This is the author's version of the work. The definitive work was published in *Proceedings of the International Workshop on FPGAs for Software Programmers (FSP) co-located with International Conference on Field Programmable Logic and Applications (FPL), Ghent, Belgium, September 7, 2017.*

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

M. Akif Özkan, Oliver Reiche, Frank Hannig, and Jürgen Teich
Friedrich-Alexander University Erlangen-Nürnberg (FAU), Germany

Abstract

Field Programmable Gate Arrays (FPGAs) are proved to be among the most suitable architectures for image processing applications. However, accelerating algorithms using FPGAs is a time-consuming task and needs expertise. Whereas the recent advancements in High-Level Synthesis (HLS) promise to solve this problem, today's HLS tools require efficient hardware descriptions of algorithms to be able to provide favorable implementations. A solution is developing highly parameterizable and optimized HLS libraries for the fundamental image processing components. Another solution is providing a higher level of abstraction in the form of a Domain-Specific Language (DSL) and a corresponding efficient back end for hardware design. In this paper, we provide a highly efficient and parameterizable C++ library for image processing applications, which would be the cornerstone for both approaches. In our library, nodes of a stream-based data flow graph can be described as C++ objects for specified functions, and the whole application can be efficiently parallelized just by defining a global constant as the parallelization factor. Moreover, the key hardware design elements, i. e., line buffers and sliding windows with different border handling patterns, can be utilized individually to ease the design of more complicated applications.

1 Introduction

FPGAs have a great potential for improving throughput per watt in many applications. However, one needs to *design* hardware for FPGA-based acceleration, unlike traditional software *programming*. This is a time-consuming task and requires hardware expertise. HLS has received lots of attention over 30 years and has become a lot more sophisticated in the last decade with its *second wave* [1]. Yet, today's HLS tools can provide favorable performance only from hardware friendly descriptions [2], [3]. This is a setback for FPGAs to be widely used, but not a surprise since similar requirements can be observed for other parallel processing platforms. For instance, an algorithm should be described as the combinations of map, gather, and scatter operations with a different language, i. e., Open Computing Language (OpenCL), CUDA, when Graphics Processing Units (GPUs) are targeted [4]. Similarly, a suitable language such as OpenMP or Cilk Plus is necessary for programming multi-core Xeon Phi architectures to employ a proper thread level parallelism [5]. Even Central Processing Unit (CPU) implementations can significantly be tuned up through explicit vectorization [6].

Considering that the memory architecture and the parallelism that unleash the benefits of FPGAs are significantly different than CPUs and GPUs, the next step of enabling FPGAs to people with little or no expertise in hardware design might be developing highly efficient and comprehensible libraries or DSLs for the target application domains. Efficiency is vital since FPGAs are usually more expensive than other accelerators, and thus, can be promoted only with a significantly better performance. Yet, harder to be provided. The first reason is that the *best design* of an algorithm for an FPGA depends on the design objectives and the resource constraints. Furthermore, FPGAs are susceptible to acquiring multiple Pareto-optimal design points for the same algorithm, which minimize different resource types but facilitate the same performance. For instance, a Pareto-optimal architecture might require less Block Ram (BRAM), while another uses less Look-up Tables (LUTs). In this case, an efficient algorithm implementation would be the one that requires the less budget-critical type of FPGA resources, and thus maximizes the desired design objective, i. e., throughput or energy efficiency. Second, a Pareto-optimal architecture of an algorithmic instance might depend on the input parameters of the same algorithm. For example, an optimal border handling architecture for a small kernel might become inefficient for a large kernel.

A solution to the latter challenge can be addressed by comprehensive libraries that consist of more than one template design for each algorithm component. Subsequently, an automated design space exploration that offers different hardware architectures for the given constraints and objectives might be a solution to the both. On the other hand, Pareto-selection algorithms within a hardware design template library in cases where the optimal architecture, according to the specification parameters, is known might be sufficient for the second challenge and would simplify the design space exploration.

In this work, we provide a highly efficient and comprehensive library for image processing. Our contributions are summarized as follows:

- An approach for developing HLS libraries that facil-

itate a high performance thanks to multiple template architectures considered for different specifications of the same algorithmic instances, but sustain ease of use by hiding the architecture knowledge through a compile-time selection.

- A comprehensive stream-based image processing library for Vivado HLS that facilitates different border handling patterns and efficient parallelization according to template parameters.

2 Background

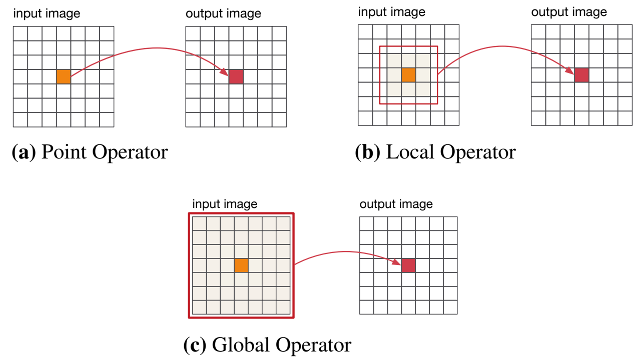
A plethora of image processing applications can be described as a data flow graph of point, local, and global operators (see Figure 1). Point operators calculate an output pixel using only one pixel of an input image, whereas the result of a local operator depends on neighboring pixels in a local window. Distinctively, a global operator is not constrained to a local window, and therefore, the calculation might depend on the whole input image.
Hardware implementation of point operators is rather straightforward since no dependency is implied. A new input could be read from the corresponding *data path*, which refers to arithmetic calculations of a node, in each cycle. On the other hand, a more sophisticated memory architecture is needed for the efficient implementation of local operators as explained in Section 2.1. Whereas a huge variety of global operators exist, a great portion can be implemented as the combination of global and local operators with global memory instances.

2.1 Local Operators

A local operator processes an input image using a local window after a stencil pattern-based computation. Characteristically, two adjacent local operators have a high spatial locality since a local operator with a (w, h) window requires the same $(w-1)\cdot h$ pixels to calculate results at the image coordinates (x, y) and $(x+1, y)$. A line-buffered implementation is a well-known architecture for FPGA-based acceleration that exploits this spatial locality to maximize the throughput at the cost of initial latency. In this architecture, an image is processed in raster order to enable burst-mode external memory reads, and all the dependency pixels of a local operator are read from on-chip memory blocks, which are called *line buffers*. Once the dependency pixels of the first calculation are cached, h pixels are read to a *sliding window* from the line buffers in each cycle, allowing the data path function to operate on a (w, h) window with a throughput of one cycle.

2.2 Loop Coarsening

A recent work [7], [8] revealed that Data Level Parallelism (DLP) can be increased with an improved resource sharing when a factor of *pFactor* coarsens the outer horizontal loop of an image processing algorithm, and temporally consecutive local operators are bound together to process a so called *data beat*, which is the group of *pFactor* pixels, in parallel at a cycle. Thus, memory reads remain in burst mode, and the overlap between consecutive local operators is loaded only once to on-chip memory as shown in Figure 2.



(a) Point Operator **(b)** Local Operator

(c) Global Operator

Figure 1 Operator types.

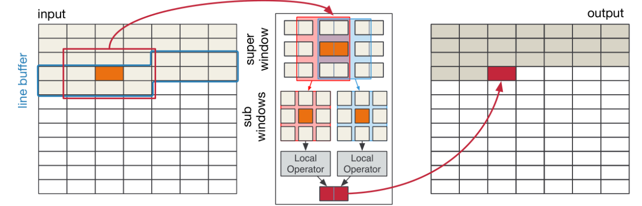
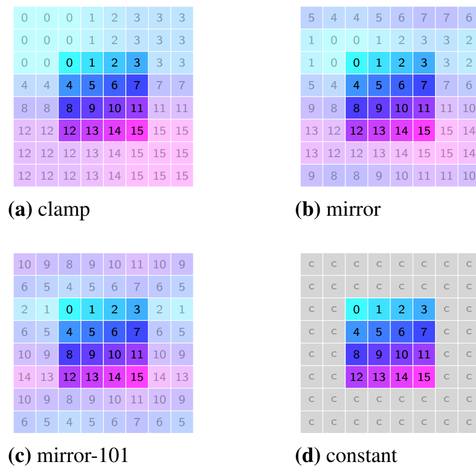


Figure 2 Loop coarsening.



(a) clamp **(b)** mirror

(c) mirror-101 **(d)** constant

Figure 3 Common border handling modes.

2.3 Border Handling

A local operator indicates dependencies outside of an input image while processing the pixels at the borders. A well known algorithmic solution to avoid artifacts at the borders of an output image is handling data according to the border patterns shown in Figure 3. An efficient hardware implementation that does not stall an input stream thus sustains the input throughput, is deploying selection circuits within the sliding window that choose the appropriate data according to image coordinates. A detailed explanation to these architectures can be found in [9].

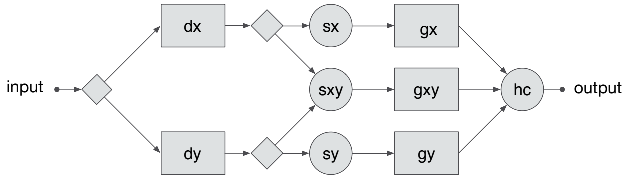


Figure 4 DFG for stream-based implementation of Harris corner detection. (*line*: stream, *rhombus*: split stream, *square*: local operator, *circle*: point operator)

3 Motivation

This paper focuses on FPGA implementations of image processing applications that can be expressed as Data Flow Graphs (DFGs). Unlike most other approaches, we provide multiple Pareto-optimal architectures for the same library instances to allow a designer or a design space exploration algorithm to tailor the implementation. However, we sustain the highest abstraction, in which a designer can only describe an instance, and thus depends on automatic template selection algorithms. Also, image border handling modes can be selected just by a template parameter and the whole algorithm can efficiently be parallelized with a global constant.

As a motivational example, the DFG representation of a Harris corner detector, shown in Figure 4, can be described as easy as Listing 1. The algorithm works as follows: 1. Horizontal and vertical derivatives of the input image are calculated by Sobel local operators. 2. Multiplication and squares of the derivatives are calculated through point operators and then smoothed by a Gaussian kernel. 3. Finally, the ranking of every pixel, which is the determinant of the derivatives, is calculated and the corners of the image are selected through a binary thresholding within the last point operator.

4 Overview

Section 5 presents the specifications for the essential instances of FPGA-based image processing pipelines in our proposed library. Section 6, in which we discuss the software architecture of the library, takes a deeper look into hardware templates and provides interfaces for the basic instances and policies. This lower abstraction layer allows the library to be utilized from another Design Space Exploration (DSE) framework, DSL, or just can be used for the description of more complicated nodes, i. e., a hierarchical node or a local operator consisting of two sliding windows. Finally, we evaluate the library for different image processing applications in Section 7.

5 Stream-Based Image Processing

Stream processing is a very useful paradigm to express an algorithm as a DFG without explicitly managing allocation, synchronization, or communication. As the memory architectures can be grasped from the types of the nodes and their connections in a DFG, scheduling of an algorithm and the

```
#define W 1024     // Image Width
#define H 1024     // Image Height
#define pFactor 1 // Parallelization factor
typedef uchar DataT;

// Parallelized data type
newDataType(VecDataT, DataT, pFactor)

// Local operator definitions
localOp<W, H, 3, 3, VecDataT, pFactor,
        DataT, MIRROR> sobelX, sobelY;

localOp<W, H, 5, 5, VecDataT, pFactor,
        DataT, MIRROR, LessLUTMoreRegister
        > gaussX, gaussY, gaussXY;

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

  // Stream definitions
  hls::stream<VecDataT> in_sx, in_sy;
  hls::stream<VecDataT> Dx_s, Dx_s1, Dx_s2;
  hls::stream<VecDataT> Dy_s, Dy_s1, Dy_s2;
  hls::stream<VecDataT> Dxy_s;
  hls::stream<VecDataT> Mx_s, My_s, Mxy_s;
  hls::stream<VecDataT> Gx_s, Gy_s, Gxy_s;

  // Data path construction
  splitStream(in_sx, in_sy, in_s);

  sobelX.run(Dx_s, in_sx);
  sobelY.run(Dy_s, in_sy);

  splitStream(Dx_s2, Dx_s1, Dx_s);
  splitStream(Dy_s2, Dy_s1, Dy_s);

  pointOp<pFactor>(Mx_s, Dx_s1, square_k);
  pointOp<pFactor>(My_s, Dy_s1, square_k);
  pointOp<pFactor>(Mxy_s, Dy_s2, Dx_s2,
                   mult_k);

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

  pointOp<pFactor>(out_s, Gxy_s, Gy_s, Gx_s,
                   threshold_kernel);
}
```

Listing 1 Harris corner HLS code in the proposed library.

data paths of the nodes can be described separately, allowing to focus on functionality rather than the implementation. Correspondingly, one can define a DFG as in Listing 1 via our proposed library and describe the data path functions with fewer considerations of implementation details.

5.1 Specification of a Data Path

Our proposed library expects to have a data path function without parallelization concerns for any defined node of a DFG. Assume that input and output data types of a data path are named as `inDataT`, `outDataT`, respectively. Then, the data path of a point operator, which inputs and outputs

one data element, can be described as in Listing 2.

```
outDataType datapath(inDataType in_d){
  #pragma HLS inline
  return in_d * in_d;
}
```

Listing 2 Datapath of a multiplication point operator.

A data path description, for a local operator, that reads from a sliding window is shown in Listing 3.

```
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));
}
```

Listing 3 Datapath of a mean filter local operator.

5.2 Specification of a Data Flow Graph

This section presents the specifications in our proposed library for describing DFGs mentioned above.

5.2.1 Data Types for Parallelization

As Loop coarsening of a DFG increases the data bandwidth of the input, output, and interconnecting streams by the parallelization factor *pFactor*, all the operator nodes process data beats instead of data tokens as explained in Section 2.2. A preprocessor macro, `newDataType`, is provided in our proposed library for defining so called *parallelizable* data types, whose size increases at the compile time according to the parallelization factor.

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

Listing 4 Specification of a parallelizable data type.

As can be conceived, all inputs, outputs, and interconnecting streams should have a parallelizable data type as in Listing 1 to enable the automatic parallelization feature. Moreover, a data element or consecutive elements can be read partially from a data beat using `EXTRACT` as in Listing 5.

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

Listing 5 Partially reading from a data beat.

Similarly, a data beat can be updated partially from smaller data types via `ASSIGN` as shown in Listing 6.

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

Listing 6 Updating a data beat from smaller data types.

5.2.2 Interconnecting Streams

The nodes are connected via streams according to a producer-consumer relation in which streams can be considered as First In First Out (FIFO) buffers. Therefore, care should be taken when the output of a stream is read from multiple nodes. As a solution, a stream can be replicated via `splitStream`.

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

Listing 7 Replicating one stream to multiple streams.

5.2.3 Point operators

A point operator can be specified with the `pointOperator` template function whose input parameters are output/input streams and a data path function as shown in Listing 8.

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

Listing 8 Specification of a point operator.

Moreover, multiple input streams can be given as inputs. Input and output streams should have the parallelizable data types as discussed in Section 5.2.1. The parallelization factor should be specified as a template parameter.

5.2.4 Local Operators

A local operator node can be described in two steps. First, a C++ object should be instantiated with the corresponding template parameters, which are kernel and image sizes, data types, parallelization factor and the border handling mode.Then, input and output streams as well as a data path

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

Listing 9 Specification of local operators.

function should be registered with the `run` method.

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

Listing 10 Invocation of local operators.

5.2.5 Global operators

A global operator can be described with global or static variables using local, global operator specifications, or as a custom node. Note that global arrays are interpreted as BRAMs whereas static variables are considered as registers in Vivado HLS.

5.2.6 Memory Instances and Custom Nodes

Whereas a lot of operators can be described through the specifications above, we provide individual specifications for the basic memory instances, line buffers and sliding window, to ease custom node description.
A line buffer can be instantiated as a C++ object.

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

Listing 11 Specification of a line buffer.

Then, it can be used within a program via the `shift` method which pushes `newDataBeat` as the `colIdx`th element and reads the corresponding col for the sliding window assignment.

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

Listing 12 Specification of a line buffer

Similarly, a sliding window can be specified as in Listing 13.

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

Listing 13 Specification of a sliding window.

Then, a new col can be pushed. Thus the window can be shifted, via the Application Programming Interfaces (APIs) provided in Listing 14. For border handling, the Boolean arrays `leftBorderFlags`, `rightBorderFlags`, should be given as inputs to the `shift` method. As an example, for a 7-by-7 kernel, `leftBorderFlags` should be `[false, true, false]` when the `colIdx` is 1. These flags can also be set easily with the control path policies explained in Section 6.

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

Listing 14 APIs for shift.

Whereas a sliding window can be read via the `get` method, the public member `win_out` can also be used, for instance within the data path functions expecting an array input.

```
DataType pix = swin.get(j, i);
DataType pix = swin.win_out[j][i];
```

Listing 15 APIs for read from a sliding window.

Furthermore, a custom node that does not use any of the provided specifications can be described, but then the parallelization should be supported manually as illustrated in Listing 16.

```
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;
}
```

Listing 16 Custom description of a global operator.

6 A Deeper Look into the Library

As motivated afore, an HLS library could be designed to comprehend alternative architectures for the same algorithmic instances to sustain efficiency for different design specifications and objectives. Whereas the details of the hardware architectures used in this paper can be found in [9], this paper deals with the design of such a library as a proof-of-concept example.

The proposed library has a policy-based structure. Template classes are defined for parametrization, and dynamic linking is avoided. An architecture selection algorithm that picks the expected best policy according to template parameters at compile time is integrated into the library for facilitating simple APIs. Yet, any supported policy can be individually utilized for advanced usage, i. e., for a DSE or a DSL design.

The Unified Modeling Language (UML) diagram of the proposed library is shown in Figure 5. A local operator is instantiated through three composite object classes. First, the registers and the shifting mechanism of the sliding window are set according to selected loop coarsening policy. Then selections, multiplexers (MUXs), and thus, final data assignments are determined through a border handling class, which composites a loop coarsening policy. Finally, the control path policy corresponding to the border handling policy employs the sliding window for a stencil calculation with appropriate control signals.

6.1 Loop Coarsening Policies

Two different loop coarsening architectures, one of which performs better than the other depending on the parallelization factor, kernel size, border handling pattern, and input/output data bit widths, are proposed in previous work [9]. The design of the data paths and the sliding window in these architectures, Fetch and Calc (F&C) and Calc and Pack (C&P) are illustrated in Figure 6. F&C has a larger sliding window, in which at least two consecutive data beats are fetched for handing the expected local sub-windows over to the parallelized data path. On the other hand, C&P hands over any local sub-window to the data path at the moment it is read and packs the results as appropriate output data beats after they are calculated at the cost of additional delay registers. A coarsening object, registers of a sliding window, can be specified as in Listing 17. Then,

```
// Policies: FetchAndCalc, CalcAndPack
// h: windowHeight, w: windowWidth
CoarseningPolicy<h, w, DataBeatType,
                 pFactor, DataType> swin;

// Reading from the coarsening object
swin.get(y, x);

// Updating the coarsening object
swin.setNewCol(colNew);   // colNew[h]
swin.setWindowNext(winN); // winN[h][w]
```

Listing 17 Specification of coarsening policies.

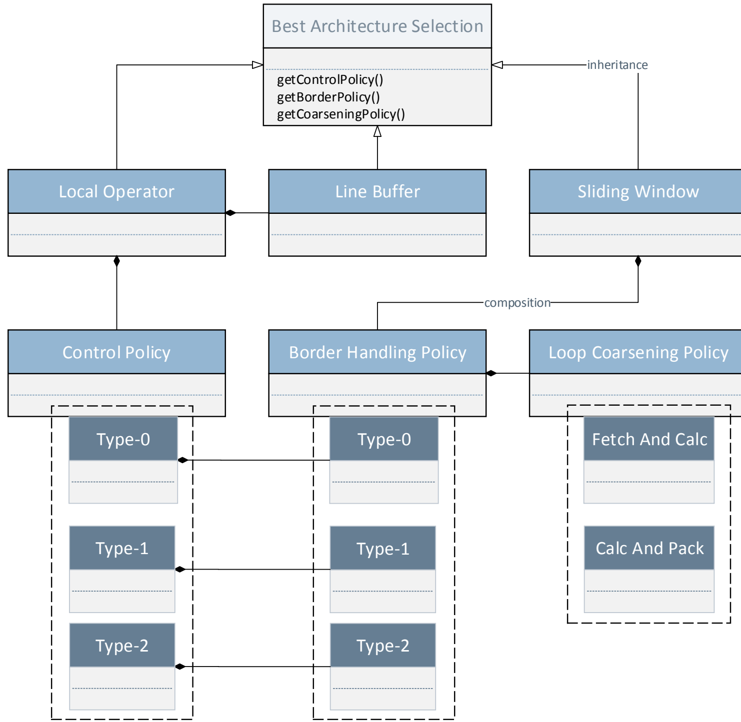
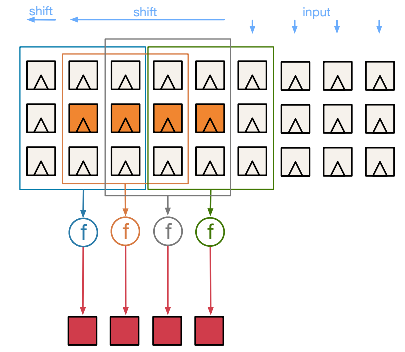
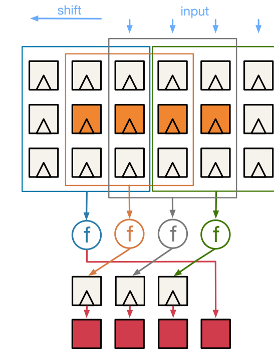


Figure 5 An object relationship diagram for our proposed library.

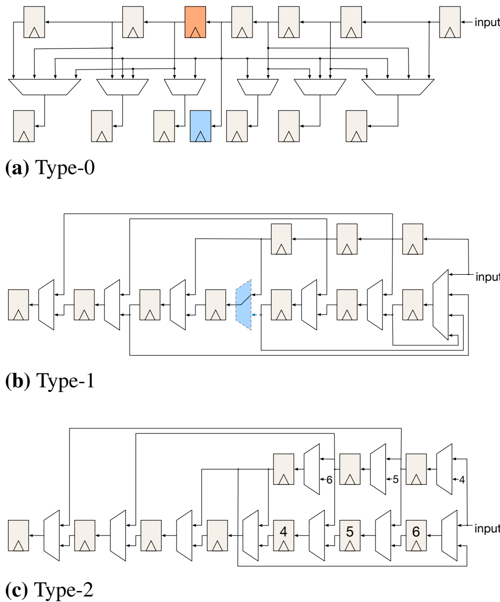


(a) Fetch And Calc



(b) Calc And Pack

Figure 6 Considered loop coarsening architectures for a 3-by-3 kernel where the parallelization factor is 4.



(a) Type-0

(b) Type-1

(c) Type-2

Figure 7 Column selection architectures for the mirror border mode.

a new col can be read via `setNewCol` and the new values after a shift can be assigned via `setWindowNext`. Finally, any pixel in a sliding window plus the latest read column can be read via `get`.

6.2 Border Handling Policies

Border handling conditions can be separated into vertical and horizontal sets to decrease the total number of data selections, thus facilitate significantly better implementations. The proposed library consists of templates for three different border handling architectures given in [9]. Type-1 minimize the area whereas Type-2 optimize the critical path where the parallelization factor is a factor of the image width. On the other hand, Type-0 can be used with any specification of a local operator but requires more area resources. Whereas the details of the architectures are not our main focus, their specifications require different APIs as shown in Listing 18.

```
// Policies: TYPE-0, TYPE-1, TYPE-2
BorderHandlingPolicy<ImageWidth, ImageHeight,
    WindowHeight, WindowWidth, DataBeatType,
    pFactor, DataType, BorderHandlingMode,
    CoarseningPolicy> brdrWin;

// APIs
swin.setNewCol(newCol, topBorderFlags,
                bottomBorderFlags);
// Type-0, Type-1
swin.shift(leftBrdrFlags, isCol0);
// Type-2
swin.shift(leftBrdrFlags, rightBrdrFlags);
```

Listing 18 Specifications for border handling policies.

6.3 Local Operator Control Path Policies

Line-buffered implementation of a local operator maximizes the throughput at a cost of initial latency, causing a more complicated HLS description. Similarly, border handling mentioned above and loop coarsening architectures require different control paths. Thus, they have different APIs. As a simplification, the proposed library is extended to offer control path policies for local operator implementations that can be used via the specifications in Listing 19.

```
// Policies: TYPE-0, TYPE-1, TYPE-2
ControlPathPolicy<ImageWidth, ImageHeight,
    KernelWidth, KernelHeight,
    DataBeatType, pFactor, DataType,
    BorderHandlingMode, CoarseningPolicy
    > cntrlWin;
```

Listing 19 Specifications for control path policies.

Furthermore, border handling policies are integrated to be components of the control policies to make a local operator description as simple as in Listing 20.

```
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);
}
```

Listing 20 Local operator control path via the interfaces of the control policies.

6.4 Automatic Architecture Selection

Providing multiple architectures for the same algorithmic instances complicates its usage. Yet, facilitating high performance in this way without sacrificing high productivity is possible with a compile time automatic architecture selection. As a proof-of-concept, the proposed library estimates and selects the best policies according to template parameters. The selection algorithm is based on the deterministic analysis in [9]. Whereas one of the loop coarsening architectures always performs better than the other, different border handling designs use different types of resources. Therefore, selection of a border handling policy is based on a design objective that can be given as a template parameter to local operator specification as shown in Listing 21. Nonetheless,

```
// designObjective LessLUTMoreRegister
// designObjective LessRegisterMoreLUT
localOp<ImageWidth, ImageHeight, KernelWidth,
        KernelHeight, DataBeatType, pFactor,
        DataType, BorderHandlingMode,
        designObjective> localOprtr;
```

Listing 21 Specification of a local operator with a design objective.

the library optimizes the number of LUTs as the default objective for the sake of simplicity.

6.5 RTL Level Optimizations

Current HLS tools mostly benefit from considerations at register-transfer level [2]. In fact, major FPGA vendors encourage considering the hardware architecture in their HLS optimization guides [10], [11]. These hardware specific improvements significantly increase the performance but result in tedious HLS codes. On the other hand, the performance can be sustained without the sacrifice of productivity by providing simple interfaces from a library or a framework. In this context, we discuss some of these optimizations at the register-transfer level in the following by providing proof-of-concept examples described using our proposed library.

Optimizations that can be enabled according to algorithm specifications can be considered with compile time flags. For instance, resource requirements for the control path of a local operator can be reduced in the case the image width is a power of two. The *if-else* condition in Listing 22 depends on the template parameters in the proposed library.

Arbitrary bit widths can be specified for the variables instead of typical data types.

Wire assignments at register-transfer level can be described in HLS through temporary registers updated in each iteration. For instance, no registers will be allocated for the *colIm* and *rowIm* variables when the *if* condition in Listing 22 satisfies.

Expressions in an application should be compressed wherever possible. In the case of multiple expressions in different parts of a program requiring the same comparison, the result can be assigned to a 1-bit flag once and used multiple times as in Listing 23. Note that the compile time flags are used in Listing 23 to exploit different possible reductions in different specifications.

Similarly, temporal locality of a control path can be exploited. For instance, the proposed library deploys only one comparison in the horizontal access for checking if a local operator enters to the border handling area and uses pipelined registers for handling the control signals accordingly, as shown in Listing 24.

```
// 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;
  }
}
```

Listing 22 Bit-level optimizations in the control flow.

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

Listing 23 Efficient usage of flags in the control flow.

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

Listing 24 Temporal locality optimizations in the control flow.

7 Evaluation and Results

In this section, we evaluate our proposed library by investigating implementation results for varying image processing algorithms. We targeted Zynq xc7z100ffg900-2 FPGA, in which a rate of 1,024 bits per cycle can be reached for streaming the data from the main memory to the reconfigurable logic. We provide implementation results for the Vivado HLS IP blocks obtained through Vivado HLS 2016.3 using *export* feature.

One of the main advantages of Vivado HLS is the ability to set different speed targets. The logic is highly pipelined in the case of ambitious speed targets, causing utilization of additional registers between LUT. Therefore, different target speed constraints should be taken into consideration when implementation results of an HLS code are evaluated. Moreover, we observed that the drastically high resource utilization for the same speed or relatively less logic speed despite the same area could be acquired when too ambitious or too relaxed target speed constraints are set. In this paper, we evaluate our algorithms with two different target speed constraints, which are 50 MHz and 200 MHz. As expected, the higher speed constraint slightly increases the number of the registers, and thus, the initial latency for all the architectures. Moreover, the effects of architectural differences to implementation results are amplified with faster logic frequencies.

Table 1 shows a comparison of different border modes regarding resources used for the implementation of a 5-by-5 integer Gaussian filter. The cost of any border handling mode comparing to no border handling can be seen by the comparison of Table 1 with Table 2. The constant border handling mode is the cheapest in terms of resources. A higher parallelization factor simplifies the data selection in border handling [9]. Thus, the implementation overhead of all the considered border handling modes converges for the large parallelization factors.

Table 3 shows the implementation results of the same kernel for different parallelization factors. It can be seen that the increase in throughput is linear to parallelization factor while the increase in resource usage is sublinear. Moreover, comparing Table 3 with Table 1 reveals that the C&P provides better results when no border handling is applied, which is not the case for all the local operators [9]. On the other hand, the user does not need to consider the implementation efficiency of the discussed architectures, since a seamless policy selection is applied by the integrated automatic selection when a local operator is described.

Finally, we compare our library with HIPA^{cc} [8], [12] for varying image processing algorithms in Table 4. HIPA^{cc} is a DSL that generates target efficient code from the description of a higher abstraction level. Its hardware back-end analysis the described algorithm, eliminates unnecessary memory transfers and generates efficient HLS code. In addition to its code generation, HIPA^{cc} uses a C++-based template library that consists of memory architectures for local and point operators. In order to make a fair comparison, we generated HLS codes for the algorithms in Table 4 via HIPA^{cc}, and replaced the corresponding memory architectures in the generated codes with our proposed library. In the end, all the competitors had exactly the same DFG and data path functions but different template libraries. We were able to describe all these algorithms easily with our proposed library and parallelize just with a global macro parameter.

Table 4 shows that the proposed library significantly improves the results for all the considered applications. More interestingly, we observe that the improvement in resource usage becomes becomes more significant for the more complicated data path functions. For instance, bilateral filter can only be parallelized by the factor of 2 using HIPA^{cc}, whereas the factor of 8 was possible with our proposed library. This reveals that the more a HLS code is written with hardware design manner, the better Vivado HLS performs. In addition, the latency results of our proposed library fits to the equations in [9], and smaller than the HIPA^{cc}.

8 Related Work

Vivado provides a video memory library for the algorithmic constructs sliding window and line buffer [10]. However, a user must write loops for local operator implementations himself/herself. For instance, the wait at the initial latency

Table 1 5-by-5 Gaussian filter with different border handling modes for a 1024×1024 gray scale image. The results are obtained for 50 MHz (left) and 200 MHz (right) speed constraints.

CF	Brdr Mode	SLICE	LUT	FF	BRAM	SRL	CPimp	Latency	SLICE	LUT	FF	BRAM	SRL	CPimp	Latency
1	CONST	119	301	360	4	0	4.4	1050630	198	374	881	4	10	2.1	1050639
1	CLAMP	121	335	361	4	0	3.6	1050630	222	453	922	4	9	2.3	1050639
1	MIRROR	128	356	361	4	0	3.8	1050630	226	511	922	4	10	2.3	1050639
1	MIRROR101	131	374	361	4	0	3.8	1050630	215	508	882	4	9	2.3	1050639
32	CONST	2431	6246	3035	32	0	7.3	32838	3465	8094	15607	32	198	3.0	32846
32	CLAMP	2406	6935	3042	32	0	8.3	32838	3591	8890	16188	32	200	2.8	32846
32	MIRROR	2240	6927	3042	32	0	7.1	32838	3576	8939	16272	32	200	3.4	32846
32	MIRROR101	2291	6928	3042	32	0	7.4	32838	3648	8928	16280	32	195	3.3	32846

Table 2 5-by-5 Gaussian filter with no border handling for a 1024×1024 gray scale image. The results are obtained for 50 MHz (left) and 200 MHz (right) speed constraints. The coarsening policy is F&C as in Table 1. Yet, this policy would not be selected since C&P is better as shown in Table 3.

CF	Brdr Mode	SLICE	LUT	FF	BRAM	SRL	CPimp	Latency	SLICE	LUT	FF	BRAM	SRL	CPimp	Latency
1	UNDEF	94	248	249	4	0	4.0	1050630	157	316	625	4	9	2.3	1050637
32	UNDEF	2238	6099	2683	32	0	6.2	32838	3211	8032	13380	32	202	2.8	32845

Table 3 5-by-5 Gaussian filter with different coarsening factors for a 1024×1024 gray scale image.

CF	Brdr Mode	SLICE	LUT	FF	BRAM	SRL	CPimp	Latency	SLICE	LUT	FF	BRAM	SRL	CPimp	Latency
1	UNDEF	105	247	249	4	0	3.4	1050631	161	316	625	4	10	2.2	1050637
2	UNDEF	156	438	295	4	0	3.3	525319	257	562	1029	4	16	2.4	525325
4	UNDEF	265	818	388	4	0	8.6	262663	439	1058	1823	4	28	2.4	262669
8	UNDEF	477	1576	577	8	0	9.5	131335	818	2055	3381	8	52	2.5	131341
16	UNDEF	981	3106	958	16	0	9.2	65671	1563	4125	6506	16	101	2.8	65677
32	UNDEF	1919	6160	1723	32	0	9.3	32839	3092	8034	12416	32	204	3.1	32845

Table 4 Benchmarking of the proposed library for different image processing applications.

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

and stalling the input should manually be described for a convolution. Therefore, a more software like algorithm instead of the optimal hardware implementation is given in the Xilinx optimization guide [10]. Another solution could be using the OpenCV library of Vivado HLS, which constraints the developer to provided functions only. How-

ever, their border handling solution applies padding, thus increases the latency. A similar approach to our solution, a library that provides template functions for point and local operators is proposed in [13]. Whereas the concept of line buffers and sliding windows are encapsulated in their local operator description, their specifications are not facilitated in this framework. Moreover, each algorithm construct maps to only one architecture. Neither the discussed solutions provided by Vivado nor the work in [13] facilitates any parallelization. Moreover, our implementations use less resources and processes faster than the both even without parallelization.

An alternative approach to HLS design in hardware manner could be Hardware Descriptive Language (HDL) inlining, which is not supported in Vivado HLS or Altera OpenCL. Furthermore, this would create a new challenge against automatic pipelining for higher speed constraints, which is not provided in HDL synthesis.

DSLs, [2], [12], [14], [15] for image processing are another alternative to the library approach. These image processing DSLs, Halide, PolyMage, HIPA^{cc}, are initially proposed for other computing platforms, i. e., CPU, GPU, but then, extended for the FPGAs using Vivado HLS. The loop coarsening architecture implemented in the proposed library uses less resources compared to HIPA^{cc}' [9]. Furthermore, our proposed library could be used as a building block for a DSL even to ease the description.

9 Conclusion

In this paper, we showed that apt descriptions in HLS are necessary for efficient FPGA implementations similar to other parallel computing devices such as GPUs. Moreover, we argued that a template library or a DSL can consist of more than one architectures to sustain implementation efficiency for different specifications of the same algorithm constructs. Besides, ease of use can be provided with automatic selection mechanisms that seamlessly utilize the expected best architecture, when a user instantiates an algorithmic construct via a simple API.

Additionally, our library can be used for the development of a DSL or a DSE as each policy can be instantiated individually. In particular, it can be useful since the provided parallelization approaches are missing in most of the existing DSLs and better than the others. Moreover, as the loop coarsening is vital for inference design in FPGA implementations of Convolutional Neural Network (CNN) or stencil-based iterations, the library can be extended to other domains.

Acknowledgment

This work is supported by the German Research Foundation (DFG), as part of the Research Training Group 1773 "Heterogeneous Image Systems."

References

[1] G. Martin and G. Smith, "High-level synthesis: Past, present, and future", *IEEE Design & Test of Computers*, vol. 26, no. 4, 2009.

[2] M. A. Özkan, O. Reiche, F. Hannig, and J. Teich, "FPGA-based accelerator design from a domain-specific language", in *26th Intl. Conf. on Field-Programmable Logic and Applications (FPL)*, IEEE, 2016.

[3] E. Kalali and I. Hamzaoglu, "FPGA implementation of HEVC intra prediction using high-level synthesis", in *The 6th IEEE International Conference on Consumer Electronics - Berlin*, IEEE, 2016.

[4] J. D. Owens, M. Houston, D. Luebke, S. Green, J. E. Stone, and J. C. Phillips, "Gpu computing", *Proceedings of the IEEE*, vol. 96, no. 5, pp. 879–899, 2008.

[5] A. Tousimojarad and W. Vanderbauwhede, "Comparison of three popular parallel programming models on the intel xeon phi.", in *Euro-Par Workshops (2)*, 2014.

[6] O. Reiche, C. Kobylko, F. Hannig, and J. Teich, "Auto-vectorization for Image Processing DSLs", in *18th Intl. Conf. on Languages, Compilers, Tools, and Theory for Embedded Systems (LCTES)*, (Barcelona), 2017.

[7] M. Schmid, O. Reiche, F. Hannig, and J. Teich, "Loop coarsening in c-based high-level synthesis", in *26th IEEE Intl. Conf. on Application-specific Systems, Architectures and Processors (ASAP)*, IEEE, 2015.

[8] O. Reiche, M. A. Özkan, F. Hannig, J. Teich, and M. Schmid, "Loop parallelization techniques for FPGA accelerator synthesis", *Journal of Signal Processing Systems*, 2017.

[9] 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 Annual IEEE Intl. Conf. on Application-specific Systems, Architectures and Processors (ASAP)*, (Seattle), Jul. 10–12, 2017.

[10] *Vivado design suite user guide, high-level synthesis*, UG902, Xilinx, 2017.

[11] *Intel FPGA SDK for OpenCL best practices guide*, Intel, 2017.

[12] O. Reiche, M. Schmid, F. Hannig, R. Membarth, and J. Teich, "Code generation from a domain-specific language for c-based hls of hardware accelerators", in *Hardware/-Software Codesign and System Synthesis (CODES+ ISSS), 2014 Intl. Conf. on*, IEEE, 2014, pp. 1–10.

[13] M. Schmid, N. Apelt, F. Hannig, and J. Teich, "An image processing library for C-based high-level synthesis", in *Proceedings of the 24th Intl. Conf. on Field Programmable Logic and Applications (FPL)*, (Munich, Germany), Sep. 2–4, 2014.

[14] J. Pu, S. Bell, X. Yang, J. Setter, S. Richardson, J. Ragan-Kelley, and M. Horowitz, "Programming heterogeneous systems from an image processing DSL", *CoRR*, vol. abs/1610.09405, 2016.

[15] N. Chugh, V. Vasista, S. Purini, and U. Bondhugula, "A dsl compiler for accelerating image processing pipelines on fpgas", in *Proceedings of the 2016 Intl. Conf. on Parallel Architectures and Compilation*, Haifa, Israel, 2016.