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:

CPU: 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

![Point Operator Diagram](image)

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

![Local Operator Diagram](image)

**Global Operators:**
Output data is determined by all of the input data

![Global Operator Diagram](image)
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

```c
#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)

```cpp
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)

```cpp
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

\[ \text{newDataType}(\text{DataBeatType}, \text{DataType}, \text{pFactor}) \]

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

```c
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

```cpp
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

```cpp
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

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

replicating one stream to multiple streams
Operator Descriptions

Local Operator:  template class

\[
\text{localOp}\langle \text{ImageWidth}, \text{ImageHeight}, \\
\text{KernelWidth}, \text{KernelHeight}, \\
\text{DataBeatType}, \text{pFactor}, \\
\text{DataType}, \text{MIRROR}\rangle \text{ locObObj;}
\]

\[
\text{locObObj.run(outStream, inStream, datapath);}
\]

Point Operator:  template function

\[
\text{pointOp}\langle \text{pFactor}\rangle(\text{outStream, inStream, dataPath});
\]

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:**

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

```cpp
linebuf.shift(col2swin, newDataBeat, col1m);
```

**Sliding Window:**

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

```cpp
// 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

func selectParetoOptimal(w, h, borderMode, v, k\_{out}, k\_{in}, designGoal)

\[
\begin{align*}
 r_w &= \lfloor w/2 \rfloor \\
 \text{if} \ borderMode = UNDEFINED \text{ then} \\
 &\quad \text{if} \ k_{out} < k_{in} \cdot h \text{ then} \\
 &\quad \quad \text{CoarseningArch} \leftarrow \text{Calc and Pack} \\
 &\quad \text{else} \\
 &\quad \quad \text{CoarseningArch} \leftarrow \text{Fetch and Calc} \\
 &\quad \text{BorderHandlingPattern} \leftarrow \text{none} \\
 \text{else} \\
 &\quad \text{if} \ r_w \cdot (k_{in} \cdot h - k_{out} + 1) < v \cdot (k_{in} \cdot h - k_{out}) \text{ then} \\
 &\quad \quad \text{CoarseningArch} \leftarrow \text{Calc and Pack} \\
 &\quad \text{else} \\
 &\quad \quad \text{CoarseningArch} \leftarrow \text{Fetch and Calc} \\
 &\quad \text{if} \ borderMode = (\text{CLAMP} \lor \text{CONSTANT}) \text{ then} \\
 &\quad \quad \text{BorderHandlingPattern} \leftarrow \text{Type-1} \\
 \text{else} \\
 &\quad \quad // borderMode = (\text{MIRROR} \lor \text{MIRROR-101}) \\
 &\quad \quad \text{if} \ (\text{designGoal} = \text{speed}) \lor ((r_w + 1) \text{MUX}[2] \leq \text{MUX}[r_w + 1] < \text{MUX}[2] < 0) \text{ then} \\
 &\quad \quad \quad \text{BorderHandlingPattern} \leftarrow \text{Type-2} \\
 &\quad \quad \text{else} \\
 &\quad \quad \quad \text{BorderHandlingPattern} \leftarrow \text{Type-1} \\
 &\quad \text{end} \\
 &\text{end} \\
 &\text{end} \\
 &\text{end}
\end{align*}
```

\(^a\text{M. A. Özkan et al., “Hardware Design and Analysis of Efficient Loop Coarsening and Border Handling for Image Processing”, in 28th IEEE International Conference on Application-specific Systems, Architectures and Processors (ASAP), (Seattle), Jul. 2017.}\)
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

```cpp
// designObjective LessLUTMoreRegister
// designObjective LessRegisterMoreLUT
local0p<..., designObjective> local0ptr;
```

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

```c
// 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

```c
// 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

```c
// 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

```c
// Program control flags
if( isImageWidthPowerOf2 == true ||
    (BorderPattern != UNDEFINED)){
    initLatPASS = isRow0 && isXEnd;
    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

```c
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.

```c
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

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

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

\[ r_w \cdot (k_{in} \cdot h - k_{out} + 1) < v \cdot (k_{in} \cdot h - k_{out}) \]

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