A Functional Approach to Border Handling in Image Processing

Leonard G. C. Hamey
Department of Computing
Faculty of Science
Macquarie University
Sydney NSW Australia
len.hamey@mq.edu.au

Abstract—**Domain Specific Languages for image processing offer simplified programming and efficient execution on a variety of platforms. These languages are particularly suitable for implementing local image processing operators that produce each output pixel by processing a corresponding window of input pixels. Such operators are easily parallelised and vectorised for efficiency. However, border handling is required to define the results for pixels close to the image border where the input window partially falls outside the bounds of the input image. We propose a declarative approach to border handling in a functional image processing Domain Specific Language called Halide*. The compiler uses code analysis to infer the bounds of output images and to optimise border padding implementations. Experimental results demonstrate good execution efficiency for large images.**

I. Introduction

Image processing and low-level image analysis operators are computationally intensive. Local operators such as sharpening, smoothing and edge detection involve performing the same computation at every pixel location, yielding another image or a list of extracted features. These local operators are easily parallelised and suitable for graphics hardware. Programming graphics hardware is more difficult than normal programming and requires reprogramming as new GPU programming models arise. Domain Specific Languages (DSLs) simplify programming and provide portability between different hardware while still obtaining an efficient implementation.

A variety of DSL approaches have been proposed. Apply [1], [2] expresses the computation to be performed at a single pixel in a language based on Ada. The Apply module is compiled into hardware-specific code that efficiently processes entire images. Brook [3] is a stream processing language that similarly expresses per-pixel computation as kernel functions in a language based on C and compiled to GPU code. ZPL [4] is an array processing language that expresses image processing using whole-array operations such as element-by-element addition. In ZPL, local window operators are implemented by translating the array coordinates to access neighbouring pixels, and conditionals are implemented as an array select operator. All of these DSLs are imperative languages based on the traditional concept of sequences of statements.

Although not strictly programming languages, libraries such as OpenCV [5] and HALCON [6] provide image processing operators as primitives embedded in a general purpose language. These libraries can be seen as Domain Specific Languages providing features including image data representations and image processing operators. Such libraries support applications development but are not intended for the programmer to express new image processing operators.

Functional languages offer a declarative programming paradigm. The programmer defines functions that are evaluated to compute the desired outcome, but there is little or no concept of explicit order of execution. Functional image processing DSLs such as IMPEL [7] and Halide [8], [9] express the computation as functions from one image to another. In functional image processing, images are functions and operators are higher-order functions that transform input image functions into an output image function.

Our work aims to address some of the limitations of functional image processing, especially in Halide. We remove Halide's assumption that the image functions are defined over infinite domains. Real images are defined over finite domains, so Halide's assumption is unrealistic. Halide programmers must specify the actual output image dimensionsand errors in this specification produce unexpected runtime failures. We introduce finite image domains as a solution to this problem.

The issue of image domains is related to border handling. When a local image processing operator is applied, the output is usually defined on the same domain as the input. However, if the operator uses a neighbourhood of input pixels then the output value is strictly undefined near the borders of the image. Border handling is the process of modifying the algorithm or its input data so as to compute approximate output values for the border-dependent pixels. For example, a common border handling method is to pad the input image by replicating the border pixels.

As a concrete example of these ideas, this paper introduces Halide*, a modified version of Halide that includes domain inference and border handling. We discuss the programmer's view of this approach, along with issues of execution efficiency. Our study shows that domain inference and functional border handling extend the capabilities of Halide without loss of execution efficiency.

Section II introduces functional image processing in Halide*, explaining key aspects of the language. Section III then introduces key ideas of border handling and discusses

```
Func sobel (Func in)
{
  Func h("sobel_horiz"), v("sobel_vert");
  Func sob("sobel");
  Var x("x"), y("y");

  h(x,y) = in(x+1,y-1) + 2*in(x+1,y) +
           in(x+1,y+1) - in(x-1,y-1) -
           2 * in(x-1,y) - in(x-1,y+1);
  v(x,y) = -in(x-1,y+1) - 2*in(x,y+1) -
           in(x+1,y+1) + in(x-1,y-1) +
           2 * in(x,y-1) + in(x+1,y-1);
  sob(x,y) = (abs(h(x,y)) +
              abs(v(x,y))) / 4;

  return sob;
}
```

Fig. 1: Halide implementation of Sobel

applying different padding methods to different dimensions of an image. Domain inference is discussed in section IV and execution efficiency is discussed in section V.

II. Functional Image Processing in Halide*

Halide [8], [9] is a functional image processing DSL embedded in C++. Halide modules are written using C++ syntax then Just-In-Time (JIT) compiled for run time execution. The C++ embedding imposes some boilerplate on the program code, such as C++ type declarations. C++ arithmetic operators are overloaded to provide Halide semantics, and Halide library functions and class methods provide additional semantics. Halide*, as described in this paper, is a development branch of Halide that includes domain inference and border handling with relevant code optimisations.

For example, consider the well known 3×3 Sobel edge detector which is defined with two convolution masks as follows.

$$H = \begin{bmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{bmatrix}, \qquad V = \begin{bmatrix} -1 & -2 & -1 \\ 0 & 0 & 0 \\ 1 & 2 & 1 \end{bmatrix}$$

Here, H and V compute horizontal and vertical discrete derivatives centered at the pixel under consideration. The Sobel edge magnitude is a norm of the resulting gradient vector; for speed of computation, the ℓ_1 norm is often used. This computation can be expressed with three function definitions as shown in figure 1.

The C++ objects `in`, `h`, `v` and `sob` are Halide image functions where the parameters `x` and `y` are the column and row indices respectively. The three function definitions of `h`, `v` and `sob` are Halide's functional representation of the Sobel operator while the remainder of the body of `sobel` is boilerplate resulting from the embedding in C++. In particular, C++ type declarations are required for Halide functions

```
Func sobel (Func in)
{
  Func h("sobel_horiz"), v("sobel_vert");
  Func sob("sobel");

  h() = in[1][-1] + 2 * in[1][0] +
        in[1][1] - in[-1][-1] -
        2 * in[-1][0] - in[-1][1];
  v() = -in[-1][-1] - 2 * in[0][-1] -
        in[1][-1] + in[-1][1] +
        2 * in[0][1] + in[1][1];

  sob() = (abs(h) + abs(v)) / 4;
  return sob;
}
```

Fig. 2: Basic Halide* implementation of Sobel

(`Func`) and index variables (`Var`). The C++ function `sobel` itself is a higher-order Halide function that applies the Sobel computation to the Halide image function parameter `in` and returns a newly defined image function `sob`.

The three function declarations tidily define the Sobel operator without introducing explicit sequence of execution. This makes it relatively easy for the Halide compiler to introduce parallel and vectorised computation, or even to off-load the computation to a Graphics Processing Unit (GPU).

For comparison, figure 2 shows the equivalent Halide* implementation of the Sobel edge magnitude operator. Halide* introduces a neighbourhood notation for expressing local operators without the need of explicit index variables. This notation is not only more compact and less error prone than the standard Halide notation, but it also identifies the code as an implementation of a local operator, information that is required for domain inference. However, this Halide* implementation of Sobel does not include any border handling — it is exactly equivalent to the Halide implementation in figure 1.

III. Border Handling

Border handling is an important concept in image processing, particularly for local operators. Border handling provides approximate output pixels close to the borders of the image where insufficient data is available to fully compute the local operator. For example, the Sobel 3×3 operator (see section II) requires the immediately adjacent pixels of each input pixel. This operator cannot be computed in the normal manner at the borders of the input image because the adjacent pixels are not available. In general, local operators that use a non-trivial input window require some form of border handling. For this reason, border handling is commonly provided in libraries and tools that support image processing; some example are shown in table I.

There are many possible approaches to border handling. The three main approaches are: cropping the output image so as to eliminate the border-dependent pixels, padding the

TABLE I: Available border handlers in image processing environments

Environment	Replicate	Reflect	Reflect101	Wrap	Constant	Zero
OpenCV[10]	Replicate	Reflect	Reflect101	Wrap	Constant	
Matlab[11]	Replicate	Symmetric		Circular		
Scipy[12]	Nearest	Reflect	Mirror	Wrap	Constant	
Java Advanced Imaging[13]	Copy	Reflect		Wrap	Constant	Zero
Intel IPP[14]	Repl	MirrorR	Mirror	Wrap	Const	Zero
Mathematica[15]	Fixed	Reversed	Reflected	Periodic	by value[a]	
HALCON[16]	Continued		Mirrored	Cyclic	by value[a]	

[a]A constant pixel value for the Padding parameter specifies constant border padding.

input image(s) to approximate the missing input pixels, and modifying the processing operator itself so that it computes approximate results when the available data is limited. The first two approaches are the most common and are considered in the current work.

Cropping computes only the pixels where the local operator's input window lies fully inside the input image. This elimiinates the need for any approximation at the cost of reducing the image size, so it is only appropriate when the size reduction is acceptably small. For example, cropping is used in the denoising and demosaicing algorithm employed by a commercial camera [8] where the sensor size is considerably larger than the desired final image size. Cropping is not a viable solution when the processing operator uses a large window, or when multiple stages or iterations combine to produce a large effective window size. In these cases, it is better to obtain approximate results for the border-dependent pixels than to completely discard them.

Border padding is the most common way to compute approximate results in the border-dependent region. Border padding extrapolates the input image, providing synthetic input pixels beyond the bounds of the image. The quality of the approximate results will depend on the padding technique employed, but it will also depend on the processing algorithm and on the nature of the image data. This means that the programmer needs a variety of padding methods from which to choose and the ability to select the padding method to be used in a particular operator.

There are five well known padding methods that are commonly provided in image processing environments. Table I displays the support for each method in several environments and table II demonstrates the implementation of each padding method in a single dimension. The names vary, so we adopt OpenCV's names where appropriate.

"Replicate" copies the border pixels as padding beyond the image bounds. For example, all row positions above the top of the image are replications of the top row. Replicate padding produces stripes radiating from the image.

"Reflect" and "Reflect101" mirror the pixels at the border of the image. The difference between them is that "Reflect" copies the border pixels producing two identical rows and columns at the border, whereas "Reflect101" does not copy the border pixels. Reflection produces corner artifacts in visual edges that cross the image boundary at an angle.

TABLE II: Image Padding Methods

Method	Example of padded pixels		
	Pad	Image	Pad
Replicate	a a a a	a b c d e f	f f f f
Reflect	d c b a	a b c d e f	f e d c
Reflect101	e d c b	a b c d e f	e d c b
Wrap	c d e f	a b c d e f	a b c d
Constant m	m m m m	a b c d e f	m m m m
Tile 2	a b a b	a b c d e f	e f e f

"Wrap" copies pixels from the opposite side of the image, effectively mapping the image onto a torus. For example, the wrapped row immediately above the top row of the image is a copy of the bottom row. Although wrapping is not typically useful for natural images, it is very appropriate for computed images such as Fourier transforms and polar coordinate transforms where pixels on opposite borders are computationally adjacent.

"Constant" supplies a constant value for pixels outside the borders of the image. "Zero" is a special case that pads the image with constant zero pixels. Constant is typically not suitable for natural images, but zero padding can be appropriate for processed images such as edge magnitude where zero would mean "no edge detected".

Halide*'s library contains an implementation of these five padding methods, together with "Crop". The library also includes "Tile", a novel method that pads the image by replicating small tiles of pixels adjacent to the border as shown in table II. A parameter specifies the size of the tiles. Tile padding is particularly useful for Bayer coded colour images because it preserves the Bayer colour pattern.

A. Declarative Border Handling

In libraries such as OpenCV [10], border handling is indicated by a parameter to the image processing operator. This declarative specification allows the programmer to select the desired border handling method but avoids an extra processing step by deferring processing of the border handler to the

```
Func sobel (Func in,
     BorderFunc bdr = Border::replicate)
{
  Func h("sobel_horiz"), v("sobel_vert");
  Func sob("sobel"), b;

  b = bdr(in);

  h() = b[1][-1] + 2 * b[1][0] +
        b[1][1] - b[-1][-1] -
        2 * b[-1][0] - b[-1][1];
  v() = -b[-1][-1] - 2 * b[0][-1] -
        b[1][-1] + b[-1][1] +
        2 * b[0][1] + b[1][1];

  sob() = (abs(h) + abs(v)) / 4;
  return Border::crop(sob);
}
```

Fig. 3: Halide* implementation of Sobel with Border Handling

operator. In Halide*, function applications are declarative so multiple processing steps can be combined by the compiler into a single pass through the image. In particular, border padding could be applied explicitly by the programmer before passing the input image for processing and the compiler could inline the border padding into the subsequent processing step. However, we prefer to specify the border handler as a declarative parameter to a higher-order function. This approach is easier to use, more consistent with existing approaches, and it provides the opportunity for future expansion to include more sophisticated border handlers that cannot be written as modifications to the input image before processing.

Halide* implements border handlers as a special type of higher-order function called `BorderFunc`. These functions have additional capabilities including the ability to be applied selectively to individual dimensions of an image as discussed below. Advanced programmers can define novel border handlers by writing a C++ subclass of BorderFunc with methods that embody the Halide* code to implement the border handler. The existing library supports all the methods shown in table II.

Figure 3 presents a Halide* module that implements the 3×3 Sobel operator with border handling. The parameter `bdr` is optional and defaults to "Replicate" border padding which is likely to be an appropriate choice for this operator when applied to ordinary images. The application of the border handler within `sobel` is straightforward and the implementation of the Sobel operator itself is essentially the same as in figure 2.

The application of `crop` to the final result in figure 3 deserves explanation. This is a recommended boilerplate statement that prevents other image processing operators from accidentally exploiting sobel's border padding.

B. Combined Padding

In some situations it is appropriate to apply different padding methods to different dimensions of an image. For example, an image in polar coordinates could be padded with "Wrap" in the angle dimension and "Replicate" in the distance dimension. In Halide* this capability is provided by a higher-order function called `border` that accepts several border padding functions and returns a border handler that applies each padding method to a separate dimension. Thus, `border(wrap, replicate)` returns a border handler that applies "Wrap" to the first dimension of the image and "Replicate" to the second dimension.

Since `border` applies different handlers to different dimensions of the image, the padded result could potentially depend on the order in which the padding methods are applied. In particular, the padded corners may be computed in one of two ways. If the rows are padded first, then the corners are column padded from the previously padded rows. On the other hand, if the columns are padded first, then the corners result from row padding of the previously padded columns. If both methods of computation always yield the same results then the padding methods are said to commute.

Commuting padding methods benefit the programmer because they do not have to be concerned about dependencies upon the order of application of the padding methods. For example, when padding an image in polar coordinates, the programmer can apply "Wrap" padding to the angle dimension and "Replicate" to the distance dimension and the results will be semantically identical irrespective of whether the angle dimension is `x` or `y`.

Commuting padding methods also benefits the compiler because it is free to apply the padding in any order it chooses. Depending on the padding methods, this may allow the compiler to simplify the computation.

Figure 4 explores the commutativity of different padding methods. The figure shows the top-left corner of an image, with three rows and columns of padding above and to the left. Figure 4a shows the result of padding rows with "Replicate" and columns with "Reflect". This padding commutes, because the upper-left corner of the figure is consistent with padding either the rows to the right or the columns below. Similarly, figure 4b demonstrates that "Constant" padding commutes with "Reflect". In this case, if the rows are padded first then "Reflect" copies the constant into the corner, whereas if the columns are padded first then "Constant" fills the corner with the constant pad value. In constrast, figures 4c and 4d demonstrate that padding with two different constants yields different results depending on whether rows or columns are padded first.

It can be shown that all the padding methods listed in table II commute with each other, except that "Constant" paddings with two different constants do not commute. Each of the padding methods "Replicate", "Reflect", "Reflect101" and "Wrap" is implemented by applying a particular range limiting function to the index expressions used to access the underlying

g	g	g	g	h	i
d	d	d	d	e	f
a	a	a	a	b	c
a	a	a	a	b	c
d	d	d	d	e	f
g	g	g	g	h	i

(a) Replicate, Reflect padding

m	m	m	g	h	i
m	m	m	d	e	f
m	m	m	a	b	c
m	m	m	a	b	c
m	m	m	d	e	f
m	m	m	g	h	i

(b) Constant, Reflect padding

m	m	m	n	n	n
m	m	m	n	n	n
m	m	m	n	n	n
m	m	m	a	b	c
m	m	m	d	e	f
m	m	m	g	h	i

(c) Columns padded first

n	n	n	n	n	n
n	n	n	n	n	n
n	n	n	n	n	n
m	m	m	a	b	c
m	m	m	d	e	f
m	m	m	g	h	i

(d) Rows padded first

Fig. 4: Commuting and non-commuting padding

pixels. For example, "Replicate" is implemented by truncating index values that are out of bounds to the nearest limit. With these *index limiting* padding methods, each dimension's index expression is range limited independently of the others, so the order of evaluation has no impact on the result. It follows that all range limiting padding methods commute provided that the range limits do not introduce dependencies between the index expressions.

A similar argument shows that index limiting padding methods commute with constant padding, as demonstrated in figure 4b. If the constant is padded first, then the index limiting method replicates the constant into the corners. On the other hand, if the index limiting method is applied first then the corners are directly filled by the constant padding method. In both cases, the corners are filled with the constant pad value. In contrast, constant padding methods do not commute with each other unless the constant values are the same because whichever constant is applied last will pad the corners as shown in figures 4c and 4d.

The argument given here can be further generalized to padding methods that compute weighted linear combinations of pixels selected from the same row or column, provided that the sum of the weights is one. For example, the "Extend" padding method is defined as twice the "Replicate" pad value minus the "Reflect101" pad value; this padding method, adapted from Szeliski's description [17], has weights summing to one. The method is attractive because it is continuous in the first derivative unlike the index limiting padding methods. It is easy to show that such weighted linear padding methods commute with others of the same kind because the corners are padded with a linear combination of a linear combination of pixels. Index limiting padding methods are a special case of these weighted linear combinations so they also commute with each other. Finally, with the weights summing to one, the linear combination faithfully copies constant padding into the corners if the constant padding is applied first, so these padding methods commute with constant padding.

With this understanding of commuting padding operators, the semantics of `border` is usually well-defined without considering the order of application of padding methods. The analysis highlights the case where different constants are used to pad different dimensions as a potential problem where the programmer should carefully consider how the corners are to be padded.

IV. Domain Inference

In Halide, computations produce image functions that are defined on infinite discrete domains. However, concrete images are arrays of pixels values that are treated as functions defined on finite discrete domains. This distinction between Halide functions and concrete images presents a problem for border handling because border handling is only applicable to finite domains. Although Halide functions are conceptually infinite, they are often actually partial functions that can only be computed over the restricted domain where the concrete input images are defined. Halide does not provide the programmer with any information about these implied restrictions on the function domain except that Halide detects a run time error if an attempt is made to access outside the bounds of a concrete image. The Halide programmer is responsible for ensuring that their program will not cause such access violations, either by padding the concrete input images, or by explicitly computing the appropriate domain for the output image. Halide provides no language support for the concept of image functions with finite domains.

Halide* introduces finite rectangular domains for image functions. This makes image functions and concrete images equivalent — border handling can be applied to both kinds of images. Thus, Halide* makes it possible to apply border handling to each processing step. This means that the programmer can select the most appropriate border handler to use at each stage of processing, whereas Halide only supports border handling for the initial concrete images. An additional benefit of Halide*'s support for finite domains is that the compiled program can compute the dimensions of the result image, relieving the programmer of this task.

Halide* defines two rectangular domains for each image function. The *valid* domain represents the bounds of the meaningful image data while the *computable* domain represents the bounds of pixels that can be computed without causing an access violation. Thus, the valid domain represents semantic limits while the computable domain captures computational limits. Because pixels that cannot be computed are certainly not valid, Halide* also enforces a rule that the valid domain cannot exceed the computable domain.

For concrete images, the two domains are identical and represent the actual image bounds. For image functions, the two domains may be different. In particular, padded images have an expanded computable domain that is usually infinite, but the valid domain is identical to the original image because

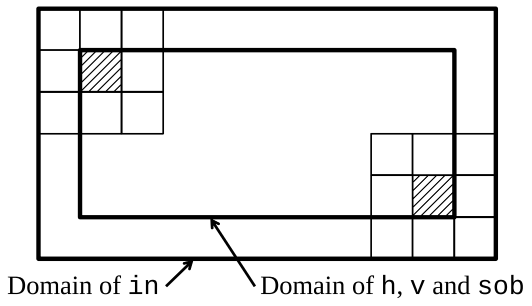


Fig. 5: Computable domains of sobel (fig. 1)

padding does not create additional meaningful image data. Padding always expands on the valid domain, so if padding is applied to an image that has already been padded, the previous padding is overridden.

When a computation is applied to images (including both concrete images and image functions), Halide* determines the domain of the resulting image function using *domain inference*. Domain inference involves examining the calls to input image functions and symbolically computing the range of pixel locations that are computable and valid for the result image function.

The computable domain of a Halide* function is determined by inspecting the index expressions that access the input image functions. If the computable domain of the input image is finite then computability implies that the index expressions must fall within the domain of the input function. Since domains are rectangular, the input function's computable domain constrains the range of the index expression. Solving each constraint equation for the current function's index variables yields a range limit for the computable domain of the current function. The computable domain of the function is obtained by intersecting the relevant ranges. Trivially, if the computable domain of the input image is infinite then it does not constrain the computable domain of the result function.

For example, suppose a function contains the index expressions `x-1` and `x+2` in separate references to an image that is defined over the computable domain $[0, 100]$. In this case, solving the constraints $0 \leq \mathtt{x} - 1 \leq 100$ and $0 \leq \mathtt{x} + 2 \leq 100$ yields ranges of $[1, 101]$ and $[-2, 98]$ for `x`. The intersection of these ranges yields $[1, 98]$ as the computable domain of the function.

As a further example of computable domains, figure 5 shows the computable domain of the Sobel functions `h`, `v` and `sob` relative to the input computable domain. The figure shows two input windows that represent the limits of computability. The relevant index expressions in figure 1 are `x-1`, `x+1`, `y-1` and `y+1` in the definitions of `h` and `v`.

The valid domain of a Halide* function is obtained similarly to the computable domain but with one significant difference: The valid domain of a local operator is enlarged to the intersection of the input domains when sufficient padding is present. This captures the intent of border handling which is to preserve the dimensions of the input image in the result. To achieve this, Halide*'s domain inference ignores the local operator index offsets when it is computing the valid domain, whereas the computation of the computable domain includes the offsets. For example, the functions `h`, `v` and `sob` in figure 3 have the same valid domain as the input when padding is provided. In fact, Halide* computes the valid domain of a local operator by intersecting the input valid domains together with the computable domain of the result. If there is insufficient padding, the computable domain will be smaller than the intersection of the input valid domains and the valid domain of the result will then be restricted to its computable domain. In this way, Halide* correctly computes the valid domain of local operators whether padding is absent, partial or complete.

Domain inference as defined by Halide* is well defined for local image processing operators, even when they are combined with image flips, rotations by multiples of 90 degrees and resizing. For general image warping operators, the domain of the result may not be rectangular and therefore may not be representable as one of Halide*'s rectangular domains. In specific cases of image warping, it may not be possible to solve the constraints that define the domain. In these cases, Halide* ignores the unsolved constraints and provides a domain that is no smaller than the actual domain together with a flag indicating that the domain is inexact. In these cases, the programmer may choose to explicitly compute the result domain themselves. The partially solved domain is available to the programmer and may be useful in this computation.

V. Efficient Border Handling

Border handling introduces potential efficiency problems. Padding an image requires computing the pad pixels, either in a precomputation step or on demand. Precomputation of padded pixels introduces an additional computational pass and additional memory accesses, typically involving copying the entire image to a larger memory buffer. On-demand padding introduces conditional expressions that must be evaluated during pixel accesses to return the pad values. These overheads can dominate execution time and should be reduced as much as possible.

By default, Halide in-lines function calls, eliminating processing passes. However, in-lining can result in repeated computation of the in-lined function, so Halide provides *schedules* that can advise the compiler to buffer the intermediate computation results in memory and perform other optimisations [8]. Whether memory buffering is beneficial or not depends on the computation and the target CPU, so scheduling Halide functions is left to the programmer. In Halide*'s border handling library, the same schedule capabilities can be used to select between precomputation and on-demand padding.

A distinctive feature of image padding is that a large portion of the computation of the output image does not require the use of padded pixels. This border-independent region can be computed in the same way irrespective of whether padding is being applied or not. On the other hand, computation of the border-dependent region is only possible with padding and requires the computation of padded pixels. Efficiency can thus be improved by using specialised code to compute the border-independent region with the conditional expressions that result from padding. This optimisation is known as loop splitting or index-set splitting [18], [19].

Some existing image processing DSLs have included border handling together with index-set splitting for efficiency. This capability is easily implemented in DSLs where the dimensions of the processing window are specified; for example, the Apply compiler splits image processing into nine regions where the border handling conditions are different [2]. In functional languages, the dimensions of the processing window must be determined by analysis the index expressions in function calls. IMPEL [7] uses index-set splitting to optimise boundary conditions in local operators with simple integer offsets. Both of these DSLs apply index-set splitting to relatively simple border padding in local operators with concrete images as their input.

Halide* takes the idea of index-set splitting further. Halide* applies index-set splitting to border padding where the input image may itself be the result of other computations including resizing, rotations by 90 degrees and flipping. In-lining these image transformations yields complicated index expressions that must be analysed to identify the border-independent region. The Halide* compiler extracts the conditional expressions from these index expressions and simplifies them to obtain bounds for the border-independent region. This is similar to domain inference, and can use the same constraint equation solver, but the equations to be solved are derived from the conditional expressions that implement the border handlers.

Having determined the border-independent region, Halide* recursively splits the loops that process the image and then optimises the code in each loop based on the loop bounds. Using the known range of the loop variables, these optimisations eliminate the conditional expressions from the border-independent region and also simplify the conditionals in the border-dependent regions. Using index-set splitting in this way improves the efficiency of both precomputed and on-demand padding.

A. *Performance Results*

We tested the performance efficiency of Halide*'s border handling optimizations by measuring execution time over repeated runs of a collection of simple Halide* programs with various schedule options. The schedules included precomputation of padding, code vectorization, parallelization and index-set splitting (ISS) options in various combinations. The test programs were separable 2D convolutions with different window sizes, adding two diagonally separated pixels, and Sobel edge detection. These programs are representative of typical simple image processing operators.

Performance tests were performed on square images of different sizes. The experiments included 8, 16 and 32-bit integer pixels, and 32-bit floating point pixels. The test environment was an unloaded Intel Core-i7/870 2.93GHz system with 16GB RAM running Ubuntu 10.4.1. The CPU has an 8MB level 3 cache that is dynamically split between the processors. Each processor also has an individual 256KB level 2 cache.

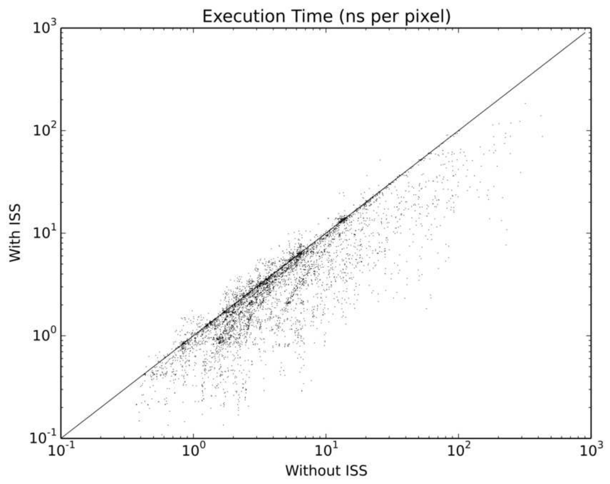


Fig. 6: Halide* execution time comparison

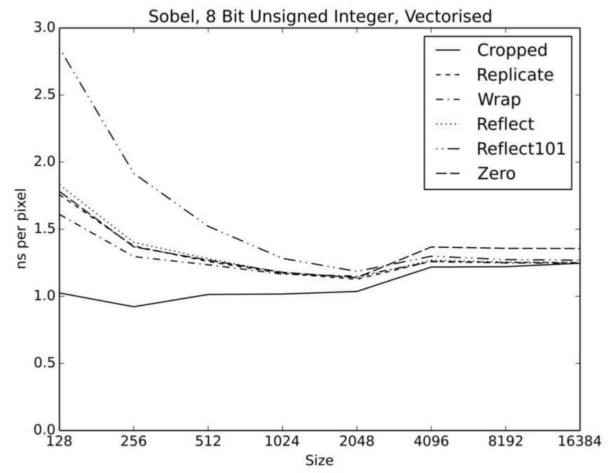
As an overview of the ISS optimization results, figure 6 directly compares the execution times for a variety of paired test cases with and without ISS. The 6000 plot points represent 12000 compiled modules — 2000 paired tests of each of the convolution, diagonal and Sobel operators. In 19% of our test cases, the ISS schedule was slower than the non-ISS schedule by up to a factor of two, but in 81% of cases, ISS improved execution time by a factor of up to 31 by reducing the unnecessary computation of border padding conditionals.

Figure 7 shows the execution time per pixel for the Sobel and 7×7 convolution operators for different image sizes and border handlers. Cropped border handling is included as a baseline that has no border handling overhead because the output image is computed only over the border-independent region. We expect larger images to have less border handling overhead per pixel because the border-dependent region is a smaller portion of the result image. The figure supports this, with an overhead of at most 16% for images at least 2048×2048. The increase in the baseline execution at the image size 4096×4096 is believed to result from misses on the level 3 cache when processing the 16MB images.

For comparison, without ISS the execution time per pixel is consistent across all image sizes. The execution time for 7×7 convolution ranges from 37 ns per pixel for "Wrap" padding to 61 ns for "Zero" padding. For Sobel, it ranges from 5.5 ns per pixel for "Wrap" to 14.2 ns for "Reflect101". Thus, ISS reduces execution time in these examples by a factor of up to 10 for large images.

VI. Conclusion

Border handling is important for local image processing operators to avoid reducing the dimensions of the output image. Libraries such as OpenCV provide border handling options that pad the input image to produce approximate output values in the border-dependent regions. We have adapted the



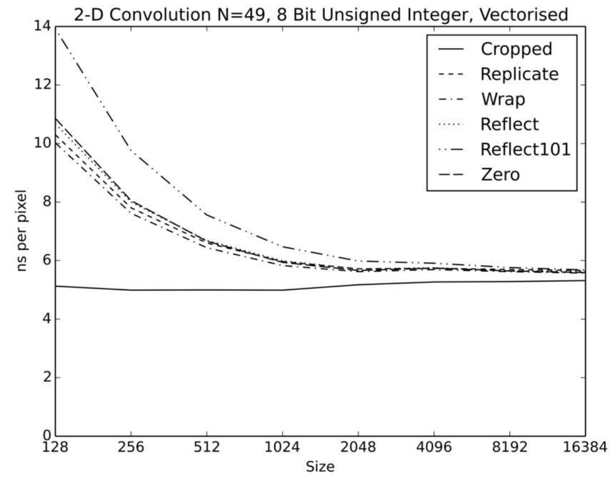


Fig. 7: Effect of image size on processing time per pixel

same idea to functional image processing, showing how the declarative style of functional programming naturally supports border handling options for higher-order functions.

We have implemented these ideas as an extension to the functional image processing language Halide. Because border handling is only relevant for finite images, we added domain inference to Halide* so that intermediate and output images can have finite domains. This allows border handling to be applied not only to concrete images but also to Halide* image functions. Halide*'s domain inference uses two domains — the valid domain represents the semantically meaningful portion of the image while the computable domain represents the region that can be computed without access violations. Using these two domains makes it possible to infer the result domain for local image processing operators whether or not the input images are padded.

Border handling is potentially computationally expensive due to introduced conditional expressions. Halide* addresses this problem through code optimisation. Our experimental results demonstrate that these optimisations are particularly effective for larger image dimensions.

Acknowledgment

The author is grateful for discussion and suggestions from Prof. Tony Sloane and Matthew Roberts and to the authors of Halide who have developed it as an open source project.

References

[1] L. G. C. Hamey, J. A. Webb, and I.-C. Wu, "An architecture independent programming language for low-level vision," *Computer Vision, Graphics and Image Processing*, vol. 48, no. 2, pp. 246–264, Nov. 1989.

[2] L. Hamey, "Efficient image processing with the Apply language," in *Digital Image Computing Techniques and Applications, 9th Biennial Conference of the Australian Pattern Recognition Society on*, 2007, pp. 533–540.

[3] I. Buck, T. Foley, D. Horn, J. Sugerman, K. Fatahalian, M. Houston, and P. Hanrahan, "Brook for GPUs: Stream computing on graphics hardware," *ACM Trans. Graph.*, vol. 23, no. 3, pp. 777–786, Aug. 2004. [Online]. Available: http://doi.acm.org/10.1145/1015706.1015800

[4] B. Chamberlain, S.-E. Choi, C. Lewis, C. Lin, L. Snyder, and W. Weathersby, "ZPL: a machine independent programming language for parallel computers," *Software Engineering, IEEE Transactions on*, vol. 26, no. 3, pp. 197–211, 2000.

[5] OpenCV development team, "OpenCV 2.4.11.0 documentation," 2015 (accessed July 7, 2015). [Online]. Available: http://docs.opencv.org/

[6] MVTec Software GmbH, "HALCON operator reference version 12.0," 2014 (accessed July 7, 2015). [Online]. Available: http://www.mvtec.com/download/reference/

[7] T. L. Kay and L. I. Rudin, "IMPEL: a domain-specific image processing environment, language, and compiler," in *Proc. SPIE 2277, Automatic Systems for the Identification and Inspection of Humans*, 1994, pp. 265–274.

[8] J. Ragan-Kelley, A. Adams, S. Paris, M. Levoy, S. Amarasinghe, and F. Durand, "Decoupling algorithms from schedules for easy optimization of image processing pipelines," *ACM Trans. Graph.*, vol. 31, no. 4, pp. 32:1–32:12, Jul. 2012.

[9] J. Ragan-Kelley, C. Barnes, A. Adams, S. Paris, F. Durand, and S. Amarasinghe, "Halide: A language and compiler for optimizing parallelism, locality, and recomputation in image processing pipelines," in *Proceedings of the 2013 ACM Sigplan Conference on Programming Language Design and Implementation*, 2013.

[10] OpenCV development team, "OpenCV 2.4.11.0 documentation: Image filtering," 2015 (accessed July 7, 2015). [Online]. Available: http://docs.opencv.org/modules/imgproc/doc/filtering.html

[11] The MathWorks, Inc, "Pad array - MATLAB padarray," 2013. [Online]. Available: http://www.mathworks.com.au/help/images/ref/padarray.html

[12] The Scipy community, "SciPy v0.12 reference guide (DRAFT): scipy.ndimage.filters.convolve," 2013. [Online]. Available: http://docs.scipy.org/doc/scipy/reference/generated/scipy.ndimage.filters.convolve.html

[13] Oracle, "Java advanced imaging: Class BorderExtender," 2013. [Online]. Available: http://docs.oracle.com/cd/E17802_01/products/products/java-media/jai/forDevelopers/jai-apidocs/javax/media/jai/BorderExtender.html

[14] Intel, "Intel integrated performance primitives for Intel architecture reference manual 7.1: FilterGaussBorder," 2012. [Online]. Available: http://software.intel.com/sites/products/documentation/doclib/ipp_sa/71/ipp_manual/IPPI/ippi_ch9/functn_FilterGaussBorder.htm

[15] Wolfram Research, "Mathematica 9 documentation: Padding," 2013. [Online]. Available: http://reference.wolfram.com/mathematica/ref/Padding.html

[16] MVTec Software GmbH, "HALCON operator reference version 12.0: convol_image," 2014 (accessed July 7, 2015). [Online]. Available: http://www.mvtec.com/download/reference/convol_image.html

[17] R. Szeliski, *Computer Vision: Algorithms and Applications*. Springer, 2011, pp. 101–102.

[18] R. Sakellariou, "Partitioning loop nests containing conditionals for automatic parallelisation," in *Proceedings of the 3rd Hellenic-European Conference on Mathematics and Informatics*, E. A. Lipitakis, Ed. Athens: Lea Press, September 1996, pp. 580–587.

[19] M. J. Wolfe, *High Performance Compilers for Parallel Computing*. Addison-Wesley, 1996.