A Highly Efficient and Comprehensive Image Processing Library for C++-based High-Level Synthesis

M. Akif Özkan, Oliver Reiche, Frank Hannig, and Jürgen Teich
Hardware/Software Co-Design, Friedrich-Alexander University Erlangen-Nürnberg
FSP, September 7, 2017, Ghent

Motivation

Opportunity: FPGAs have a great potential for improving throughput per watt

Challenge: Hardware **design** is time consuming and needs expertise

Solution: High Level Synthesis (HLS) for providing the best suitable architecture from a traditional C++ code

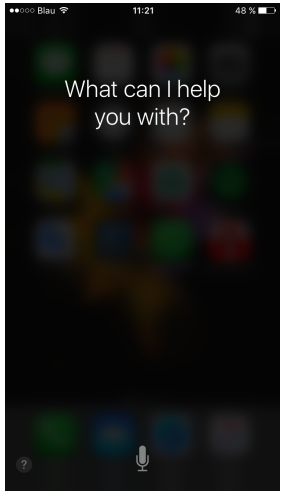
Motivation

Opportunity: FPGAs have a great potential for improving throughput per watt

Challenge: Hardware design is time consuming and needs expertise

Solution: High Level Synthesis (HLS) for providing the best suitable architecture from a traditional C++ code

What would be better is asking to Siri;
“Siri, could you please design a ConvNet accelerator for my 200 dollars FPGA!”



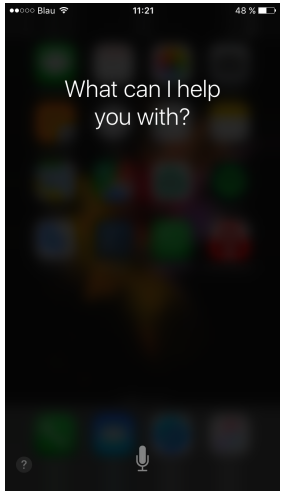
Motivation

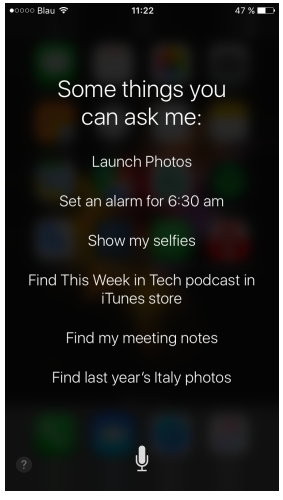
Opportunity: FPGAs have a great potential for improving throughput per watt

Challenge: Hardware design is time consuming and needs expertise

Solution: High Level Synthesis (HLS) for providing the best suitable architecture from a traditional C++ code

What would be better is asking to Siri;
“Siri, could you please design a ConvNet accelerator for my 200 dollars FPGA!”
Unfortunately, we are not there yet!





Motivation

Opportunity: FPGAs have a great potential for improving throughput per watt

Challenge: Hardware design is time consuming and needs expertise

Solution: High Level Synthesis (HLS) for providing the best suitable architecture from a traditional C++ code

Programming methodologies for **other platforms** are not there yet as well:

GPUs: map, gather, and scatter operations with a different language, i. e., OpenCL, CUDA

Multi-core CPUs: OpenMP or Cilk Plus for proper thread level parallelism for programming Xeon Phi architectures

CPUs: explicit vectorization

Motivation

Opportunity: FPGAs have a great potential for improving throughput per watt

Challenge: Hardware design is time consuming and needs expertise

Solution: High Level Synthesis (HLS) for providing the best suitable architecture from a traditional C++ code

Maybe it is the time to reconsider abstractions for FPGA design?

- Computational parallel patterns: i. e. gather, scatter
- Domain Specific Languages: HIPAcc, Halide, Polymage
- Hardware favorable library objects for essential algorithmic instances

Motivation

Opportunity: FPGAs have a great potential for improving throughput per watt

Challenge: Hardware design is time consuming and needs expertise

Solution: High Level Synthesis (HLS) for providing the best suitable architecture from a traditional C++ code

"Best" is hard to reach:

- Definition of the "best" depends on the design objectives (i. e. speed, area)
- Multiple alternative architectures exist for the same algorithmic instances
- The Pareto-optimal hardware architecture of an algorithmic instance for given design objectives might not be the optimal for different scheduling specifications (i. e. filter size, parallelization factor)

Motivation

Opportunity: FPGAs have a great potential for improving throughput per watt

Challenge: Hardware design is time consuming and needs expertise

Solution: High Level Synthesis (HLS) for providing the best suitable architecture from a traditional C++ code

"Best" is hard to reach: A design space exploration is needed!

- Definition of the "best" depends on the design objectives (i. e. speed, area)
- Multiple alternative architectures exist for the same algorithmic instances
- The Pareto-optimal hardware architecture of an algorithmic instance for given design objectives might not be the optimal for different scheduling specifications (i. e. filter size, parallelization factor)

Efficiency is important when the cost is considered!

Motivation

Opportunity: FPGAs have a great potential for improving throughput per watt

Challenge: Hardware design is time consuming and needs expertise

Solution: High Level Synthesis (HLS) for providing the best suitable architecture from a traditional C++ code

Not all bad news:

- HLS became sophisticated enough for data path design
- Different speed constraints are possible
- Support for deploying FPGAs in a heterogeneous system

Outline

Analysis of the Domain

Proposed Image Processing Library

A Deeper Look Into the Library

Evaluation and Results

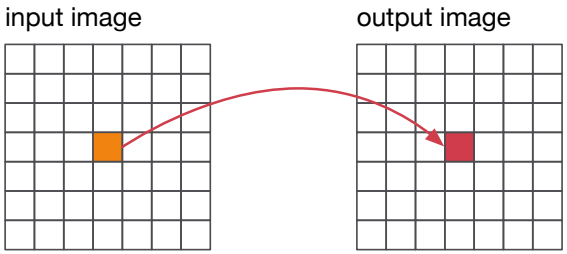
Analysis of the Domain

Image Processing Applications

We can define three characteristic data operations in image processing applications:

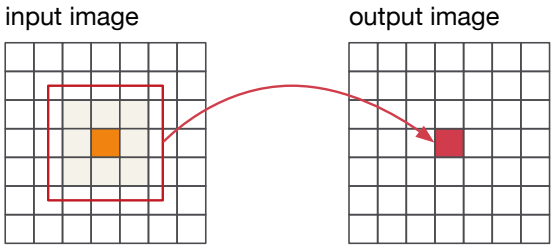
Point Operators:

Output data is determined by single input data



Local Operators:

Output data is determined by a local region of the input data (stencil pattern-based calculations)



Global Operators:

Output data is determined by all of the input data

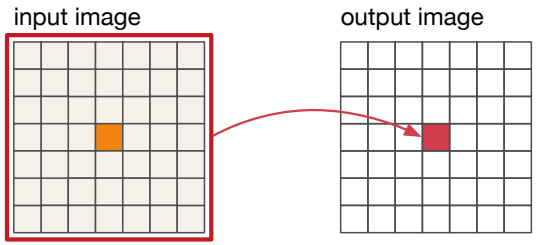
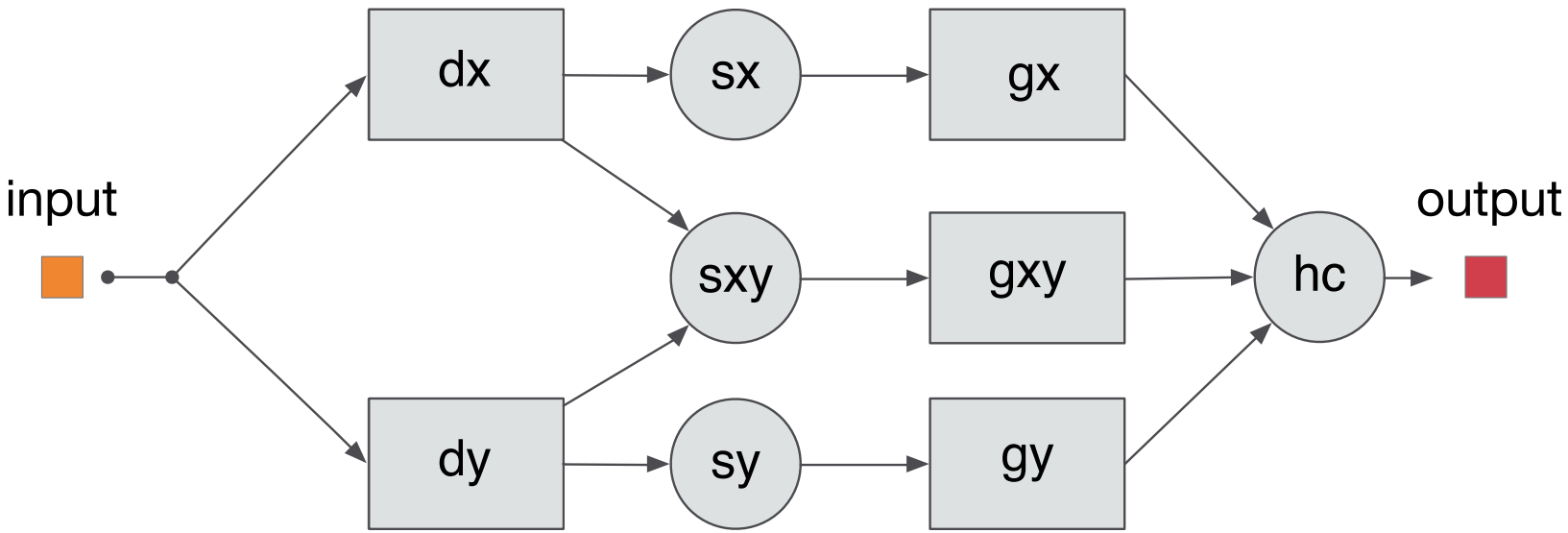


Image Processing Applications

A great portion of image processing applications can be described as task graphs of point, local, and global operators:



An example task graph for Harris Corner Detection
(square: local operator, circle: point operator)

Coarse-Grained Parallelism

Memory bandwidth limits can be reached by processing multiple pixels per cycle

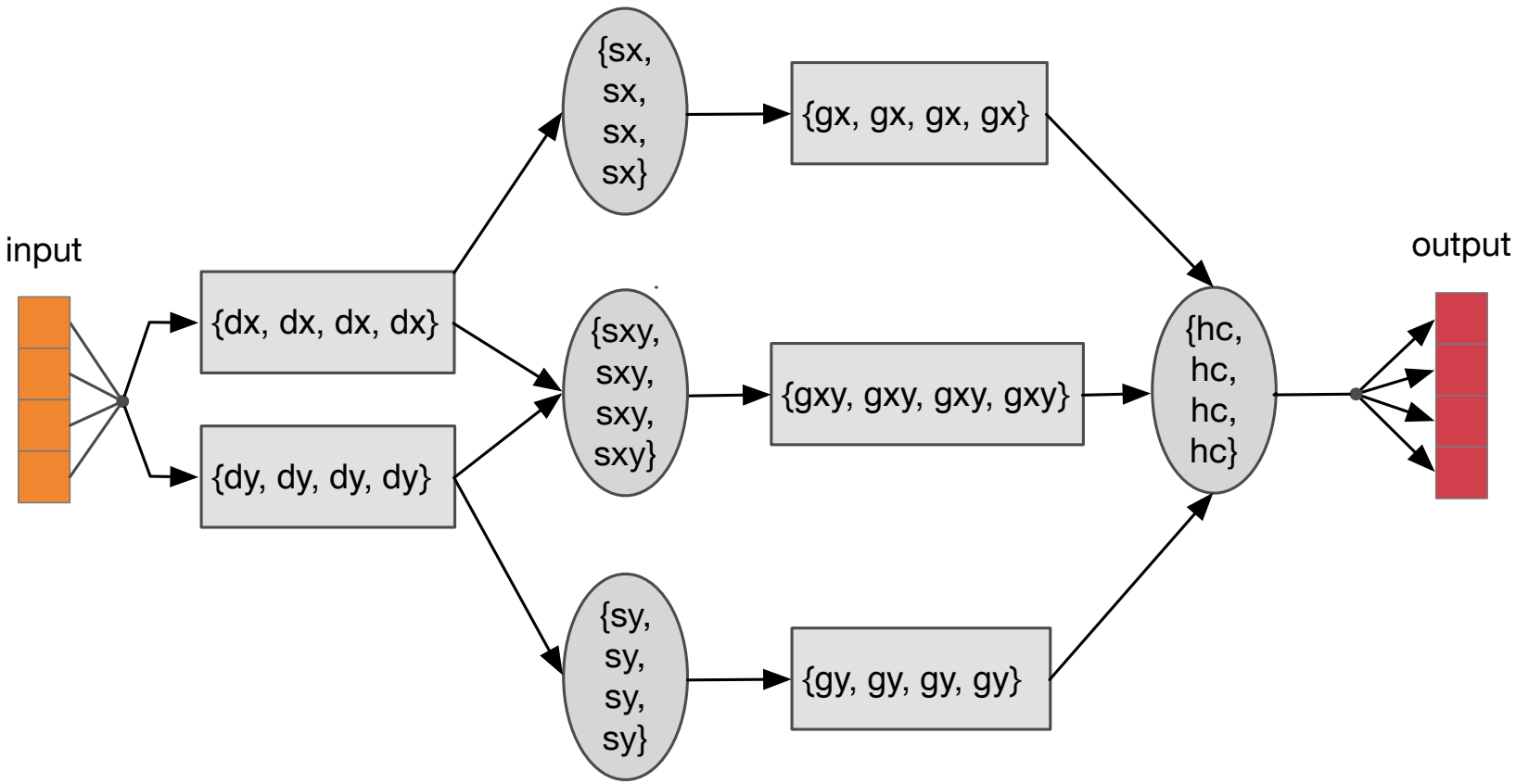
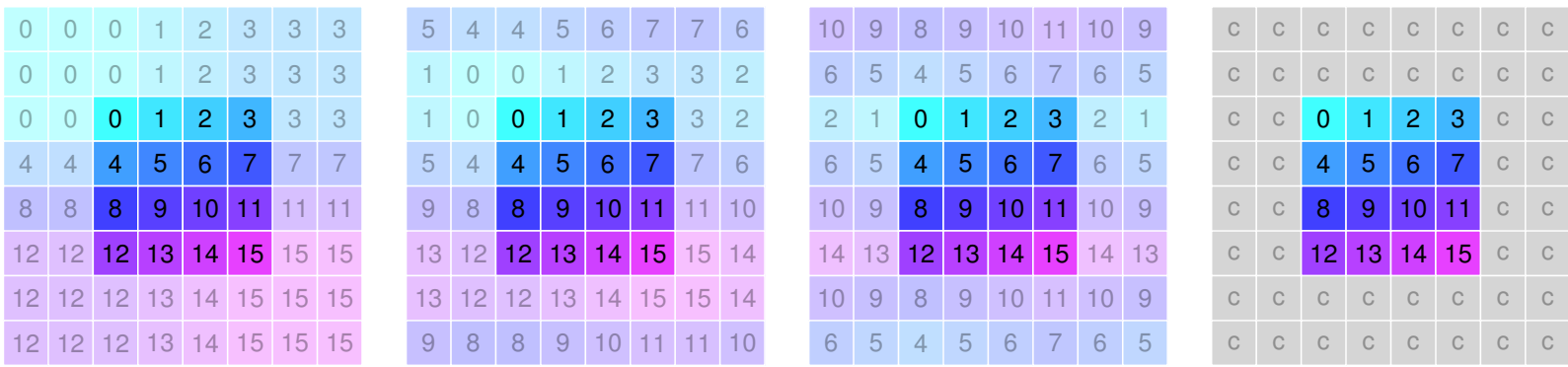


Image Border Handling

- a fundamental image processing issue for local operators
- should be considered together with coarse-grained parallelization



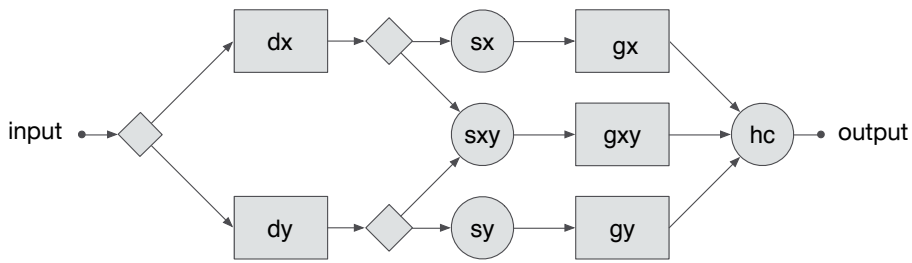
(a) clamp (b) mirror (c) mirror-101 (d) constant

Common border handling modes.

Proposed Image Processing Library

Description of an Application Data Flow Graph

```
#define W 1024    // Image Width
#define H 1024    // Image Height
#define pFactor 1 // Parallelization factor

// Data type descriptions
...

// Local operator definitions
localOp<W, H, pFactor, ..., MIRROR> sobelX, sobelY;

localOp<W, H, pFactor, ...> gaussX, gaussY, gaussXY;

pointOp<W, H, pFactor, ...> square, mult, harrisCorner;

// Hardware top function
void harris_corner(hls::stream<inVecDataType>  &out_s,
                   hls::stream<outVecDataType> &in_s) {
  #pragma HLS dataflow

  // Stream definitions
  hls::stream<VecDataType1> in_sx, in_sy, ...;
  hls::stream<VecDataType2>  ...;
  ...

  // Data path construction
  sobelX.run(Dx_s, in_sx);
  sobelY.run(Dy_s, in_sy);

  square.run(Mx_s, Dx_s1, square_kernel);
  square.run(My_s, Dy_s1, square_kernel);
  mult.run(Mxy_s, Dy_s2, Dx_s2, mult_kernel);

  gaussX.run(Gx_s, Mx_s, gauss_kernel);
  gaussY.run(Gy_s, My_s, gauss_kernel);
  gaussXY.run(Gxy_s, Mxy_s, gauss_kernel);

  harrisCorner.run(out_s, Gxy_s, Gy_s, Gx_s,
                   threshold_kernel);
}
```



Specification of a Data Path

Data path is a regular C++ function

point operator reads from an input data element

local operator reads from a window (2D array)

```
outDataType datapath(inDataType in_d){
  #pragma HLS inline

  return in_d * in_d;
}
```

Datapath of a multiplication (point operator).

Specification of a Data Path

Data path is a regular C++ function

point operator reads from an input data element

local operator reads from a window (2D array)

```
outDataT datapath(inDataT win[KernelH][KernelW]){
  #pragma HLS inline

  unsigned sum=0;
  for(uint j=0; j<KernelH; j++){
  #pragma HLS unroll
    for(uint i=0; i<KernelW; i++){
    #pragma HLS unroll
      sum += win[j][i];
    }
  }
  return (outDataT)(sum / (KernelH*KernelW));
}
```

Datapath of a mean filter (local operator).

Parallelizable Data Types

Objective: parallelize DFG according to a preprocessor constant (pFactor)

Challenge: data types depend on pFactor

Solution: pre-processor macros for data type definitions

```
newDataType(DataBeatType, DataType, pFactor)
```

specification of a parallelizable data type

Parallelizable Data Types

Objective: parallelize DFG according to a preprocessor constant (pFactor)

Challenge: data types depend on pFactor

Solution: pre-processor macros for data type definitions

```
newDataType(DataBeatType, DataType, pFactor)
```

specification of a parallelizable data type

```
// Data = DataBeat[index]
EXTRACT(Data, DataBeat, index);
```

partially reading from a data beat

```
// DataBeat[i] = Data
ASSIGN(DataBeat, Data, index);
```

updating a data beat from smaller data types

Interconnecting Streams

Vivado HLS streams are FIFO buffers, which

- \+ stalls the execution of the next node when there is no data
- \+ can have a depth that is higher than one data element

=> can be used as interconnecting streams between the nodes of a DFG

```
hls::stream<DataBeatType> repl1, repl2, in;
```

Definition of a stream in Vivado HLS.

Interconnecting Streams

Vivado HLS streams are FIFO buffers, which

- \+ stalls the execution of the next node when there is no data
- \+ can have a depth that is higher than one data element

=> can be used as interconnecting streams between the nodes of a DFG

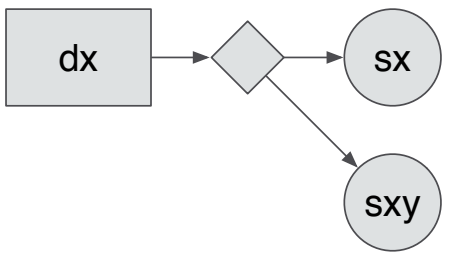
```
hls::stream<DataBeatType> repl1, repl2, in;
```

Definition of a stream in Vivado HLS.

Output stream of a node must be replicated when multiple following nodes are connected

```
splitStream(repl2, repl1, in);
```

replicating one stream to multiple streams



Operator Descriptions

Local Operator: template class

```
localOp<ImageWidth, ImageHeight,
        KernelWidth, KernelHeight,
        DataBeatType, pFactor,
        DataType, MIRROR> locObObj;
```

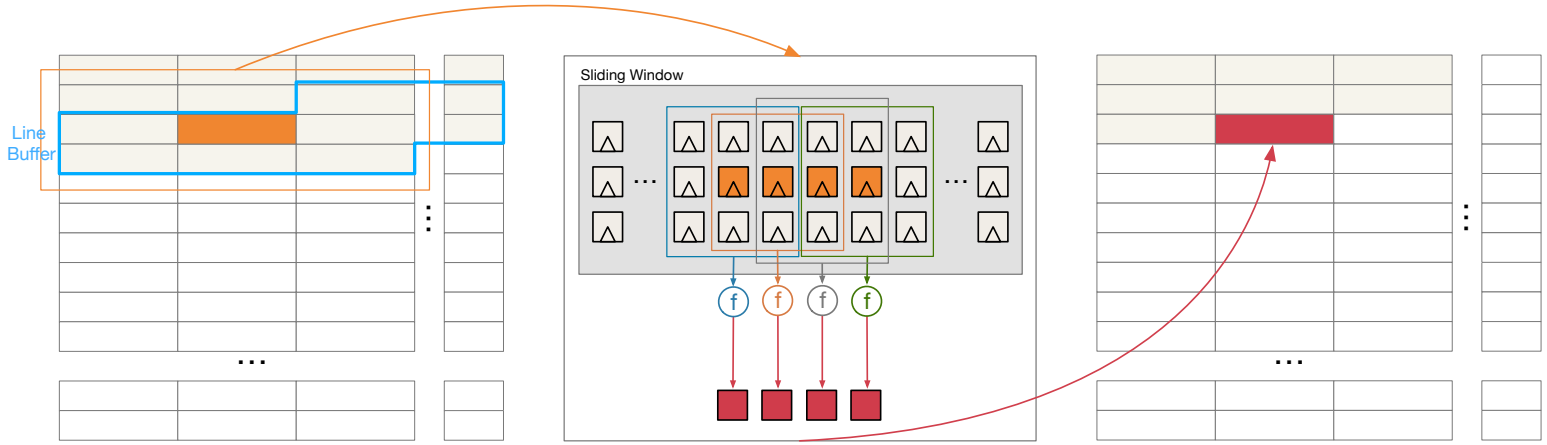
```
locOpObj.run(outStream, inStream,
             datapath);
```

Point Operator: template function

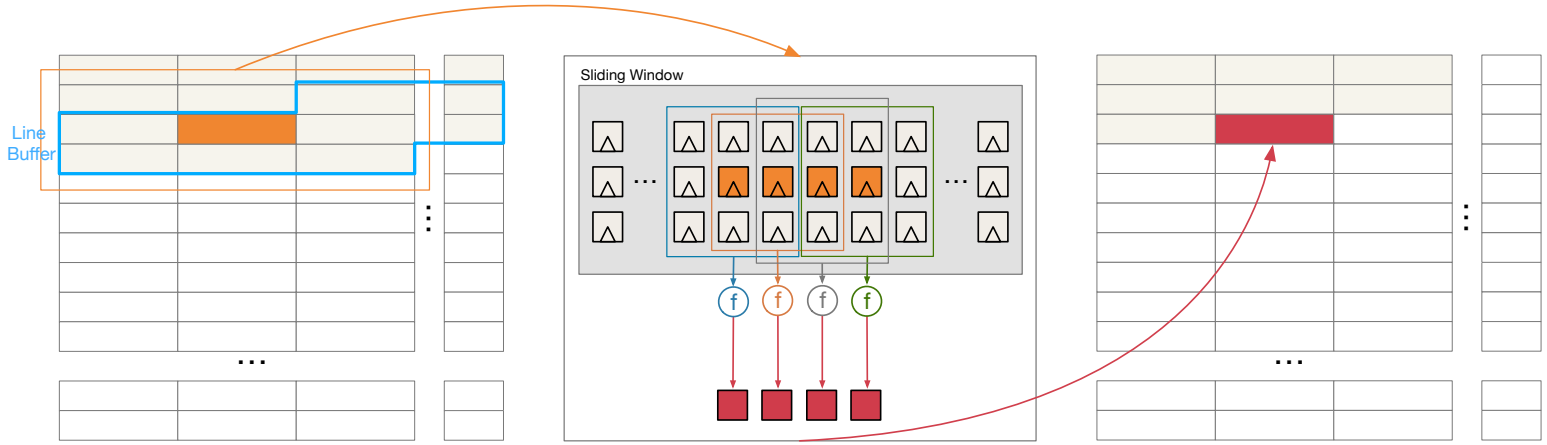
```
pointOp<pFactor>(outStream, inStream, dataPath);
```

Global Operator: Custom functions with global or static variables/arrays

Custom Node Descriptions: Stencil-based Applications



Custom Node Descriptions: Stencil-based Applications




```
for(size_t i = 0; i < ImageSize/pFactor; y++)
{
  // ...
  dataBeatIn << inStream;
  for(v = 0; v < pFactor; v++){
    #pragma HLS unroll
    EXTRACT(pixIn, dataBeatIn, v);
    // ...
    ASSIGN(dataBeatOut, pixOut, v);
  }
  outStream << dataBeatOut;
}
```

Custom Node Descriptions: Memory Instances

Supported specifications:

Line Buffer:

```
LineBuffer<KernelHeight,
           ImageWidth,
           DataBeatType> linebuf;
```

```
linebuf.shift(col2swin, newDataBeat,
              colIm);
```

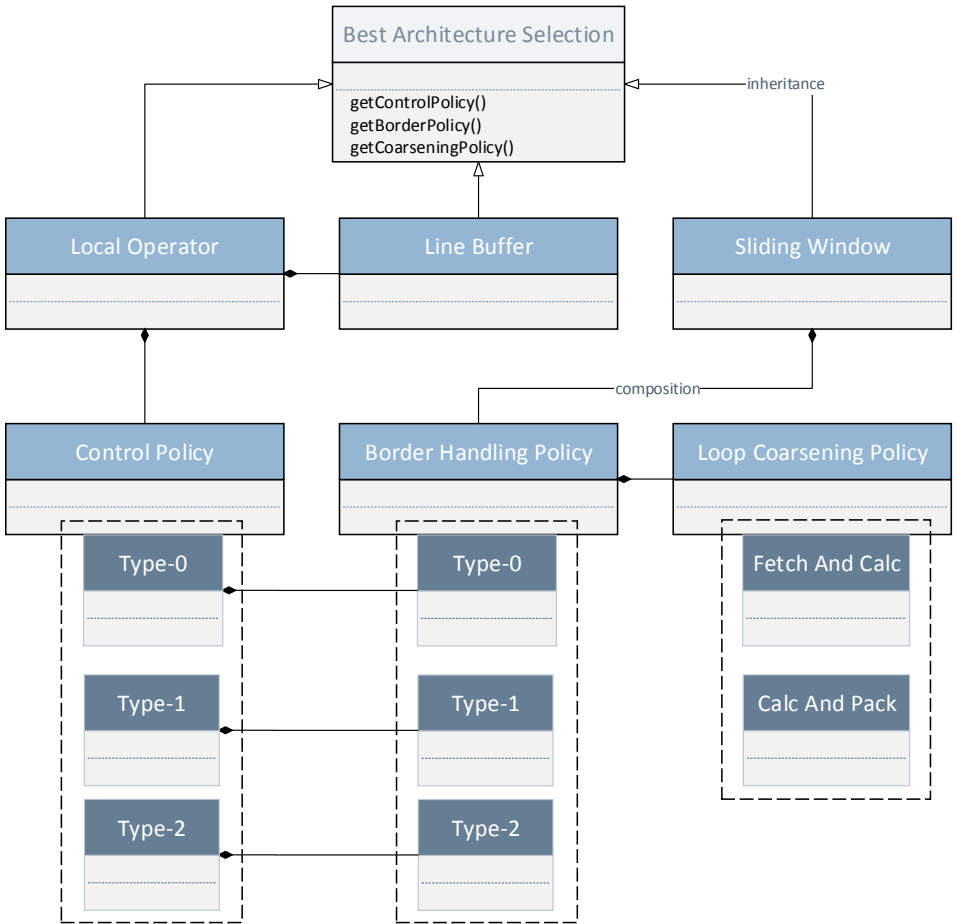
Sliding Window:

```
SlidingWindow<KernelWidth, KernelHeight
              DataBeatType, v, DataType
              MIRROR> sWin;
```

```
//Shift
swin.shift(col);
swin.shift(col, leftBorderFlags,
           rightBorderFlags);

// Read
DataBeatT pix = swin.get(j, i);
DataBeatT pix = swin.win_out[j][i];
```

A Deeper Look Into the Library

Software Architecture: Local Operator Class



An object relationship diagram for our proposed library.

Best Architecture Selection

Facilitate high performance without sacrificing high productivity with a compile time automatic architecture selection.

input : w, h, borderMode, v, k_{out}, k_{in}, designGoal
output: BorderHandlingPattern, CoarseningArch

```
1  func selectParetoOptimal[a](BorderHandlingPattern, CoarseningArch,
2                              w, h, borderMode, v, k_out, k_in, designGoal)
3      r_w = ⌊w/2⌋
4      if borderMode = UNDEFINED then
5          if k_out < k_in · h then
6              CoarseningArch ← Calc and Pack
7          else
8              CoarseningArch ← Fetch and Calc
9          end
10         BorderHandlingPattern ← none
11     else
12         if r_w · (k_in · h − k_out + 1) < v · (k_in · h − k_out) then
13             CoarseningArch ← Calc and Pack
14         else
15             CoarseningArch ← Fetch and Calc
16         end
17         if borderMode = (CLAMP ∨ CONSTANT) then
18             BorderHandlingPattern ← Type-1
19         else
               // borderMode = (MIRROR ∨ MIRROR-101)
20             if (designGoal = speed) ∨ ((r_w + 1)MUX[2] − MUX[r_vw + 1] − MUX[2] < 0) then
21                 BorderHandlingPattern ← Type-2
22             else
23                 BorderHandlingPattern ← Type-1
24             end
25         end
26     end
27 end
```

[a] M. A. Özkan *et al.*, "Hardware Design and Analysis of Efficient Loop Coarsening and Border Handling for Image Processing", in *28th IEEE International Conference on Application-specific Systems, Architectures and Processors (ASAP)*, (Seattle), Jul. 2017.

Best Architecture Selection

Facilitate high performance without sacrificing high productivity with a compile time automatic architecture selection.

Coarsening Selection

a seemless selection based on template parameters

Border Handling Selection

border handling architectures optimize different types of resources

a default design objective simplifies the specification

```
// designObjective LessLUTMoreRegister
// designObjective LessRegisterMoreLUT
localOp<..., designObjective> localOprtr;
```

Specification of a local operator with a design objective

RTL Level Optimizations

HLS tools mostly benefit from considerations at register-transfer level.

- arbitrary bit widths for the variables
- exploiting bit-specific properties for conditional assignments
- temporary registers updated in each iteration for describing wire assignments
- exploiting similarities in expressions through flags
- exploiting the temporal locality of the both control flow and data path

```
// Update Image indexes and isColRead
if(isImageWidthPowerOf2 == true){
  colIm = clkTick[BW_col-1:0];
  rowIm = clkTick[BW_row+BW_col-1:BW_col];
  isColRead = (colIm == imageWidth-1);
}
else{
  isColRead=false;
  colIm++;
  if(colIm == imageWidth){
    colIm=0; rowIm++;
    isColRead=true;
  }
}
```

Bit-level optimizations in the control flow

RTL Level Optimizations

HLS tools mostly benefit from considerations at register-transfer level.

- arbitrary bit widths for the variables
- exploiting bit-specific properties for conditional assignments
- temporary registers updated in each iteration for describing wire assignments
- exploiting similarities in expressions through flags
- exploiting the temporal locality of the both control flow and data path

```
// Update Image indexes and isColRead
if(isImageWidthPowerOf2 == true){
  colIm = clkTick[BW_col-1:0];
  rowIm = clkTick[BW_row+BW_col-1:BW_col];
  isColRead = (colIm == imageWidth-1);
}
else{
  isColRead=false;
  colIm++;
  if(colIm == imageWidth){
    colIm=0; rowIm++;
    isColRead=true;
  }
}
```

Bit-level optimizations in the control flow

RTL Level Optimizations

HLS tools mostly benefit from considerations at register-transfer level.

- arbitrary bit widths for the variables
- exploiting bit-specific properties for conditional assignments
- temporary registers updated in each iteration for describing wire assignments
- exploiting similarities in expressions through flags
- exploiting the temporal locality of the both control flow and data path

```
// Update Image indexes and isColRead
if(isImageWidthPowerOf2 == true){
  colIm = clkTick[BW_col-1:0];
  rowIm = clkTick[BW_row+BW_col-1:BW_col];
  isColRead = (colIm == imageWidth-1);
}
else{
  isColRead=false;
  colIm++;
  if(colIm == imageWidth){
    colIm=0; rowIm++;
    isColRead=true;
  }
}
```

Bit-level optimizations in the control flow

RTL Level Optimizations

HLS tools mostly benefit from considerations at register-transfer level.

- arbitrary bit widths for the variables
- exploiting bit-specific properties for conditional assignments
- temporary registers updated in each iteration for describing wire assignments
- exploiting similarities in expressions through flags
- exploiting the temporal locality of the both control flow and data path

```
// Program control flags
if( isImageWidthPowerOf2 == true ||
    (BorderPattern != UNDEFINED) ){
   initLatPASS = isRow0 && isXBndEnd;
   imREAD = !(isRowRead && isColRead);
}else{
  initLatPASS = (clkTick > initialLatency);
  imREAD      = (clkTick < imageSize);
}
```

Efficient usage of flags in the control flow

RTL Level Optimizations

HLS tools mostly benefit from considerations at register-transfer level.

- arbitrary bit widths for the variables
- exploiting bit-specific properties for conditional assignments
- temporary registers updated in each iteration for describing wire assignments
- exploiting similarities in expressions through flags
- exploiting the temporal locality of the both control flow and data path

```
isXleftBnd[0] = isXrightBnd[kRx-1];
for(int i = kRx - 1; i > 0; i--){
  isXrightBnd[i] =  isXrightBnd[i-1];
}
isXrightBnd[0] = isColRead;
```

Efficient usage of flags in the control flow

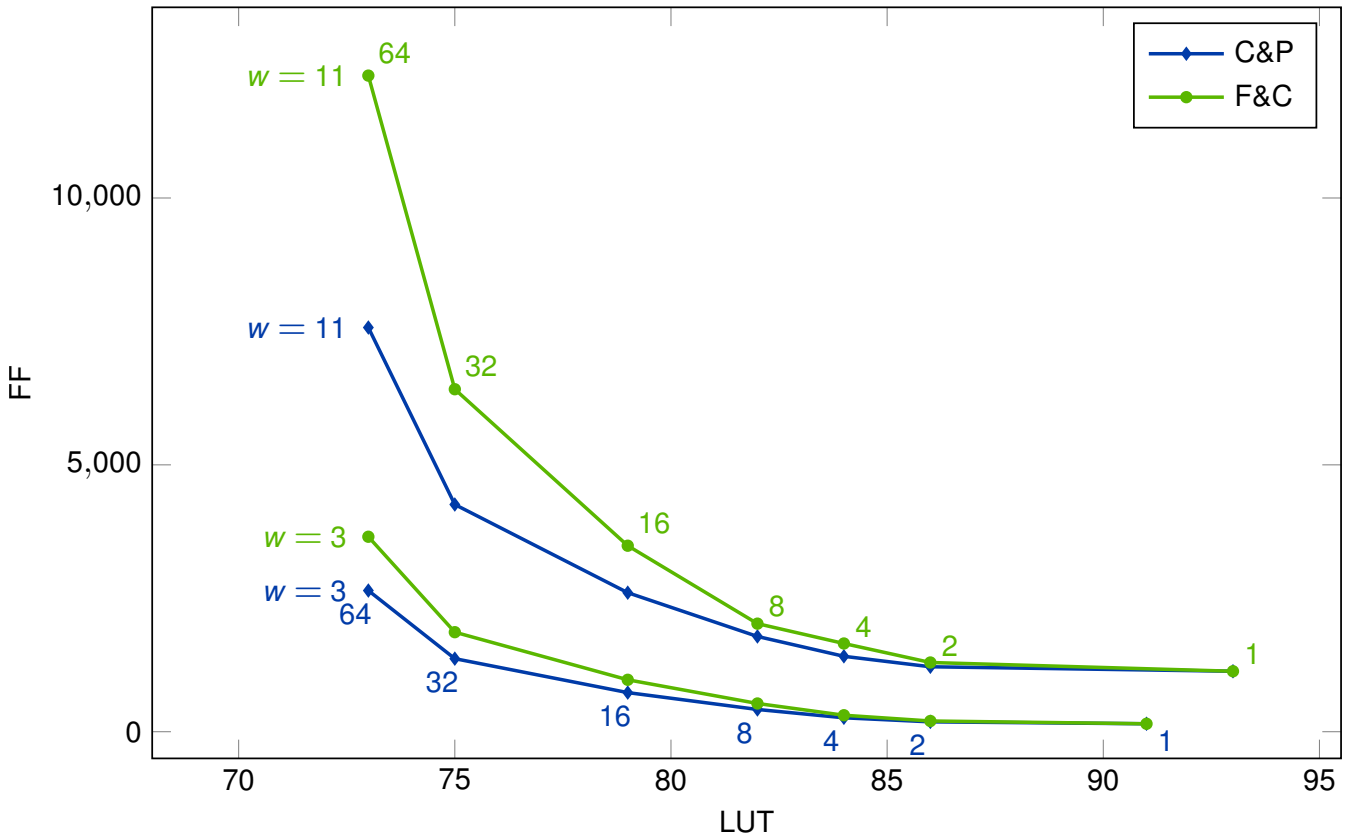
Control Path of a Local Operator

Optimizations at register-transfer level make an HLS code cumbersome, but can be hidden within a good software architecture.

```
local_operator_loop:
for(size_t clkTick=0;
    clkTick <= initialLatency+imageSize;
    clkTick++){
  #pragma HLS pipeline ii=1

  // Update Control Flags (1/2)
  control.UpdateBeforeShift(clkTick);

  // Run Data-path
  outPixel = datapath(control.SlidingWin);

  // Write Result
  if(control.initLatPASS == true ){
    out_s.write(data_out);
  }

  // Get New Input
  if(control.imREAD == true){
    in_s >> data_in;
  }

  // Shift Line Buffers and Sliding Window
  control.shift(data_in);

  // Update Control Flags (2/2)
  control.UpdateAfterShift(clkTick);
}
```

Evaluation and Results

Comparison of Loop Coarsening Architectures



HLS estimation results of the proposed coarsening architectures (target clock frequency is 200 MHz, and no border handling is applied)

Proposed Library vs. HIPAcc

Application	Framework	CF	SLICE	LUT	FF	DSP	BRAM	SRL	CPimp	Latency
Mean Filter	proposed	1	106	206	409	0	4	0	2.96	1050633
		32	1698	4722	6073	0	32	1	4.16	32841
	Hipacc	1	151	253	581	0	4	1	2.77	1052684
		32	2078	5008	8487	0	32	121	2.70	33866
Laplace	proposed	1	469	1126	1762	0	8	17	3.90	1050634
		32	12235	40157	33440	0	116	2	4.85	32842
	Hipacc	1	581	11307	2057	0	8	0	3.88	1052684
		32	12430	41349	36514	0	116	1404	4.85	33868
Sobel Edge	proposed	1	1113	2809	4942	8	4	85	3.94	1049687
		32	26716	76667	137267	256	14	2560	4.73	33878
	Hipacc	1	1138	2899	5028	8	4	85	3.82	1050632
		32	27770	83470	145072	256	32	2565	4.87	33878
Harris Corner	proposed	1	763	1731	2528	14	10	38	3.88	1049633
		32	8293	20017	31399	363	39	998	4.34	33825
	Hipacc	1	936	2125	3086	15	10	72	4.15	1050637
		32	14739	37424	56691	480	80	1081	4.89	33837
Bilateral	proposed	1	6049	15691	18535	190	2	811	4.26	1049763
		8	38776	119123	135711	1520	4	5604	4.87	131364
	Hipacc	1	15875	43859	50453	558	4	2638	4.48	1052967
		2	29669	85228	96159	1116	4	4307	4.84	526630

https://github.com/akifoezkan/implib-hls

Thanks for listening.
Any questions?

Title A Highly Efficient and Comprehensive Image Processing Library for C++-based High-Level Synthesis

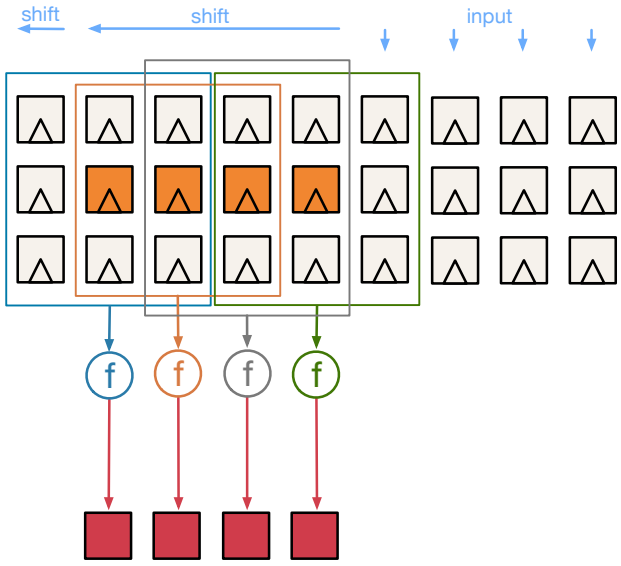
Speaker M. Akif Özkan, akif.oezkan@fau.de

References I

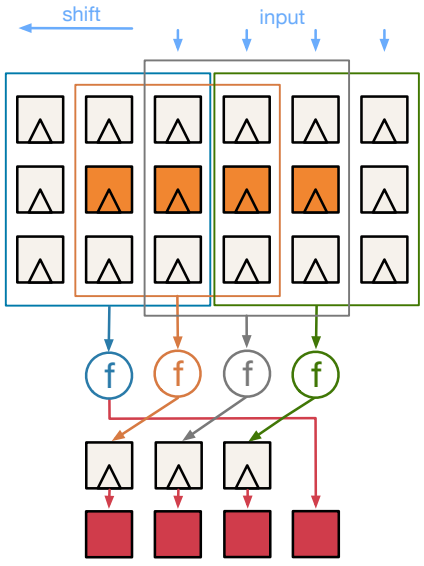
[1] M. A. Özkan, O. Reiche, F. Hannig, and J. Teich, "Hardware Design and Analysis of Efficient Loop Coarsening and Border Handling for Image Processing", in *28th IEEE International Conference on Application-specific Systems, Architectures and Processors (ASAP)*, (Seattle), Jul. 2017.

Related Hardware Architectures

Loop Coarsening Architectures



(a) Fetch And Calc (F&C)



(b) Calc And Pack (C&P)

C&P uses fewer registers than F&C when

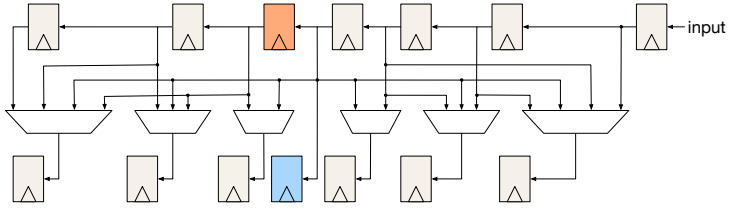
$$r_w \cdot \left(k_{\text{in}} \cdot h - k_{\text{out}} + 1\right) < v \cdot \left(k_{\text{in}} \cdot h - k_{\text{out}}\right) \text{ satisfies}$$

where r_w: radius of the width, h: height, v: pFactor, k: bitwidth

Column Selection Architectures: Mirror border mode

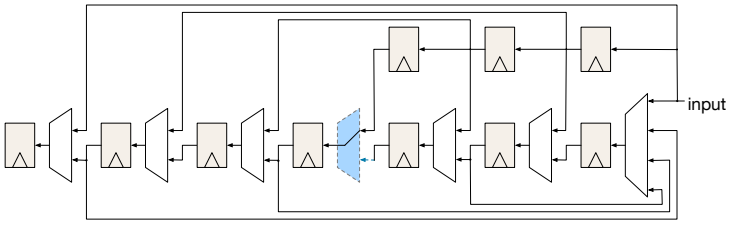
Type-0:

- not resource efficient
+ full flexibility for all the border modes



Type-1:

+ resource efficient for a great portion of design space



Type-2:

+ fastest architecture
+ Pareto-optimal depending on *w*, *v*, and technology mapping