This is the author's version of the work. The definitive work was published in *Proceedings of the 28th Annual IEEE International Conference on Application-specific Systems, Architectures and Processors (ASAP), Seattle, USA, July 10th-12th, 2017.*

Hardware Design and Analysis of Efficient Loop Coarsening and Border Handling for Image Processing

M. Akif Özkan, Oliver Reiche, Frank Hannig, and Jürgen Teich
Hardware/Software Co-Design, Department of Computer Science,
Friedrich-Alexander University Erlangen-Nürnberg (FAU), Germany.
{akif.oezkan,oliver.reiche,hannig,teich}@cs.fau.de

***Abstract*—Field Programmable Gate Arrays (FPGAs) excel at the implementation of local operators in terms of throughput per energy since the off-chip communication can be reduced with an application-specific on-chip memory configuration. Furthermore, data-level parallelism can efficiently be exploited through so-called loop coarsening, which processes multiple horizontal pixels simultaneously. Moreover, existing solutions for proper border handling in hardware show considerable resource overheads.**

In this paper, we first propose novel architectures for image border handling and loop coarsening, which can significantly reduce area. Second, we present a systematic analysis of these architectures including the formulation of analytical models for their area usage. Based on these models, we provide an algorithm for suggesting the most efficient hardware architecture for a given specification. Finally, we evaluate several implementations of our proposed architectures obtained through Vivado High-Level Synthesis (HLS). The synthesis results show that the proposed coarsening architecture uses 32 % less registers for a 5-by-5 convolution with a 64 coarsening factor compared to previous works, whereas the proposed border handling architectures facilitate a decrease in the Look-up Table (LUT) usage by 36 %.

I. Introduction

A great portion of image processing applications can be expressed as a task graph composed of point, local, and global operators. A point operator computes an output pixel only from one corresponding input pixel, whereas the access patterns of local operators also involve neighboring pixels. Global operators, on the other hand, consider the entire input image to calculate a single output pixel. FPGAs become top-notch platforms in terms of throughput per energy for the implementation of fixed point applications based on local operators. As nearly all image processing applications include multiple local operators. Even the smallest improvements in their implementations may provide significant improvements in terms of resource requirements and throughput.

A drawback of FPGAs is the typically quite low achievable clock frequency in comparison to state-of-the-art CPUs. Yet, this can be circumvented by reading at external memory speed and streaming multiple data elements in parallel into the reconfigurable logic. A ZYNQ-zc706 FPGA, as an example, can stream 1,024 bits of data concurrently into the logic. Consequently, algorithms can run at the memory bandwidth limit if the data path is sufficiently parallelized. For instance, a local operator performing on a 8-bit grayscale image must be parallelized by a factor of 128 to reach the memory bandwidth limit. A naive approach would be to clone the entire accelerator by a factor of 128. However, the area, thus power, can significantly be reduced through better resource sharing by replicating only the local operator, as shown in Figure 1, which Schmid denotes as *loop coarsening* [1], [2].

In this paper, we investigate two fundamental aspects of implementing local operators in hardware: image border handling and techniques efficient parallelization. Our contributions are summarized as follows:

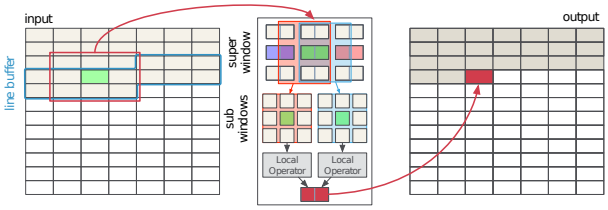


Figure 1. Schmid's loop coarsening with two pixels processed at once ($v = 2$) for a local operator of size 3×3.

- Two novel architectures for loop coarsening that use significantly less registers than previous work. Depending on the choice of operator parameters, either the first or the second architecture requires less resources.
- The analysis and design of hardware architectures that support image border handling with a guaranteed throughput of the input stream for the processing of local image operators in combination with loop coarsening.
- A Pareto-optimal selection algorithm for a given kernel size, coarsening factor and border handling mode.
- An analysis of the results obtained by Vivado HLS for both proposed variants and mentioned naive implementations.

II. Background and Scope

A. Notation

This section introduces the notation of important parameters used in this paper to increase clarity. Column and row image coordinates are represented by x and y. W/H and w/h refer to the width and height of an image and the width and height of a local operator, respectively. Based on that, $r_w = \lfloor w/2 \rfloor$ and $r_h = \lfloor h/2 \rfloor$ represent the integer radius of the operator. Parameter v represents the so-called coarsening factor and k_{in}, k_{out} are input and output bit widths of a local operator.

Moreover, we introduce two types of area cost functions: C_{reg} and C_{mux}. $C_{\text{reg}}^{\text{arch}}$ denotes the number of registers that are estimated to be used for *arch*. Similarly, $C_{\text{mux}}^{\text{arch}}$ indicates the number of multiplexers (MUXs). $C_{\text{reg}}^{\text{arch}}(\text{type})$ and $C_{\text{mux}}^{\text{arch}}(\text{type})$ represent the cost functions for a selected border handling *type*. Furthermore, we denote MUX$[n]$ to refer to a MUX with n inputs. As a speed measure , $T_{\text{CriticalPath}}^{\text{type}}$ denotes the time estimation of the critical path of an architecture that implements border handling of *type*.

B. Local Operators

Image processing functions that operate on neighboring pixels within a local window to calculate the output pixel are classified as local operators. Naturally, two adjacent local operators have

high spatial locality as many reads of their windows overlap. Therefore, raster order processing, which enables burst mode read from external memory, features high horizontal temporal locality for local operators. For instance, raster order processing of a $w \times h$ local operator uses $(w-1) \cdot h$ identical pixels from the previous cycle. A powerful caching design for FPGA implementations is deploying $(h-1)$ line buffers followed by a $w \times h$ sliding window to keep neighboring pixels in on-chip memory. Thus, a throughput (Th) of one cycle can be achieved at a cost of an initial latency (L_{initial}) for reading the neighboring pixels of the first window, as indicated by Eq. (1). Note that L and L_{calc} refer to the latencies of the local operator and its calculation data path, respectively.

$$\begin{aligned} Th^{\text{out}} &= Th^{\text{in}} = 1\,\text{pixel/cycle} \\ L &= L_{\text{initial}} + L_{\text{process}} \\ &= (\lfloor h/2 \rfloor \cdot W + \lfloor w/2 \rfloor + L_{\text{calc}}) + (W \cdot H) \end{aligned} \tag{1}$$

C. Image Border Handling

At the image borders, a local operator depends on the pixels from outside the image, the area defined by Eq. (2).

$$x < \lfloor w/2 \rfloor \ \vee \ x > W - \lceil w/2 \rceil \ \vee \ y < \lfloor h/2 \rfloor \ \vee \ y > H - \lceil h/2 \rceil \tag{2}$$

A solution is handling the data according to well-known border patterns as shown in Figure 3. Not handling borders appropriately creates artifacts, which becomes a serious problem in case of multiple consecutive local operators. In this paper, we only investigate solutions that do not stall the input, thus, sustain the throughput of the input. Consequently, solutions that pad the input image to circumvent border handling are out of the scope.

D. Schmid's Loop Coarsening

Data Level Parallelism (DLP) of local operators reading data in raster order can be increased by processing groups of data elements, so-called *data beats*, in each cycle. Schmid et al. [1] denote this technique as *loop coarsening*, since coarsening is applied to the outer loop of a stencil operation, reading data in raster order. Consequently, coarsening the loop of a local operator by a factor v transforms the processing speed equation given in Eq. (1) to Eq. (3).

$$\begin{aligned} Th^{\text{out}} &= Th^{\text{in}} = v\,\text{pixels/cycle} \\ L &= (\lfloor h/2 \rfloor \cdot \lceil W/v \rceil + \lfloor \lceil w/v \rceil / 2 \rfloor + L_{\text{calc}}) + (\lceil W/v \rceil \cdot H) \end{aligned} \tag{3}$$

A representation of Schmid's loop coarsening architecture is given in Figure 1. The line buffer and sliding window are modified to store data beats consisting of v pixels. The calculation is replicated by a factor of v followed by a unit for packing the results into a single output data beat.

III. Improved Loop Coarsening

In this paper, we propose two novel architectures for implementing loop coarsening with equal performance as in Eq. (3), but with less area usage. While the first one, Fetch and Calc (F&C) is an optimized version of Schmid's loop coarsening architecture [1], the other, Calc and Pack (C&P) uses a different schedule. In Section VI, it is shown that which of these is the more efficient architecture depends on the parameters w, h, k_{in}, k_{out}, and the border handling mode. Both architectures have in common that the sliding window is shifted in each cycle by v steps.

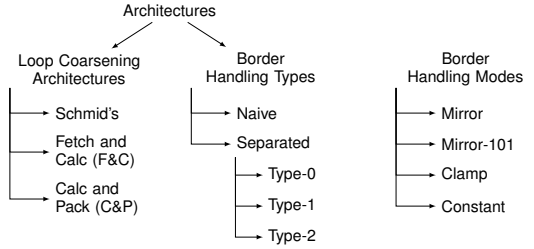
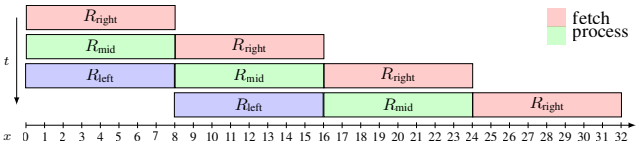


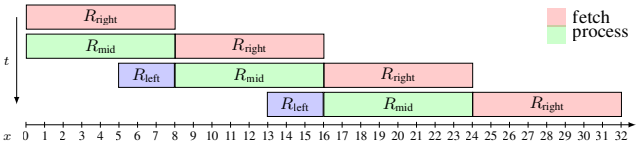
Figure 2. Loop coarsening architectures, border handling types, and border handling modes considered in this paper.

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

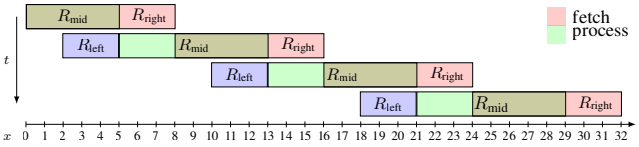
Figure 3. Common border handling modes.



(a) Schmid's: Processing starts at the second cycle. Each row holds $v + 2 \cdot (v \cdot \lceil r_w/v \rceil) = 24$ pixels starting with the third cycle.



(b) Fetch and Calc: Processing starts at the second cycle. Each row holds $r_w + v + (v \cdot \lceil r_w/v \rceil) = 19$ pixels starting with the third cycle.



(c) Calc and Pack: Processing starts at the first cycle. Writing starts at the second cycle. Writing the overlap of *fetch* and *process* is delayed by one cycle. Each row holds $v + 2 \cdot r_w = 11$ pixels starting with the second cycle.

Figure 4. First 4 cycles of the schedule for assigning pixels to register regions for the sliding window rows, whose width $w = 7$, radius $r_w = 3$ and coarsening factor $v = 8$. Red and green areas represent pixels fetched and processed in each cycle, respectively. All of them start writing v outputs at the second cycle.

For explanation purposes, we group the registers holding the pixels of a sliding window into regions R according to their temporal content as shown in Figure 4. R_{left} and R_{right} denote the registers that hold the neighboring pixels to process v pixels at R_{mid}. Similarly, R_{fetch} labels the registers that store the newest data beats. We also use colors in the figures for the labels R_{fetch} (red) and R_{mid} (green) since they can overlap.

A. Fetch and Calc

Schmid's loop coarsening reads v new pixels into register region R_{right} and processes v pixels at R_{mid} in each cycle. This is shown in Figure 4a. Red regions show the pixels read in each cycle, while green regions show the pixels processed and written in each cycle by the local operator. Yet, redundant

pixels, which are not subject to calculations in the current cycle, must be fetched to sliding window in Schmid's architecture as follows:

i $v > r_w$: causes R_{right} to store v pixels instead of only r_w.
ii $v < r_w$: $v - (v \bmod r_w)$ extra registers are necessary to fetch the next v pixels.

The number of necessary registers in Schmid's sliding window can be expressed by Eq. (4).

$$C_{\text{reg}}^{\text{Schmid's}} = k_{\text{in}} \cdot h \cdot v \cdot (2 \cdot \lceil r_w / v \rceil + 1) \tag{4}$$

While the unnecessary pixels in R_{right} are necessary for calculations in subsequent cycles, it is not necessary to store more than r_w pixels in R_{left}. Hence, as an improvement to [1], the number of pixels in R_{left} may be reduced to r_w. as shown in Figure 4b. Eq. (5) gives the number of pixels that need to be fetched in a row of a coarsened sliding window for any given v and r_w.

$$C_{\text{reg}}^{\text{F\&C}} = k_{\text{in}} \cdot h \cdot (r_w + v \cdot (\lceil r_w / v \rceil + 1)) + C_{\text{reg}}^{\text{F\&C}}(b) \tag{5}$$

B. Calc and Pack

C&P further reduces the fetching cost by using an alternative schedule. In this schedule, the sliding window holds the ideal number of pixels according to Eq. (6), so that the newest input *data beat* is not only stored in R_{right} but also in R_{mid}.

$$|R_{\text{left}}| = |R_{\text{right}}| = r_w, \quad |R_{\text{mid}}| = v \tag{6}$$

Naturally, the sliding window is shifted by v in each cycle. This splits the processing order at R_{mid} into two cycles. Consequently, an output *data beat* is partially calculated in two cycles and written in the second cycle using the memory packing scheme in Eq. (7).

$$\begin{aligned}\text{output}([x, x+v], t) = \text{pack}\{&\text{out}([0, v - r_w - 1], t - 1),\\ &\text{out}([v - r_w, v - 1], t)\}\end{aligned} \tag{7}$$

The described schedule can be implemented with additional registers that are placed after the leftmost $v - r_w$ local operators in order to the hold results from the previous cycle. For instance, the described schedule for a kernel with $w = 7$ and $v = 8$ is given in Figure 4c. As it can be seen, the number of pixels stored in a row remains ideal $w - 1 + v = 14$, but $v - r_w = 5$ additional registers are necessary in order to delay the results of the previous cycle. Eqs. (8) and (9) give the costs necessary per row of a coarsened sliding window for any given v and r_w.

$$C_{\text{reg}}^{\text{C\&P}}(d) = k_{\text{out}} \cdot (v - (r_w \bmod v)) \tag{8}$$

$$C_{\text{reg}}^{\text{C\&P}} = k_{\text{in}} \cdot h \cdot (2 \cdot r_w + v) + C_{\text{reg}}^{\text{C\&P}}(d) + C_{\text{reg}}^{\text{C\&P}}(b) \tag{9}$$

IV. Analysis of Border Handling

In this section, we analyze hardware implementation properties of border handling modes shown in Figure 3 to provide an analytical basis for the hardware architectures we propose in Section IV.

A. Naïve Border Handling

A straight forward approach for border handling, which is also common in software implementations, is to have a conditional read on the sliding window. Considering W, H, w, h to be constant,and the read coordinates for the sliding window depend on the combination of input indices x, y, i, j. The indices i and j are addressing window elements in the range $[0, w-1]$ and $[0, h-1]$, respectively. Depending on the current position of the operator within the image, we have to consider w different border cases for x, e. g., $[0, 1, \text{else}, W-2, W-1]$ for $w = 5$. Similarly, h different border cases for y need to be considered. In total, this results in $w^2 \cdot h^2$ different input index combinations for (i, x) and (j, y).

Assuming I_s is the union of all input index combinations, then the cost of the data selection hardware would be a function of $|I_s|$. Luckily, set I_s can be separated into two sets $I_w \times I_h$. Here, I_w depends only on x and i indices, resulting in $|I_w| = w^2$ index combinations and $|I_h| = h^2$ analogously. Depending on the image size, not all operator positions are relevant to every window element, e. g., the center element is never subject to any border case, as it is always fetched from a valid image region. Consequently, for an image size of at least $\lceil w/2 \rceil \times \lceil h/2 \rceil$, the number of input index combinations can be reduced to a maximum of $|I_w| = 1 + 2 \cdot \sum_{i=2}^{\lceil w/2 \rceil} i$. According to Eq. (10), $|I_s|$ can be reduced to only 121 instead of 625 output indices for a 5×5 local operator.

$$\begin{aligned}&|I_s| = |I_w| \cdot |I_h|\\ &|I_w| = C(w),\ |I_h| = C(h)\end{aligned} \qquad C(n) = 1 + 2 \cdot \sum_{i=2}^{\lceil n/2 \rceil} i \tag{10}$$

B. Separated Border Handling

The following properties imply the spatial and temporal features that can further improve the so called naive border handling architecture:

1) None of the considered border handling modes needs $w^2 \cdot h^2$ MUXs. One reason is that the number of cases that
a pixel from outside of an image is requested decreases towards the center of a local operator.
2) Assume that I_h and I_w consist of all the y and x index combinations, then Eq. (11) always satisfies.

$$|I_s| = |I_w \times I_h| = |I_w| \cdot |I_h| \tag{11}$$

3) Every pixel read to the first register of a sliding window row is reused in the following $w - 1$ cycles, shown in Eq. (12).

$$\begin{aligned}\text{in}[x, y] = \{\text{wind}[i, j], t\} &= \{\text{wind}[i-1, j], t+1\}\\ &= \ldots = \{\text{wind}[i - w + 1, j], t + w - 1\}\end{aligned} \tag{12}$$

The first feature implies that the conditional selection through x and y axises are orthogonal to each other. This means that the row selection, in y direction, can be separated from the column selection, in x, which is the same for all columns of a local operator. Furthermore, the row selection can be implemented only once before any pixel is read to the sliding window, which reduces the border handling cost from $|I_s|$ to $|I_h| + h \cdot |I_w|$. By separating the row and column selection from each other, $|I_s|$ reduces from $11 \cdot 11 = 121$ to $6 \cdot 11 = 66$. The top level architecture that separates row selection and column selection is given in Figure 5.

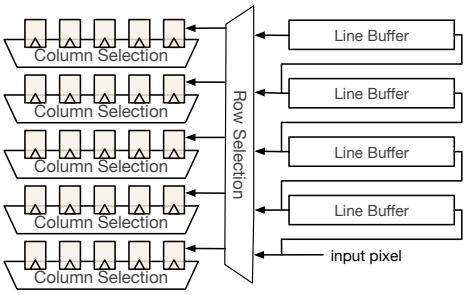


Figure 5. Separated border handling architecture: Row and column selection consist of MUXs for implementing border handling index combination sets I_w and I_h, respectively. One row selection is placed before the sliding window, contrary to the naïve type, deploying $w = 5$ of them after each column.

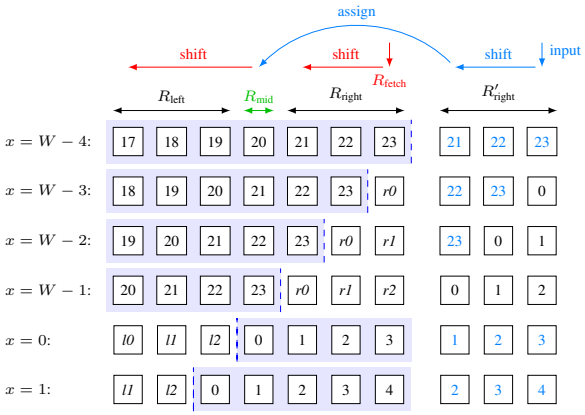


Figure 6. Temporal data flow for row data selection Type-1 with a local operator of size $w = 7$. Blue background denotes valid image regions, while *lX* and *rX* represent variable values depending on the corresponding border handling mode.

1) Row Selection: The row selection circuit reads from line buffers and writes to the sliding window without increasing the critical path of the local operator. Moreover, no locality can be exploited.

2) Column Selection: Whereas the column selection can use the same circuits with row selection, its output is directly connected with the local operators' data path, needing extra registers in between for pipelining the critical path. Moreover, temporal locality can be exploited. Therefore, we conduct a formal analysis on finding the minimum number of registers and MUXs under certain assumptions.

A timed data flow for column selection is given in Figure 6 R_{mid} represents the center pixel of the local operator. Investigating the steps from $x = W - 4$ to processing the next image row at $x = 1$ reveals all the corner cases for border handling. Investigating all border handling modes reveals that they can be implemented with similar design patterns. It can be observed that the window does not need to fetch a new pixel in the interval $x = [W - 3, W - 1]$. However, multiple pixels for the next image row must be read at once when $x = 0$. Assuming that the streaming is not stalled and one pixel is read in each cycle, at least r_w pixels per row[1] must be fetched at $x = 0$ in order be able to initialize all column pixels. Eq. (13) defines the minimum number of registers in a sliding window in terms of bits.

[1]Note that additional registers utilized for border handling are represented by R'_{right}.

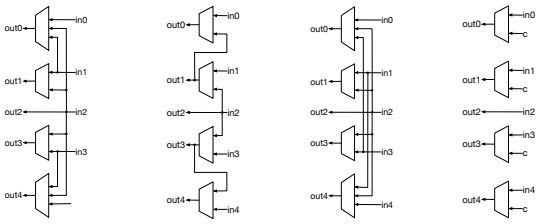


Figure 7. Border handling row data selection

$$C^{\min}_{\text{reg}} = h \cdot k_{\text{in}} \cdot (w + C^{\min}_{\text{reg}}(b)),\ C^{\min}_{\text{reg}}(b) = r_w \tag{13}$$

Under this assumption, a column selection architecture using the minimum number of registers has the following features:

i Except at $x = 0$, border handling can be achieved only through data selection that appropriately feeds R_{fetch} and shifts the content stored in R_{right}, R_{mid} and R_{left}.
ii All registers, except R_{fetch}, should be able to read from R'_{right} in order to initialize all column pixels at $x = 0$.
iii R'_{right} fetches one pixel in every cycle, but only at $x = 0$, reads from R'_{right} become indispensable. This fact renders the blue highlighted area in Figure 6 to be needless.

Under the assumption that a selection can only be implemented using a MUX, the minimum column selection for border handling uses at least the following resources:

i There must be at least one MUX[2] before any register in R_{right} and R_{left} since they must be able to read from R'_{right} at $x = 0$.
ii R_{mid} can be implemented to read only from the leftmost register in R'_{right}. If the blue area is turned off, there must be at least one MUX[2] to read from R_{right}.
iii The size of the MUX before R_{fetch} depends on the border handling mode.
iv Only the data selection before R_{fetch} and blue portion of R'_{right} can be optimized. Resource usage is identical for all border handling modes.

In conclusion, for further optimization, the blue highlighted portion of R'_{right} can be shut down for the interval $x = [0, W - r_w)$ by modifying the selection before R_{fetch} and R_{mid}. Eq. (14) shows the minimum number of MUXs for the discussed border handling modes.

$$C^{\min}_{\text{mux}} = ((2 \cdot r_w - 1)\text{MUX}[2] + \text{MUX}[2]) \cdot h \cdot k_{\text{in}} + C^{\text{new}}_{\text{mux}} \tag{14}$$

V. Hardware Architectures for Image Border Handling

In this section, we propose hardware implementations of the row and column selection circuits, provide their cost functions, and investigate their effects on the circuit speed. In addition, we extend our proposed implementations to loop coarsening.

A. Row Selection

Row selection circuits for the considered border handling modes are shown in Figure 7. *Clamp* and *mirror* may be implemented by the same architecture with different control paths. For *clamp*, I_w can be reduced even more by wiring out(y) to out$(y + 1)$ and out$(y - 1)$ at the top and lower

border, respectively. A corresponding row selection circuit, called *clamp2*, is shown in Figure 7b. Despite the simplification in selection size for *clamp2*, its longer critical path makes it barely appealing for fast implementations. Cost functions for all architectures are given in Eqs. (15) and (16).

$$C_{\text{mux}}^{\text{RowSelect}} = 2 \cdot \begin{cases} \sum_{i=2}^{r_h+1} \text{MUX}[i], & \text{mirror-101, mirror, clamp} \\ r_h \cdot \text{MUX}[2], & \text{clamp2} \\ \text{MUX}[2], & \text{constant} \end{cases} \tag{15}$$

$$T_{\text{CriticalPath}}^{\text{RowSelect}} = \begin{cases} T(\text{MUX}[r_h+1]), & \text{mirror-101, mirror, clamp} \\ T(r_h \cdot \text{MUX}[2]), & \text{clamp2} \\ T(\text{MUX}[2]), & \text{constant} \end{cases} \tag{16}$$

B. Column Selection

In this section, we present three types to design the column selection hardware. Type-0 is explained for comparison reasons only, since it is a commonly known approach. On the other hand, Type-1 and Type-2 follow the analysis results discussed in Section II-C. Register requirements for both types are equal to the amount claimed to be minimum in Section II-C.

$$C_{\text{reg}}^{\text{Type-2}}(b) = C_{\text{reg}}^{\text{Type-1}}(b) = C_{\text{reg}}^{\text{min}}(b) \tag{17}$$

Similarly, their design approach assumes that there is one MUX[2] before every register in R_{right} and R_{left}. The optimization focuses on the selection circuit before R_{fetch} and R_{mid} by restructuring R'_{right}.

1) Type-0: Column selection can be implemented by transposing row selection architectures discussed in Section V-A. Figure 8a shows the mirror border handling. Whereas the cost of this common approach, shown in Eq. (18), is twice the registers of Eq. (17).

$$C_{\text{reg}}^{\text{Type-0}}(b) = h \cdot k_{\text{in}} \cdot (2 \cdot r_w) \tag{18}$$

Similarly, the selection is also larger since there is more than one MUX[2] before R_{right} and R_{left}. The reason is that this type does not exploit temporal locality in x direction. The MUX cost for the selection is the same as the cost in Eq. (15), where r_h is replaced by r_w. On the contrary, the critical path for clamp2, as defined by Eq. (19), is shorter since the selection and temporal direction is the same.

$$T_{\text{CriticalPath}}^{\text{Type-0}} = \begin{cases} T(\text{MUX}[r_w+1]), & \text{mirror-101, mirror, clamp} \\ T(\text{MUX}[2]), & \text{clamp2, constant} \end{cases} \tag{19}$$

2) Type-1: Temporal data flow for Type-1 for a $h \times 11$ kernel is already given in Figure 6. Registers in the blue highlighted region in R'_{right} read from the previous one, thus, no MUX is needed for any register in R'_{right}. Moreover, the MUX before R_{mid} can be eliminated if R'_{right} is not switched off. This MUX is highlighted in blue in Eq. (20). By analyzing the registers for temporal data flow, it can be seen that there are types for any arbitrary size. Example architectures for all border handling modes are given in Figure 8b. The critical path and the selection cost in terms of MUXs are given in Eqs. (20) and (21).

$$C_{\text{mux}}^{\text{Type-1}} = (\text{MUX}[2] + \begin{cases} \text{MUX}[r_w+1], & \text{mirror-101, mirror} \\ \text{MUX}[2], & \text{clamp, constant} \end{cases} + (2 \cdot r_w - 1)\text{MUX}[2]) \cdot h \cdot k_{\text{in}} \tag{20}$$

$$T_{\text{CriticalPath}}^{\text{Type-1}} = \begin{cases} T(\text{MUX}[r_w+1]), & \text{mirror-101, mirror} \\ T(\text{MUX}[2]), & \text{clamp, constant} \end{cases} \tag{21}$$

3) Type-2: The size of the MUX before R_{fetch} depending on r_w, as Eq. (20) indicates, could drastically decrease the speed of the entire circuit for large windows. Section IV suggests that if a more optimized architecture exists, it can be found by rescheduling the blue highlighted region of R'_{right} to minimize $C_{\text{mux}}^{\text{new}}$ in Eq. (14). Based on that, we propose an alternative schedule in Figure 9. Here, the blue highlighted region is restructured in a way that the leftmost register in R'_{right} can always output the proper input for R_{fetch}. In this way, the selection before R_{fetch} can always be implemented with only a MUX[2] at a cost of additional MUX[2]s between the registers in R'_{right}. An example architecture for this type is given in Figure 8c. The critical path and the selection cost in terms of MUXs are equal for all border handling modes and are given in Eqs. (22) and (23). Note that R'_{right}, thus the selection in between, can be switched off in the interval $x = [0, W - r_w)$.

$$T_{\text{CriticalPath}}^{\text{Type-2}} = T(\text{MUX}[2]) \tag{22}$$

$$C_{\text{mux}}^{\text{Type-2}} = ((3 \cdot r_w + 1)\text{MUX}[2]) \cdot h \cdot k_{\text{in}} \tag{23}$$

C. Loop Coarsening

Our analysis on border handling remains valid for both loop coarsening architectures. Thereby, separated border handling types can be used as an efficient top level architecture. However, F&C and C&P require different column selection patterns.

1) Fetch and Calc: A local operator can be considered as a F&C architecture with $v = 1$. Correspondingly, column selection architectures of Type-0, Type-1 and Type-2, can be used for F&C with slight modifications. Section II-C explains that the time interval between a local operator entering and leaving the border region is r_w cycles in raster order processing. As the local operator moves faster in the horizontal direction by the increase in v, this interval shrinks to r_{wv} as in Eq. (24) for the left and right borders.

$$r_{wv} = \lceil r_w / v \rceil \tag{24}$$

Consequently, column selection of a loop coarsening architecture with the parameters of w, v, consists of $\min(r_w, v)$ parallel column selection architectures, whose $r_w = r_{wv}$. Examples to two corner cases, which are $r_w > v$ and $r_w < v$, for Type-1 border handling in loop coarsening are shown in Figures 10 and 11. As a result, the increase in v reduces C_{mux} of border handling whereas $C_{\text{reg}}(b)$ remains the same as Eqs. (25) and (26) indicate.

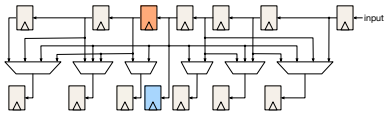
$$C_{\text{mux}}^{\text{F\&C}}(b) = \min(r_w, v) \cdot C_{\text{mux}}^{\text{Type}}(b, r_{wv}) \tag{25}$$

$$C_{\text{reg}}^{\text{F\&C}}(b) = C_{\text{reg}}^{\text{Type}}(b, r_w) \tag{26}$$

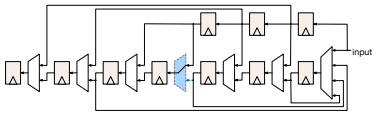
2) Calc and Pack: The regions R'_{right} and R'_{left} in Figure 12 illustrate the difference to F&C. As it can be seen, the first pixel of an image row is not processed in the first pixel of R_{mid}. Therefore, R_{left} cannot be initialized at $x = 0$ and additional registers R'_{left} should be used for border handling instead. An example architecture is given in Figure 11. However, since it consists of an extra register for each register at R_{left} and R_{right} any selection in border handling can be implemented via a MUX[2]. Hence, C_{mux} and C_{reg} of the border handling can be defined by Eqs. (27) and (28).

$$C_{\text{reg}}^{\text{C\&P}}(b) = k_{\text{in}} \cdot h \cdot (2 \cdot r_w) \tag{27}$$

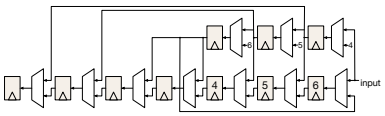
$$C_{\text{mux}}^{\text{C\&P}}(b) = \min(r_w, v) \cdot C_{\text{mux}}^{\text{Type-0}}(b, r_{wv}) \tag{28}$$



(a) Type-0: Transpose of row selection following with pipelining registers.



(b) Type-1: Minimal number of registers are used and the selection at the upper right register region ($R_{\text{right'}}$) is optimized according to Section II-C.



(c) Type-2: Minimal number of registers are used and the selection before the input register (R_{fetch}) is optimized according to Section II-C.

Figure 8. Column data selections for *mirror* with $w = 7$. Vivado HLS removes the blue register in Figure 8a since it stores the same data as the orange one.

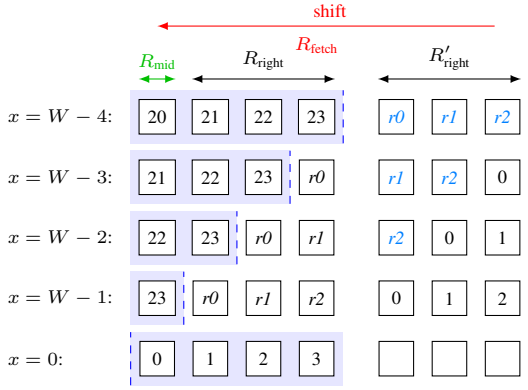


Figure 9. Temporal data flow that minimizes the selection before R_{fetch} (Type-2 row data selection). R_{left} was omitted for demonstration purposes only.

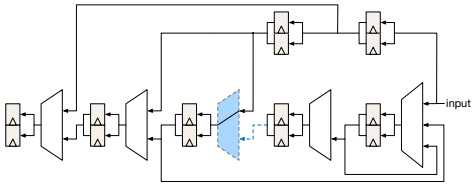


Figure 10. F&C Type-1 mirror border handling for $w = 9$ and $v = 2$, which basically is $\min(r_w, v) = 2$ parallel Type-1 column selection for $w = 5$.

VI. Selecting the Best Architecture

A hardware architecture is considered as Pareto-optimal only if it requires less resources and a lower $T_{\text{CriticalPath}}$ than other implementations. In this section, we analyze the analytical models of the considered architectures and provide Algorithm 1.

A. Border Handling Pattern Selection

The difference between C&P and F&Cin terms of $C_{\text{reg}}(b)$ cost, can be found through Eq. (29) by merging Eqs. (13), (17) and (27).

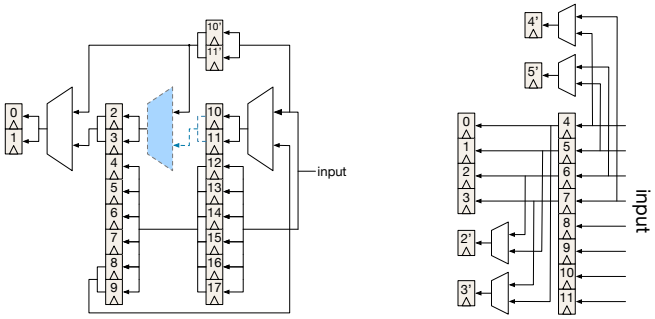
$$C_{\text{reg}}^{\text{C\&P - F\&C}}(b) = k_{\text{in}} \cdot h \cdot \begin{cases} 0, & \text{undefined, Type-0} \\ r_w, & \text{Type-1, Type-2} \end{cases} \tag{29}$$

Combining Eqs. (15), (20), (23), (26) and (28) reveals that the selection circuits of all border handling types except Type-2 converges to the same implementation for $v \geq r_w$, as Eq. (30) indicates.

$$\begin{aligned} &C_{\text{mux}}^{\text{Type-0- Type-1}} \leq 0 \quad C_{\text{mux}}^{\text{Type-0- Type-2}} \leq 0, \text{ when } r_{wv} > 1 \\ &C_{\text{mux}}^{\text{Type-2}}(b) > C_{\text{mux}}^{\text{Type-0}}(b) = C_{\text{mux}}^{\text{Type-1}}(b), \text{ when } r_{wv} = 1 \end{aligned} \tag{30}$$

Merging Eq. (28) with Eq. (22) shows that Type-2 is the fastest.

$$T_{\text{CriticalPath}}^{\text{Type-0}}(b, r_w) \geq T_{\text{CriticalPath}}^{\text{Type-1}}(b, r_w) \geq T_{\text{CriticalPath}}^{\text{Type-2}}(b, r_w) = \text{MUX}[2] \tag{31}$$



(a) F&C Type-1, which basically is $\min(r_w, v) = 2$ parallel Type-1 column selection for $w = 3$

(b) C&P

Figure 11. Mirror border handling for $w = 5$, $v = 8$.

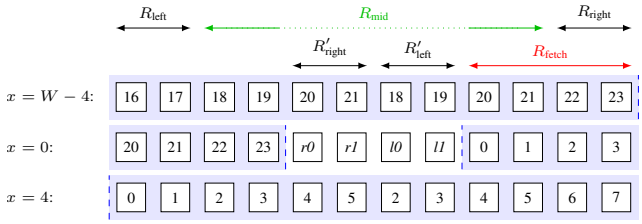


Figure 12. Temporal data flow for $w = 5$ with coarsening $v = 4$.

Considering Eq. (29) with Eq. (33), derives the conclusion that Type-1 is Pareto-optimal to Type-0. On the other hand, Type-1 and Type-2 are two different design points. Type-1 is more efficient than Type-2 for the clamp and constant.

$$C_{\text{mux}}^{\text{Type-2- Type-1}}(b) = (r_w+1)\text{MUX}[2] - \text{MUX}[r_{vw}+1] - \text{MUX}[2] < 0 \tag{32}$$

As it can be seen, Eq. (32) depends on technology mapping. Finally, merging Eqs. (30), (32) and (33) shows that border handling for F&C can be implemented not only with less resources but also faster than C&P, as Eq. (33) indicates.

$$C^{\text{C\&P - F\&C}}(b) \geq 0, \qquad T_{\text{CriticalPath}}^{\text{C\&P - F\&C}}(b) \geq 0, \tag{33}$$

B. Loop Coarsening Architecture Selection

C&P is more efficient than F&C when Eq. (34) satisfies.

$$0 < C^{\text{F\&C - C\&P}} \tag{34}$$

Eq. (34) turns to Eq. (35) by substituting Eqs. (5) and (9).

$$0 < (v - (r_w \bmod v)) \cdot (k_{\text{in}} \cdot h - k_{\text{out}}) + C^{\text{F\&C - C\&P}}(b) \tag{35}$$

1) Undefined: Substituting Eq. (35) into Eq. (29) shows the region that C&P is more efficient.

$$C^{\text{F\&C - C\&P}} \geq 0, \qquad \text{where } k_{\text{out}} \leq k_{\text{in}} \cdot h \tag{36}$$

2) Mirror-101, Mirror, Clamp, Constant:

a) $v > r_w$: Combining Eqs. (29), (30) and (35), C&P is more efficient when Eq. (37) satisfies.

$$r_w \cdot (k_{\text{in}} \cdot h - k_{\text{out}} + 1) < v \cdot (k_{\text{in}} \cdot h - k_{\text{out}}) \tag{37}$$

b) $v \leq r_w$: Merging Eqs. (29), (33) and (35), shows that C&P is always worse since the left hand side of the comparison in Eq. (38) is always positive, whereas the right hand side is negative.

$$(k_{\text{in}} \cdot h) \cdot (r_w - v + (r_w \bmod v)) < k_{\text{out}} \cdot ((r_w \bmod v) - v) \tag{38}$$

C. Pareto-optimal Architecture Selection

Algorithm 1 summarizes the comparisons given in Section VI and the column selection architectures discussed in Section II-C.

Algorithm 1: Picking the best architecture.

```
input : w, h, borderMode, v, k_out, k_in, designGoal
output: BorderHandlingPattern CoarseningArch
func selectParetoOptimal(BorderHandlingPattern, CoarseningArch
                         w, h, borderMode, v, k_out, k_in, designGoal)
    r_w = ⌊w/2⌋
    if borderMode = UNDEFINED then
        if k_out < k_in · h then
            CoarseningArch ← Calc and Pack
        else
            CoarseningArch ← Fetch and Calc
        end
        BorderHandlingPattern ← none
    else
        if r_w · (k_in · h − k_out + 1) < v · (k_in · h − k_out) then
            CoarseningArch ← Calc and Pack
        else
            CoarseningArch ← Fetch and Calc
        end
        if borderMode = (CLAMP ∨ CONSTANT) then
            BorderHandlingPattern ← Type-1
        else
            // borderMode = (MIRROR ∨ MIRROR-101)
            if (designGoal = speed) ∨ (Eq. (32) = true) then
                BorderHandlingPattern ← Type-2
            else
                BorderHandlingPattern ← Type-1
            end
        end
    end
end
```

VII. Evaluation and Results

In this section, we evaluate the FPGA implementation results of the considered loop coarsening and border handling architectures. Due to paper space limitations, we narrowed our results to individual evaluations of the proposed coarsening and border handling architectures in Sections VII-A and VII-B respectively, and then in Section VII-C, investigation of additional pipelining overhead that a higher target logic speed might introduce to these. We used Vivado HLS instead of a traditional Register Transfer Level (RTL) design in order to understand whether the proposed architectures contribute from a higher level description. In order to examine the analytical models of the proposed architectures, we investigate the synthesis of a single kernel algorithm, the mean filter, and subtract the overhead of the data path from the total cost using estimation results. Confirming analytical models in this way facilitates the confidence that we avoid reporting any overhead caused by the HLS tool's heuristics. Finally, we compare the FPGA implementation results with the estimation results to examine their validity in Section VII-D.

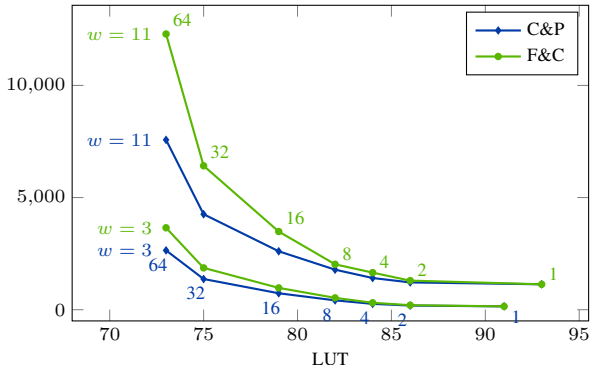


Figure 13. HLS estimation results of the proposed coarsening architectures for different kernel sizes and 5.0 ns target clock period. (no border handling)

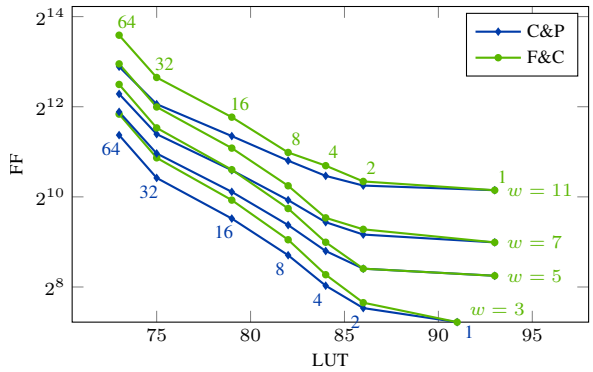


Figure 14. HLS estimation results of the proposed coarsening architectures for different kernel sizes and 5.0 ns target clock period. (no border handling)

A. Coarsening Types

In this section, we compare the coarsening architectures that are proposed in Section III. As already discussed, F&C is a more optimized version of Schmid's loop coarsening that eliminates the redundant registers. Therefore, we compared F&C with C&P in Figures 13 and 14. Moreover, some of these results are given in Table I. Algorithm 1 suggests using C&P in case of *undefined* border handling mode for any v and r_x. Moreover, it can be derived from Eq. (36) that C&P should use less registers than F&C in this case according to $((v-(r_w \bmod v)) \bmod v) \cdot (k_{\text{in}} \cdot h - k_{\text{out}})$. F&C uses 960 more registers (48 %) than C&P for $w = 5$ and $v = 32$, as given in Table I. The improvement reaches up to 2160 registers (50 %) for $w = 11$ and $v = 32$, as shown in Figure 13. Not only this validates our coarsening formulas, but also reveals that thousands of registers can be saved with an appropriate architecture even for an application consisting of a simple single kernel.

B. Border Handling Architectures

Figure 16 shows that separated border handling significantly improves the naive approach. This indicates that neither Vivado HLS and synthesis tools are able to conduct the analysis in Section II-C. As Eq. (30) indicates, all the discussed separated border handling architectures except Type-2 converge to the same design point for $v \geq r_w$ as shown in Figure 11. On the other hand, Figure 16 clearly shows that Type-1 uses considerably less logic then Type-0 for $v < r_w$. Investigating the generated Hardware Descriptive Language (HDL) codes reveals that Vivado HLS optimizes the Type-0 column selection by eliminating the blue colored registers in Figure 8a. This can

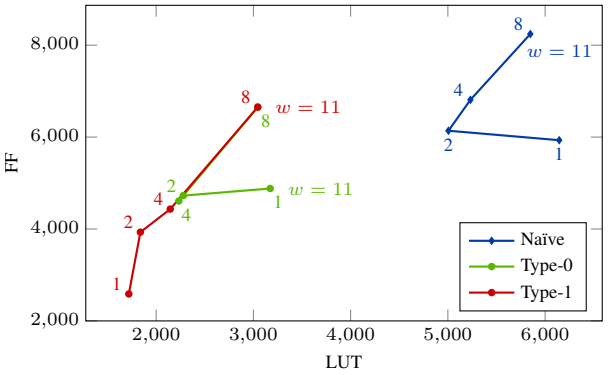


Figure 15. HLS estimation results of the proposed mirror border handling architectures for a 11×11 kernel with F&C coarsening.

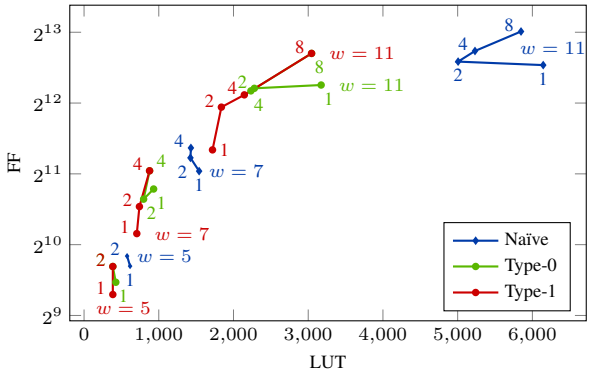


Figure 16. HLS estimation results of the proposed mirror border handling architectures for different kernel sizes F&C coarsening.

be achieved with a simple analysis that detects the registers set via the same wire. Considering Eqs. (17) and (18) with this optimization Type-1 uses $h \cdot k_{in}$ $(r_w - 1)$ less registers than Type-0 for the sliding window as expected. Speed optimization introduced via Type-2 highly depends on technological mapping, since MUX[2] is mapped via LUTs in FPGA. Table I shows that Eq. (32) doesn't satisfy in our evaluation environment, therefore Type-1 uses less LUTs than Type-2. Yet Type-2 uses less registers, which makes it a different Pareto design point.

C. Effects of Target Speed Constraint

Vivado HLS generates different HDL codes for different speed targets. We evaluated the pipelining overheads of the considered architectures using the target speeds of 50 MHz and 300 MHz for a ZYNQ-zc706 FPGA. Estimation results revealed that the number of registers and MUXs match our equations for the low speed target. This can be seen by Comparing resource utilization of different coarsening factors in Table I Note that, we conducted a more detailed analysis on MUXs usage from the expressions that Vivado reports for any data selection. Likewise, comparing the estimation results of architectures that have the same loop coarsening parameters but different target speeds shows that the improvement gets more significant for higher speed constraints.

D. Implementation Results

Reporting implementation results is good practice to eliminate estimation errors. On the other hand, using estimation reports make it possible to subtract the calculation overhead of a local operator and distinguish the selection cost of border handling among other MUXs. Thankfully, we observed that the estimation results, from Vivado HLS v2016.3, are reliable enough for our implementations other than Type-2. As a measure of this reliability, Table I consists of implementation results for the provided estimation results. It can be seen that the estimation differences between coarsening and border handling architectures reflect the implementation results with only very minor deviations. On the other hand, we investigated the results of Type-2 only through its implementation results in Section VII-B.

VIII. Related Work

A popular approach for increasing data level parallelism is imitating Graphics Processing Unit (GPU) programming model by deploying accelerators in parallel from Open Computing Language (OpenCL) kernels as in [3], [4], [5]. [6] proposes parallelization through vectorization of outer loops for programs without loop-carried data dependencies. The *polyhedron model* [7] is often the method of choice for automatic parallelization, in case a loop program contains loop-carried data dependencies or is not defined over a rectangular iteration space. Recently, researchers considered the polyhedral model also in combination with C-based HLS. Alias et al. presented in [8] an approach how to optimize off-chip memory traffic and on-chip data reuse in case of FPGA accelerators, and integrated their concept into the Altera C-to-Hardware (C2H) toolchain. Cong et al. [9], [10] studied the impact of several polyhedral transformations in order to improve data reuse and throughput in HLS, and prototypically implemented the transformations in PolyOpt/HLS. All of the aforementioned polyhedron-model-based HLS approaches perform aggressive pipelining and loop tiling, but none of them reaches to the efficiency level of so called loop coarsening proposed in [1], [2].

Image border handling is a fundamental subject in image processing. Yet, only few work consider the hardware implementation in detail [11], [12].

IX. Conclusion

Being inspired by Schmid's [1] RTL considerations on HLS, in this paper, we proposed two new loop coarsening architectures. While the first one always uses less resources than [1], a Pareto-optimal architecture depends on the local operator's parameters. Moreover, as a novel contribution, we extended the problem of image border handling to loop coarsening.

We conducted a systematic analysis for the minimal resource utilization on image border handling and proposed novel architectures based on that analysis. Previous work on image border handling [11], [12] are slightly less efficient than our Type-1, but in none of them switching off additional registers is considered, although utilizing an additional MUX for it. Moreover, we proposed faster architectures for border handling modes mirror and mirror-101, and discussed how other Pareto-optimal architectures can be designed based on the analysis. Aside from that, their architectures do not consider loop coarsening at all.

Finally, we analyzed the area usage of every architecture discussed in this paper, formulated it in equations, compared those equations, and eventually provided an algorithm that selects the best coarsening and border handling architectures for given parameters of a local operator. As a side contribution, we investigated the Vivado HLS implementations of our architectures as well as the naïve approach. Thus, we were able to show

Table I
HLS ESTIMATION RESULTS FOR A LOCAL OPERATOR AND IMPLEMENTATION RESULTS FOR A MEAN FILTER.

Parameters			Estimation (CPtar = 20 ns)				Estimation (CPtar = 3.1 ns)				Implementation (CPtar = 3.1 ns)						
v	w/h	Coars.	BRAM	FF	LUT	CPes	BRAM	FF	LUT	CPes	SLICE	BRAM	FF	LUT	DSP	CPpsyn	CPimp
1	5	C&P	4	304	93	13.50	4	378	93	3.10	152	4	600	270	29	2.48	2.55
1	5	F&C	4	304	93	13.50	4	378	93	3.10	139	4	600	269	29	2.48	2.61
2	5	C&P	4	339	86	14.43	4	446	88	3.09	215	4	873	448	14	2.52	2.63
2	5	F&C	4	339	86	14.43	4	446	88	3.09	222	4	873	449	14	2.52	2.46
8	5	C&P	8	663	82	14.43	8	954	84	3.06	675	8	2589	1565	6	2.39	2.56
8	5	F&C	8	855	82	14.43	8	1146	84	3.06	603	8	2781	1566	6	2.39	2.74
32	5	C&P	32	1995	75	14.43	32	3045	77	3.03	1951	32	9256	5367	6	2.38	2.83
32	5	F&C	32	2955	75	14.43	32	4005	77	3.03	2023	32	10216	5367	6	2.38	3.09

(a) Coarsening Architectures

Parameters			Estimation (CPtar = 20 ns)				Estimation (CPtar = 3.1 ns)				Implementation (CPtar = 3.1 ns)					
v	w/h	BH Patt.	BRAM	FF	LUT	CPes	BRAM	FF	LUT	CPes	SLICE	BRAM	FF	LUT	CPpsyn	CPimp
1	5	Naïve	4	471	575	13.5	4	643	576	3.10	213	4	879	533	2.63	2.67
1	5	Type-0	4	438	381	13.5	4	522	385	3.10	192	4	772	422	2.53	2.65
1	5	Type-1	4	398	341	13.5	4	482	345	3.10	176	4	732	419	2.52	2.58
1	5	Type-1	4	403	459	13.5	4	487	468	3.10	179	4	742	475	2.52	2.53
2	5	Naïve	4	495	538	14.4	4	664	542	3.09	288	4	1182	741	2.46	2.65
2	5	Type-0	4	432	344	14.4	4	565	349	3.09	268	4	1053	625	2.52	2.65
2	5	Type-1	4	432	344	14.4	4	565	349	3.09	244	4	1053	626	2.52	2.60
2	5	Type-1	4	435	501	14.4	4	569	511	3.09	262	4	1061	700	2.53	2.80
1	7	Naïve	6	855	1443	15.0	6	1434	1446	3.10	482	6	1976	1174	2.55	2.74
1	7	Type-0	6	806	867	15.0	6	1332	871	3.10	415	6	1843	878	2.56	2.85
1	7	Type-1	6	693	642	15.0	6	1107	646	3.10	381	6	1620	741	2.56	2.83
1	7	Type-1	6	698	808	15.0	6	885	817	3.10	316	6	1409	851	2.52	2.83
2	7	Naïve	6	957	1308	15.9	6	1589	1339	3.09	653	6	2715	1595	2.53	2.70
2	7	Type-0	6	862	730	15.9	6	1496	735	3.09	595	6	2570	1219	2.52	2.95
2	7	Type-1	6	806	674	15.9	6	1384	679	3.09	554	6	2459	1189	2.52	2.76
2	11	Type-1	6	923	1287	17.2	6	1107	1298	16.1	514	6	2194	1405	2.53	2.94
1	11	Naïve	10	2033	5390	16.6	10	4324	5393	3.10	1352	10	5601	3852	2.55	2.87
1	11	Type-0	10	1952	2997	16.6	10	3549	3018	3.10	1028	10	4781	2627	2.55	2.95
1	11	Type-1	10	1597	1583	16.6	10	1892	1594	3.45	711	10	3104	1883	2.55	2.67
1	11	Type-1	10	1601	1845	16.6	10	1891	1862	3.10	685	10	3114	2062	2.52	2.90
2	11	Naïve	10	2184	4562	16.7	10	4597	4566	3.09	1607	10	6839	4366	2.56	2.91
2	11	Type-0	10	2025	2160	16.7	10	3372	2228	3.09	1286	10	5663	3183	2.53	2.82
2	11	Type-1	10	1760	1719	16.7	10	2843	1787	3.09	1204	10	5136	2956	2.56	2.95
2	11	Type-1	10	1989	3564	25.3	10	2230	3639	25.3	1221	10	4537	3724	2.53	2.85

(b) Border Handling Architectures

that focusing on the underlying architecture can significantly improve HLS' performance.

REFERENCES

[1] M. Schmid, O. Reiche, F. Hannig, and J. Teich, "Loop coarsening in C-based high-level synthesis", in *Proc. of the 26th IEEE Intl. Conf. on Application-specific Systems, Architectures and Processors (ASAP)*, (Toronto, Canada), IEEE, Jul. 27–29, 2015, pp. 166–173.

[2] 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.

[3] A. Papakonstantinou, K. Gururaj, J. Stratton, D. Chen, J. Cong, and W.-M. Hwu, "Fcuda: Enabling efficient compilation of CUDA kernels onto FPGAs", in *Proc. of the IEEE 7th Symposium on Application Specific Processors (SASP)*, (San Francisco, CA, USA), Jul. 27–28, 2009, pp. 35–42.

[4] M. Owaida, N. Bellas, K. Daloukas, and C. Antonopoulos, "Synthesis of platform architectures from OpenCL programs", in *Proc. of the Intl. Symposium on Field-Programmable Custom Computing Machines (FCCM)*, Salt Lake City, UT, USA, May 1–3, 2011, pp. 186–193.

[5] D. Singh, "Implementing FPGA design with the OpenCL standard", *Altera whitepaper*, 2011.

[6] M. Lattuada and F. Ferrandi, "Exploiting outer loops vectorization in high level synthesis", in *Proc. of the 28th Intl. Conf. on Architecture of Computing Systems (ARCS)*, ser. Lecture Notes in Computer Science (LNCS), vol. 9017, Springer, 2015, pp. 31–42.

[7] P. Feautrier and C. Lengauer, "Polyhedron model", in *Encyclopedia of Parallel Computing*, D. Padua, Ed., Springer, 2011, pp. 1581–1592.

[8] C. Alias, A. Darte, and A. Plesco, "Optimizing remote accesses for offloaded kernels: Application to high-level synthesis for FPGA", in *Proc. of the Conference on Design, Automation and Test in Europe (DATE)*, 2013, pp. 575–580.

[9] L.-N. Pouchet, P. Zhang, P. Sadayappan, and J. Cong, "Polyhedral-based data reuse optimization for configurable computing", in *Proc. of the ACM/SIGDA Intl. Symposium on Field Programmable Gate Arrays*, ACM, 2013, pp. 29–38.

[10] P. Li, L.-N. Pouchet, and J. Cong, "Throughput optimization for high-level synthesis using resource constraints", in *Proc. of the 4th Intl. Workshop on Polyhedral Compilation Techniques*, (Vienna, Austria), S. Rajopadhye and S. Verdoolaege, Eds., Jan. 2014.

[11] D. G. Bailey, "Image border management for FPGA based filters", in *Proc. of the Sixth IEEE Intl. Symposium on Electronic Design, Test and Application (DELTA)*, IEEE, 2011, pp. 144–149.

[12] M. Rafi and Najeeb-ud-Din, "A novel arrangement for efficiently handling image border in FPGA filter implementation", in *Proc. of the 3rd Intl. Conf. on Signal Processing and Integrated Networks (SPIN)*, IEEE, 2016, pp. 163–168.