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), Dublin, Ireland, August 31, 2018.*

A Journey into DSL Design using Generative Programming: FPGA Mapping of Image Border Handling through Refinement

M. Akif Özkan[‡], Arsène Pérard-Gayot[†], Richard Membarth[†,*], Philipp Slusallek[†,*], Jürgen Teich[‡], and Frank Hannig[‡]
[‡]Friedrich-Alexander University Erlangen-Nürnberg (FAU), Germany
[†]Saarland University (UdS), Germany
[*]German Research Center for Artificial Intelligence (DFKI), Germany

Abstract

Field Programmable Gate Arrays (FPGAs) are continually improving their computing capabilities and energy efficiency. Yet, programming FPGAs remains a time-consuming task and requires expert knowledge to obtain good performance. Recent advancements in High-Level Synthesis (HLS) promise to solve this problem. However, today's HLS tools still require vendor-specific low-level optimizations in the form of compiler hints and code restructuring. Despite the pursuit of new programming methodologies for many-core, multi-threading, or vector architectures, the FPGA community mostly tries to improve the design techniques from existing programming languages that are either sequential or developed for other computing platforms. In this paper, we use a state-of-the-art functional language that offers explicit control over code refinement to design border handling circuits. This allows us to produce high-level, elegant code descriptions that can be easily refined to low-level hardware designs. Additionally, these descriptions can be exposed to software developers in the form of either a DSL or library.

1 Introduction

FPGAs provide high energy efficiency by reducing the off-chip communication through an application-tailored on-chip memory architecture. Yet, expert knowledge remains necessary for FPGAs, since an application-specific on-chip memory architecture and pipelined hardware units need to be *designed*. This is a time-consuming task, even for *specialists*. In general, there is no *optimal* implementation for a given algorithm. Even small modifications in large Hardware Description Language (HDL) projects become error-prone and time-consuming. Moreover, most of the algorithm developers are not familiar with hardware design at all. HLS was introduced to remedy these issues and has received a lot of attention over the last 30 years. After some early attempts, with the *second wave* [7], HLS tools have become able to generate high-quality results for data-path oriented applications. HLS vendors additionally support standard languages (e. g., C, C++, OpenCL) or system-level integration tools (e. g., Xilinx SDSoC) to help integrating the hardware accelerator into a heterogeneous system. Yet, most HLS tools are based on C-like languages and expect users to describe a hardware architecture with the help of preprocessor directives (*pragma*s), resulting in vendor-locked solutions.

We believe that the next step for HLS requires an increased level of abstraction on the language side, which might eliminate the need for expert knowledge when combined with modern metaprogramming approaches. One solution to solve this challenge are domain-specific languages and libraries. In fact, Domain-Specific Languages (DSLs) allow to raise the level of abstraction and to separate the *algorithmic description* from *hardware-specific transformations* such as parallelization, vectorization, or memory related optimizations. This facilitates fast prototyping while achieving high performance on different hardware platforms from the same high-level description. Furthermore, it relieves the programmer from hardware-specific optimizations which requires architecture expertise and is often error-prone, non-portable, and time-consuming.

Recent language and compiler research such as LMS [15] and AnyDSL [5, 6] has focused on simplifying the development of domain-specific libraries by decoupling language abstractions from compiler development. Thereby, a DSL can be developed just by code refinements of a functional language, which eliminates the need for laborious compiler changes. In this setting, the platform mapping of a DSL can be realized within the same language as the algorithm specification. In contrast to this, other approaches rely on a dedicated "back end" for a DSL, often working on the Abstract Syntax Tree (AST). Using code refinement removes the *black boxes* of a DSL compiler and makes the existent transformations extensible as shown in Figure 1. This is especially useful for the FPGA community, since in the literature, for many algorithms, more than a few Pareto-optimal hardware designs exist, instead of one *best* solution. Therefore, a DSL library that can easily be extended by any FPGA designer could facilitate the existence of a very comprehensive and efficient framework.

In this paper, we investigate the application of such language features for FPGA designs utilizing the infrastructure provided by AnyDSL. We use as a case study border handling for stencil functions, and explore how the *algorithm* can be decoupled from the *platform-specific mapping*. Then,

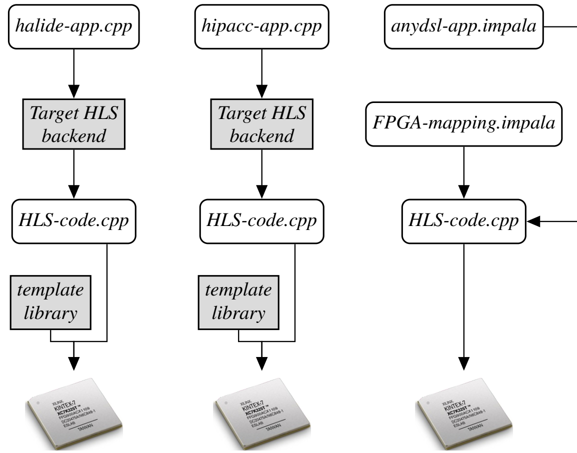


Figure 1 FPGA code generation flows for AnyDSL, Hipacc, and Halide. Hipacc and Halide have AST-based back ends to generate target HLS. The final code still has references to a template library that includes design patterns, such as line buffers and sliding windows. On the right, an application code written in Impala can be refined to efficient, target-specific codes with a DSL-mapping also implemented in Impala. This eases not only the DSL development but also allows developers to improve and understand the provided FPGA mapping.

we discuss the differences of an efficient FPGA mapping to the other platforms by applying *hardware-inspired* optimizations to the window-based implementation. Finally, we present border handling implementations for 2*D* stencil functions based on functions developed in the prior sections.

2 Background

2.1 AnyDSL and Impala

AnyDSL is a recently proposed compiler framework[1] [5, 6] that promises to ease DSL development via *shallow embedding* and *partial evaluation*. DSLs are embedded into Impala, an imperative and functional programming language that borrows from the language Rust. In contrast to Rust, Impala integrates a partial evaluator that allows the compiler to remove abstractions at compile time.

2.2 Impala Compiler

By default, only calls to higher-order functions are specialized so that closures are eliminated. As a consequence, there is no overhead in taking functions as parameters, as everything is resolved at compile time. In addition, the programmer can annotate the signature of a function to specify further conditions under which a function should be partially evaluated. For example, the following `@(?n)` annotation will only specialize calls to `pow` when `n` is known:

[1] https://anydsl.github.io

```
fn @(?n) pow(x: i32, n: i32) -> i32 {
  if n == 0 {
    1
  } else {
    if n & 1 == 0 {
      let y = pow(x, n / 2);
      y * y
    } else {
      x * pow(x, n - 1)
    }
  }
}
```

The compiler will look at every call site of the `pow` function, and specialize it when needed. As an example, the following calls

```
let z = pow(x, 5);          let z = pow(3, 5);
```

will result in the following equivalent sequences of instructions after specialization:

```
let y = x * x;              let z = 243;
let z = x * y * y;
```

The programmer can place arbitrary Boolean expressions after the `@`, and if no expression is present, the function will always be specialized.

Because iteration on various domains is a common pattern, Impala provides syntactic sugar to call a higher-order function using the **for** protocol. The following loop

```
for var1, ..., var2 in iter_func(arg1, ..., arg2) {
  /* ... */
}
```

translates to:

```
iter_func(arg1, ..., arg2,
  |var1, ..., var2| {
    /* ... */
  });
```

Here, the body of the **for** loop and the iteration variables constitute an anonymous function:

```
|var1, ..., var2| { /* ... */ }
```

that is passed to `iter_func` as the last argument. It can then be called from within the function `iter_func`. A programmer can, for instance, leverage this feature to write custom iteration functions that take advantage of the domain knowledge to increase performance.

2.3 Stencil Codes in Impala

A one-dimensional stencil function operating on a 1D data array can be written in C-like imperative style:

```
for i in range(0, size) {
  out(i) = 0.2f*arr(i-1) + 0.5f*arr(i) + 0.3f*arr(i+1);
}
```

Such a description is specific to the stencil kernel and the target device architecture. Impala's functional nature and partial evaluator allows decoupling of the algorithm from its schedule while sustaining the same optimizations of a low-level implementation. For instance, a kernel-independent `apply` function can be *refined* to equally optimized code as shown above for a constant `mask` array.

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

Figure 2 Common border handling patterns (see Equations (4a) to (4d) for the mathematical expressions).

```
fn @apply(arr: &[f32], mask: Mask, x: i32) -> f32 {
  let mut res = 0.0f;

  for i in unroll(-mask.lower, mask.upper) {
    let coeff = mask(i + mask.size / 2);
    if coeff != 0.0f {
      res += arr(x + i) * coeff;
    }
  }

  res
}
```

Then, a platform-specific schedule can be realized with a custom iteration function that applies the kernel body to the input data:

```
let mask = Mask { data : [ 0.2f, 0.5f, 0.2f ], ...  };

for x in platform_loop(arr) {
  out(x) = apply(arr, mask, x);
}
```

The iteration abstraction `platform_loop` can be refined to data-parallel processing for a GPU implementation as well as to a serial, vectorized, or parallel execution in the case of a CPU implementation.
Note that the platform-specific mappings of the schedule abstractions are intended to be provided in form of a library that can be linked to the common algorithmic abstractions and the application code according to the selected target platform.

3 Border Handling

Stencil functions depend on out-of-bounds pixels at the image borders. This creates artifacts that could become a severe problem for large filters or consecutive local operator processes. A solution is handling the data according to well-known border patterns as shown in Figure 2.
In this section, we investigate functional descriptions that refine algorithmic expressions derived from the mathematical expressions to efficient hardware architectures. Section 3.2 presents the basic descriptions and features of the considered border handling functions, which are used for a window-based implementation in Section 3.3.

3.1 Preliminaries

This section introduces the notation of important parameters used in this paper to increase clarity.
Let the range of a one-dimensional input space *in* be denoted by $[L,U]$ and f be a function operating over a neighborhood within a region $[wl_{in}, wu_{in}]$, then the stencil func-

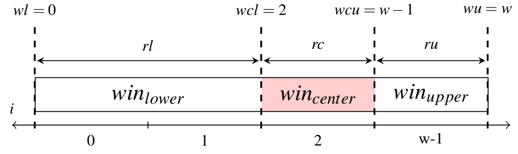


Figure 3 Representation of a stencil window array *win* with a size of w. A datum is stored in $win(i)$ for every window index $i \in [wl, wu)$. The results for the data in win_{center} depend on the elements in win_{lower} and win_{upper}. The sizes of these arrays are as follows: $|win_{lower}| = rl$, $|win_{center}| = rc$, $|win_{upper}| = ru$. The coordinates wl, wcl, wcu, and wu separates these three subarrays.

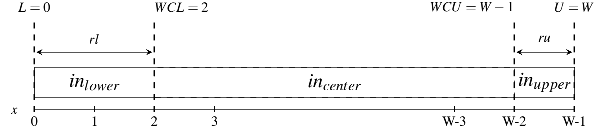
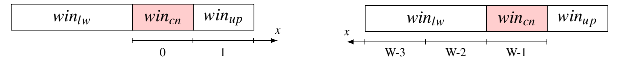


Figure 4 Representation of an input array *in* with a size of W. For every element $x \in [L,U)$ a datum can be read from *in*. $[WCL, WCU)$ denotes the region where no border handling is required for a stencil whose $rl = 2$ and $ru = 1$.



(a) Window is at the *left* border (x= 0). Border handling is necessary for the datum in win_{lower}.

(b) Window is at the *right* border (x= $W-1$). Border handling is necessary for the datum in win_{upper}.

Figure 5 Example scenarios to border handling. The x axis represents the window position at the input array, whose size is W.

tion for the output $y \in [x_{cl}, x_{cu})$ can be represented as:

$$y[x_{cl}, x_{cu-1}] = f(\{x_{wl}, \cdots, x_{wcl}, \cdots, x_{wcu}, \cdots, x_{wu-1}\}) \quad (1)$$

For the explanation purposes, we group the pixels within a stencil window to three regions, which are win_{lower}, win_{center}, and win_{upper} as shown in Figure 3. The results for the data in win_{center} depend on the elements in win_{lower} and win_{upper}. Correspondingly, sizes of these regions are denoted by;

$$rl = |wcl - wl|, \quad rc = |wcu - wcl|, \quad ru = |wu - wcu| \quad (2)$$

Thereby, the stencil function f depends on the pixels from outside the input space borders ($[L,U]$) at the area defined by;

$$L \leq x < WCL \quad \vee \quad WCU \leq x < U \quad (3)$$

Similarly, we split the input space coordinates to three regions, which are in_{lower}, in_{center}, and in_{upper}, as in Figure 4. Border handling for the datum in win_{lower} and win_{upper} is necessary for a stencil function operating at the in_{lower} and in_{upper} input space coordinates, respectively, as shown in Figure 5.

3.2 Border Handling Conditions

The different considered border handling conditions for a given input index x can be expressed as:

$$b_{mirror}(x,[L,U]) = \begin{cases} L+(L-x-1) & L > x \geq L-rl \\ U+(U-x-1) & U+ru > x \geq U \\ x & \text{else} \end{cases} \tag{4a}$$

$$b_{mirror101}(x,[L,U]) = \begin{cases} L+(L-x) & L > x \geq L-rl \\ U+(U-x-2) & U+ru > x \geq U \\ x & \text{else} \end{cases} \tag{4b}$$

$$b_{clamp}(x,[L,U]) = \begin{cases} L & L > x \geq L-rl \\ U-1 & U+ru > x \geq U \\ x & \text{else} \end{cases} \tag{4c}$$

$$fb_{constant}(x,in,cval,[L,U]) = \begin{cases} cval & L > x \geq L-rl \\ cval & U+ru > x \geq U \\ in[x] & \text{else} \end{cases} \tag{4d}$$

The functions *mirror*, *mirror*101, and *clamp* are the permutation functions mapping from the array indices, while the *constant* maps to the data.

$$f_{bh}(x,in,\cdots) = \begin{cases} in[\,b_{bh}(x)\,] & bh = mirror, mirror101, clamp \\ fb_{bh}(x,in,\cdots) & bh = constant \end{cases} \tag{5}$$

3.2.1 Decoupling the Iteration Schedule from the Data Assignments

The border handling functions in Equation (4) can be represented with Equation (6). Here, the input coordinate checks depends on the implementation schedule, which should be optimized according to the target platform. The function $f_{bh}(\cdots)$ consists of the data assignments according to the border handling algorithm. Therefore, the $f_{bh}(\cdots)$'s algorithmic description should be decoupled from the coordinate checks.

Furthermore, the lower and upper regions of the border handling functions can be detached to have an orthogonal set of the algorithm. Note that this allows calling $f_{bh_lower}(x)$ at the lower border and selecting different border patterns (i. e., mirror and constant) for the lower and upper regions.

$$f_{bh}(x,in,\cdots) = \begin{cases} f_{bh_lower}(x,in,L,\cdots) & L > x \geq L-rl \\ f_{bh_upper}(x,in,U,\cdots) & U+ru > x \geq U \\ f_{bh_center}(x,in,\cdots) & U > x \geq L \end{cases} \tag{6}$$

The algorithm (f_{bh}) depends on Equation (4), but the function of the constant pattern ($fb_{constant}$) returns a datum instead of an index compared to the other (f_{bh}) functions as shown in Equation (5). This can elegantly be described with Impala's functional enumerations, which allows the creation of a type which may be one of a few different variants.

```
enum BoundaryMode {
    Index(i32),
    Const(pixel_t)
}
```

```
fn @get_data(x: i32, read: fn(i32) -> pixel_t,
             boundary: Boundary, L: i32, U: i32,
             bh_lower: BoundaryFn,
             bh_upper: BoundaryFn) -> pixel_t {
  let mode = match boundary {
    Boundary::Lower  => bh_lower(x, L),
    Boundary::Center => BoundaryMode::Index(x),
    Boundary::Upper  => bh_upper(x, U)
  };

  match mode {
    BoundaryMode::Index(idx) => read(idx),
    BoundaryMode::Const(c)   => c
  }
}
```

Listing 1 Impala code for the reading a pixel from an input, which is expressed in Equation (4). The lower and upper parts of the expression are detached according to the Equation (6) and functional enumerations are used. Impala's partial evaluater checks compile-time known input parameters and generates static assignments that returns a pixel without function calls.

Thereby, mirroring in Equation (4a) can be described as follows:

```
fn @mirror_lower(idx: i32, lower: i32) -> BoundaryMode {
  BoundaryMode::Index(
    if idx < lower { lower + (lower - idx-1) }
    else { idx }
  )
}
fn @mirror_upper(idx: i32, upper: i32) -> BoundaryMode {
  BoundaryMode::Index(
    if idx >= upper { upper - (idx+1 - upper) }
    else { idx }
  )
}
```

Similarly, the constant in Equation (4d) can be modified to return a constant instead of a read index at the image borders.

```
fn @const_lower(idx: i32, lower: i32, cval: pixel_t)
              -> BoundaryMode {
  if idx < lower { BoundaryMode::Const(cval) }
  else { BoundaryMode::Index(idx) }
}

fn @const_upper(idx: i32, upper: i32, cval: pixel_t)
              -> BoundaryMode {
  if idx >= upper { BoundaryMode::Const(cval) }
  else { BoundaryMode::Index(idx) }
}
```

As an abstraction for reading data with a given border handling function, we consider a function that takes as input f_{bh_lower} and f_{bh_upper}, as well as the `read` function as shown in Listing 1.

Previous work [4] shows how boundary handling can be mapped efficiently for GPU architectures: The input data is divided into three different regions lower, center, and upper. For each of these regions code is generated with a static `boundary` assigned to avoid thread divergence. In this paper, we use the same interface and design FPGA mappings based on functional enumerations (see Section 4).

3.3 Border Handling from a Window

One possible border handling implementation is the padding of the input image according to the selected border pattern and cropping the output back to the original size. Yet, for FPGA designs, this stalls processing in the pipelining architecture and increases the data transfer size when reading from the host. A more efficient implementation transfers dependency elements to a faster memory, such as

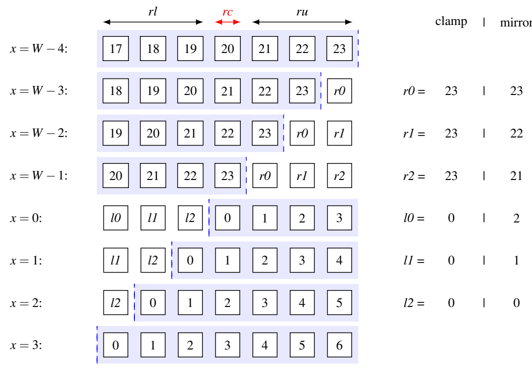


Figure 6 The proper set of values for an example 1-d window ($L = 0, WCL = 3, WCU = 4, U = 7$) at the input borders. The input coordinate of the stencil calculation is denoted by x, the input size $W = 24$. The boxes represents elements of the window and the values shows the expected input values. The blue background encapsulates the window elements within the input, thus the dashed lines show the input borders (L, U).

shared memory or registers, for a faster calculation and border handling. For instance, Figure 6 shows a 7×7 window at the borders of an input array with a size of $W = 24$. Figure 7 shows the corresponding indices to the same window for a stencil input.

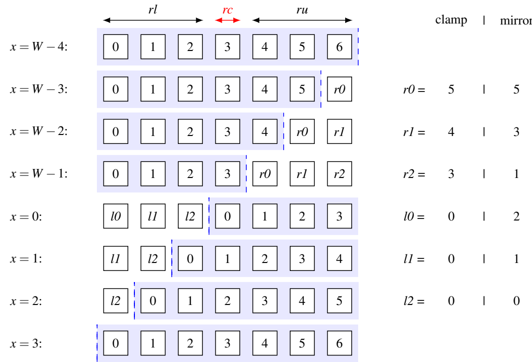


Figure 7 Analysis of the window-based border handling implementation for a 1-d window ($L = 0, WCL = 3, WCU = 4, U = 7$). The values (of the boxes) shows the permutation index of the window elements according to the Equation (8). For instance $win[4]$ should be read from the $win[3]$ at $x = 0$. (x: input coordinate of the execution, (blue) colored region of the window stays within the image, the dashed borders shows wcl_x and wcu_x in Equation (7).

However, this complicates border handling since the algorithm f_{bh} needs to be changed to read from the window instead of the input. The challenge is that the upper and lower bounds (U, L) of an input are fixed, but the bounds (wu, wl) change according to the input coordinate x. Yet, the here considered border handling functions in Equation (4) (algorithm) can be reused when the following transformations (mapping) are applied:

$$L \rightarrow wcl_x = wcl - (x - L), \tag{7a}$$

$$U \rightarrow wcu_x = wcu + (x - WCU), \tag{7b}$$

$$x \rightarrow i \tag{7c}$$

$$f_{bh}(x, in, [L, U], \cdots) \rightarrow fwb_{bh}(x, i, win, [L, U], [wcl, wcu], \cdots) =$$

```
fn @(?x) get_data_wnd(x: i32, i: i32,
                      read: fn(i32) -> pixel_t,
                      boundary: Boundary,
                      L: i32, WCL: i32, WCU: i32, U: i32,
                      wcl: i32, wcu: i32,
                      bh_lower: BoundaryFn,
                      bh_upper: BoundaryFn) -> pixel_t {
  let (wclx, wcux) = match boundary {
    Boundary::Lower  => (wcl - (x - L), 0),
    Boundary::Center => (0, 0),
    Boundary::Upper  => (0, wcu + (x - WCU))
  };
  get_data(i, read, boundary, wclx, wcux,
           bh_lower, bh_upper)
}
```

Listing 2 Pixel read within a window, which is the Impala code for Equation (7d). In this way, the border handling expressions (algorithm) in Equation (4) (see Listing 1) is mapped to the window-based implementations in Equation (8). The input parameter x, denoted by `@`, can't be known at the compile time, therefore the pixel read assignments depends on x in the generated code.

$$\begin{cases} f_{bh_lower}(i, win, wcl_x, \cdots) & WCL > x \geq L \\ f_{bh_upper}(i, win, wcu_x, \cdots) & U > x \geq WCU \\ f_{bh_center}(i, win, \cdots) & WCU > x \geq WCL \end{cases} \tag{7d}$$

Applying the transformations (*mapping*) to the border handling expressions (*algorithm*) in Equation (4) yields the following equations that express a window-based *implementation*:

$$wb_{mirror}(x, i, [L, WCL, WCU, U], [wcl, wcu]) = \tag{8a}$$
$$\begin{cases} 2 \times (wcl - (x - L)) - (i + 1) & WCL > x \geq L \\ 2 \times (wcu + (x - WCU)) - (i + 1) & U > x \geq WCU \\ i & \text{else} \end{cases}$$

$$wb_{mirror101}(x, i, [WCL, WCU], [wcl, wcu]) = \tag{8b}$$
$$\begin{cases} 2 \times (wcl - (x - L)) - i & WCL > x \geq L \\ 2 \times (wcu - 1 + (x - WCU)) - i & U > x \geq WCU \\ i & \text{else} \end{cases}$$

$$wb_{clamp}(x, i, [WCL, WCU], [wcl, wcu]) = \tag{8c}$$
$$\begin{cases} wcl - (x - L) & WCL > x \geq L \\ wcu - 1 + (x - WCU) & U > x \geq WCU \\ i & \text{else} \end{cases}$$

$$fwb_{constant}(x, i, win, cval, [WCL, WCU], [wcl, wcu]) = \tag{8d}$$
$$\begin{cases} cval & WCL > x \geq L \\ cval & U > x \geq WCU \\ i & \text{else} \end{cases}$$

A window-based data read function using the `get_data` of Listing 1 is shown in Listing 2.

4 FPGA Mapping

Data selection can be implemented with multiplexer (MUX) digital circuit elements, and border handling can be implemented as data-selection circuits for FPGAs. Current HLS tools are prone to generate non-optimal hardware from regular `if/else` expressions [11]. This issue is addressed in Section 4.1 and a mapping function for implementing a

```
fn @(?cnd) MUX(cnd: bool, assign_true: fn()->(),
               assign_false: fn()->()) -> () {
  assign_false();
  if cnd {
    assign_true();
  }
  // if cndN { .. } for an N-input MUX
}
```

Listing 3 Impala representation of a MUX following the hardware-inspired design rules [11]. The parameter `cnd` is only specialized if known at compile-time as implied by the `@(?)` annotation. The `assign` functions contain data assignments rather than data-path calculations.

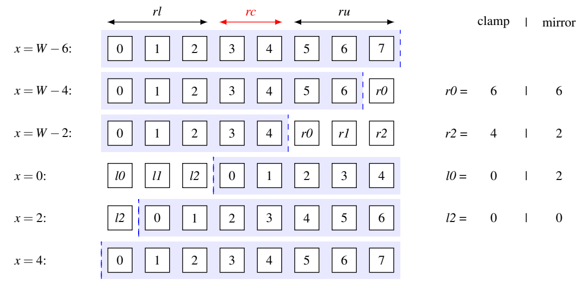
MUX array is explained in Section 4.2. Finally, the hardware implementation of image border handling is shown in Section 4.3.

4.1 Description of a MUX for HLS

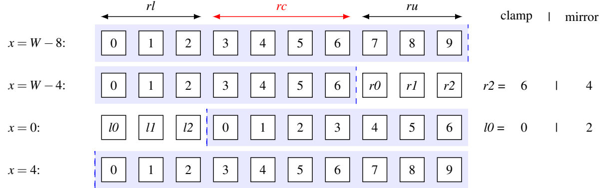
The main difference between hardware and software is as follows [11]: On the one hand, conditional expressions of an optimized software implementation execute only the relevant operations of one branch while the arithmetic operations for the other branches are skipped. On the other hand, an FPGA implementation allocates corresponding MUXs for a conditional logic, applies all the arithmetic operations of all cases, and selects the relevant output according to the input. This is similar to the *predicated execution* of Very-Long Instruction Word (VLIW) architectures.
The performance can be significantly improved with *hardware-inspired code* descriptions [2, 11]: **(Rule-1)** Similar to the digital logic of a MUX, input and output switching for all possible conditional cases should be known at compile time. Moreover, assigning data instead of array indices wherever possible simplifies the HLS compiler's task. **(Rule-2)** Arithmetic operations of both *true* and *false* cases should always be calculated before the data assignments. **(Rule-3)** All data assignments depending on the same conditional signal should be written in the same `if/else` body. Otherwise, control logic is replicated. A MUX following the above guidelines can be represented as in Listing 3. Having as input parameter the data assignment functions instead of data itself allows describing the MUX invariant to the type and amount of the data read and written.

4.2 MUX Network for 1D Border Handling

Efficient FPGA implementations for border handling consists of MUXs and operates on on-chip memory instances, even for the input padding. However, the window-based `get_data_wnd` function should be modified according to the guidelines explain in Section 4.1. In this section, we explain the following modifications that can be applied to Listing 2 for designing a generic MUX network generator functions (*mappings*) in Listing 4b:
(R1) static data assignments for all possible conditions: The parameters wcl_x and wcu_x depend on the input array coordinate x, which is not know at compile time. This implies that neither Impala's partial evaluator nor the HLS compilers can deduce all possible inputs for each output, as this can escalate to the input size depending on the use. Therefore, we need to write a *hardware inspired* code that



(a) Coarsening factor = 2



(b) Coarsening factor = 4

Figure 8 Dashed lines show the input borders (wcl, wcu). The number of possible coordinates that wcu decreases with larger steps (coarsening factors v). This simplifies the border handling since number of permutations to every element decreases.

generates static assignments for every output and for every possible (wcl_x, wcu_x). The described hardware mapping can mathematically be expressed as follows (which changes (wcl_x, wcu_x) in Equation (8)):

$$wcl_x = \begin{cases} wcl - wi & x = L + wi \\ wcl - \cdots & \cdots \\ wcl - (wl - 1) & x = L + (wl - 1) \end{cases}, wi \in [0, rl) \quad (9a)$$

$$wcu_x = \begin{cases} wcu + wi & x = U - 1 - wi \\ wcu + \cdots & \cdots \\ wcu + (wu - 1) & x = U - wu \end{cases}, wi \in [0, ru) \quad (9b)$$

This variable amount of regions can conveniently be described with Impala's partial evaluator with unrolled loops as follows (lines 17–36 in Listing 4b):

```
Boundary::Lower => {
  for wi in unroll(0, wl) { // for every possible wclx
    if x == L + i {
      // data assignments ( wclx = wcl - wi )
      ... = get_data(..., wcl - wi, 0, ...);
    }
  }
}
```

(R2-R3) input parameter data assignments function: **(R2)** Having an input function that consists of data assignments as in Listing 3 enables different code mappings such as writing all assignments at the deepest body of an `if` expression. Xilinx considers such a loop nest to be *perfect* [17]. Thus, we depend on the DSL developer to write a data assignment function as in Listing 4a. Data reads can use the `get_data` (see Listing 1) function for the border handling. **(R3)** Separating data assignments from the data selection expression facilitates a more generic function that describe only the MUXs. Moreover, this allows to generate a cleaner final code where the same MUX array can be shared for different data assignments. For instance, the following two data selections (MUXs) have the same structure:

```
fn @assign_data(/* similar to get_data */) -> () {
  for /* all the assignments */ {
    let data = get_data(i, read, boundary, wclx, wcux,
                        bh_lower, bh_upper);
    out.write(xw, ..., data);
  }
}
```

(a) Data assignment function for the MUX network function (see Listing 4b). This function should not consist the arithmetic operations.

```
fn @(?x) muxes_for_the_bound(/*x, boundary, v, in_bounds,
    get_bounds, bh_lower, bh_upper*/) -> () {

  // parameters from the API
  let (L, U) = in_bounds;
  let (wbl, wbu) = get_bounds(boundary); // win
  let rb = wbu - wbl; // rb = {rl, rc, ru}

  // extend assign_data to the boundary
  fn @for_the_bound(/* boundary, wclx, wcux */) -> () {
    for i in unroll(lower_w, upper_w) {
      assign_data(i, boundary, wclx, wcux,
                  bh_lower, bh_upper);
    }
  }

  // Boundary::Center (default)
  for_the_bound(Boundary::Center, 0, 0);

  match boundary {
    Boundary::Lower => {
      for wi in unroll_step(0, rb, v) { // rb = wcl - wl
        let wclx = wcl - wi;
        if x == L + wi / v {
          for_the_bound(boundary, wclx, 0);
        }
      }
    },
    Boundary::Upper => {
      for wi in unroll_step(0, rb, v) { // rb = wu - wcu
        let wcux = wcu + wi;
        if x == U - 1 - wi / v {
          for_the_bound(boundary, 0, wcux);
        }
      }
    },
  }
}
```

(b) A function that generates a MUX array for a given boundary.

Listing 4 A mapping function that takes the *algorithm* description of the border handling as input and generates a MUX array as an efficient FPGA *implementation*. Corresponding mapping for a GPU or a CPU can be developed with a window based data read function given in Listing 2.

```
// write(i, get_data_wnd(i, read, ...));
if common_cond { win(0, 3) = get_data(..., wclx, ...); }
if common_cond { win(1, 3) = get_data(..., wclx, ...); }
```

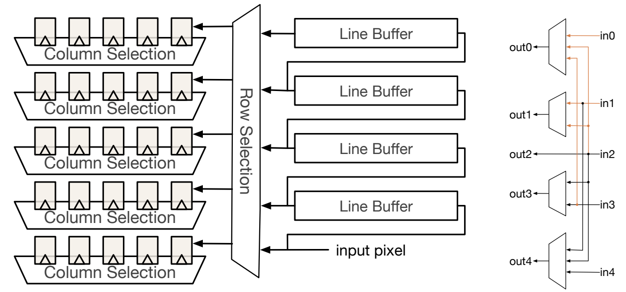
This can better be described as follows:

```
fn @ex_assign(common_cond: bool) -> () {
  win(0, 3) = get_data(..., wclx, ...);
  win(1, 3) = get_data(..., wclx, ...);
}
if common_cond { ex_assign(...); }
```

A cleaner HLS code eliminates the chance of unnecessary replications of control logic for the MUXs with the same control signal.

4.2.1 Loop Coarsening

It is often the case that the stencil window moves with large steps and processes multiple data at each coordinate (loop coarsening [12]). This implies that the set of wi in Equation (9) gets correspondingly reduced since the number of possible wcl and wcu decreases as shown in Figure 8. Thereby, wi in Equation (9) changes to Equation (10) (lines



(a) selections from on-chip memory **(b)** selection

Figure 9 Separated border handling.

22 and 30 in Listing 4b) for the coarsening factor v.

$$wi = \in \begin{cases} 0, v, \cdots, \lfloor rl/v \rfloor & WCL > x \geq L \quad \text{// lower} \\ 0, v, \cdots, \lfloor ru/v \rfloor & U > x \geq WCU \quad \text{// upper} \end{cases} \tag{10}$$

4.2.2 Specification of the API

As indicated through Equation (7d), data assignments (f_{bh}) can be called with the same function for a border region (i. e., lower and upper). As we argued before, (f_{bh}) corresponds to the *platform-mapping* where the border region is a *schedule* element, Listing 4b specializes to the border region with Impala's partial evaluation. Correspondingly, it extends the input data assignment function for all the outputs of a given boundary (`for_the_bound` function in lines 10–15).

On the downside, such a function requires too many input parameters that can decrease the usability. An elegant solution is having as input parameter a function that provides the relevant data as follows:

```
let bounds = new_bounds(0, wl, wcu, wu);
```

Then, the parameters rl, rc, ru, wl, wcl, wcu, wu can be fetched just with lines 6–7 of Listing 4b. Similarly, input parameters L and U can be given as a bound structure as shown in line 5 of Listing 4b.

4.3 Border Handling for Stencil Codes

The two-dimensional border handling is orthogonal to the 1D border handling (expressed with Equation (4)):

$$f_{bh}(y, x, in_{(x,y)}) = f_{bh}(y, f_{bh}(x, in_x)) \tag{11}$$

Therefore, border handling for a $w \times h$ 2D stencil computation can be implemented with two 1D MUX nets with sizes of w and h. Such an implementation for a stencil micro-architecture implementation is shown in Figure 9a (see also [12]). The border handling for the horizontal scan is called column selection and row selection for the vertical scan. Note that the row selection is applied before the data is fetched to the on-chip memory registers in order to reduce resource utilization. An alternative implementation could implement w times row selection before or after column selection.

This can easily be implemented using the MUXs network abstraction in Listing 4b. First, a 1D MUX net for all the boundary regions of the line buffer and sliding window

memories can be generated as follows:

```
1 fn @(?x) muxes_1d(/*in_bounds, get_bounds, v,
2                     assign_data*/) -> () {
3   for boundary in (/* Lower, Center, Upper */) {
4     muxes_for_the_bound(x, boundary, ...);
5   }
6 }
```

This generates a circuit as shown in Figure 9b. Next, the `muxes_1d` function is called with the corresponding parameters. The corresponding row selection of Figure 9a for a $W \times H$ input can be described as follows:

```
fn assign_data = @|...| {
  // assignments from line buffer to sliding window
  swin.write(4, wi, get_data(wi, lbuf.read, ...)
}

let v = 1;
let in_bounds = (0, H);
let bounds = new_bounds(0, 2, 3, 5);
muxes_1d(in_bounds, bounds, v, assign_data);
```

`assign_data` reads from line buffer and writes to the sliding window (to the row index 4 above) according to the border handling. Window moves by 1 data steps on the vertical axis, thus $v = 1$. Note that a y-coordinate counter with an offset still can be used by changing `in_bounds`.
Similarly, the column selection can be called as follows:

```
fn assign_data = @|...| {
  // assignments from sliding window to output image
  for j in unroll(0, 5) { // assigns of 5 col. sel.
    write(wi, j, get_data(wi, swin.read, ...));
  }
}

let v = 1;
let in_bounds = (0, W); // (0, 1 + (W - 1) / v)
let bounds = new_bounds(0, 2, 3, 5);
muxes_1d(in_bounds, bounds, v, assign_data);
```

Different to the row selection above, `assign_data` consists of data assignments for all 5 column selection circuits in Figure 9a since all depends on the same schedule conditionals (Rule-3 in Section 4.1). Moreover, border handling for loop coarsening can be specified just by setting the `v`, and the corresponding W as above.
As can be seen, the specified API (in Section 4.2.2) simplifies descriptions while directing the DSL developer to writing the expected code. Impala's functional features and partial evaluation allows powerful code refinement ability. Furthermore, all these mapping functions are meant to be hidden from the application developer, which can be a software developer with no FPGA knowledge.

5 Evaluation and Results

We present results for a Cyclone 5GT-5CGTD9 FPGA using Intel Intel SDK for OpenCL (AOCL) v17.1. The implementation with no border handling is named as *nobh*. Table 1b shows the results for a common NDRange kernel that reads input similar to the `mirror_lower` in Section 3.2.1. Table 1a shows the results for a single-threaded OpenCL kernel using our DSL approach, in which a MUX network is generated as explained in Section 4.3.
The cost of the border handling is negligible when implemented with our DSL approach. On the other hand, a severe slow-down and increased resource usage is observed for the FPGA implementation derived from an OpenCL kernel initially targeted for another platform. In fact, we couldn't measure the latency of the mirroring of the NRange kernel, since the execution hangs within a deadlock loop.

Table 1 Implementation and runtime execution results of a 5×5 mean filter for an input of size 1024×1024. It can be seen that the cost of the border handling is very minor with our DSL approach.

(a) Single-threaded OpenCL kernel generated from Impala

Border	latency (ms)	M10K	ALM	ALUT	FF
nobh	12.009	346	27589	26507	45153
clamp	12.358	347	28324	27147	45968
mirror	12.276	347	28288	27010	45951

(b) NDRange OpenCL kernel

Border	latency (ms)	M10K	ALM	ALUT	FF
nobh	141.980	392	30087	40068	53688
clamp	83.359	570	31099	43178	59086
mirror	-	570	31569	44108	59679

6 Related Work

DSLs for FPGAs: The ever-increasing variety of electronics and embedded computing platforms escalates the need for efficient design concepts. Thus, DSLs are gaining traction for digital hardware design. In the domain of image processing, Rigel [3], the work of Pu et al. [13] (based on Halide), HIPA^{cc} [14], and PolyMage [1] create image processing pipelines from a DSL. Rigel/Halide and PolyMage are declarative DSLs whereas HIPA^{cc} is embedded into C++. While Rigel directly generates HDL code, the others leverage datapath optimizations of HLS, that is, automatic pipelining for higher clock frequencies and, hence, use domain knowledge for efficient FPGA designs.
Generative Programming: Algorithms perform better when they are tailored to one use-case. In particular for FPGAs, a lot of the performance is obtained through domain-specific knowledge. In order to take advantage of such knowledge, generative programming can be used to specialize code. Metaprogramming is a typical way to reach that goal, and has been used for image processing [9, 10], machine learning [16], parallel processing [8]. Our work uses partial evaluation, a technique which has already been employed for this purpose [5, 6]. Compared to metaprogramming, partial evaluation operates on the source language and preserves the well-typedness of programs.

7 Conclusion

In this paper, we show how generative programming can help in making FPGAs more accessible to software developers. Using the AnyDSL compiler framework allows us to describe FPGAs designs in a library or DSL. We argue that a framework that can target multiple platforms can be developed by decoupling the iteration schedule from the data assignments. We present our approach by taking the image

border handling algorithms as examples, expressing them in mathematical equations and discussing all the intermediate steps until we derive a mapping function that generates a hardware-favorable HLS code from the *data assignment* functions of the border handling patterns. In the end, we show that HLS code (in OpenCL language) generated by our approach performs significantly better than an OpenCL code written for many-core architectures.

Acknowledgments

This work is supported by the German Research Foundation (DFG) as part of the Research Training Group 1773 "Heterogeneous Image Systems", the Federal Ministry of Education and Research (BMBF) as part of the Metacca and ProThOS projects, and the Intel Visual Computing Institute (IVCI) as well as the Cluster of Excellence on Multimodal Computing and Interaction (MMCI) at Saarland University.

Literature

[1] N. Chugh, V. Vasista, S. Purini, and U. Bondhugula. "A DSL compiler for accelerating image processing pipelines on FPGAs". In: *Proceedings of the International Conference on Parallel Architecture and Compilation Techniques (PACT)*. IEEE. 2016, pp. 327–338. DOI: 10.1145/2967938.2967969.

[2] R. Domingo, R. Salvador, H. Fabelo, D. Madroñal, S. Ortega, R. Lazcano, E. Juárez, G. Callicó, and C. Sanz. "High-level design using Intel FPGA OpenCL: A hyperspectral imaging spatial-spectral classifier". In: *12th International Symposium on Reconfigurable Communication-centric Systems-on-Chip (ReCoSoC)*. IEEE. 2017, pp. 1–8. DOI: 10.1109/ReCoSoC.2017.8016152.

[3] J. Hegarty, R. Daly, Z. DeVito, J. Ragan-Kelley, M. Horowitz, and P. Hanrahan. "Rigel: Flexible multi-rate image processing hardware". In: *ACM Transactions on Graphics (TOG)* 35.4 (2016), 85:1–85:11. DOI: 10.1145/2897824.2925892.

[4] M. Köster, R. Leißa, S. Hack, R. Membarth, and P. Slusallek. "Platform-Specific Optimization and Mapping of Stencil Codes through Refinement". In: *Proceedings of the 1st International Workshop on High-Performance Stencil Computations (HiStencils)*. 2014, pp. 1–6.

[5] R. Leißa, K. Boesche, S. Hack, R. Membarth, and P. Slusallek. "Shallow embedding of DSLs via online partial evaluation". In: *Proceedings of the International Conference on Generative Programming: Concepts & Experiences (GPCE)*. ACM. 2015, pp. 11–20. DOI: 10.1145/2814204.2814208.

[6] R. Leißa, K. Boesche, S. Hack, A. Pérard-Gayot, R. Membarth, P. Slusallek, A. Müller, and B. Schmidt. "AnyDSL: A partial evaluation framework for programming high-performance libraries". In: *Proceedings of the International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*. ACM. 2018, pp. 1–28.

[7] G. Martin and G. Smith. "High-level synthesis: Past, present, and future". In: *IEEE Design & Test of Computers* 26.4 (2009), pp. 18–25. DOI: 10.1109/MDT.2009.83.

[8] C. J. Newburn, B. So, Z. Liu, M. D. McCool, A. M. Ghuloum, S. D. Toit, Z. Wang, Z. Du, Y. Chen, G. Wu, P. Guo, Z. Liu, and D. Zhang. "Intel's array building blocks: A retargetable, dynamic compiler and embedded language". In: *Proceedings of the 9th International Symposium on Code Generation and Optimization (CGO)*. (Chamonix, France). IEEE. 2011, pp. 224–235. DOI: 10.1109/CGO.2011.5764690.

[9] G. Ofenbeck, T. Rompf, A. Stojanov, M. Odersky, and M. Püschel. "Spiral in Scala: Towards the systematic construction of generators for performance libraries". In: *Proceedings of the International Conference on Generative Programming: Concepts & Experiences (GPCE)*. (Indianapolis, IN, USA). ACM. 2013, pp. 125–134. DOI: 10.1145/2517208.2517228.

[10] M. A. Özkan, O. Reiche, F. Hannig, and J. Teich. "A Highly Efficient and Comprehensive Image Processing Library for C++-based High-Level Synthesis". In: *Proceedings of the Fourth International Workshop on FPGAs for Software Programmers (FSP)*. (Ghent). 2017.

[11] M. A. Özkan, O. Reiche, F. Hannig, and J. Teich. "FPGA-based accelerator design from a domain-specific language". In: *Proceedings of the International Conference on Field Programmable Logic and Applications (FPL)*. IEEE. 2016, pp. 1–9. DOI: 10.1109/FPL.2016.7577357.

[12] 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: *Proceedings of the International IEEE Conference on Application-specific Systems, Architectures and Processors (ASAP)*. IEEE. 2017, pp. 155–163. DOI: 10.1109/ASAP.2017.7995273.

[13] J. Pu, S. Bell, X. Yang, J. Setter, S. Richardson, J. Ragan-Kelley, and M. Horowitz. "Programming heterogeneous systems from an image processing DSL". In: *ACM Transactions on Architecture and Code Optimization (TACO)* 14.3 (2017), 26:1–26:25. DOI: 10.1145/3107953.

[14] O. Reiche, M. A. Özkan, R. Membarth, J. Teich, and F. Hannig. "Generating FPGA-based image processing accelerators with Hipacc". In: *Proceedings of the International Conference on Computer Aided Design (ICCAD)*. (Irvine, CA, USA). Invited Paper. IEEE. 2017, pp. 1026–1033. DOI: 10.1109/ICCAD.2017.8203894.

[15] T. Rompf and M. Odersky. "Lightweight modular staging: A pragmatic approach to runtime code generation and compiled DSLs". In: *Proceedings of the International Conference on Generative Programming and Component Engineering (GPCE)*. ACM. 2010, pp. 127–136. DOI: 10.1145/1868294.1868314.

[16] A. K. Sujeeth, H. Lee, K. J. Brown, T. Rompf, H. Chafi, M. Wu, A. R. Atreya, M. Odersky, and K. Olukotun. "OptiML: An implicitly parallel domain-specific language for machine learning". In: *Proceedings of the 28th International Conference on Machine Learning (ICML)*. (Bellevue, Washington, USA). Omnipress. 2011, pp. 609–616.

[17] Xilinx. *Vivado Design Suite user guide high-level synthesis UG902*. 2017.