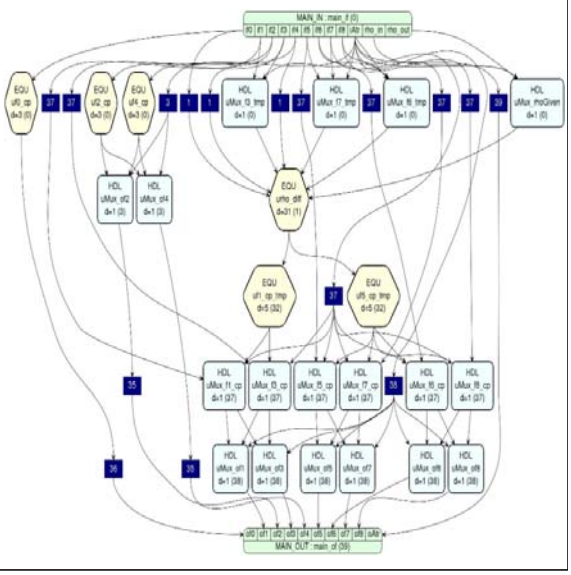
Translation Module




```
Name mLBM_Trans2Dx1;
Main_In   {Mi::if0,if1,if2,if3,if4,if5,if6,if7,if8,if9, isop,ieop};
Main_Out  {Mo::of0,of1,of2,of3,of4,of5,of6,of7,of8,of9, osop,oeop};

DRCT (of0)                  = (if0);
HDL strBwd1,   1, (of1_tr)() = mStreamBackward(if1,ieop[0])(), <.pConstWord(0),.pBwdCycles(1)>     # +1
HDL strBwd2,   1, (of2_tr)() = mStreamBackward(if2,ieop[0])(), <.pConstWord(0),.pBwdCycles(720)>   # +L
HDL strBwd5,   1, (of5_tr)() = mStreamBackward(if5,ieop[0])(), <.pConstWord(0),.pBwdCycles(721)>   # +(L+1)
HDL strBwd6,   1, (of6_tr)() = mStreamBackward(if6,ieop[0])(), <.pConstWord(0),.pBwdCycles(719)>   # +(L-1)
HDL strFwd3,   2, (of3_tr)() = mStreamForward(if3,ieop[0])(),  <.pConstWord(0),.pFwdCycles(1)>     # -1
HDL strFwd4, 721, (of4_tr)() = mStreamForward(if4,ieop[0])(),  <.pConstWord(0),.pFwdCycles(720)>   # -L
HDL strFwd7, 722, (of7_tr)() = mStreamForward(if7,ieop[0])(),  <.pConstWord(0),.pFwdCycles(721)>   # -(L+1)
HDL strFwd8, 720, (of8_tr)() = mStreamForward(if8,ieop[0])(),  <.pConstWord(0),.pFwdCycles(719)>   # -(L-1)

## Prevent from divergence related to contacting solid
HDL uMux_f1,   1, (of1)()    = mMux_sync(of1_tr, if1, if9[3])(); ## if9 : attribute : of1_tr for 0, if1 for 1
HDL uMux_f2,   1, (of2)()    = mMux_sync(of2_tr, if2, if9[4])(); ## if9 : attribute
HDL uMux_f3,   1, (of3)()    = mMux_sync(of3_tr, if3, if9[1])(); ## if9 : attribute
HDL uMux_f4,   1, (of4)()    = mMux_sync(of4_tr, if4, if9[2])(); ## if9 : attribute
HDL uMux_f5,   1, (of5)()    = mMux_sync(of5_tr, if5, if9[7])(); ## if9 : attribute
HDL uMux_f6,   1, (of6)()    = mMux_sync(of6_tr, if6, if9[8])(); ## if9 : attribute
HDL uMux_f7,   1, (of7)()    = mMux_sync(of7_tr, if7, if9[5])(); ## if9 : attribute
HDL uMux_f8,   1, (of8)()    = mMux_sync(of8_tr, if8, if9[6])(); ## if9 : attribute

DRCT (of9)                  = (if9);   # Attribute
DRCT (osop,oeop)            = (isop,ieop);  # sop,eop
```

Boundary Module

```
Name mLBM_boundary;
Main_In   {main_if::if0,if1,if2,if3,if4,if5,if6,if7,if8,iAtr, rho_in, rho_out};
Main_Out  {main_of::of0,of1,of2,of3,of4,of5,of6,of7,of8,oAtr};

Param P_2_3             = 0.66666666666666666666;  # 2/3
Param P_1_6             = 0.16666666666666666666;  # 1/6

#--------------------------------------------------------------------------#
# Const-press stage (only left and right boundaries are implemented.
#--------------------------------------------------------------------------#
HDL uMux_f6_tmp,   1, (f6_tmp)()   = mMux_sync(if6, if8, iAtr[11])();
HDL uMux_f3_tmp,   1, (f3_tmp)()   = mMux_sync(if3, if1, iAtr[11])();
HDL uMux_f7_tmp,   1, (f7_tmp)()   = mMux_sync(if7, if5, iAtr[11])();
HDL uMux_rhoGiven, 1, (rhoGiven)() = mMux_sync(rho_in, rho_out, iAtr[11])();

EQU urho_diff,          rho_diff   = ( rhoGiven - ((if0 + if2) + (f3_tmp + if4) + (f6_tmp + f7_tmp)) );
EQU uf1_cp_tmp,         f1_cp_tmp  = ( P_2_3 * rho_diff );
EQU uf5_cp_tmp,         f5_cp_tmp  = ( P_1_6 * rho_diff );

EQU uf0_cp,             f0_cp      = if0;
EQU uf2_cp,             f2_cp      = if2;
EQU uf4_cp,             f4_cp      = if4;

HDL uMux_f1_cp,    1, (f1_cp)()    = mMux_sync(if1, f1_cp_tmp, iAtr[9])();
HDL uMux_f5_cp,    1, (f5_cp)()    = mMux_sync(if5, f5_cp_tmp, iAtr[9])();
HDL uMux_f8_cp,    1, (f8_cp)()    = mMux_sync(if8, f5_cp_tmp, iAtr[9])();
HDL uMux_f3_cp,    1, (f3_cp)()    = mMux_sync(if3, f1_cp_tmp, iAtr[11])();
HDL uMux_f7_cp,    1, (f7_cp)()    = mMux_sync(if7, f5_cp_tmp, iAtr[11])();
HDL uMux_f6_cp,    1, (f6_cp)()    = mMux_sync(if6, f5_cp_tmp, iAtr[11])();

#--------------------------------------------------------------------------#
# Bounce-back stage
#--------------------------------------------------------------------------#
DRCT (of0) = (f0_cp);
HDL uMux_of3,      1, (of3)()      = mMux_sync(f3_cp, f1_cp, iAtr[1])();
HDL uMux_of4,      1, (of4)()      = mMux_sync(f4_cp, f2_cp, iAtr[2])();
HDL uMux_of1,      1, (of1)()      = mMux_sync(f1_cp, f3_cp, iAtr[3])();
HDL uMux_of2,      1, (of2)()      = mMux_sync(f2_cp, f4_cp, iAtr[4])();
HDL uMux_of7,      1, (of7)()      = mMux_sync(f7_cp, f5_cp, iAtr[5])();
HDL uMux_of8,      1, (of8)()      = mMux_sync(f8_cp, f6_cp, iAtr[6])();
HDL uMux_of5,      1, (of5)()      = mMux_sync(f5_cp, f7_cp, iAtr[7])();
HDL uMux_of6,      1, (of6)()      = mMux_sync(f6_cp, f8_cp, iAtr[8])();
#--------------------------------------------------------------------------#

DRCT (oAtr) = (iAtr);
```



Performance and Power (LBM2D, S10)

10% more PEs are expected. We target **450MHz for 3 TF+** performance.

Outline

- Introduction
  - ✓ Why computing with FPGAs? >> **Spatial custom computing!**
- FPGA cluster prototype
- Data-flow stream computing
  its compiler : SPGen
- **Upgrade plan**
- Summary

Stratix10 FPGA board (PAC)

Upgrade Plan for Multiple-Precision

- **Present:**
  - ✓ Numerical ops: single-precision FP / Intel's FP core (32-bit)
  - ✓ Bit-wise ops: 32-bit word
- **Upgrade plan:**
  - ✓ Introduce various bit widths:
    - + Predefined 8 bits, 16 bits, 32 bits, ... or
    - + User-programmable for arbitrary widths
  - ✓ Introduce "types" and their extension
    - + We need to implement "cast" in hardware...
  - ✓ Introduce HLS-synthesized DMA (memory access module)
    - + Arbitrary pattern of memory access

Outline

- Introduction
  - ✓ Why computing with FPGAs? >> **Spatial custom computing!**
- FPGA cluster prototype
- Data-flow stream computing
  its compiler : SPGen
- Upgrade plan
- **Summary**

Stratix10 FPGA board (PAC)

Summary

- **Need to change architecture for post-Moore era**
  - ✓ Solution : Spatial custom computing ?
- **Data-flow stream computing**
- **Compiler: SPGen**
  - ✓ Computing model, Hardware model, and tool flow
  - ✓ Description examples
  - ✓ Preliminary results with Stratix10 FPGA

- **Future work**
  - ✓ Extension for multi-precision computing
  - ✓ Various wit width, various types, their programmability