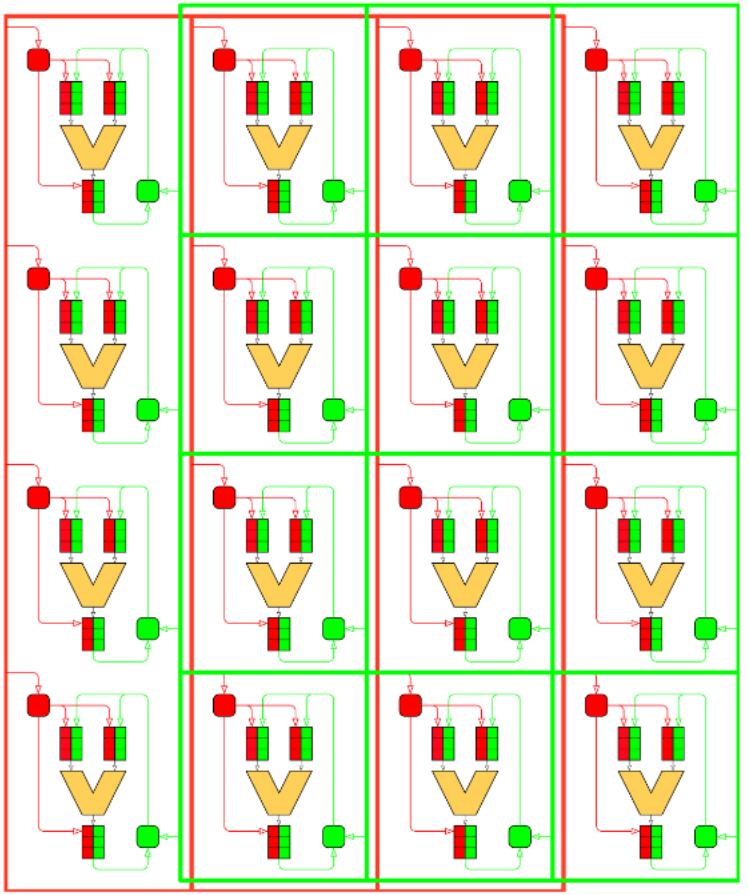
Evaluating Interconnection Networks for Exposed Datapath Architectures

Motivation: SCAD-Machines

- **S**ynchronous **C**ontrolflow **A**synchronous **D**ataflow
- Processing Units with Buffers
  - Operate as soon as inputs are avaiable
- Move instructions move data between buffers
  - Data Transport Network
  - Interconnect *N* PUs with *N* PUs

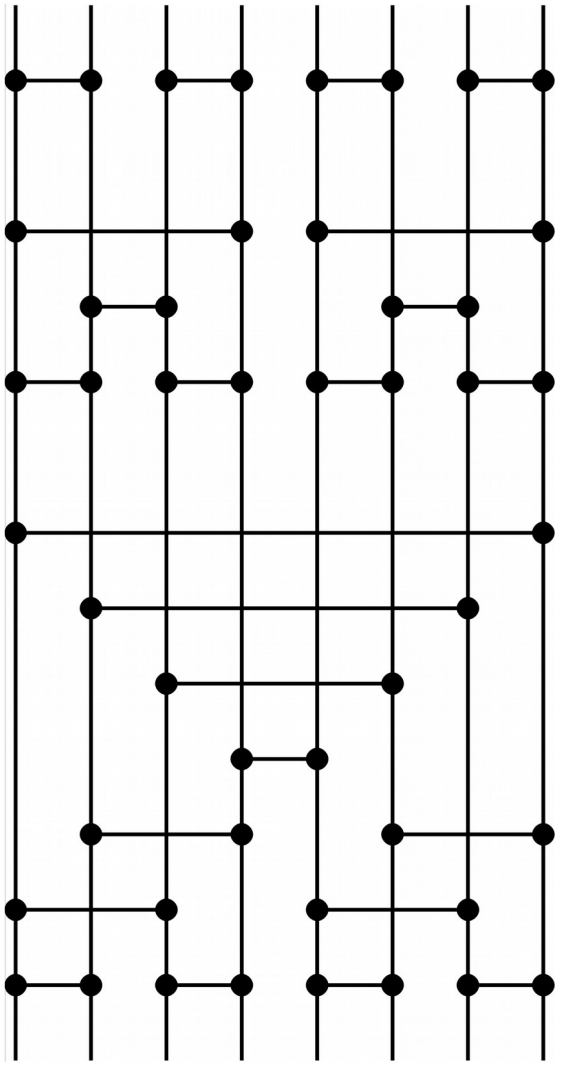


Interconnection Networks

- Classic approaches:
  - Buses
    - Only one move at a time
    - Multiple parallel buses may not scale well
  - Crossbar Switches
    - Matrix with *N* x *N* switches
    - Allows to route N pieces of data from N source to N destinations in parallel
    - Needs global configurations for switches
- Better: Interconnection Networks

Sorting Networks

- Structure
  - Compare and Swap units
    - Switches with attached control logic
  - Interconnected by wires
- Transport/Sort all inputs in parallel
- Routing decisions are local
- Interconnection Networks
  - Transport Address-Data-Tuples
  - Sort by address

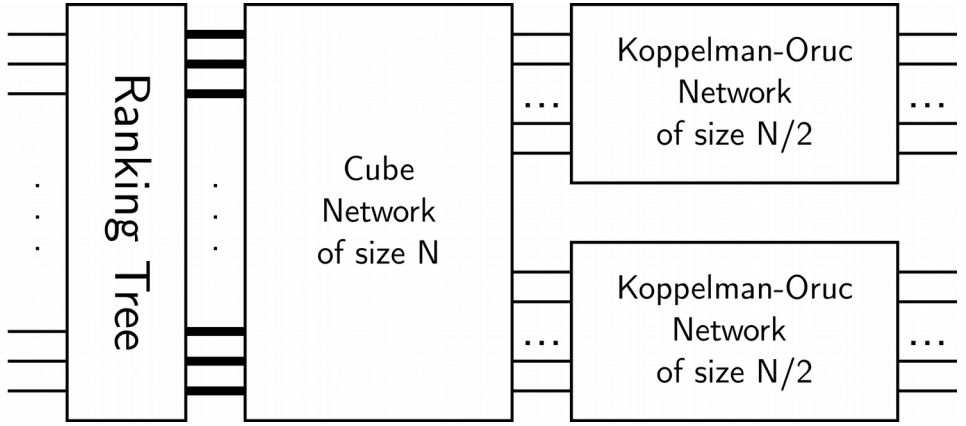


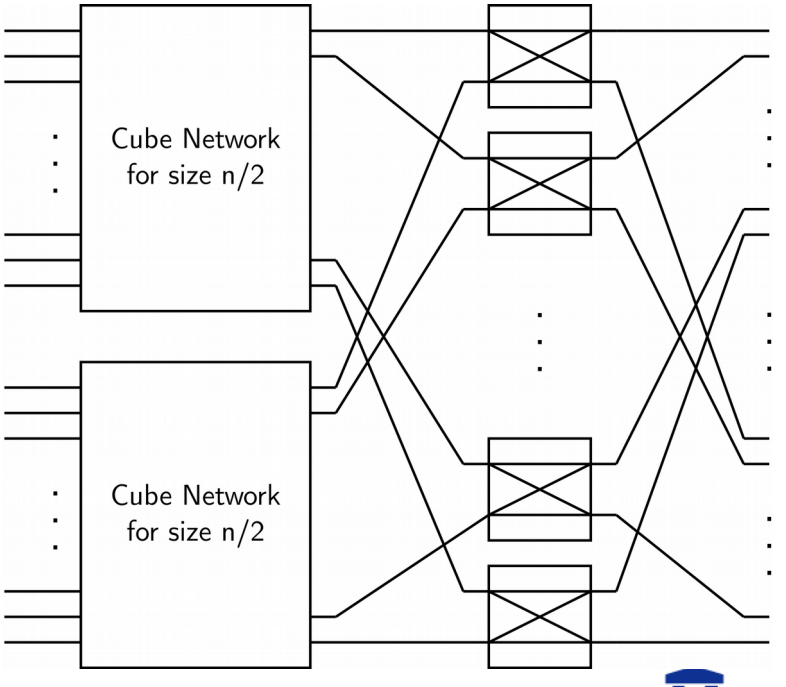
Koppelmann-Oruç Network

- Computes ranks of input addresses
  - Prefix sum over address MSBs
  - Uses tree of adders
- Routed using Cube Network
  - Switch position based on

    rank LSB and address MSBs

    $$p = (a_u \wedge \neg r_l) \vee (\neg a_u \wedge r_u)$$
  - Rank LSBs are dropped after each column
  - Address MSBs are dropped on recursion



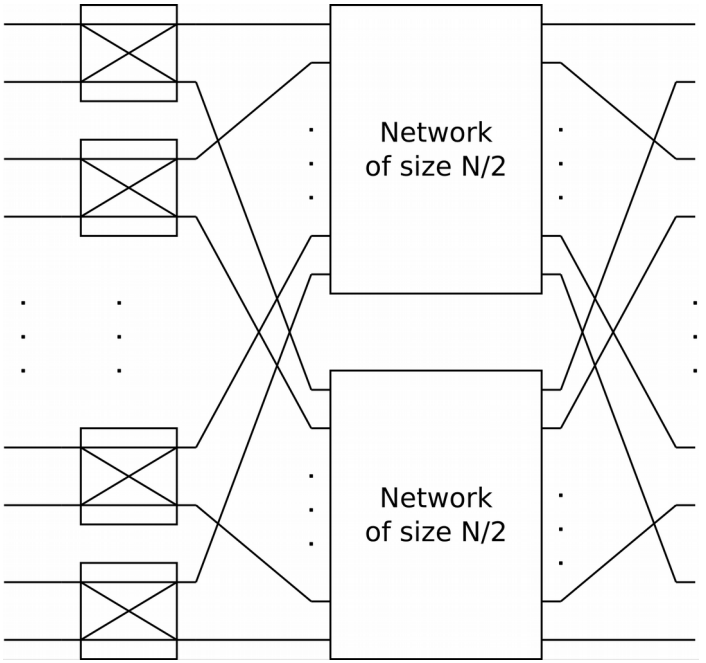


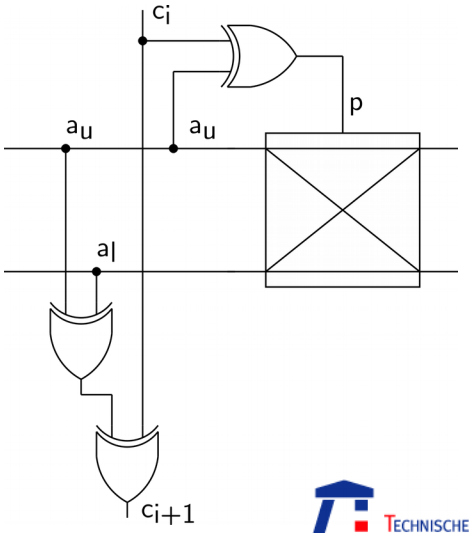
Narasimha's network

- Routing based on the address MSBs and a „carry“

$$p = c_i \oplus a_u \qquad c_{i+1} = (a_u \oplus a_l) \oplus c_i$$

- Carry spans the entire first column of the network
- Address MSBs are dropped before recursion





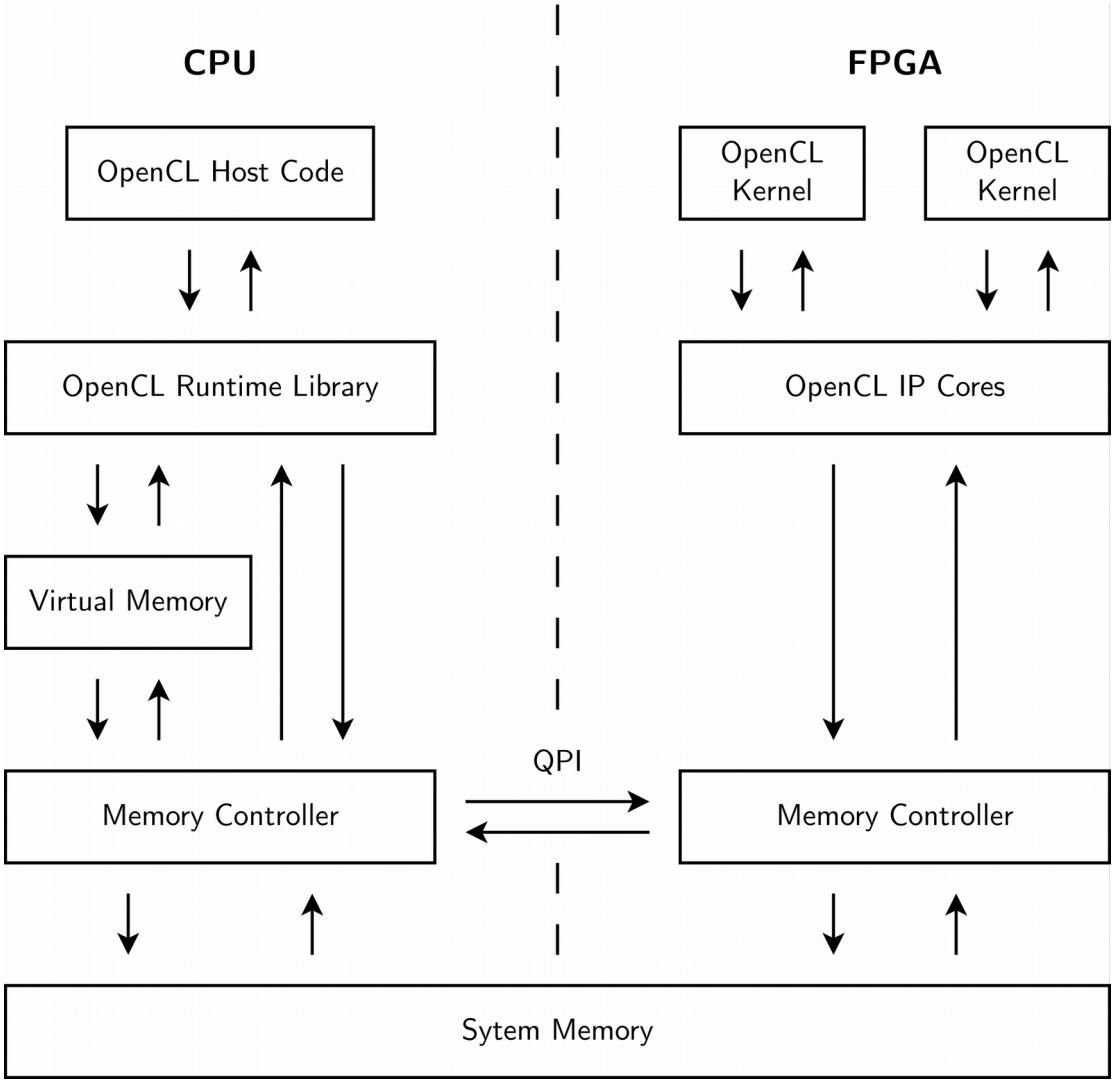
Intel HARP

- Hardware Accelerator Research Project
- Xeon CPU with FPGA based accelerator
- hardware platform and software framework
- Primarily aimed at Datacenter applications
- FPGA and CPU share the same Memory
  - Low latency data exchange
- Programmable in VHDL OpenCL

OpenCL

- Programming language and runtime environment
- Based on the C language and C99 standard
  - Removes: Recursive functions, pointer arithmetic, dynamic memory management
  - Adds: vector operations and vector datatypes
  - Memory segmented in regions
- Runs Kernels on Compute Units attached to a host system
- Multiple kernels can be run in parallel

Intel HARP and OpenCL

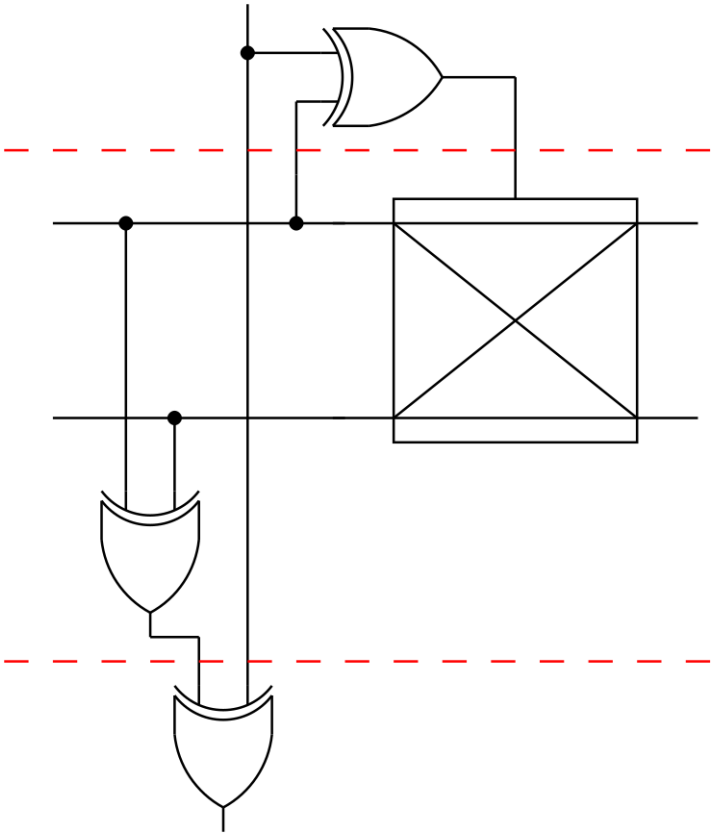


Network Generator

- Translates hardware circuits for networks into OpenCL
  - Complete network in a single Kernel
  - Using vector operations for parallelism
  - Limited to 16 inputs by OpenCL vector types
- Four stages:
  - Leveled circuit
  - Integer circuit
  - Vector operations
  - OpenCL codes

Leveled Circuits

- Netlist format: gates connected by wires
- Split into levels of gates
  - No data dependencies within levels
- Mostly bit-wise operations
- Data and addresses routed in the same bitvector



Leveled Circuits: Gates

Type	Inputs	Output	Operation
NOT	x	$[\neg x]$	Boolean negation
AND	x, y	$[x \wedge y]$	Boolean conjunction
OR	x, y	$[x \vee y]$	Boolean disjunction
XOR	x, y	$[x \oplus y]$	Exclusive disjunction
HA	x, y	$[x \oplus y,\ x \wedge y]$	Half adder
FA	x, y, z	$[x \oplus y \oplus z,\ x \wedge y \vee z \wedge (x \oplus y)]$	Full adder
PG	g_1, p_2, g_2, p_2	$[p_1 \wedge p_2,\ g_2 \vee (p_2 \wedge g1)]$	Propagate-generate gate
MX	$c,\ [x_0, x_1,\ \ldots\ x_n],$ $[y_0, y_1,\ \ldots\ y_n]$	$[c?y_0{:}x_0,\ c?y_1{:}x_1,\ \ \ldots\ c?y_n{:}x_n]$	2:1 multiplexer
SW	$c,\ [x_0, x_1,\ \ldots\ x_n],$ $[y_0, y_1,\ \ldots\ y_n]$	$[c?y_0{:}x_0,\ c?y_1{:}x_1,\ \ \ldots\ c?y_n{:}x_n,$ $c?y_0{:}x_0,\ \ldots\ c?y_n{:}x_n]$	Switch gate

Integer Ciruits

- Basic datatype in OpenCL is an integer
  - Bitvectors need to be translated to integers
  - Single wires need to become integers as well
- Splits Switches into separate address and data switches
  - Simplifies code generation later
- Introduces SubSig gates in new levels
  - Needed to extract single bits from bitvectors/integers
- Replaces ranking tree with special gate
- Reschedules resulting circuit in new levels

Vector Operations

- Use vector operations to exploit parallelism in the circuit
- Main Goal:

  Merge switches of one level into shuffle instruction
- Secondary Goal:

  Vectorise address logic if possible
- Levels of the circuit already group gates suitable for vectorising
- Turn the circuit into a stream of vector operations

Vector Operations

Type	Inputs	Output	Operation
NOT	x	$\neg x$	Boolean negation
AND	x, y	$x \wedge y$	Boolean conjunction
OR	x, y	$x \vee y$	Boolean disjunction
XOR	x, y	$x \oplus y$	Exclusive disjunction
MX	c, x, y	$c?y{:}x$	2:1 Multiplexer
SW	c, x, y	$c?y{:}x,\ c?y{:}y$	Switch gate
SUBSIG	x	y with $y \subseteq x$	Extract the bits in y from x
RANKINGTREE	$x_0, x_1 ... x_n$	$r_0, r_1, ... r_n$	Compute ranks of inputs

Vectorisation Strategy

- Start with vectors for addresses and data
- Vectorise gates of the same type in the same level if:
  - All elements of a vector are inputs of the same type of gate
  - Ignore the control input for switches
  - Ranking tree gates are already vectorised
- Derive output vectors

OpenCL: Logic and Arithmetic

- Direct translation
- Problem: Different values for true then scalar operators
- Only relevant for arithmetic operations

```
1  int4 x = (int4) (0,0,1,1);
2  int4 y = (int4) (0,1,0,1);
3
4  // z1 = NOT(x)
5  int4 z1 = !x;       // (-1,-1,0,0)
6  // z2 = AND(x,y)
7  int4 z2 = x && y;    // (0,0,0,-1)
8  // z3 = OR(x,y)
9  int4 z3 = x || y;    // (0,-1,-1,-1)
10 // z4 = XOR(x,y)
11 int4 z4 = x ^^ y;    // (0,-1,-1,0)
```

OpenCL: Extracting Bits

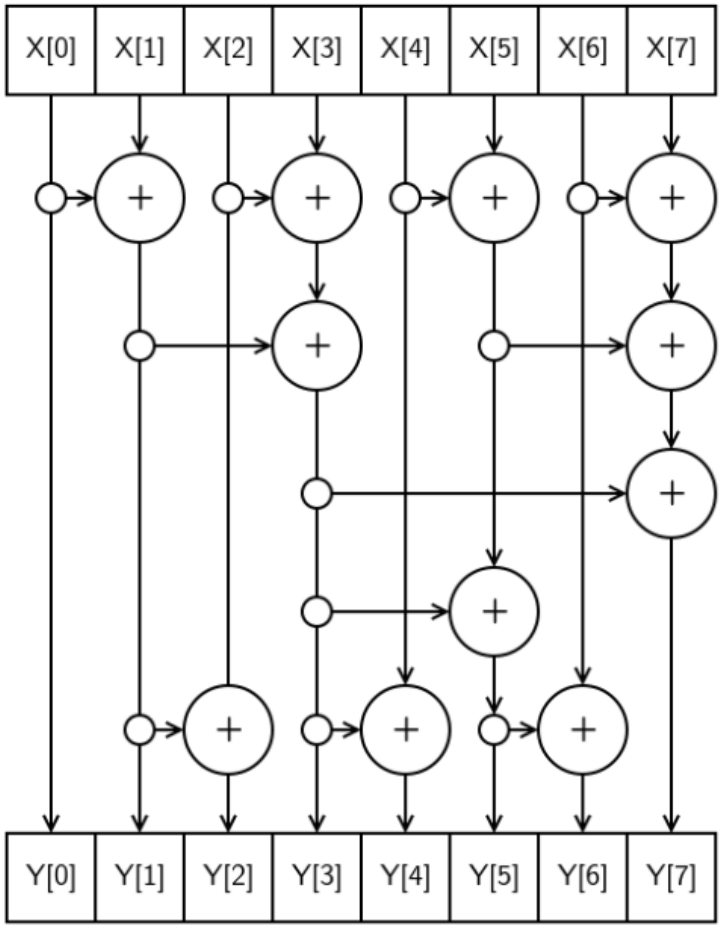
```
1  int  sub_0 = (((addr_in[0] & (1 << 2)) >> 2) << 0);
2
3  int4  sub_4 = (((v_4 & (1 << 1)) >> 1) << 0)
4                | (((v_4 & (1 << 2)) >> 2) << 1);
```

- Mask out the relevant bits
- Shift to right to LSB then to left to target position
- Use bitwise or to construct new integer

OpenCL: Mux

```
1  // z = MUX(c,x,y)
2  int4 z;
3  if(c) {
4     z = y;
5  }
6  else {
7     z = x;
8  }
```

OpenCL: RankingTree




```
1  //--- start ranking tree ---
2  int8 rank0 = (int8) (sig_1, sig_3, sig_5, sig_7,
3                       sig_9, sig_11, sig_13, sig_15);
4  rank0 = rank0 + (int8) (-1, 0, 0, 0, 0, 0, 0, 0);
5  int8 rank0_add_0 = (int8) (0, rank0[0], 0, rank0[2],
6                             0, rank0[4], 0, rank0[6]);
7  int8 rank0_0 = rank0 + rank0_add_0;
8  int8 rank0_add_1 = (int8) (0, 0, 0, rank0_0[1],
9                                0, 0, 0, rank0_0[5]);
10 int8 rank0_1 = rank0_0 + rank0_add_1;
11 int8 rank0_add_2 = (int8) (0, 0, 0, 0,
12                            0, 0, 0, rank0_1[3]);
13
14 int8 rank0_2 = rank0_1 + rank0_add_2;
15 int8 vec_24_add_1 = (int8) (0, 0, 0, 0,
16                            0, rank0_2[3], 0, 0);
17 int8 vec_24_1 = rank0_2 + vec_24_add_1;
18 int8 vec_24_add_0 = (int8) (0, 0, vec_24_1[1], 0,
19                      vec_24_1[3], 0, vec_24_1[5], 0);
20 int8 vec_24 = vec_24_1 + vec_24_add_0;
21 //--- end ranking tree ---
```

OpenCL: Switches

- Switch vector operations
  - Input vector, Control vector, Output vector
  - Element pairs in the vectors are swapped
- Shuffel instruction
  - Input vector **x**, mask vector **m**, Output vector **y**
  - `y[i] = x[m[i]]`
  - Mask has to computed from control vector
  - Two possible value per mask element

$$\vec{m} = \vec{p}_0 + \vec{c} \cdot \left(\vec{p}_1 - \vec{p}_0 \right)$$

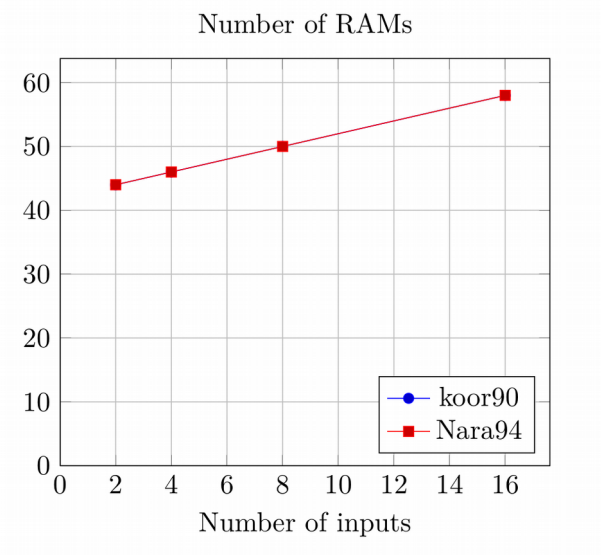
OpenCL: Switches

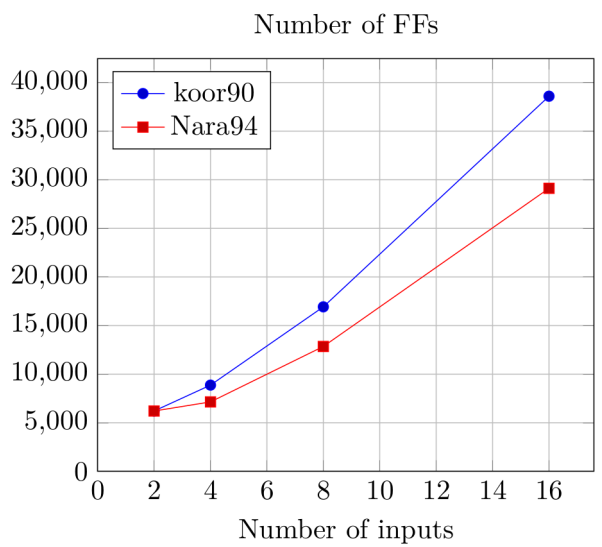
```
1  //--- start switch ---
2  int8 switchCrtlVec0 = (int8)(sig_22, sig_15,
3                               sig_13, sig_0, 0, 0, 0, 0);
4  switchCrtlVec0 = switchCrtlVec0 & 1;
5  int8 shuffleInputVec0 = (int8)data_input;
6  int8 preShuffleVec0 = (int8)(0, 0, 1, 1,
7                               2, 2, 3, 3);
8  int8 shuffleCtrlVec0 = shuffle(switchCrtlVec0,
9                          convert_uint8(preShuffleVec0));
10 int8 shuffleOffsetVec0 = (int8) (6, 7, 4, 5,
11                              2, 3, 0, 1);
12 int8 shuffleCoeffVec0 = (int8) (1, -1, 1, -1,
13                              1, -1, 1, -1);
14 int8 shuffleVec0 = shuffleOffsetVec0 +
15              shuffleCoeffVec0 * shuffleCtrlVec0;
16 int8 vec_1 = shuffle(shuffleInputVec0,
17                       convert_uint8(shuffleVec0));
18 //--- end   switch ---
```

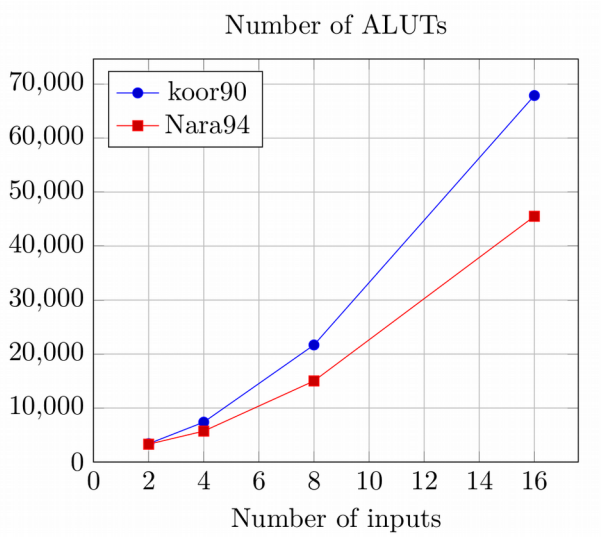
Evaluation

- Test runs on GPU and CPU
  - Networks with 2,4,8 inputs tested exhaustively
  - Network with 16 inputs tested for evenly distributed sample
- Test in Simulations
  - Failed for more then 4 inputs
  - Likely compiler bug in shuffle

FGA Hardware Resources







Inputs	ALUTs		Flip Flops		RAM Cells	
	KoOr90	Nara94	KoOr90	Nara94	KoOr90	Nara94
2	3396	3304	6211	6210	44	44
4	7390	5729	8873	7147	46	46
8	21686	15020	16925	12849	50	50
16	67887	45519	38593	29119	58	58

FGA Hardware Resources: Switch

```
1  //--- start switch ---
2  int4 switchCrtlVec1 = (int4)(sig_addr_13, sig_addr_15,
3                  0, 0);
4  //              ALUTs: 0  FFs: 0  RAMs: 0
5  int4 shuffleInputVec1 = (int4)data_input;
6  //              ALUTs: 0  FFs: 0  RAMs: 0
7  int4 preShuffleVec1 = (int4)(0, 0, 1, 1);
8  //              ALUTs: 0  FFs: 0  RAMs: 0
9  int4 shuffleCtrlVec1 = shuffle(switchCrtlVec1,
       convert_uint4(preShuffleVec1));
10 //              ALUTs: 0  FFs: 0  RAMs: 0
11 int4 shuffleOffsetVec1 = (int4) (0, 1, 2, 3);
12 //              ALUTs: 0  FFs: 0  RAMs: 0
13 int4 shuffleCoeffVec1 = (int4) (1, -1, 1, -1);
14 //              ALUTs: 0  FFs: 0  RAMs: 0
15 int4 shuffleVec1 = shuffleOffsetVec1
16           + shuffleCoeffVec1 * shuffleCtrlVec1;
17 // 32-bit Int Subtract(x2)  ALUTs: 66   FFs: 0  RAMs: 0
```

FPGA Hardware: Ranking Tree

```
1  //--- start ranking tree ---
2  int4 rankInputVec0 = (int4) (sig_addr_1, sig_addr_3,
3                 sig_addr_5, sig_addr_7);
4  //                ALUTs: 0  FFs: 0  RAMs: 0
5  rankInputVec0 = rankInputVec0 + (int4) (-1, 0, 0, 0);
6  //                ALUTs: 0  FFs: 0  RAMs: 0
7  int4 rankInputVec0_add_0 = (int4) (0, rankInputVec0[0],
8                 0, rankInputVec0[2]);
9  //                ALUTs: 0  FFs: 0  RAMs: 0
10 int4 rankInputVec0_0 = rankInputVec0 + rankInputVec0_add0;
11 // 32-bit Integer Add (x2)  ALUTs: 66 FFs: 0  RAMs: 0
12 int4 rankInputVec0_add1 = (int4) (0, 0,
13                0, rankInputVec0_0[1]);
14 //                ALUTs: 0  FFs: 0  RAMs: 0
15 int4 rankInputVec0_1 = rankInputVec0_0 +
      rankInputVec0_add1;
16 // 32-bit Integer Add (x1)  ALUTs: 33   FFs: 0  RAMs: 0
17 int4 vec_8_add0 = (int4) (0, 0, rankInputVec0_1[1], 0);
18 //                ALUTs: 0  FFs: 0  RAMs: 0
19 int4 vec_8 = rankInputVec0_1 + vec_8_add0;
20 //   32-bit Integer Add   ALUTs: 33   FFs: 0  RAMs: 0
21 //--- end ranking tree ---
```

Future work

- Test on actual HARP hardware
- Extension to larger input sizes
- Explore different vectorisation strategies
  - Vectorize more aggressively
  - Vectorize nothing but the switches
- Test with an actual scad-machine implementation

Conclusion

- The generated OpenCL kernels work
- Vectorisation of switches into shuffle instruction is possible
- Simulation fails, due to compiler bugs
- Hardware synthesis works, but has not been tested
- Scaleability might be an issue for larger inputs
- Vector operations are optimized efficiently
- Narashima's network is preferable in terms of size

Thank you for your attention